Voting-based Approximation of Dependability Attributes and its Application to Redundancy Schemata in Distributed Computing Environments

Proefschrift voorgelegd tot het behalen van de graad
van Doctor in de Wetenschappen: Informatica aan
de Universiteit Antwerpen te verdedigen door

Jonas Buys

Promotoren
dr. Vincenzo De Florio
prof. dr. Chris Blondia

Faculteit Wetenschappen
Departement Wiskunde-Informatica
Antwerpen, 2020

Universiteit Antwerpen

"What though the radiance which was once so bright
be now for ever taken from my sight,
though nothing can bring back the hour
of splendor in the grass, of glory in the flower;
we will grieve not, rather find
strength in what remains behind.
In the primal sympathy
which having been must ever be;
in the soothing thoughts that spring
out of human suffering;
in the faith that looks through death,
in years that bring the philosophic mind" [1].

— William Wordsworth (1770–1850)

I will forever cherish the view where, upon every depart, you both waved me goodbye:
you with your sparkling eyes accompanied with the most sincerest of barking art,
and you with your loving smile whilst tenderly holding our beloved pet.

In commemoration of my caring, sanguine grandmother Norma, and Blacky, my
faithful quadrupedal companion.

With love, and the most sincere feelings of thankfulness and gratitude.

Universiteit Antwerpen

Voting-based Approximation of Dependability Attributes and its Application to Redundancy Schemata in Distributed Computing Environments

Proefschrift voorgelegd op 12 oktober 2020 tot het behalen van de graad van Doctor in de Wetenschappen: Informatica, bij de faculteit Wetenschappen, aan de Universiteit Antwerpen te verdedigen door

Jonas Buys

PROMOTOREN:
dr. Vincenzo De Florio
prof. dr. Chris Blondia

Antwerpen, 2020

IDLab
INTERNET & DATA LAB

Acknowledgements

After a long period of hard work — a period that occasionally seemed to be endless and the task to be completed often inachievable — it is with a sense of pride that I am submitting this dissertation for review, hoping that it will be accepted as the final piece of my doctoral research programme. It was an enlightening journey during which I have learned much about myself and struggled with my flaws. It was a period during which little successes caused sporadic blisses with sentiments of joy and victory, which were often offset by prolonged periods of frustration and feelings of doubt and agitation. Time and time again, I have been confronted with my strong as well as my weak characteristics. Looking back on the past ten years, I have learned one valuable lesson: that sometimes weak characteristics like being stubborn and extremely demanding, can actually be leveraged for the greater good.

Before continuing, I insist on expressing my **sincere gratitude** to many people who have supported me in this endeavour. People without whom this result could not possibly have been achieved. People who have shown to believe in me, who have offered me many unique opportunities. People who have shown remarkable patience...

First and foremost, I'd like to thank my tutor, **dr. Vincenzo De Florio**. He has been there all along the way, guiding me, offering valuable advice and insights, building on his experience. In times of doubt and concern due to lack of success, he was there to seek refuge, to listen and express his sympathy, and to offer advice and suggesting how to proceed. He was there at night, during the weekend, even during his holidays abroad. He has shown to be a most remarkable and amiable person, and I want to emphasise how privileged I feel to have had the opportunity to work together with him.

During my stay at the department of Mathematics and Computer Science, there have been numerous opportunities to seise: teaching assignments, research projects, attending and/or presenting at conferences, just to list a few. Most of these were arranged by **prof. dr. Chris Blondia**, to whom I am indebted for arranging all of these, and for his proficient management of the ledgers and balance sheets.

Another person from my doctoral committee to thank is **prof. dr. Serge Demeyer**, for repeatedly expressing his belief in me, and for his sharp, albeit right questioning. Questions that necessitated self-reflection and that urged an assessment of the current state of affairs.

I would also like to thank the other members of the jury: prof. dr. Francky Catthoor, prof. dr. Peter Hellinckx and prof. dr. Mohamed Bakhouya. I really appreciate you took the time to review this dissertation, and providing me with very constructive and useful comments and suggestions. Without doubt, they have

led to some adjustments in this manuscript that can only benefit its readability, completeness, and — in general — the applicability of this work.

In addition to expressing my gratitude for the support and help received from the members of my doctoral committee, I find it appropriate to take the opportunity and thank all those that may not have been directly involved in this research, but that did contribute to the realisation of this dissertation in particular.

I have been blessed with **two wonderful parents**, whom I love dearly. Amid the daily hustle and bustle, the care and unwavering, wholehearted support they are giving me — and that they have been giving me ever since my very existence — is too easily taken for granted or forgotten. I vividly remember their reaction when I entered the house and told them I graduated and obtained my master's degree. This memory exemplifies their pure desire for my well-being and personal and professional development. They have always given all, sustaining me all along. Thank you for providing me the environment and the tranquillity needed to concentrate and to write this dissertation.

Next, there is the **love of my life**: my lovely Jessica. Being a cute procrastinate herself, she has confronted me with my very own procrastinatory skills in a subtle way that characterises her, urging me to get the job done and turn the page and start the next chapter in our lives.

Adding to that, there are **many friends** who should be mentioned, in particular Silas De Munck, Joachim Ganseman, Nicolas Letor, Christophe Mertens and Bart Cornelis, to name a few. Some I have known for over two decades; others I met during my stay at university. Some were former colleagues, and we've enjoyed many pleasant talks at the office. Their recurring questions like "are you going to finish?" and "when do you hope to submit?", or statements like "just wrap it up and submit!" were the final incentive to complete this task.

It is with a warm, yet modest feeling of complacency and thankfulness that I am looking back on the past decade. Without doubt, this has been one of the greatest challenges in my life so far. There is no regret, in spite of the many moments of doubt, frustration and anxiety, for I have learned more about myself, and especially: I have had the privilege to enjoy working and living with all the extraordinary people who have supported me all along.

Contents

List of Figures

Abstract

Business- and mission-critical distributed applications are increasingly expected to exhibit highly dependable characteristics, particularly in the areas of availability and QoS-related factors such as timeliness. For this type of applications, a complete cessation or a subnormal performance of the service they provide, as well as late or invalid results, are likely to result in significant monetary penalties, environmental disaster or human injury. However, software components deployed within distributed computing systems may inherently suffer from several types of impairments, such as long response times or temporary unavailability; the former which is mainly to be attributed to data exchanges (susceptible to network latencies and latencies resulting from potential resource contention), the latter due to functional failures having occurred (or an accumulation of parametric failures). Considering the compositional nature of many large-scale distributed applications, it is easily foreseeable that failures in the constituent components not properly dealt with can propagate and may subsequently perturb the service provided by the application.

Traditional redundancy-based fault-tolerant schemes, which were conceived to tolerate permanent hardware faults primarily and transient faults caused by external disturbances secondarily, do not offer sufficient protection for tolerating software faults (often referred to as design or specification faults) [2, 3]. Indeed, the peculiar characteristics to software require design diversity rather than simply replicating a specific software component [4, 5]. Replicating software would obviously incur replicating any residual dormant software fault. The rationale is that redundantly deploying multiple functionally-equivalent but independently implemented software components will hopefully reduce the probability of a specific software fault affecting multiple implementations simultaneously, thereby keeping the system operational.

The n-version programming (NVP) mechanism, a well proven design diversity pattern for software fault tolerance, was first introduced in 1985 as "the independent generation of $n > 1$ functionally-equivalent programs from the same initial specification" [6]. An n-version module constitutes a fault-tolerant software unit — a client-transparent replication layer in which all n programs, called versions, receive a copy of the user input and are orchestrated to independently perform their computations in parallel. It relies on a generic decision algorithm to determine a result from the individual outputs of the version employed within the unit. Many different types of decision algorithms have been developed, which are usually implemented as generic voters. Examples include, amongst others, majority, plurality and consensus voting [7, 8] [5, Chapt. 4].

Adopting classic redundancy-based fault-tolerant design patterns, such as NVP, in highly dynamic distributed computing systems does not necessarily result in the anticipated improvement in dependability. This primarily stems from the statically

predefined redundancy configurations hardwired within such dependability strategies, *i.e.* a fixed degree of redundancy and, accordingly, an immutable selection of functionally-equivalent software components, which may negatively impact the schemes' overall effectiveness, at least from the following two angles.

Firstly, a static, context-agnostic redundancy configuration may in time lead to a more rapid exhaustion of the available redundancy and, therefore, fail to properly counterbalance any disturbances possibly affecting the operational status (context) of any of the components integrated within the dependability scheme. The effectiveness of an NVP composite is largely determined by the dependability of the versions employed within. As elucidated in [3, Sect. 4.3.3], the use of replicas of poor reliability can result in a system tolerant of faults but with poor reliability. It is therefore crucial for the system to continuously monitor the operational status of the available resources and avoid the use of resources that do not significantly contribute to an increase in dependability, or that may even jeopardise the schemes' overall effectiveness.

Secondly, the amount of redundancy, in conjunction with the voting algorithm, determines how many simultaneously failing versions the NVP composite can tolerate. For instance, an NVP/MV scheme applying majority voting (MV) can mask failures affecting the availability of up to a minority of its versions. A predetermined degree of redundancy is, however, cost ineffective in that it inhibits to economise on resource consumption in case the actual number of disturbances could be successfully overcome by a lesser amount of redundancy. Reversely, when the foreseen amount of redundancy is not enough to compensate for the currently experienced disturbances, the inclusion of additional resources (if available) may prevent further service disruption.

In this thesis, a novel dependability strategy is introduced encompassing advanced redundancy management, aiming to autonomously tune its internal redundancy configuration in function of the observed disturbances. Designed to sustain high availability and reliability, this adaptive fault-tolerant strategy may dynamically alter the amount of redundancy and the selection of functionally-equivalent resources employed within the redundancy scheme. The remainder of this thesis is structured as follows: We first present the concept of NVP/MV schemata in Chapt. 2 and show how disturbances emerging from the activation of software design faults may put their effectiveness into jeopardy. A set of ancillary metrics is then set forth in Chapt. 3, allowing to capture contextual information regarding the environment in which the scheme is operating, particularly with respect to disturbances that challenge its effectiveness, and that enable to detect the proximity of hazardous situations that may require the adjustment of the redundancy configuration. After introducing another metric designed for approximating the operational status of individual resources in terms of reliability in Chapt. 4, we move on to elaborate on the internals of the proposed adaptive fault-tolerant strategy in Chapt. 5. An overview of the architectural framework for the discrete-event simulations used for the validation of the proposed dependability strategy is given in Chapt. 6, followed by an overview of the options are available to the designer for modelling the system and analysing its performance. Proceeding by reporting on the strategy's effectiveness analysis in Chapt. 7, we then elaborate on a prototypical service-oriented implementation before concluding by summarising the main findings reported and conclusions drawn throughout this dissertation.

CHAPTER 1

Introduction

"It is [...] estimated that the vast majority of computer failures originate from software faults, estimations ranging from 60 up to 90 percent" [9–11].

In this introductory chapter, we elaborate on how the dependability of software applications may be impaired, and how it can be safeguarded through the use of design techniques for application-level fault tolerance. We then proceed by outlining how the research reported throughout this dissertation relates to various research domains in the area of computer science in general, its objectives, and the challenges it is supposed to address.

Business- and mission-critical distributed applications are increasingly expected to exhibit highly dependable characteristics, particularly in the areas of availability and QoS-related factors such as timeliness. For this type of applications, a complete cessation or a subnormal performance of the service they provide, as well as late or invalid results, are likely to result in significant monetary penalties, environmental disaster or human injury. However, software components deployed within distributed computing systems may inherently suffer from several types of impairments, such as long response times or temporary unavailability; the former which is mainly to be attributed to network latency as the result of data exchange, the latter due to failures having occurred. Considering the compositional nature of many large-scale distributed applications, it is easily foreseeable that failures in the constituent components not properly dealt with can propagate and may subsequently perturb the service provided by the application.

However, what precisely is meant by service? And what makes software truely dependable? Exactly by what types of disturbance can a distributed application be impaired? How can it be shielded to avoid that such impairments would result in failure? And can failures be contained at all, preventing them from damaging and perturbing system-wide operations?

1.1 A Short Introduction to Software Dependability

Largely structured according to the dependability taxonomy set forth in [12], this section contains a broad overview of several dependability aspects, with a particular

focus on fault-tolerant techniques, and how they may be successfully leveraged for attaining and sustaining specific QoS levels. Before elaborating on software dependability, however, some fundamental concepts of software systems need to be clarified and explicitly defined.

Compositionality of Enterprise Applications From a functional perspective, an **enterprise application** is an ICT solution that has been purposefully designed and implemented so as to provide an automated end-to-end software solution in support of specific functionality and/or a specific set of business processes, in line with the customer's commercial, public, societal, and/or environmental objectives. From a merely technical perspective though, an ICT solution is synonymous with a **distributed computing solution**: it is the whole of software applications that are deployed and being executed on different computing devices, potentially located at geographically dispersed sites, and that rely on a dedicated network communication infrastructure for exchanging information[1].

Found to be an effective branch of software engineering, **component-based development** has proved successful in structuring and modularising the functionality, the implementation logic and complexity of ICT solutions into a set of dedicated software components [13, Sect. 5.2] [14, Chapt. 19]. "The essence of using components is to divide [...] functionality into units with maximal internal cohesion and minimal external coupling. Components are truly self-contained units of functionality" [15]. A software solution is typically assembled from a specific set of dedicated and interconnected software components, supplemented by adequate orchestration logic responsible for coupling the underlying components to one another, and coordinating any exchanges of data. A comprehensive definition can be found in [16, Chapt. 16], in which a component is defined as "a standardised and interchangeable software module that is fully assembled and ready to use and that has well-defined interfaces to connect it to clients or other components". The definition well and truly highlights three elementary properties: it must (i) implement a public interface, (ii) respond to invocation requests, possibly using a message exchange mechanism, and (iii) encapsulate its implementation logic and hide the implementation details within. In order to allow for remote access, the interface may accommodate non-functional information in addition to a mandatory list of supported operations and message formats — *cf.* WSDL and SOAP [17, 18].

Considering this compositional nature of ICT solutions and the role of software components as building blocks, the abstract notion of software entity is introduced for generalisation purposes [14, Sect. 11.3]. At the very lowest level, it can be used to refer to stand-alone, single-component software entities without dependencies of any kind. One level upward, composite software entities encapsulate logic for properly orchestrating other software entities encompassed or used within. A *binding* is said to have been established between two interacting software entities

[1] The focus lies on application components that can interact in a distributed context. Based on principles in SoA and microservices-oriented paradigms, these software entities pass data by exchanging messages with one another. The architectural trait of modularity can also be applied in the context of embedded devices, where it is more likely data (messages) are passed using shared storage, *e.g.* a commonly accessible area in a shared memory.

when they have been successfully linked such that they can exchange information[2]. In this arrangement, the entity supplying the information fulfils the role of *producer*; the entity that accepts this information fulfils the role of *consumer*. At the highest level of composition, the term is used to refer to the enterprise application in its entirety, that is to say the software system itself.

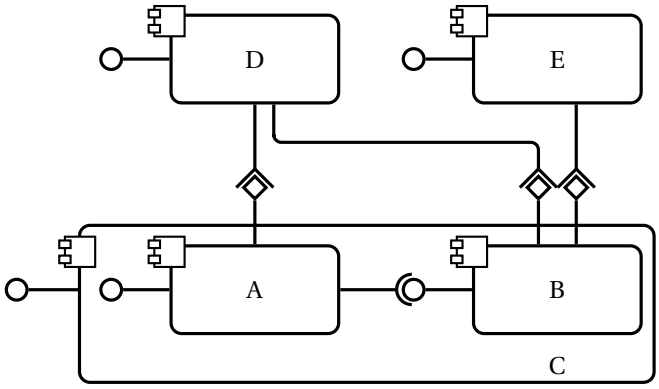


Figure 1.1: Software applications as assemblies of multiple software entities; interconnectivity can be achieved by composition and/or eventing. The software entities have been modeled as CCM components.

As the desire has been growing to integrate legacy IT assets in enterprise applications, so did the need for standardisation in order to bridge the technological disparities that inevitably surface due to the wide range of technologies used for the implementation of software entities. One such type of standard is the CORBA® Component Model (CCM) developed by the OMG®, which defines a comprehensive model that illustrates how software entities can be linked and assembled in contemporary distributed computing systems [20]. In doing so, it reveals two distinct, complementary architectural patterns for binding software entities: client-server architectures and event-driven architectures (EDAs). When zooming into the mechanism that data is passed between system components, the former architectural style maintains a request-response invocation pattern, where the initiative for triggering the exchange is *requested* by the consuming party (the client). The request may contain input data to be processed by some known functionality or software routine on the server. Rather than sending request arguments, in the event-driven approach, a set of events are defined that represent significant and interesting changes in system state. Upon their occurrence, a publish-and-subscribe messaging pattern will ensure that these events and the associated event data is *published* and propagated to any interested party that *subscribed* to be *notified* for a specific type of event. Clearly, it is the producing party (the server) that takes the initiative to notify the interested party, which will then have the responsibility to process the event data locally and obtain meaningful information from that.

[2]The terminology used here was inspired by the concept of the same name defined as part of the WSDL specification, although it is used here with a strong focus on the loose coupling of services and service composition at runtime. The naming is used in [19] in a very similar context.

– In the traditional **client-server architecture**, which remains the predominantly used architecture to date, a binding is established between two software entities: a client and a server, which correspond to the roles of consumer and producer, respectively [14, Sect. 12.2]. The invocation of the server entity is initiated by the client entity by issuing a request message destined for the server entity. As soon as the request message has been handed over to the server entity, the request will be served and processed, after which a response message is usually sent back to the client entity. When invoking an entity, a single operation — a handle to a specific software processing routine — is called that is defined in the entity's interface. An interface is a structured textual representation for describing the "specifics of how to access the underlying" functionality offered by the entity [21]. It comprises at the very least a set of exposed operations, the permissible message (payload) data types, and the applicable interaction patterns [22, Chapt. 7–9]. Several interface definition languages are available, though WSDL has become extremely popular, and is now widely and successfully used in XML-based SoA solutions [18]. A competing, and more recent language, is the OpenAPI specification (commonly referred to as Swagger), that has become the *de facto* standard for representational state transfer (REST)-based interfaces [23].
  An example of an invocation-based binding can be seen in Fig. 1.1, linking the software entities (components) A and B: the crescent-like receptacle is the client entity's sole point of interaction with the interface of the server entity B, which is depicted by the circular shape at the left. The receptacle represents a stage in the operations of the client entity A, during which some part of the functionality of the server entity B is explicitly called for by sending out the request message, until the response message has been received.
  The composite entity C is composed of the entities A and B, including the binding between these two underlying entities. This means that whenever C is invoked, it will call for the functionality of entity A, which in turn will call for the functionality of B. This call may be preceded and/or followed by additional programming logic, which typically includes data transformations and conversions. The underlying entities may or may not be packaged in the same deployment unit.

– **Event-driven architectures (EDAs)** have recently gained popularity, and rely on publish-and-subscribe models for the asynchronous exchange of specific data fragments called *events* [24]. In such type of models, there is an interested party — the event sink — that issues a subscription request, *i.e.* it states its interest in a specific type of events. The other party — commonly referred to as the event source, or the publisher — may accept the request, thereby pledging to send out event messages asynchronously[3] [22, Sect. 7.7]. That is to say, events are published without delay, avoiding the need for the sink to periodically poll.
  An event-driven binding links a specific event type that is advertised by a given software entity fulfilling the roles of event source and producer, to another entity that takes on the roles of event sink and consumer. As can be seen from software entity B in Fig. 1.1, an event source can produce several types of events. Each supported event type is depicted by a rhombic shape and corresponds with a specific data structure. Event instances and their payload are usually wrapped and

[3] Note that the initial subscription request follows a request-response messaging pattern. Event messages are, however, exchanged using a publish-and-subscribe model.

carried inside a predefined message format that depends on the standard used; an example is the WS-Notification set of specifications, which is frequently used in XML-based SoA solutions [24]. Entity D shows that an event sink can be capable to receive events of different types, though this depends on the implementation. A y-shaped receptacle can be observed for each supported event type the sink can interpret and process.

Both architectural patterns show great similarity indeed. The difference is to be found however in the role that initiates data exchanges in bindings. In EDA architectures, it is the event source (producer) that takes the initiative to exchange, or, as it is frequently said, to push event data. The even sink will be waiting and listening for any such events to arrive. In client-server architectures though, this initiative is taken by the client (consumer) by explicitly requesting data and waiting for a reply.

Applications, and the (Quality of) Service they Provide A commonly used design paradigm in the development of enterprise applications is service-orientation, which is "a [popular, multi-disciplinary] design paradigm indended for the creation of solution logic units [— software entities, that is —] that are individually shaped to be collectively [...] utilised in support of the realisation of the strategic goals [and objectives set out by the customer]" [25, Sect. 3.1]. Careful scrutinisation of this definition shows that structuring enterprise applications as service-oriented architectures (SoAs) implies the application of principles such as modularisation and compositionality, in which the collective behaviour of the underlying software entities contributes to the overall mission the application as a whole is expected to support. According to [25, Sect. 3.1], each of these "units of service-oriented solution logic" constitutes a service. In other words, the service associated with a specific software entity corresponds to the functionality of the set of related software routines and their implementation encapsulated within, and its ability to successfully perform some production act and deliver a desirable result [26]. Furthermore, the SoA Reference Model as proposed by OASIS® emphasises that "a service is provided by a [software entity] — the service provider — for use by others [...]". It then continues to define a service as a "mechanism to enable access to one or more capabilities, where the access is provided using a prescribed interface and is exercised consistent with constraints and policies as specified by the service description" [21]. The external party mentioned in the definition above is usually referred to as the service consumer, or simply the end-user, which can be "[any] other system, human or physical, [or software entity] interacting with the considered [service provider]" [12, 27]. From the end-user's point of view, a software entity appears to be a black box; the end-user has little or no knowledge on its internals. As the sole point of interaction takes place via its interface, the **service** it delivers corresponds to the (functional) behaviour resulting from the invocation of any of the defined operations within, as it is perceived by the end-user.

The term **quality of service (QoS)** is typically used in networked telephony, (real-time) streaming multimedia services, and distributed/cloud computing solutions, as an indication of the overall performance of the service associated to the whole software system, or to an individual software component, as it is perceived by

the end-user[4] [28, Sect. 5.4.1]. It is from this user perspective that it is also frequently referred to as **quality of experience (QoE)**. A QoS estimate allows to quantitatively measure the extent to which the intended service is actually being delivered, and to assess whether specific service level agreements (SLAs) are met. Such assessment is the result of comparing a combination of quantifiable properties that adequately capture the characteristics that the service is expected and observed to exhibit. "This allows the quality of service to be benchmarked and, if stipulated by the agreement, rewarded or penalized accordingly" [29,30]. From a functional perspective, suitable properties allow to quantify and assess if the entity is performing and operating as intended and expected, and if it manages to deliver desirable results. One suitable attribute proposed in the QoS taxonomy of [31] is fidelity, or a measure to assess "how well [the] service is [actually] being rendered". Unfortunately, it "is often difficult to define and measure [this attribute] because it is subject to [varying] judgements and perceptions". Other examples include dependability measures such as availability and reliability. From a non-functional perspective, the quality aspects of the service under consideration primarily reduce to performance measures that capture the service throughput and latency. These can be complemented by other measures and rule abidance with respect to characteristics such as operational cost, interoperability, security, integrity (transactionality), accessibility *etc.* [31,32].

Software Dependability: Characteristics and Properties With business- and mission-critical enterprise applications increasingly expected to exhibit highly dependable characteristics, it is of the utmost importance that sufficient attention is paid to an effective composition of suitable software entities, which are themselves highly dependable [31, Sect. 5.1.1] [3, Sect. 4.3]. A comprehensive definition to informally describe the property of dependability can be found in [33]: for any given software entity, dependability is "the trustworthiness and continuity of the [associated,] delivered service, such that reliance can justifiably be placed on this service". An essential quality in this definition is trustworthiness, which, according to [27], can be seen as "the degree of user confidence that the [entity] will operate as [expected] and that the system will not fail in normal use" [27]. Untrustworthy applications may occasionally fail to meet their objectives, and, in doing so, result in significant monetary penalties, or even catastrophic failures causing environmental disaster or human injury.

A better insight into dependability may be reached by breaking it down into a number of consituent sub-properties and measures, of which the three most influential have been shown in the figure above:

– **Availability** is a dependability measure that allows to assess and quantify the ability of a software entity to deliver its associated service as is intended, *i.e.* conform its specifications, whenever requested. An entity is said to be available during that part of its operational life during which the service is accomplished in full whilst processing a specific invocation [12]. It is formally defined as the probability that a given entity is operational and ready for immediate use at a given point in time.

[4] Throughout this dissertation, we focus on software programs and functionality exposed through well-defined interfaces, so it can be used in a distributed context. We do not specifically target streaming or telephony services.

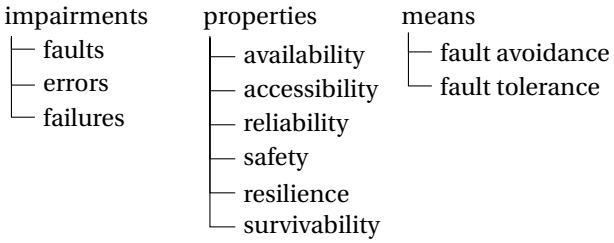


Figure 1.2: Reproduction of the dependability taxonomy introduced in [12].

– Another quality aspect of a service is **accessibility**: it "represents the degree [the service] is capable of serving a [...] request. It may be expressed as a probability measure denoting the success rate or chance of a successful service instantiation at a point in time. There could be situations when a [...] service is available but not accessible. High accessibility [...] can be achieved by building highly scalable systems. Scalability refers to the ability to consistently serve the requests despite variations in the volume of requests" [34, pp. 17–18]. The property is commonly overlooked and thought to be part of availability, since the service itself will be perceived as unavailable when it cannot be accessed. Accessibility is usually determined by decisions taken on capacity planning, and various settings and configuration details of the deployment environment, including any underlying hardware, network and middleware infrastructure — *cf.* Sect. 8.5.

– A software entity capable of maintaining failure-free operation over a specified time period, is said to be reliable. **Reliability** is a dependability measure of the "continuous service accomplishment [...] from a reference initial instant", which corresponds to a period of sustained availability during which all issued invocations will be serviced correctly [12]. It is a probabilistic measure, and is defined as the "probability, over a given period of time, that [an entity] will correctly deliver services as expected by the user" [27]. Even though this measure has been successfully used for hardware appliances, it remains extremely hard to find exact estimates for complex software components, and an alternative context-aware approximation method is introduced in Chapt. 4. Similar to availability, reliability is concerned with the operations of a software entity, and the level of conformance to its specifications.

– Another dependability concern is **safety**, a property which is of the utmost importance to bear in mind when engineering safety-critical applications. This class of applications is usually found in enterprises such as nuclear power plants, waste treatment facilities and chemical processing plants. In such industries, system failure could potentially lead to grave catastrophic consequences, thereby (in)directly threatening the environment and causing damage to people. Safety is a "system property that reflects the system's ability to operate (normally or abnormally) without [catastrophic failure], ensuring [that it] cannot damage [the environment in which it is operating, and this] regardless of its conformance (or nonconformance) to its specifications" [27].

In addition to the above four principal properties of dependability, other system properties are also occasionally considered under the heading of dependability:

– The concept of **survivability** has been rigorously analysed in [35], in which it is defined as "the ability of a networked computing system to [sustain] essential services in the presence of attacks and failures, and recover full services in a timely manner". As implied, the service associated with a survivable software entity may temporarily drop to a degraded service QoS level, during which part of the underlying system resources may be intentionally disabled, before resuming operations in full. Unlike other dependability facets, survivability calls for additional requirements, which describe the acceptable levels of post-disturbance functionality, and the maximum duration that such degraded service levels are acceptable.

– A related property is that of software **resilience**, which, according to [36], refers to the robustness of a given software entity or system. It can be defined as "the trustworthiness of a software system to adapt itself so as to [effectively] absorb and tolerate the consequences of failures, attacks, and changes within and without the system boundaries" that could potentially affect system availability [36] [37, Chapt. 23, 26]. The adaptive capabilities of resilient software entities call for self-configuring and self-healing capabilities, and so seem to suggest that this type of entities are themselves in fact self-managing autonomic software entities [38]. Furthermore, the resilient behaviour of a software entity seems to correspond to the self-optimising characteristic of a similar autonomic entity: it will pursue its objective to maximally sustain the availability of the associated service, though occasionally, despite its perseverance, this objective may not always be met, in which case it might exhibit a suboptimal performance of its operations. Additional information on the properties of autonomic computing systems can be found further down this chapter in Sect. 1.2.

Dependability Threats and Impairments Throughout the operational life of a software entity, the associated service may be temporarily and/or repeatedly disrupted, potentially resulting in a violation of one or more of the intended dependability attributes. As shown in Fig. 1.2 and 1.3, "the accepted terminology [...] distinguishes three levels" in the materialisation of dependability impairments: faults, errors and failure [12, 39]. Each term (level) marks a specific type of problem affecting the given software entity or system, the difference being that "in case of a fault, the problem is on the physical level (that is, located in the program source code in the context of software); in case of an error, the problem occurred on a computational level (the values of a program state); in case of a failure, the problem occurred on [entity (system)] level" [2, 40]. More specifically:

– A **failure** or **malfunction** is any situation under which the delivered service is behaving in an unexpected way, and deviates from the expected service as it was anticipated and specified. It indicates a period of time in the operational life of the affected entity during which some functionality or action that is due or expected, cannot be fully accomplished conform the entity's specifications. At the worst, a failure will translate in a "a total cessation of [the associated service]", though

it may also result in "a performance of some function in a subnormal quality or quantity, like deterioration or instability of operation" [2].

– An **error** is a condition in which the affected software entity exhibits an undesirable or unexpected internal state, and "which is liable to lead to [a subsequent] failure" [12]. As long as such condition holds, the error has become effective, and unfinished computations are likely to result in incorrect or inaccurate results, resulting in a perturbed service being rendered. So it can be seen that a "failure is [indeed] the manifestation on the service of an error" [12]. Examples of erroneous system state include, for instance, program execution being caught in an infinite loop, deadlock, truncation of numeric values, and arithmetic over- or underflow[5]. As soon as the undesirable effects of an error have dissipated, *i.e.* the error has become latent, the corresponding failure may, given time, dematerialise as well.

– Finally, as shown in the cause-and-effect relationship depicted in Fig. 1.3, the root cause of an error, and, consequentially, a failure, is to be found in a **fault**: a physical defect, a design or implementation flaw that was overseen and remains within a software entity. Such imperfections introduce latent errors, which, when activated, become effective, and result in an unintentional deviation of the entity's function. A fault does not necessarily have to manifest as an effective error though: whether it is activated or not depends on the internal state of a susceptible entity, and the frequency and timing of requesting the associated service. Software faults typically include programming mistakes and wrong design assumptions (endogenous causes). Furthermore, unrealistic assumptions regarding the entity's deployment environment — exogenous conditions that seemingly do not affect the entity's correct behaviour itself — may prevent successful service delivery. This would include inadequate capacity planning, which could lead to, *e.g.*, frequent congestion of the communication network, resulting in performance failures.

With software entities deployed on hardware devices, and ever more often relying on middleware solutions and implementation frameworks, it can easily be seen how the dependability of a given entity is highly dependent on the robustness of the underlying infrastructure [40] [3, Chapt. 4] [42, Fig. 1]. Middleware solutions and implementation frameworks are themselves compositions of software entities, and are designed to support several ancillary specifications, and the services exposed by the constituent entities usually directly correspond to standardised predefined features.
Examples include, *e.g.* messaging solutions supporting publish-and-subscribe models, and runtime libraries in support of WS-* specifications — *v.* Chapt. 8. The definitions for the dependability properties and impairments introduced throughout this section are also applicable to hardware devices and peripherals [3]. Examples of faults that may result in failure can be physical flaws in the design of digital circuitry, or the underlying electronic components and wires. Unlike software components which are a purely intellectual product, hardware components typically contain electronic and mechanical parts, and therefore do wear out over time [39]. Moreover,

[5] These examples are specific to the application layer. One may also consider errors in lower layers, *e.g.* memory or storage malfunctions. As we focus on software reliability and the effectiveness of applying redundancy schemata based on design diversity, within the scope of this dissertation, we will merely focus on software-specific faults — so-called design faults [41].

they can be quite sensitive to unforeseen environmental conditions, including exogenous factors such as temperature, humidity, and radiation.

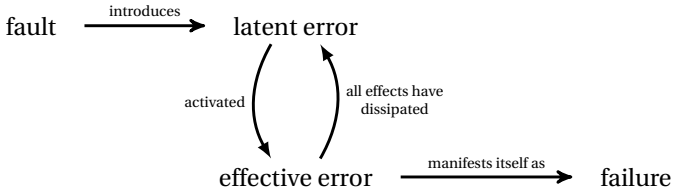


Figure 1.3: Dependability impairments: terminology and cause-and-effect relationship.

"Because a system is defined by a collection of objects [— components —] working towards a singular goal or collection of goals, their efficiency, [dependability] and security are directly tied to one another" [43]. For instance, "[the occurrence of a failure] may, and in general does, propagate from one component to another; by propagating, [it is likely to cause] other (new) errors", affecting the operations of entities relying on the component from which the initial failure originated [12]. Failure propagation poses a serious challenge in the effectiveness of software applications, as these are often composed of multiple software entities, which in turn depend on hardware devices and middleware components. Unless the impact of a failure can be limited to the immediate proximity of the component from which it originated, it may indirectly put at risk the service that the application as a whole is expected to provide [40, 44] [5, Chapt. 8]. This calls for adequate validation, compensation and/or recovery logic to be put in place, which is typically accomplished by structuring the application as a composition of **fault containment units (FCUs)** [2, Sect. 2.2]. An FCU is a dedicated wrapper component designed to accommodate this additional type of logic for a specific component, or a subset of the application's constituent components. In doing so, it is responsible for maximally isolating (containing) any failures that may originate from the use of the underlying components, and prevent the effects of that failure from propagating throughout the system any further. FCUs can be implemented in a variety of styles, some of which will be addressed later on in this section.

Following the accepted terminology for dependability impairments, another term is now introduced for improved readability. Throughout this dissertation, the notion of **disturbance** will be used whenever there is no specific need to distinguish between the different stages in the phenomenological failure emergence model as depicted in Fig. 1.3. In the context of a given software entity, the notion is used to indicate the event of a specific request being struck by some type of failure, resulting in the perturbation and, consequently, the (temporary) unavailability of the service that the software entity that is processing the request was expected to provide. When used, the emphasis lies on the unavailability of the service, and the entity's inability to serve this request, *i.e.* the failure matters, and which precise design fault and erroneous state lead to its occurrence, is considered irrelevant.

After careful analysis of the way failures materialise and how (seriously) they affect their surroundings — that is, the software entity from which they originate — they can be categorised using the failure taxonomy described in [45, Sect. 11.3], which is composed of the following three dimensions: duration, effect, and scope.

– A first element in characterising the seriousness of a failure is its **duration**. It corresponds to a specific time interval in the entity's operational life, during which the failure (continuously) affects the associated service; after this interval, all the effects of the failure will have dissipated, and the corresponding error will return to a latent state. When categorising failures in terms of their duration, a distinction is commonly made between transient, intermittent and permanent failures. Unlike transient failures, which are "characterised by a relatively short duration" and may clear up without "major recovery actions", permanent failures are unrecoverable, and will last until the system is shut down for repair, *i.e.* it is instructed to cease its operations. Intermittent failures are transient *in se*, but they become active periodically, usually when some specific environmental condition, a specific workload pattern, or even another fault (*e.g.* memory malfunction) results in a specific system state [2, Sect. 3.4.1].

– A second dimension considers a failure's **effect** on the behaviour of the service being rendered, from a functional and/or non-functional perspective. When affected by a functional failure, an entity will not operate in accordance with its functional specifications, and thus fail to deliver desirable results. In case of a non-functional failure, the entity "may be executing the requested functions correctly", though at a subnormal performance and fail to meet some of the instated SLAs[6]. Further analysis and insight of the commonest software failures and their effects can be found in Sect. 2.6 and the software failure classes depicted in Fig. 2.3.

– Probably the most important element in characterising the seriousness of a failure is its **scope**, or the extent to which it affects (part of) the application. Ideally, a partial failure should be confined to cause anomalous behaviour only in its immediate locality, hence perturbing the service directly associated to the affected entity, or the service wrapped inside the enclosing FCU. In case failures are successfully contained, "some of the services provided by the [consituent application entities may] become unavailable, while others can still be used". At the other extreme of the spectrum are total failures, which are characterised by a complete disruption of all the functionality and services offered by the application. These could result from the occurrence of software faults such as, *e.g.*, crash failures, and from environmental factors such as power outages. Finally, the dimension of failure scope can also be used to indicate whether the service associated to a software entity is disrupted in full, or in part, in which case only some, and not all of the operations defined in the entity's interface are affected.

1.2 Software Resilience and Autonomic Computing

It was already pointed out that resilient software entities, being adaptive and reconfigurable *in se*, exhibit some of the properties inherently linked to autonomic

[6] Non-functional failures are oftentimes referred to as parametric failures in the hardware community. It refers to any condition in which a device or circuit fails to meet datasheet specifications. Examples could be a higher consumption of power than that what was specified, lower performance, longer processing times *etc.*

computing. Considering the compositional and distributed nature of contemporary enterprise applications, the autonomic computing paradigm acknowledges the need for additional computational intelligence so as to "provide users with a [service] running at peak performance 24/7", and to overcome any disturbance that could potentially result from the operational, compositional and environmental complexity of such type of applications [46]. The construction of an autonomic software application calls for the introduction of a set of dedicated **autonomic managers**, in addition to the underlying resources from which an application is traditionally composed — be it other software entities, hardware or middleware components. An autonomic manager is a purpose-built software entity that is responsible for managing a specific selection of the application's resources — *v.* Fig. 1.5. In this capacity, it implements and encapsulates some type of adaptation logic responsible for adequately responding "to unpredictable [environmental] changes [and localised problems], while hiding the intrinsic complexity [of doing so] to operators and users".

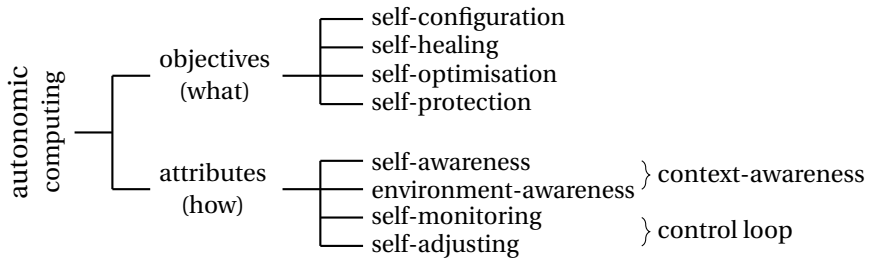


Figure 1.4: Reproduction of the autonomic computing properties tree, as it was originally published in [47].

A distinctive characteristic of such type of logic is that it should be **self-adjusting**, *i.e.* it should be able to change the operations, the configuration and/or the functions of the underlying managed resources, which should allow "to cope with temporal and spatial changes in [the] operational context, either long term (environment customisation/optimisation) or short term (exceptional conditions such as malicious attacks, [failures], *etc.*)" [48]. Furthermore, in order for adaptation logic to qualify as autonomic, it needs to exhibit some additional properties; it should be **context-aware**, and it should be able to change the system automatically. The trait of context-awareness can be used to direct the adaptation procedure and can, in itself, be further decomposed into the properties of **self-awareness** and **environment-awareness** [47]. "In order to be able to assess if its current operation serves its purpose", context-aware adaptation logic "must be able to monitor its internal state [(self-awareness)], as well as its [current external operating conditions (environment-awareness)]" [47, 48]. In the etymological sense of the word, satisfying the trait of automaticity means that the adaptation logic is able to "self-control its internal functions and operations", and must be able to "operate without any manual intervention or external help" by system administrators and operators [48].

The self-managing behaviour of an autonomic manager is typically decomposed into four self-managing capabilities: self-configuring, self-healing, self-optimising and self-protecting. "Collectively, [they] enable more efficient [...] operations by automatically and proactively monitoring [the behaviour of the underlying

managed resources, and their environmental operating conditions, so as] to improve performance, prevent outages, and recover from failures" [49].

– The capability of **self-configuration** corresponds to its ability to, within its own scope, adjust its configuration "automatically, [and] in accordance with high-level policies — representing business-level objectives, for example — that specify what is desired, not how it is accomplished" [46]. Adjustments can only be made to the underlying managed resources, or to the self-managing intelligence contained within the autonomic manager.

– "The **self-healing** objective must be to minimize all outages in order to keep enterprise applications up and available at all times" [50]. Such capability is responsible for "preventing and recovering from failure by automatically discovering, diagnosing, circumventing, and recovering from [*localised* problems] that might [otherwise] cause service disruptions" [46, 51]. Such recovery procedure should be able to detect and isolate a faulty component, and, if possible, restore it into a correct operational state, prior to reintroducing the fixed or a replacement component into service [50]. In fact, these objectives correspond to the principles of fault masking and containment, and highlight that a self-healing autonomic manager is *de facto* an FCU *per se*.

– An autonomic manager should be **self-protective** and should be able to anticipate, detect, identify and "defend the [managed unit] as a whole against large-scale, correlated [*external* threats] arising from malicious attacks or cascading failures that remain uncorrected by self-healing measures" [38,46]. This objective, bearing a strong resemblance to the survivability dependability attribute, can be achieved either reactively by applying adequate recovery actions, or by proactively intervening so as to prevent such threats from materialising, or to mitigate their effects.

– The objective of a **self-optimising** autonomic software entity is to "continually [tune itself and] seek ways to improve [its] operation, identifying and seising opportunities [— proactively or reactively —] to make [itself] more efficient in performance or cost" [46, 51]. In trying to maximise the use of the underlying resources, an attempt could be made, *e.g.*, to attain a better balance and distribution of the workloads.

The adaptive behaviour necessary for achieving the objectives set out by the self-managing capabilities listed here above, is usually modelled as a control loop before it is implemented. Control loops, or feedback loops as they are often referred to, are a well known concept originating from system theory and have been successfully used to realise the self-monitoring and self-adjusting traits of adaptive software [4, Chapt. 4] — *cf.* Fig. 1.4. These traits, in addition to context-awareness, correspond to the three quintessential attributes of autonomic behaviour, in which "changing circumstances [in the context, be it internal or environmental, should be] detected through **self-monitoring**, and adaptations [should be] made accordingly (self-adjusting)"[7] [47]. The architectural model for the design of autonomic manager software entities, featured in Fig. 1.5, shows how a self-contained software entity

[7] With all the self-monitoring properties ascribed to autonomic computing systems, and their ability to derive knowledge from available information, one may be inclined to believe that these systems,

results from the composition of a comprehensive set of dedicated software entities for implementing the autonomic, self-managing behaviour, while structuring this behaviour as a **MAPE-K control loop**. As can be seen from Fig. 1.5, such "control loop is composed of four parts that share [contextual] **knowledge**, including parts that **monitor, analyse, plan and execute**" [49]. The monitoring part is responsible for capturing contextual information within its own activity scope. Intent upon understanding the current system state, this information is then periodically analysed: data and events are diagnosed and correlated to identify relevant situations and symptoms of conditions that may require adaptation. Desirable adjustments can be applied — executed — to the managed resources if an appropriate response could be found — planned — so as to improve or sustain their operational performance in view of the environment in which they are operating. At the core of this MAPE-K subsystem is a knowledge source; not only does it hold information about the managed resources and their environment, it frequently holds business and IT policies as well, whose goals and objectives will be honoured whenever a plan for adjustment is effectuated [38, 51].

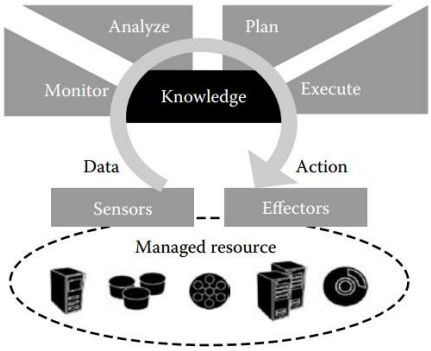


Figure 1.5: Taken from [51, Chapt. 1], this picture illustrates the internals of an autonomic manager, and how its adaptive capabilities are modeled as a MAPE-K control loop. ©2007 by Taylor & Francis Group, LLC

Furthermore, the architectural model in Fig. 1.5 seems to imply that managed resources are themselves wrapped inside manageable software entities, and thus accessible through an interface. However, in order to accommodate for the self-monitoring and self-adjusting traits of a MAPE-K control loop, each of these interfaces will need to be extended so as to provide for adequate sensing and effectuation capabilities. An abstraction of these capabilities results in the concepts of sensor

especially when combined with techniques for artificial intelligence, are self-conscious. Even though there is no globally accepted definition that lists the traits associated with consciousness, it is generally accepted that (i) a system should have access to information, and (ii) that system to be self-aware, in the sense that there exists a "self-referential relationship in which the cognitive system is able to monitor its own [functioning] and obtain information about itself" [52]. The cognitive properties of an autonomic computing systems — although capable of acting upon their internal state, based on contextual information — are not (directly) able to understand the state they find themselves in. They lack perceptional abilities, although they are aware by applying basic rules and predefined algorithms on a limited set of data representing specific characteristics of the environment in which they are set to operate — *cf.* Sect. 9.1.

and effector; both correspond to part of the manageability interface of a specific managed resource. A sensor "exposes information about the [resource's] state and state transitions", and is used for monitoring purposes. An effector aids to bring about intentional state changes and, as such, "enact a desired alteration in the [...] resource", *i.e.* to execute an adaptation plan [38]. Precisely how to implement these additional manageability capabilies is left aside, but clearly, the autonomic computing initiative has laid the conceptual foundations for the design of the WSRF and WSDM families of WS-* specifications [53] — *v.* Chapt. 8. That being said, sensing and effectuation capabilities have been successfully implemented as WSDM-enabled resources, in which relevant sensorial information may be retrieved by periodic polling or asynchronously by exchange over a publish-and-subscribe eventing model such as WS-Notification. Another option, albeit less widespread and robust, is to use reflective and refractive variables, as suggested in [53].

We already pointed out the existence of two alternative interface definition languages on p. 6: WSDL, and, closely linked to that, the XML-based SOAP messaging format, *vs* a REST-based approach that is mainly promoting the use of the JavaScript Object Notation (JSON) messaging format [23, 54]. Nowadays, REST has widely been adopted and is very much preferred over SOAP. This is mainly because JSON is found to be a much more lightweight messaging format than XML that lies at the basis of SOAP. Clearly, from a pure performance point of view, REST has the edge on WSDL, as it will require less bandwidth for the exchange of data messages. However, there is no mature match for the extensive capabilities that XSD schemas bring for XML validation of SOAP messages. Furthermore, there is a whole range of WS-* specifications available for interfaces defined in the WSDL definition language, which promote standardisation and cover several functional aspects, ranging from basic messaging aspects, security, eventing, service discovery, management of resources *etc.* [22]. No counterpart can be found for either of these standards in REST, resulting in proprietary, oftentimes, incompatible service interfaces and ad-hoc implementations — functionality that would otherwise largely be present as part of the middleware offering and that would ensure interoperability. In the context of NVP, when application logic is exposed as web services, the functional components that will represent versions are *in se* a managed resource. And, as will be explained in great detail in Chapt. 8, some of these specifications — in particular WSDM and WSRF — have shown to be very useful in implementing a generic framework that allows to implement A-NVP for any type of functionality.

It was already pointed out in Sect. 1.1 that the property of software resilience shows great similarity to the self-managing properties of autonomic software. A better insight into software resilience may be obtained by decomposing it into a base of ancillary consituent attributes: perception, awareness, planning and dynamicity [36]. Each of these attributes (except the latter) directly corresponds to one or more of the phases in a MAPE-K autonomic control loop. More specifically, **perception** is the objective of the monitor phase in the loop; it is defined as the ability to perceive change, either in the operations of the underlying managed resources, or environmental. During the next stage in the loop — the analysis — the available contextual information is diagnosed so as to recognise conditions and situations that may threaten a managed resource from operating correctly (**situation-awareness**). The essential property of resilient behaviour is, without doubt, the "ability to plan reactive or proactive strategies to compensate for current (or, respectively, future)

threatening conditions, and the ability to enact [— execute —] those strategies by performing the actual parametric and structural adaptations" (**planning**) [36, 55]. The final property of **dynamicity** characterises, in a way, the extent to which the former three properties can become acclimatised to "past experience and continuously improve" the effectiveness of the former three properties (self-learning), and the extent to which the system is able to adapt itself accordingly [36].

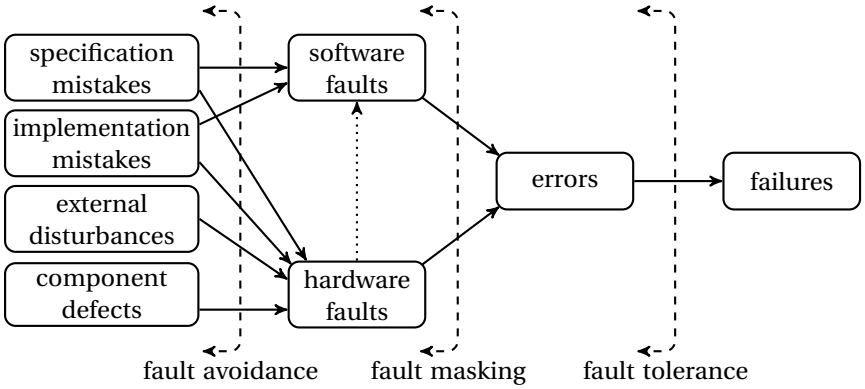


Figure 1.6: Dependability means categorised into 3 design techniques. Reproduction of [3, Fig. 2.12].

1.3 Design Philosopohies to Combate Faults

The "techniques for attempting to improve or maintain a system's normal performance in an environment where faults are of concern" are typically categorised into three categories: **fault avoidance**, **fault tolerance** and **fault masking** [3]. However, masking faults that have been activated (in other words: errors) is actually what is used by most techniques for software fault tolerance that rely on design diversity. The three design techniques are shown in Fig. 1.6, which illustrates how each technique may be applied so as to form an effective barrier against a specific dependability impairment. The key discriminating factor is that "fault avoidance techniques try to reduce the probability of fault occurrence, while fault tolerance techniques try to keep the system operational despite the presence of faults" [41]. This section will provide a bird's-eye view of the primary **means** to deal with faults in constructing dependable software solutions.

In this dissertation, the focus is on software fault tolerance, and the application and optimisation of redundancy schemata. As such, we will only cover functional software (design) faults[8]. Even though hardware faults can have an impact on the availability of software, and may (in)directly trigger latent software faults[9], this relationship is kept out of scope[10] — *v.* dashed arrow in Fig. 1.6.

[8] "Researchers agree that all software faults are design faults" [41]. Hence, both terms are will be used interchangeably.

[9] Moreover, recent discoveries seem to indicate that specific workload patterns can accelerate the malfunctioning of hardware circuitry (both in terms of parametric and functional failures) [56].

[10] The reason for this is threefold: (i) such investigation would represent a separate research topic in

1.3.1 Fault Avoidance

The concept of fault avoidance — also known as **fault prevention** — covers any technique that is used to *prevent*, by construction, or minimise, by specification and verification, the presence or introduction of latent errors in a (software) computing system during its construction or implementation [12]. Unlike fault tolerance, it is a proactive strategy mean to identify all potential areas where faults can occur. It includes various precautionary quality assurance and design methodologies; approaches that are quite different from a philosophical point of view:

– "In the design and implementation phase, fault avoidance can be achieved by [...] formal verification and validation techniques" [57]. Such formal methods aim to "eliminate errors at the requirements specification and design stages of the development" [41].

  o It is useful to *validate* every design before it is used for further development. If conducted appropriately, "many of the specification mistakes that might otherwise result in faults can be eliminated" [3]. Furthermore, ample attention should be spent on ensuring a proper system-environment fit [55]. This also affects systems that integrate with various other external systems. If needed, the design should be altered to introduce additional "[shielding] to prevent external disturbances from causing faults in the system" [3]. Mostly, such external disturbances affect the underlying hardware and connectivity infrastructure. Typical examples include lightning, electromagnetic interference, cooling, *etc.*

  o The correctness of a design and its implementation can be supported through the use of formal specification and *verification* techniques [58]. Most of these techniques apply a type of mathematical notation to explicitly describe the desired system properties and behaviour. This allows a clear, unambiguous specification of the requirements for software, which highlight misinterpretations and incorrect system assumptions in the detailed technical design. An example of one such specification language is the **Z** notation [59]. In particular, the accuracy and expressive power of such languages makes them useful for the description of core components in mission- and safety-critical systems. Admittedly, such formalisms cannot easily be applied to highly complex, large, distributed systems. Hence, they are usually only applied to specific parts of the system that pose most risk to the overall system operations and performance.

  o Finally, the concept of fault *forecasting* is a rather different approach. It "includes a set of methods and techniques [...] to estimate the presence, the creation, and the consequences of faults", usually by monitoring and analysing the operational behaviour of the system (once it was placed in production). It may leverage machine learning (ML) technologies to predict when errors can materialise, and to report on this so that any residual (latent) fault can be addressed and fixed in subsequent software releases[11].

itself, and (ii) we would likely have to make assumptions on specific hardware that is used, whereas we deliberately wanted to abstract the intricacies of the underlying middleware and hardware layers, and zoom in specifically on software redundancy schemata. Furthermore (iii), such investigation would be more valuable in the context of low-level dedicated/embedded systems, whereas the focus here is on real application functionality deployed on general commercial-off-the-shelf (COTS) hardware.

[11] Note that forecasting has primarily been used for hardware, where it is used to support predictive maintenance.

– "In the test and debugging phase, fault removal can be performed" by subjecting the implementation to rigorous **testing procedures**, aiming to *falsify* the assumption that a carefully crafted solution is sound and correct [57]. In doing so, "many of the faults that [remain] in the system after manufacture can be detected and eliminated before [... it ...] is placed in operation" [3].

1.3.2 Fault Tolerance

Even though fault avoidance techniques may be effective in identifying faults and removing them from the system *before* it is put in production, there will always remain others that eluded detection despite rigorous and extensive testing and debugging. In short: one should simply face the fact that "it is practically impossible to build a perfect [(software)] system" [2]. Hence the need for fault tolerance, which can be described as the introduction of **compensatory measures** to ensure that the system will continue to perform its task and to deliver its intended service, even *after* the occurrence of faults — in other words, when a limited subset of the underlying system components are struck (disturbed) by failure. From this, it follows that methods for fault tolerance are by their very nature "applied [... in the operational phase ...] to provide proper system service in spite of faults" [57]. Apart from safety-critical applications, many fault-tolerant solutions typically also support a **graceful degradation** of the system's performance. This is applicable in case "the size of the faulty set increases, [where the system should continue to operate] and continue executing part of its workload [in a best-effort mode, rather than] suddenly [collapsing]" [60].

By adding additional circuitry and (software) components on top of a specific component, fault-tolerant design techniques aid in the development of FCUs. The additional logic should prevent faults in the core component from introducing errors into the informational structure of the system that could affect other system components, or the system as a whole [61]. Despite the naming, fault tolerance is about the recovery and masking of failures (disturbances), as can be seen in Fig. 1.3. The additional protection layer that an FCU adds on top of a specific software component will therefore include techniques for **effective error processing** that are meant to rectify any erroneous state that violates the specification of the service that the component is expected to deliver [12]. Briefly put, it should attempt to correct the data stored in memory data before another part of the system will use that data. Error recovery techniques are usually categorised into two classes:

– In **forward error recovery**, the "error is masked without any computations having to be re-done" [60]. Such techniques will apply some type of **compensation** routine to achieve **fault masking** and contain the effect of an error or disturbance within the fault-tolerant FCU. Some commonly used techniques to achieve this type of error recovery are:

  - Running in a single thread of execution, a simplex system is by nature a potential **single point of failure (SPoF)**. Nonetheless, one can introduce and apply consistency checks to verify the correctness of the computing results returned by such component, using characteristics that are known a priori from the initial specifications. Furthermore, application-specific recovery actions can be added to the source code itself. This approach is

also referred to as **single-version fault tolerance** [5, Chapt. 8]. It is however difficult to achieve a proper separation of the error detection and correction features from the core programming logic itself. Another drawback is that this approach cannot cope "with system faults that arise from interactions between the hardware and the software" [14, p. 483], as there is no replication of hardware either[12].

- Another example is to revert to **multiple-version (software) fault tolerance**, which will be covered in more detail later in this chapter. In applying such approach, it is assumed that a sufficient degree of redundancy can "enable the delivery of an error-free service from the erroneous (internal) state" [62]. Furthermore, they rely on a comparison algorithm — so-called **voting** algorithms — that will compare and analyse potential deviations in the results computed by several distinct versions, and adjudicate the correct outcome to be returned. As an example, the architectural pattern of *n*-**version programming** will be explained in more detail further down this chapter.
- Hybrid approaches supporting failover schemes have shown their use and remain popular, in particular to improve the dependability of the deployment infrastructure. In standby sparing, for instance, the architecture of the system includes additional component(s) that are kept on standby, and which may be put in operation as a substitute for a faulty component.

– **Backward error recovery** techniques rely on a runtime environment that is responsible for "periodically [taking] checkpoints to save a correct computational state. When [an] error is detected, [the system is rolled] back to a previous checkpoint, [effectively restoring a] correct state", after which execution can resume [60]. This procedure actually corresponds with the three phases of **recovery-oriented computing** solutions: rewinding the system to an error-free state, repairing the system, and they replaying, by resuming or redoing the necessary parts of the computation [63]. System repair is achieved through **reconfiguration**, *i.e.* "the process of eliminating a faulty entity from a system and restoring the system to some operational condition or state" [3]. The result are FCUs that include adequate facilities for fault detection, fault location and fault recovery [44, 64]. Furthermore, such units are *in se* self-aware, self-configuring and self-healing, for they will autonomously monitor and adjust themselves if needed. One example of this class of error recovery techniques is the principle of **recovery blocks (RB)**, a redundancy-based pattern in which different versions are tried in sequence, one at a time, until one has produced an acceptable result.

1.3.2.1 Multiple-Version (Software) Fault Tolerance

Fault tolerance is the architectural trait of systems that can or aim to maximally sustain service delivery, compliant with the initial specifications for which they were designed and implemented, in spite of faults having occurred or occurring [12].

[12] It is, however, possible to prevent hardware errors from propagating upwards to higher abstraction layers (including the platform software) in digital systems. A classification of various "techniques for [increasing] resilience and [mitigating] functional [hardware] errors" that can increase system resilience was presented in [42].

Its achievement is, in general, "the result of some strategy exploiting some form of redundancy — time, information and/or hardware/software redundancy" [11]. The idea is that the introduced redundancy can stand up to temporary periods of malfunctioning in part of the underlying components, and, as such, sustain the system's dependability as a whole. Additional system components obviously imply a cost penalty, which cannot always be justified, as the effectiveness of redundancy schemata can vary significantly. For instance, it is possible to design highly available, yet poorly reliable systems. This would be the case if disturbances dematerialise or can be repaired quickly, having left the internal system state undamaged (possibly after recovery), or if the system is assembled using poorly performing components. Furthermore, the effectiveness of a fault-tolerant solution may evolve over time. This applies particularly to hardware systems, where components have typically suffered from age, wear and tear, and the cumulative effects of environmental maladies like dust, vibration and temperature extremes will translate into a rising failure rate [65]. Recent research has also shown that there is a tight coupling between some types of hardware aging, the architectural topology and parameters of that hardware, and the runtime workload placed on this hardware equipment [56]. Unlike hardware, software does not suffer from wear and tear; once uploaded into memory and placed in operation, it will continue to serve its purpose [2]. Although early releases of software solutions typically suffer from a diverging internal state ensuing from the accumulated effect of request processing, rigorously engineered stable production releases hardly do.

It is often said that single-version redundancy will not increase availability or reliability at all. One simply cannot expect to achieve effective fault-tolerant designs by simply replicating the same hardware/software components. Doing so can be a "possible source of **common-mode failures** [— *i.e.*, a fault that occurs] simultaneously in two or more redundant components", resulting in multiple correlated disturbances [2]. Indeed, redundant copies of the same hardware/software component are very likely to fail when subject to identical environmental conditions (specific unsupported ranges of input values, operating conditions that were not foreseen by the device manufacturer, request pattern, malfunctioning underlying or connected infrastructure *etc.*). This is particularly applicable to software, where software faults are the result of incorrect design or implementation mistakes. Hence, a "comparison procedure will not [be able to detect software failure] if the duplicated software modules are identical, because [these flaws] will appear in [all copies]" [61]. While this opinion has been repeatedly expressed in the literature, and is incontestably true for static redundancy schemata, where design faults would translate in common-mode failures, this is not necessarily the case for dynamic redundancy schemata. Such fault would only translate in failure when the internal system state would deviate from the operational conditions as expected during design — *i.e.*, when affected by an error. Since the internal state is the result of a complex interplay between state changes due to the processing of requests, as well as various types of inter-component events over time, all **replicas** of the single version would fail in a static redundancy scheme, for they are subject to the same environmental conditions. However, for dynamic redundancy schemata, where a replica can at runtime be taken out of service or put into service (again) depending on the operational context and needs, the internal states may differ, and not all may be

drifting towards an erroneous state[13].

This calls for **design diversity**; starting from the same (non-)functional specifications, several alternative implementations — called **versions** — are designed and developed independently by different project teams. At the end, several functionally-equivalent software solutions are available, each of which can be used to serve the intended service, while significantly reducing the risk of common-mode failures. In doing so, "it is hoped that [...] the same mistakes will not be made by the different [designer and implementation teams]", and that it is less likely to share a common fault across different implementations. "Therefore, when a fault occurs, the fault either does not occur in all [versions], or it occurs differently in each [version], so that the results generated by the [version] will differ" [66]. With functionally-equivalent, yet diversely designed hardware devices or software solutions in place, a fault-tolerant design typically involves a specific redundancy scheme that is composed of n versions, n being the applied level of redundancy. Throughout the next two sections, we will elaborate on the two main types of redundancy schemata: recovery blocks (RB) and n-version programming (NVP).

Recovery blocks (RB) As the name itself implies, the technique of recovery blocks is used for backward error recovery that will repeatedly recover the system state in case of failure. The term is used to refer to an architectural redundancy-based pattern in which different versions will be executed in sequence, one by one, until an **acceptance test** confirms a sufficient level of trust in a result computed by one specific version. Unlike NVP-based redundancy schemata, not all available redundancy is necessarily used: the primary implementation v_1 is always executed on invocation; other versions will be used in case the acceptance test evaluates to a value that indicates a doubtful response [67, Chapt. 1]. Central to the concept is that the (entire) system state is saved in a recovery cache before any processing is requested (including the primary implementation). This saved state will be restored prior to the execution of the next alternative module whenever the acceptance test fails — *i.e.* the system is rewound or rolled back into an error-free state.

The resulting fault containment unit heavily relies on an execution engine, which comes with support to back up the system state (so-called checkpointing), and to repeat the computation using a different implementation if the application of the acceptance test on the current alternative does not generate a checksum inside a predefined interval, in which case the current version should be considered faulty. If the test passes, the computed result is returned and the mission is accomplished; if the test fails, the correct system state is reloaded from the recovery cache. Only when an error is detected will alternative implementations be used. Alternatives $v_2 \ldots v_n$ are sequentially tried and the acceptance test is applied against the results until either a satisfactory result is obtained, or no further versions remain, in which case a failure is propagated by means of an exception. At the core of this redundancy scheme is the acceptance test: a runtime assertion that will be applied by the execution engine for validating the results computed by the underlying versions. **Consistency checks** and **self-checking functions** are commonly used for this type of test.

[13] While the author wanted to point out the impact of request processing and load patterns on faulty behaviour, by no means he wants to imply that simple replication of software components can achieve effective fault-tolerant solutions.

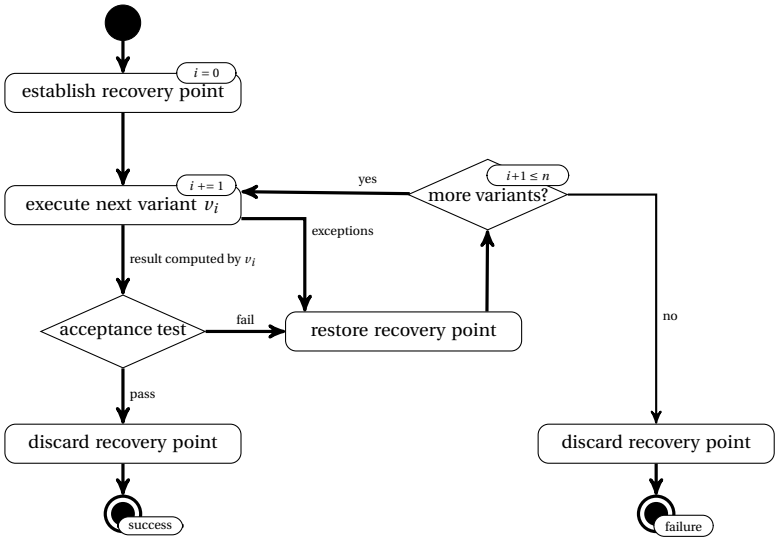


Figure 1.7: Backward error recovery using recovery blocks.

The effectiveness of recovery blocks rests to a great extent on the acceptance test. A failure of the acceptance test is a failure of the whole recovery blocks strategy. For this reason, the acceptance test must be simple, must not introduce huge run-time overheads, and it must not retain data locally [68].

Because the alternatives are executed in sequence rather than in parallel, hardware replication is, strictly speaking, not required. It also means that, given a redundancy level n, this type of redundancy scheme is capable of recovering from at most $n-1$ faulty replicas, albeit at the expensive of a considerably worse execution time. One specific version is not necessarily identical to the others that are used within a recovery blocks redundancy scheme. In determining the order in which the underlying versions are tried, the designer can actually express multiple degraded service levels. For sure, the primary version v_1 should attempt to satisfy the functional requirements in full; the first alternative v_2 would result in a less complete service. The final alternative v_n would deliver the least complete service [69] [70]. Architectures supporting failover by adding a (single) backup circuit next to the primary circuit can be classified within this architectural pattern as well.

n-version programming (NVP) The roots of NVP redundancy schemata are to be found in the domain of hardware fault tolerance [11]. They are similar to deployments using n-modular redundancy (NMR), a technique that has long been effectively used to deal with random hardware failures [14]. In such schemata, an FCU is constructed by replicating a hardware unit n times by components supplied by different manufacturers, though conforming to the same specification. The critique of wasting redundancy unneededly in spite, this architectural redundancy-based pattern has been successfully used for systems with extreme dependability requirements, examples of which can be found in aironautics and nuclear facilities. All of these components are called in parallel whenever service is requested from the FCU. The n outputs that each of the versions produce are then fed into a **comparison**

routine or **voting algorithm**, used to compare discrepancies between the received outputs, to detect potential failures, and to adjudicate a single acceptable result to be returned. If an implementation produces deviating output compared to the other variants, or it fails to produce any output at all, its output should be ignored, and this disturbance should be contained and **masked**. NVP essentially is the same architectural pattern as NMR, the only difference being that software components are used as underlying resources instead of hardware circuitry. The additional logic needed to support this type of redundancy scheme is also implemented in software rather than hardware.

A common voting mechanism is majority voting, although other algorithms exist and can be applied as well – *cf.* App. B. The selection of a suitable decision algorithm requires forethought and a realistic view on the behaviour of the environment in which the system will be operating, particularly with respect to the rate and pattern with which disturbances are likely to occur. Furthermore, the redundancy level itself should be chosen such that the decision algorithm can effectively mask a sufficient amount of failures and achieve the intended level of dependability [69]. Most of the literature has reported on triple-modular redundancy (TMR), suggesting that an odd degree of redundancy be used so as to avoid situations where a result cannot be decided. The inner working of NVP schemata is formally defined as part of the discrete event simulation model that can be found in Sect. 2.1 and 2.2.

An NVP/NMR FCU in itself eases the detection, isolation and containment of any disturbances that affect any of the n underlying resources (components). Appropriate actions can be taken when anomalies are suspected during the comparison routines: the unit can include a fault manager that may try to repair the faulty component automatically, or if this appears to be impossible, it could automatically reconfigure the component selection and purge faulting components, after which the system will proceed execution with the remaining units. Such self-configuring redundancy schemata are the scope of this thesis, in which we propose a generic framework for maximising the dependability of NVP schemata by including algorithms capable of autonomously reconfiguring and optimising the redundancy employed — *v.* Chapt. 5.

Hybrid Approaches Although NVP and RB are the main fault-tolerant techniques for design diversity, there exist hybrid variations of these redundancy schemata.

In acceptance voting (AV), the concept of acceptance test is combined with NVP, to prevent erroneous version responses (ballots) from being used as input to the voting algorithm [71, pp. 162–172]. As the name implies, the technique of two-pass adjudication (TPA) includes two voting round *passes*: the first pass is fed with the original inputs; if that fails to adjudicate an outcome, a second pass is initiated, which is fed by re-expressed parameters. By its very nature, it is the combination of traditional NVP with the basic technique of just retrying versions, hoping that disturbances would be of transient nature, and would automatically disappear. TPA was designed mainly to avoid that "small modifications to the [input] data would not adversely affect the application", especially in a context where sensor data — which, depending on the type, can be very noisy and imprecise data — is processed [71, pp. 218–231]. For more details on AV and TPA, refer to Sect. 9.3.1.

Another approach is taken in [72], in which the authors propose an algorithm that is essentially a hybridisation of a basic retry mechanism, NVP and RB. Their

approach is similar to the approach of the A-NVP algorithm that is defined throughout this dissertation, in that the optimal configuration is dynamically adjusted towards optimal performance and fault-tolerant behaviour based on estimated QoS measurements collected at runtime for individual versions. It can also vary the voting strategy, where the common MV, can be changed into plurality voting (PV) and even active voting (AV), if performance is expected to benefit from this[14]. The similarity with our work confirms once more that "[traditional] fault tolerance strategies are too static and cannot auto-adapt to different environments", and that "context-aware dynamic fault tolerance strategies" can help to improve (i) dependability, (ii) performance, and (iii) cost efficiency [72].

1.3.2.2 Combining Different Types of Resilience Techniques

The previous section discussed in great detail how the two primary techniques for fault tolerance based on design diversity can be used to achieve resilient software systems. Apart from these techniques, other basic techniques can be used, in their own right or in combination, to increase the resilience of digital systems. The effectiveness of these technique(s) is highly dependent on the fault and system models of the components used within the system, its architectural properties, and the environment in which is set to function. Furthermore, one can discriminate several layers of abstraction in the design of digital systems, ranging from the hardware platform layer, the underlying devices or electronic circuitry, the platform software stack (including operating system, runtime engines and/or interpreters, and middleware solutions), to the actual application layer. Such layered approach is nicely depicted in [42, Fig. 1, Sect. 2]. In that article, the authors point out that cross-layer resilient system design can be achieved by applying "functional reliability techniques [...] at the [hardware and software platform layers ...] complementary to techniques [at the application, circuit and device layers]."

The classification presented in [42] is particularly interesting in the context of embedded systems, where the underlying hardware is well known, and techniques can be applied on different layers to prevent functional errors from occurring and mitigate the risk they translate in a system-wide failure. Many software applications, however, are nowadays being deployed on so-called commercial-off-the-shelf (COTS) hardware and/or software components. Industry trends including the virtualisation and cloudification of deployment infrastructure, and the wide use of general-purpose middleware solutions like application servers and integration solutions have resulted in a "lack of internal knowledge [that has added] an additional challenge on deriving appropriate reliability-driven approaches" [42]. In such contexts, it is challenging to select the appropriate combination of techniques to enhance the overall system resilience by mitigating the effects of functional reliability errors.

Design-time knowledge can however be exploited to define combinations of varying sophistication levels, in which multiple error mitigation and resilience techniques can be applied at different layers to defend against functional errors occurring and to prevent them from propagating to higher-level layers. For example, hardware resources could be (re)allocated, and versions migrated/moved to improve the reliability match between software tasks and hardware modules. That approach,

[14] More information about PV and AV can be found in App. B, Sect. B.2.

or — alternatively — various failover/retry mechanisms, may further improve system reliability. The combination of various categories of techniques is a most interesting research track to pursue, but remains out of scope for the time being[15].

1.4 On the Role and Advantages of Discrete Event Simulation

Discrete event simulations have proven to be extremely useful in analysing the behaviour and properties of complex systems. They can be seen as computer programs written in such a way that they mimic the behaviour of the system under investigation [73, pp. 380–382]. In modelling the system, it is formalised from two distinct perspectives. Firstly, a set of variables and/or class objects is identified that can represent and reflect the state the system is currently in. Secondly, the system's operational and the environment's behaviour is analysed, resulting in a set of dedicated simulation **entities**. Each of these entities has a specific life cycle, during which they can affect the system, its state and/or its environment. It is the aggregate result of the state changes brought about by these identified entities that approximates the actual system's behaviour.

Even though simulation modelling and analysis is a complex task that can be time consuming and expensive, it has been profitably used in many disciplines. They are essential to scientific research, for instance, in weather and climate modelling. They have been used by utility providers in support of capacity planning, and by financial service providers for optimising their core business processes and risk analysis. Even more applications can be found in various domains: companies active in the domains of aviation, logistics in general, and even the military have long been using simulations throughout the design of complex, highly-dependable systems. They have been used to conduct experiments in order to analyse, evaluate and assess how and to what extent "new policies, operating procedures, decision rules, information flows, [changes in] hardware designs, physical layouts and transportation systems" are to affect the performance and operations of the systems being investigated [74, Chapt. 1]. This effectively allows specific hypotheses, designs and policies to be tested for feasibility and efficiency, "without disrupting the ongoing operations of the real system". Furthermore, they can be used for improving the performance of (legacy) systems, by tracing the key parameters that determine the actual quality of experience, and which value ranges show a direct link with performance hits. The only requirements for doing computer-based simulations are computing power and simulation software, both of which are readily available nowadays. Despite its many advantages, discrete event simulation in itself is no guarantee for success, and validation of the model remains hard.

Discrete event simulations are all about analysing how the overall state of the system evolves in time. Changes in state occur at distinct points in simulation time, and may occur as various entities enter the system, move through it while

[15] It was already indicated in footnote 5 on p. 11 that the scope of this dissertation is limited to the application layer. To put that in perspective: we have studied in great detail (i) the implementation and potential benefits that NVP redundancy schemata can bring, and (ii) how design faults can materialise and how they can be effectively masked in that context. Referring to the different abstraction layers defined for a digital system — or, more generally speaking, for software applications — in [42, Fig. 1, Sect. 2], we merely focus on the upper application layer. It is in that layer that implementations of NVP and RB schemata can be found.

interacting with the environment and/or other entities, and eventually exit. The lifecycle of a simulation entity is made up of numerous **events**, which have the following characteristics:

– each event is *scheduled* to occur at a distinct (discrete) point in simulation time;

– every event comes with a detailed description of the steps (actions) that are to take place and how these result in a change of state of the system as a whole, or one or more (other) entities in the system [75, Sect. 5.3.2];

– an event cannot change the past: the steps associated to events should be computed in the order that the events were scheduled and activated. This property is commonly known as the **causality constraint**, and is necessary to ensure the correctness of the simulation program [73].

As an event will move the system from one state into another instantaneously, and "the state of entities remains constant between events, there is no need to account for this inactive time" when running discrete event simulations [76]. More specifically, immediately after all the actions of the associated event have been applied, and the state of the system has been changed at the time corresponding to that particular event, the simulated time can already be advanced to the time of the next event, and the processing of that next event can start immediately. This enables a simulation to skip over inactive time, whose passage in the real world we are forced to endure [76]. Undoubtedly, the possibility to fast-forward the time between events is one of the main advantages of using discrete event simulation, and the principle is widely known as **next-time advance**. "Virtually all simulation [implementation frameworks] use the next-event approach to time advance". This is also what differentiates discrete event simulation from **emulation** – *i.e.*, injecting specific load in a real-world system (in production) with the aim of testing and analysing system properties, behaviour and/or performance.

Discrete event simulation also surpasses emulation in that the researcher can exert much more control on the influence commonly caused by the environment or specific entities:

– it allows to focus particularly on specific phenomena under investigation, and to suppress environmental behaviour that could otherwise inhibit these phenomena from materialising;

– distributions are commonly used to generate interevent time intervals of varying lengths in a deterministic manner, and to introduce a desired factor of randomness;

– hence, "time can be compressed or expanded allowing for a speedup or slowdown of the phenomena under investigation", and rarely observed phenomena can be reproduced more easily [74].

Experience has shown that it is difficult to keep track of the various intricacies of the system under investigation, when it is modelled using the basic event-scheduling approach using a single set of events. As the complexity of the analysed system increases, the designer is more likely to benefit from structuring his/her simulation model and event set using the following techniques:

– Whereas the primary emphasis of a regular discrete event simulation model is on identifying precisely how and when state changes, constructing an **activity-based model** will put focus on how various entities in the system interact. In this context, basic events are categorised and associated to a specific part (**task**) in the life cycle of an entity, and they only effectuate the impact of the entity's operations on its environment. The approach "emphasises a review of all [events] in a simulation to determine which can be begun or terminated at each advance of the clock" [76].

– Adding one further level of abstraction, a **process-interaction approach** helps in structuring the life cycle of an entity as a sequence of events and/or activities. It allows to keep track of the "progress of an entity [– which stage of its life cycle it is currently in –] from its arrival event to its departure event" [76].

– What both approaches have in common, is that the life cycle of a select number of entities is decoupled from the rest of the system, and that specific events are associated with each activity and/or process to effectuate the required state changes.

Fig. 1.8 shows how events, activities and processes can be used in discrete-event modelling and how they relate to one another. For each process or task (activity), processing will start at the arrival of an event; completion of the service that a task has received is marked by another event. Note the non-uniform inter-event time intervals:

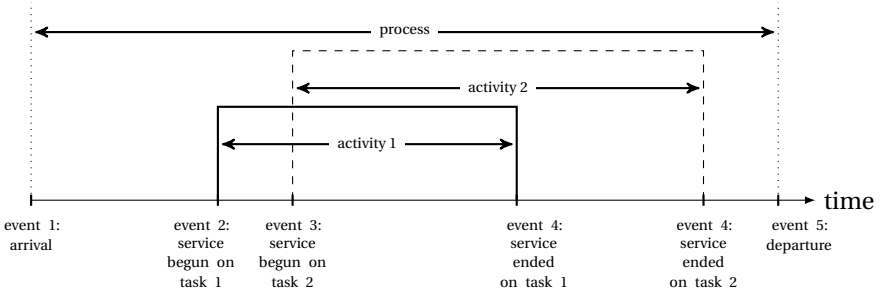


Figure 1.8: Taken from [76, p. 41]: discrete-event modelling using events, activities and processes.

Within the scope of this dissertation, discrete event simulations are used to investigate the effectiveness of various policies for dynamic redundancy management. The discrete event simulation model formalised in Chapt. 2 is supposed to unravel the operations of NVP-based redundancy schemata in detail, and identifies a number of entities, their life cycles and the interplay of events that cause the system to change state. Most of these entities are defined as processes, few examples of which are:

– Given an NVP scheme, the NVP composite is in itself an entity and is decoupled from the underlying versions. Its behaviour is described as a process that dispatches several copies of the incoming request and forwards it to the other n entities (versions) for processing. The process also relies on a voting component, again a specific entity.

– Each version can serve multiple requests concurrently, and can be affected by several types of (software) faults.

– Software faults are modeled as individual entities. The associated life cycle should foresee in a periodic transition between intervals during which the fault remains dormant, and intervals of fault activation, propagation and materialisation. More specifically, such life cycle model will include events so as to describe (i) how a latent fault is activated, (ii) how this activation leads up to a disturbance that can affect (some of) the requests being processed by the affected version, and (iii) when the effects of the fault have dissipated (in case the fault is of transient nature).

– Request handling itself is modeled as a process responsible for producing a computed response – referred to as **outcome**, if any. A request will be handled by a specific version, which can fail to produce a correct outcome when affected by disturbances of any kind. Furthermore, the execution of individual requests may depend on various scheduling policies applied at the corresponding version due to a limited processing capacity.

– Both types of request – the initial request arriving at the NVP composite and the replicated requests sent out to multiple versions – are examples of process-structured entities. The same applies for the voting procedure. The simulation model is also structured using activities, which is clearly exemplified by the fact that faults can affect the operation of versions and thus the outcome of requests.

1.5 Motivation and Problem Description

Adopting classic redundancy-based fault-tolerant design patterns, such as NVP, in highly dynamic distributed computing systems does not necessarily result in the anticipated improvement in dependability. This primarily stems from the statically predefined redundancy configurations hardwired within such dependability strategies, *i.e.* a fixed degree of redundancy and, accordingly, an immutable selection of functionally-equivalent software components, which may negatively impact the schemes' overall effectiveness, at least from the following two angles.

Firstly, a static, context-agnostic redundancy configuration may in time lead to a more rapid exhaustion of the available redundancy and, therefore, fail to properly counterbalance any disturbances possibly affecting the operational status (context) of any of the components integrated within the dependability scheme. The effectiveness of an NVP composite is largely determined by the dependability of the versions employed within. As elucidated in [3, Sect. 4.3.3], the use of replicas of poor reliability can result in a system tolerant of faults but with poor reliability[16]. It is therefore crucial for the system to continuously monitor the operational status of the available resources and avoid the use of resources that do not significantly contribute to an increase in dependability, or that may even jeopardise the schemes' overall effectiveness.

[16] Although not within the scope of this study, this phenomenon applies to various layers of the system architecture. For hardware circuitry in particular, apart from the obvious functional failures, the composition of electronic components and circuits may worsen the parametric behaviour of the composite, and therefore its general reliability [3, Sect. 4.3].

Secondly, the amount of redundancy, in conjunction with the voting algorithm, determines how many simultaneously failing versions the NVP composite can tolerate. For instance, an NVP/MV scheme applying majority voting can mask failures[17] affecting the availability of up to a minority of its versions. A predetermined degree of redundancy is, however, cost ineffective in that it inhibits to economise on resource consumption in case the actual number of disturbances could be successfully overcome by a lesser amount of redundancy. Reversely, when the foreseen amount of redundancy is not enough to compensate for the currently experienced disturbances, the inclusion of additional resources (if available) may prevent further service disruption.

1.6 Objectives, Research Questions and Contributions

In this thesis, a novel dependability strategy is introduced encompassing advanced redundancy management, aiming to autonomously tune its internal redundancy configuration in function of the observed disturbances. Designed to sustain high availability and reliability, this adaptive fault-tolerant strategy may dynamically alter the amount of redundancy and the selection of functionally-equivalent resources employed within the redundancy scheme. The proposed solution has been intentionally designed as a parameterised framework, where alternative policies can be applied at will, and a comprehensive simulation framework supports the designer in modelling the system's environmental behaviour, and in conducting extensive performance analyses in order to assess their effectiveness. While studying this dissertation, the reader may expect the following research questions to be addressed:

RQ-1. **Can the dependability of individual (software) components be approximated by aggregating runtime information and statistics?**
Rather than relying on a theoretical estimation of the dependability that is calculated at design time, it may be more accurate to monitor the functional and environmental behaviour at runtime and extract meaningful *knowledge* to obtain a better and more accurate view on the system's actual dependability attributes. Not only will it avoid potentially complex calculations during the design phase, it will also allow to better track the system-environment fit throughout the system's operational life, based on realistic estimations of the dependability of the available resources (versions).
A key contribution is to be found in the formal definition of a mathematical structure that is capable of efficiently **capturing how a specific version** — (software) component — **has affected the reliability of the fault-tolerant redundancy scheme throughout its operational life span**. Its design was inspired by the $\boldsymbol{\alpha}$-count approach, resulting in an extremely lightweight

[17] With the emphasis on software, we will consider only failures resulting from the activation of design faults, plus general performance failures where a result cannot be acquired in time. Fig. 2.3 on p. 45 provides a clear overview of the (functional) software failure classes that are covered in our study. Other failure classes, like parametric and/or functional hardware failures, specific types of failures affecting network connectivity and middleware operations, remain out of scope and are considered as future research — *v.* Sect. 9.3.6.

memory footprint[18], as described in [77]. Values for this measure are computed taking into account various types of contextual information, including instantaneous estimations of the scheme's availability, as indicated ex-post by means of the distance-to-failure (*dtof*) measure that was originally announced in [78]. This novel measure is crucial to support redundancy schemata in which the redundancy configuration is autonomically adjusted, and is used to support the attributes of perception and (environment-)awareness that autonomic computing systems require, as elucidated in Sect. 1.2.
This mathematical structure is formalised in **Chapt. 4**, and the supporting metrics that serve as the foundation are defined in **Chapt. 3**. Whereas these chapters focus exclusively on majority voting, the structure was also formalised for unanimity voting in **App. B**.

RQ-2. **Is it possible to realise an increase in dependability using a dynamic redundancy configuration in NVP schemata? Is it possible to economise on redundancy expenditure without jeopardising the overall effectiveness of an NVP redundancy scheme in terms of dependability?**
Traditional n-version programming redundancy schemata rely on a static redundancy configuration. Our objective is to see if such scheme's availability can be better sustained by steering the applied level of redundancy in function of the perceived system-environment fit, and by optimising the selection of the underlying redundant resources. At the same time, we will explore the possibility to economise on resource expenditure by avoiding inefficient or unneeded allocation of redundant resources.
The main contribution of the research reported in this dissertation is the **formal description of a parameterised algorithm responsible for the adjustment of the redundancy configuration** used in NVP redundancy schemata **in view of the environmental context** in which the scheme has been and is currently operating. From a dependability perspective, its design objective is to sustain high availability and reliability, whereas from a resource expenditure perspective it should focus on the exclusion of replicas that do not significantly contribute to the application's objectives and goals. The researcher may wish to target other non-functional properties, including, but not limited to, timeliness[19]. The proposed A-NVP/MV algorithm is helpful in building autonomic and resilient software systems. Such systems will exhibit all four of the self-management capabilities that an autonomic computing system is expected to exhibit: it is self-configuring, since it will automatically and independently tune the employed redundancy configuration as needed; it is self-healing and self-protecting, since faulty replicas that do not consistently contribute to the application's objectives and goals are (proactively)

[18] Such monitoring capability should result in negligible overhead, a low memory footprint and little additional resource cost for the deployment of dedicated software components, which should broaden its applicability and promote adoption in various environments, including resource-constrained environments.

[19] We apply a relative notion of timeliness here, in that a specific timeout t_{max} can be set in case of latency-sensitive applications. Similar to [72], in the assumption that the overhead of the A-NVP scheme is negligible, version invocation requests would be required to have returned a response in due time, for it to be used as ballot in the voting procedure — *v.* assumptions (A38) and (A37). If not, the version's result will not be considered to adjudicate an outcome. Refer to Sect. 3.2 and 5.1 for more details.

substituted; and it is self- optimising, so that the optimal selection of available replicas will be used to maximally support the objectives as defined by the application designer. This algorithm is defined in **Chapt. 5**, while the three supported application objectives (*viz.* dependability, resource consumption and timeliness) are extensively covered in **Chapt. 3**.
Another contribution is the evaluation of the effectiveness of the proposed algorithm, including a preliminary assessment of some promising policies for redundancy management, which can be found in **Chapt. 7**.

RQ-3. **Can the effectiveness of a policy for redundancy management be reliably analysed upfront?**
In order to attain a specific dependability level, one needs to ascertain there is a proper system-environment fit, in which the redundancy scheme and the underlying redundancy configuration is capable of masking any disturbance that may result from the influence of the environment in which the scheme is deployed and operating. For mission-applications, trial and error is not an option, and we need tools to analyse under which conditions the desired dependability level would be reached — well before the system is actually placed in production.
Our contribution can be found in the design and implementation of a **comprehensive discrete event simulation framework** tailored to the execution of rigorous performance analyses of new policies for (autonomic) redundancy management within the scope of various types of redundancy schemata. The framework comes with a wide range of artefacts and templates that support the designer in properly modelling the environment to a level of sufficient detail, and tools to exert control on the environment, its behaviour and properties whilst conducting large-scale experiments. This includes, amongst others, the means to define how and when a specific version may/will fail, to inject the desired type of fault at specific stages throughout a simulation run, how the load imposed on the system is to be balanced and routed to various system components *etc.* Various predefined metrics will report and provide insight on the system-environment fit or mismatch. Furthermore, the framework can easily be extended and customised, broadening its applicability to other research domains as well.
Available features in the discrete event simulation framework — including, but not limited to, failure injection, metrics and measures — are extensively described in **Chapt. 6**, whereas the formal models that define the implemented behaviour can be found in **Chapt. 2**.

RQ-4. **Can autonomous redundancy management be implemented as a well-separated concern within the context of contemporary distributed computing systems?**
Monolithic applications are difficult to debug, and complicate adding new functionality. With the growing complexity of software, simplification by means of abstraction and modular design have become key to allow for improved maintainability. Dedicated application logic for fault tolerance should be non-intrusive, in that it should be implemented well-isolated from the core application business logic.
Having scrutinised the self-managing capabilities that autonomic computing

systems are expected to implement, and how these directly map to properties of resilient software, our contribution can be found in the design of a **flexible implementation library to support the development of fault containment units**. The library was built on readily available, proven open-source software [79]. Furthermore, we argue that proper **separation of concerns** can be achieved by building solutions on top of the WS-* stack, where the WSDM and WSRF specifications in particular can aid in isolating effective implementations of autonomic capabilities from the underlying managed resources [80]. As an example, an implementation of an FCU is available, where the redundancy employed within the FCU is autonomously managed by the A-NVP/MV algorithm reported in this dissertation.
More details on the use of WS-* specifications for implementing (A-)NVP can be found in **Chapt. 8**, and **App. C**.

RQ-5. **How to define, implement, and examine the effectiveness of various strategies and policies for static and/or dynamic redundancy management?**
Applying the A-NVP algorithm calls for adequate tools to define redundancy management policies, and assess the system-environment fit in view of a specific target deployment environment. To do so, robust simulation tools are needed to both model the policies and configuration of the A-NVP configuration, and to assess its performance when varying the environmental properties.
The flexibility of the toolchain resulting from this research encourage the designer to "enable reconfiguration mechanisms that refocus the available, safe resources to support the most critical services rather than over-provisioning to build failure- proof systems". Both the simulation framework as well as the implementation library come with parameterised templates that ease the burden of **modelling redundancy allocation policies**. Furthermore, the designer is assisted in the creation of **precise models of the environment** in which the system is expected to operate, allowing him/her to exert control on specific environmental behaviour and properties while examining the scheme's effectiveness from various positions of interest. The offered toolset is built on Java™open-source software, so further customisation is possible whenever required.
An extensive overview of the feature set of the toolset is given in **Chapt. 6**, whereas the generic parameterised algorithm is defined in **Chapt. 5**, and examples of concrete redundancy allocation policies can be found in **Sect. 7.3**.

Starting from an extensive literature study on software fault tolerance, we decided to start by addressing these first two research questions by defining a formal and theoretical model to approximate the *actual* robustness of software components in terms of dependability, based on the *dtof* metric defined in [78]. The rationale for doing so was the objective of keeping track of the system-environment fit, or rather the evolution of this fit throughout the operational life of a fault-tolerant NVP composite. This approach has the advantage that, with some reasonable delay, it should be able to detect changes in the fault model. The formalism of the normalised dissent metric became the basis of our A-NVP algorithm, that was designed to provide a robust and configurable solution for autonomously steering

the redundancy configuration towards an optimum selection of versions, both quantitatively and qualitatively.

To address these research questions, emulation in the context of distributed computing solutions did not seem a viable option, because of the inherent complexity of such computing environments, where it proved to be extremely challenging to exert a proper level of control on the environment. Discrete event simulations allow to model the environment and the system under investigation to be modelled in great detail, and to achieve the needed levels of determinism and reproducibility when running simulations. It also allows to analyse particular conditions in detail, and to easily vary specific parameters — something that would be much more challenging in the case of emulation. To proceed, we formally defined several simulation models that reflect the way so-called design faults materialise in reality. These models served as the basis for the simulation tools that were developed[20] and used to assess the performance of the proposed A-NVP algorithm. Additional advantages of using simulation are:

– the system and its performance can be scrutinised in great detail before it is put in production, thereby reducing the likelihood of catastrophic failure (provided that fault and system models reflect reality);
– many predefined metrics and measures have been defined and implemented, so various attributes can easily be inspected without requiring additional effort to analyse and process results — *v.* Chapt. 6.

Our experimentation was mainly focused to highlight the advantages of A-NVP, compared to the classic NVP technique that was initially published in 1985 [6]. Using discrete event simulations, we were able to corroborate our claim that A-NVP has the potential to economise on resource expenditure, while sustaining the redundancy scheme's ability to tolerate faults in a way that is no worse than for classic NVP. Thereby, we have been able to clearly demonstrate that applying a dynamic redundancy configuration is likely to result in more robust levels of software fault tolerance, especially when it is adjusted autonomously based on detected disturbance (*i.e.*, indications of failures inferred from the majority voting procedure).

For the scope of this dissertation, we have used simplistic fault models with the aim of generating concise results *visually* by means of charts, graphs and tables (primarily to improve readability). Because of this, when injecting failures with more general distributions (for instance, by sampling from a lognormal distribution), we expect significantly more randomness and less marked trends in the number of injected failure per voting round [82]. One may therefore expect to observe a moderate slow-down in the way A-NVP is steering the redundancy configuration. Note that the simulation framework was designed to support various failure injection techniques and supports the designer in accurately defining specific fault and system models. Hence, the framework can be used to inject failures by sampling from a theoretical model, where the time between fault occurrences is sampled from some distribution, or where failures can be injected based on data that was logged/analysed for systems in production.

[20] Our simulation framework is a Java™-based implementation of the models defined in Chapt. 2 using the SSJ software library, which was developed at the Université of Montréal [81]. The framework bring additional features, like an implementation of the *dtof* and normalised dissent metrics (Chapt. 3, resp. 4), of the A-NVP algorithm (Chapt. 5), and various options to inject failures and to report different properties and attributes (Chapt. 6).

1.7 List of Publications

For completeness, this section is included to provide an overview of the publications through which the research reported throughout this dissertation were initially announced:

– Jonas Buys, Vincenzo De Florio, and Chris Blondia. **Towards Context-Aware Adaptive Fault Tolerance in SoA Applications**. In *Proceedings of the* 5^{th} *ACM International Conference on Distributed Event-Based Systems (DEBS)*, pp. 63–74, 2011. Association for Computing Machinery, Inc. (ACM) [83].
The A-NVP algorithm was first presented at the DEBS 2011 conference in New York (NY, USA). This publication covered **Chapt. 3–5 and 8**. It contained the full description of the replica selection model (Sect. 5.3), and a first, incomplete version of the redundancy dimensioning model (Sect. 5.2)[21]. A complete implementation of the A-NVP composite as a WSDM-enabled FCU, where the underlying versions are wrapped and exposed as WS-Resources, was also described — *v.* Chapt. 8.

– Jonas Buys, Vincenzo De Florio, and Chris Blondia. **Towards Parsimonious Resource Allocation in Context-Aware n-Version Programming**. In *Proceedings of the* 7^{th} *IET International Conference on System Safety and Cyber Security*, volume 607 of *IET Conference Publications*, pp. 137–144, 2012. The Institute of Engineering and Technology (IET) [84].
This paper was presented in Edinburgh, Scotland, United Kingdom. Again, the A-NVP algorithm was formally explained — covering **Chapt. 3–5** — however this time with an extended version of the redundancy dimensioning model (in its final form as it is included in Sect. 5.2). Having struggled against the shortcomings of emulation, we implemented a discrete event simulation framework to better control environmental behaviour, including several options for failure injection. Although the concepts and definitions in Chapt. 2 and 6 laid the basis of the preliminary performance analyses[22] included in the paper, due to space restrictions, neither of these chapters were shared with the community.

The following list of publications relate to a separate, albeit related, research track, in which we have shown that some of the available linguistic constructs in WS-BPEL can be used to implement redundancy schemata like NVP and RB. This work is included as **App. C**, and is somehow related to Chapt. 8, in the sense that WS-BPEL is an alternative WS-* specification that can be used to achieve a clear separation of concerns by isolating the actual business logic — encapsulated and exposed as web services — from the dedicated fault-tolerant orchestration logic — *v.* **Sect. 8.4.2**[23].

[21] Based on a combination of the redundancy allocation mechanism defined in [78], corresponding to policy Strategy A, Variant 2. Refer to p. 151 for more information.

[22] A more in-depth analysis of the initial experiment that occurred in [84] can be found as Experiment 7.9 in **Chapt. 7**. The objective was to illustrate that the use of a dynamic redundancy configuration, using the A-NVP algorithm and applying a safety margin c_{sm} — essentially an example of Strategy B, Variant 1 as defined on p. 154 — can effectively result in a higher level of reliability *and* a lower cumulative amount of allocated redundancy.

[23] We no longer advocate the use of WS-BPEL to implement NVP and RB. The WSDM-based solution described in Chapt. 8 is far more flexible, as the use of WSRF-SG service groups allows for convenient runtime discovery of functionally-equivalent versions. Furthermore, the WS-BPEL language is quite limited, and in itself does not allow to implement the A-NVP algorithm defined in Chapt. 5.

– Jonas Buys, Vincenzo De Florio, and Chris Blondia. **Applying Business Process Re-engineering Patterns to Optimize WS-BPEL Workflows**. In *IT Revolutions*, volume 11 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering (LNICST)*, pp. 151–160. Springer Berlin Heidelberg [85].
Presented at the 1^{st} international ICST Conference on IT Revolutions, organised in Venice, Italy, December 17–19, 2008.

– Jonas Buys, Vincenzo De Florio, and Chris Blondia. **Optimization of WS-BPEL Workflows through Business Process Re-engineering Patterns**. *International Journal of Adaptive, Resilient and Autonomic Systems*, 1(3), pp. 25–41, 2010 [68].

– Jonas Buys, Vincenzo De Florio, and Chris Blondia. **Optimization of WS-BPEL Workflows through Business Process Re-engineering Patterns**. In Vincenzo De Florio, editor, *Technological Innovations in Adaptive and Dependable Systems: Advancing Models and Concepts*, chapter 20, pp. 345–361. IGI Global, 2012 [86].

The above publications have been occasionally cited in the literature, which show the relevance of our contributions in the domain of fault-tolerant engineering of SoA solutions [87–89], and the domain of reliability engineering [90–93].

1.8 Structure

The remainder of this thesis is structured as follows: We first present the concept of NVP/MV schemata in Chapt. 2 and show how disturbances emerging from the activation of software design faults may put their effectiveness into jeopardy. A set of ancillary metrics is then set forth in Chapt. 3, allowing to capture contextual information regarding the environment in which the scheme is operating, primarily with respect to disturbances that challenge its effectiveness, and that enable to detect the proximity of hazardous situations that may require the adjustment of the redundancy configuration. After introducing another metric designed for approximating the operational status of individual resources in terms of reliability in Chapt. 4, we move on to elaborate on the internals of the proposed adaptive fault-tolerant strategy in Chapt. 5. Next, an overview of the architectural framework for the discrete-event simulations used for the validation of the proposed dependability strategy is given in Chapt. 6, followed by an overview of the metrics that are available to the designer, and a formal definition how each is actually measured. We proceed by reporting on the strategy's effectiveness analysis in Chapt. 7. A prototypical service-oriented implementation of the proposed adaptive fault-tolerant strategy is presented thereafter, leveraging WS-* specifications to realise proper separation of concerns, and to gather and disseminate contextual information. We conclude by summarising the main conclusions drawn and reported in this dissertation, and point out interesting future directions, as well as open research questions.

CHAPTER 2

Essential Simulation Models

In this chapter, a survey is given of how NVP-based redundancy schemata are expected to operate in the context of contemporary distributed computing environments that exhibit a timed asynchronous system model. The behaviour of such fault-tolerant schemata is formalised by means of a discrete event model that allows to unravel how potential disturbances emerging from the activation of software design faults and the occurrence of performance failures can affect these systems and may put their effectiveness into jeopardy. A fault injection and failure manifestation model is then outlined so as to characterise the different ways in which disturbances may materialise, the repercussions they may have on the proposed discrete event model, and the impact they may have on the outcome of the redundancy scheme. Related research question(s): RQ-3.

2.1 Voting Round State Transition Model

Let $\{\ell_x\}_{\mathcal{C}}$ be a sequence of monotonically increasing, strictly positive integer indices $\ell_x = x$ in $L = \mathbb{N}^+$, such that each voting round, *i.e.* a single invocation of an NVP composite $\mathcal{C}$, is uniquely identified. As shown in the state transition diagram in Fig. 2.1, the arrival of a request message at the composite interface will trigger the initialisation of a new voting round $(\mathcal{C}, \ell)$ with ℓ the next element in $\{\ell_x\}_{\mathcal{C}}$. Immediately after, the system is to retrieve the redundancy configuration to be used throughout the newly initialised voting round $(\mathcal{C}, \ell)$, *i.e.* the amount of redundancy used and, accordingly, a selection of functionally-equivalent software components (transition from state `(a)` to `(b)`). We define the set V containing all functionally-equivalent versions available in the system. For a given round $(\mathcal{C}, \ell)$, the amount of redundancy used within the NVP scheme is denoted as $n^{(\mathcal{C},\ell)} \geq 1$, such that the versions employed for round $(\mathcal{C}, \ell)$ are contained within $V^{(\mathcal{C},\ell)} \subseteq V$ and $n^{(\mathcal{C},\ell)} = |V^{(\mathcal{C},\ell)}|$.

Having acquired the redundancy configuration, the request message payload will then be replicated and forwarded to each of the selected versions $v_i \in V^{(\mathcal{C},\ell)}$ for processing. Each such invocation of a version v_i can be uniquely identified by the tuple $\langle \mathcal{C}, \ell, i \rangle$, with i a natural number that uniquely identifies the corresponding version (replica) instance.

An NVP redundancy scheme relies on a decision algorithm in an attempt to overcome any disparities in the results acquired for each of the subordinate invocations $\langle \mathcal{C}, \ell, i \rangle$, and to adjudicate a satisfactory result to be returned for the voting round $(\mathcal{C}, \ell)$ nonetheless. Such disparities may arise as the result of disturbances affecting the operational status of any of the versions $v_i \in V^{(\mathcal{C},\ell)}$ — *cf.* Sect. 2.4. The essential part of any voting mechanism is the construction of a generalised partition $\wp^{(\mathcal{C},\ell)}$ of the set of versions $V^{(\mathcal{C},\ell)}$. This partitioning procedure is initiated in state `(b)` within the scope of a specific voting round $(\mathcal{C}, \ell)$, immediately after the invocations $\langle \mathcal{C}, \ell, i \rangle$ have been issued, and runs in complete isolation of other (pending) voting rounds. Whenever a ballot has been secured for one such invocation, the partially constructed partition is updated and the corresponding version v_i will be classified as a member in an appropriate equivalence class. The equivalence classes that emerge from the partitioning procedure ultimately reflect which versions are in mutual agreement, *i.e.* which versions returned similar responses in view of the equivalence relationships defined in Sect. 2.6.1.1. As soon as a result is available for each of the versions involved, the transition from `(b)` to `(c)` will fire, the partitioning procedure terminates and the construction of $\wp^{(\mathcal{C},\ell)}$ is completed. At this stage, all information is available to adjudicate a result; if none can be found, a failure message will be returned to the client.

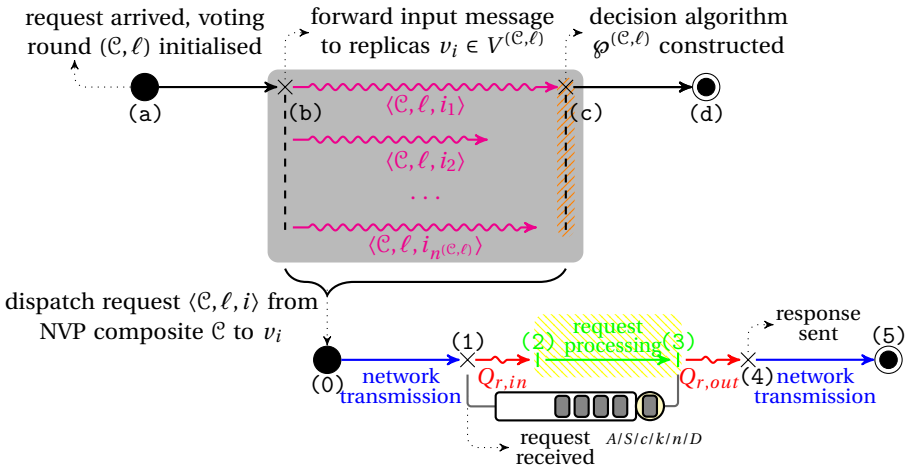


Figure 2.1: Version invocation state transition model.

Whilst an attempt is made to adjudicate a result, information will be extracted from the generated partition. For instance, in case of majority voting, the scheme will only manage to identify a result if there exists some equivalence class in $\wp^{(\mathcal{C},\ell)}$ with a cardinality no less than a qualified majority of the employed degree of redundancy $n^{(\mathcal{C},\ell)}$. If such an equivalence class is found, the associated result, based on the equivalent ballots that were reported by the members classified inside that class, will be returned.

Note that a result for a voting round $(\mathcal{C}, \ell)$ may be returned pre-emptively, depending on the type of voting mechanism used, and even before a result has been acquired for each of the subordinate invocations $\langle \mathcal{C}, \ell, i \rangle$. The conditionalities

are specific to each type of voting mechanism, which can only inspect the (partially) constructed partition immediately after each successive partition update, *i.e.* whenever a result for some version $v_i \in V^{(\mathcal{C},\ell)}$ has been acquired and the version has been classified into some equivalence class accordingly. More information regarding voting pre-emption can be found in Appendix B. Regardless of whether an adjudicated result is returned before the transition into state `(c)` has completed, results will be collected for the pending requests and accounted for by the partitioning procedure.

2.2 Version Invocation State Transition Model

The process of invoking a version v_i from within a voting round $(\mathcal{C},\ell)$ — a request denoted by $(\mathcal{C},\ell)$ — can be characterised according to the state transition diagram shown in Fig. 2.1. In the upper half, one can see the general state transition diagram of the voting round itself. In this particular case, none of the engaged versions v_i are subject to performance failures. A high-level overview of the steps in the version invocation itself, initialised when the replicated input is forwarded to v_i for processing, is provided in the lower half below. The model draws upon the Request Processing state transition model as defined in the Web Service Management: Service Life Cycle (WSLC) specification [94].

After a request $\langle \mathcal{C},\ell,i \rangle$ has been transmitted over the network, it will be handed over to the deployment environment hosting v_i, resulting in a transition to the `RequestReceived` state `(1)`.

The admission and scheduling of incoming request is henceforth managed by the host's queuing policy, which is responsible for deferring the request until the required processing capacity and/or system resources have become available. While the host acquires the necessary system resources, the request will be temporarily stalled and await further processing until the conditions allow for transition $\mathtt{Q}_{\mathtt{r,in}}$ to fire. It is only at this point with `RequestProcessing (2)` as current state, that the functionality of the version v_i will be called to process the request **(A01)**. Whenever a request has been processed, *i.e.* immediately after it has completed the transition into its `RequestProcessed (3)` state, the corresponding system resources are released, and the next request (if available) will be taken from the waiting queue: it too will then transition into its `RequestProcessing (2)`.

As soon as v_i has completed execution, the result will be handed over to the deployment environment, awaiting network transmission, which is symbolised as the transition $\mathtt{Q}_{\mathtt{r,out}}$. Finally, the invocation process $\langle \mathcal{C},\ell,i \rangle$ terminates when the response has been delivered at the NVP composite, such that its response can be utilised for the continuation of round $(\mathcal{C},\ell)$.

Throughout this dissertation, the term pending request will be used to denote a request that has accomplished the transition into state `(1)` though has yet to enter into state `(4)`. During this stage of its life span, it is being handled by the deployment environment on which the target version v_i is deployed, and therefore is susceptible to failures that may arise within that environment.

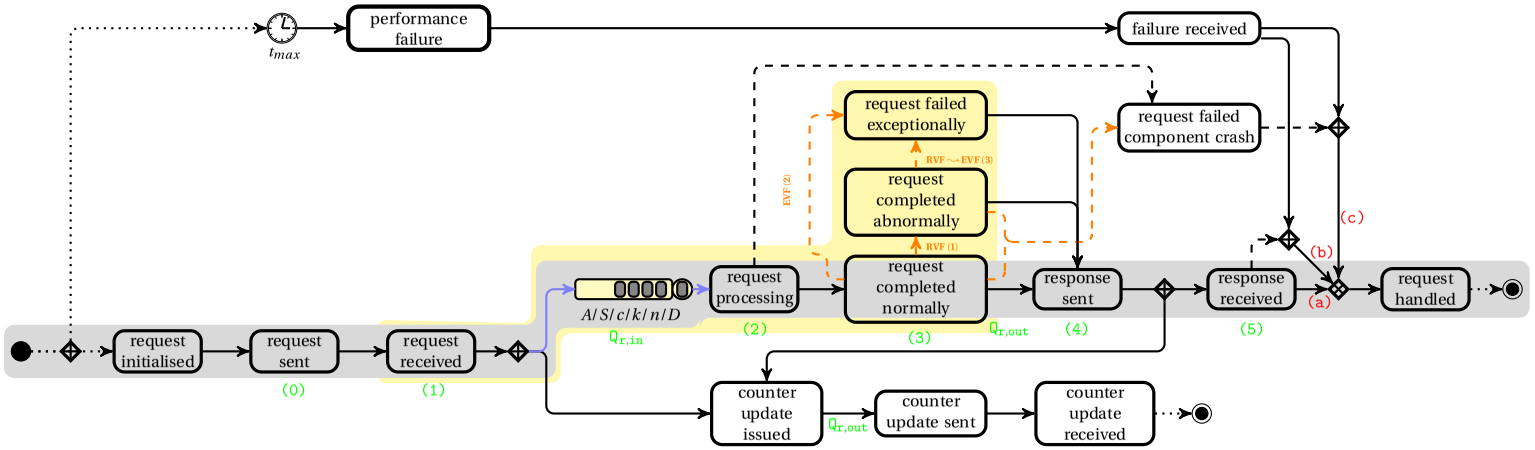


Figure 2.2: Version invocation discrete event model designed to model the ballot retrieval procedure for an invocation $\langle \mathcal{C}, \ell, i \rangle$.

2.3 Version Invocation Discrete Event Model

Fig. 2.2 shows a predefined set of discrete events and the potential chronological sequences thereof that are representative of the operation of a single version invocation. This discrete event model is at the core of the simulations used throughout this thesis for the assessment of the effectiveness of the proposed dependability strategy, and formalises the abstract invocation state transition model described in the previous section[1]. The retrieved ballot will be used as the response of version v_i when the decision algorithm is expected to adjudicate the outcome for voting round $(\mathcal{C}, \ell)$, and may be susceptible to different types of disturbances that occurred during the lifetime of invocation $\langle \mathcal{C}, \ell, i \rangle$. The shaded area highlights the normal event sequence in the absence of any type of disturbance; an overview of potential disturbances as well as their repercussions on the ballot retrieval procedure is elucidated in Sect. 2.6. Note the use of timer, divergence and convergence elements, denoted by markers taken from the BPMN specification, which may result in multiple sub-sequences of events to be followed contemporaneously[2] [95].

The event sequence(s) that will eventually result from the progression of an invocation $\langle \mathcal{C}, \ell, i \rangle$ is driven by the principle of event replacement. One can distinguish three options in the way events are replaced:

1. An event can be successively replaced by another self-instated event scheduled when it has completed processing (denoted by solid, black arcs). As per Sect. 1.4, an event is an atomic unit of work that may cause the system to change state. Once its processing starts, all steps and actions associated with it will be processed in full and without interruption. While it is processed, there can be no interleaving of actions pertaining to other events, as only a single event can be processed at any time.

2. It can be preemptively replaced when it was previously scheduled but it did not occur by the time at which another, external event occurred (dashed arcs). This type of event replacement is typically triggered by the occurrence of failure activation events.

3. Finally, event replacement can be deferred until a specific condition holds, *e.g.* until the queuing policy indicates that some processing capacity is available (blue arcs).

For the time being, we will only focus on the normal event sequence that would materialise for an invocation $\langle \mathcal{C}, \ell, i \rangle$ in the absence of any type of disturbance, as it is highlighted in the shaded area shown in Fig. 2.2. Note how this event sequence emerges almost exclusively from successive event replacements, which have the advantage that a replacement event can be scheduled to occur after some delay relative to the time the replaced event was previously scheduled to occur, thereby enabling the simulation of network transmission delays, service and waiting/sojourn times. Other event sequence(s) will be systematically expounded upon over the next few sections.

[1] The states in the lower part of Fig. 2.1 have been included in green in Fig. 2.2, to allow for more convenient cross-referencing.

[2] For a given invocation, at most one event is scheduled at any time for each sub-sequence in progress.

Upon initialisation, an event of type `RequestInitialised` will be scheduled for immediate processing, which will be replaced by a `RequestSent` event, possibly scheduled with deferred occurrence time, which may accommodate any overhead situated at the NVP composite prior to the actual transmission of the replicated request to v_i[3]. The delay applied when the `RequestSent` event instates its replacement `RequestReceived` event corresponds to the network latency for sending the invocation request message from the NVP composite to the host on which v_i is deployed.

Next, when the `RequestReceived` event is processed, two events will be scheduled, which is denoted by the parallel gateway shown at its right: the upper branch will consider the sojourn time in state `(1)` before the transition $\mathtt{Q}_{\mathtt{r,in}}$ can be fired, and will schedule the replacement `RequestProcessing` event accordingly (*cf.* Fig. 2.1); the lower branch will initiate a new context update by means of a `CounterUpdateIssued` event, a procedure which is isolated from the core invocation event sequence and that will be explained in Sect. 2.7. Observe how the conditional replacement of the `RequestReceived` event by its successor `RequestProcessing` event will be postponed (if needed) for as long as all available processing capacity at the host of v_i has been exhausted. The admission of requests for processing is coordinated using a queuing model; its scope is depicted in yellow in Fig. 2.2, which identifies the relevant event (replacements) that interact with it. The simulated invocation $\langle \mathcal{C}, \ell, i \rangle$ will then appear to be processed by v_i for a duration equal to the given service time, after which the event `RequestCompletedNormally`, replacing its `RequestProcessing` predecessor, will be processed. During its processing, it will (i) relinquish the processing capacity that had been allocated for the request's servicing[4], and (ii) schedule a replacement `ResponseSent` event to occur when network transmission of the outgoing response message has commenced — *cf.* $\mathtt{Q}_{\mathtt{r,out}}$ in Fig. 2.1.

After the `ResponseSent` event is processed, the divergence element shown at its right will again cause the scheduling of two separate events: a replacement event of type `ResponseReceived`, scheduled to occur when the response has arrived at the NVP composite, and a `CounterUpdateIssued` event. Finally, because it was assumed no disturbance of any type affected the invocation, the exclusive gateway convergence element shown at the right of the `ResponseReceived` event will be triggered by path `(a)`, resulting in the scheduling of a `RequestHandled` replacement event, which will report the result for invocation $\langle \mathcal{C}, \ell, i \rangle$ to the decision algorithm.

2.4 Fault Manifestation Model

The essential part of any voting procedure is the construction of a generalised partition $\wp^{(\mathcal{C},\ell)} = \{P_F^{(\mathcal{C},\ell)}, \dot{\cup}_{j\in\{1,\ldots,k^{(\mathcal{C},\ell)}\}} P_j\}$[5] of the set of versions $V^{(\mathcal{C},\ell)}$. This partitioning procedure is heavily influenced by the disturbances that affected any of the requests $\langle \mathcal{C}, \ell, i \rangle$ involved during the voting round $(\mathcal{C}, \ell)$. Throughout this dissertation, the notion of disturbance is used to denote the event of a single request $\langle \mathcal{C}, \ell, i \rangle$

[3] Examples could be memory — or, more general, resource — contention, or serialisation overhead of the internal data representation to a format that can be sent over the wire — *v.* p. 94.

[4] The same applies to any other event type implementing the `RequestProcessed` state. This includes events of type `RequestCompletedAbnormally` and `RequestFailedExceptionally`.

[5] The notion of a generalised partition as a partition that may contain empty blocks has been taken from [96]. A reduced notation $\dot{\cup}_{j\in\{1,\ldots,k^{(\mathcal{C},\ell)}\}} P_j$ is used to denote the partition $\{P_1, \ldots, P_{k^{(\mathcal{C},\ell)}}\}$. Unless explicitly stated otherwise, a partition is assumed to contain only non-empty equivalence classes.

struck by some type of failure, resulting in the perturbation and, consequently, the (temporary) unavailability of the service that version v_i is expected to provide **(A02)**. We will now elaborate on several types of disturbances relevant to NVP/MV schemes, how such disturbances influence the transition path in the version invocation state transition model presented in Fig. 2.1 and elaborate on the repercussions their occurrence may have on the version invocation discrete event model as depicted in Fig. 2.2, as well as their effect on the generated partition $\wp^{(\mathcal{C},\ell)}$. More specifically, disturbances will be categorised using the comprehensive list of failure classes presented in Fig. 2.3. Each of the classes shown represents a distinct manifestative behaviour for a deviation of the service an affected version is sought to provide. It reflects the total order in manifestation severity defined in Sect. 2.6.5 (for disturbances pertaining to software failures for a pending request being serviced).

Observe how the `RequestHandled` event was defined to report the result for an invocation $\langle \mathcal{C}, \ell, i \rangle$ to the decision algorithm. Throughout this section, a total order will be defined of the failure classes shown above in Fig. 2.3, which will serve as an abstraction layer of the range of possible results that can be returned for $\langle \mathcal{C}, \ell, i \rangle$ by means of `RequestHandled` events **(D01)**. In addition, this event can also signal that the result for the invocation is in accordance with the functional specifications, as it would be expected when no disturbances affected its course **(A03)**. The result to be returned for an invocation will emerge from the first event combination that activates one of the three event replacement paths comprised within the exclusive merge convergence element shown at the immediate left of the `RequestHandled` event in Fig. 2.2.

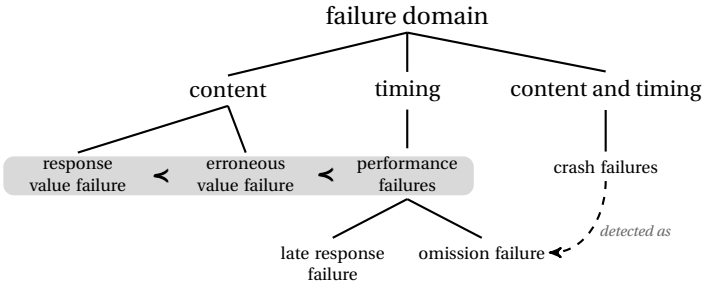


Figure 2.3: Failure classes situated in the different failure manifestation domains (generalisation of the concepts originally published in [97] and [4, Sect. 1.2.1]).

2.5 System Model

The application of NVP/MV schemata in contemporary distributed computing systems is assumed to exhibit the properties of a timed asynchronous distributed system model **(A04)** [4, 98, 99]. It is because of this reason that the WSLC Request Processing transition model was chosen as the foundation of the version invocation state transition model in the first place [94]. Assuming this type of system model allows to explicitly characterise the distinct system components on which NVP systems rely in terms of their failure semantics, *i.e.* the behaviour that each is likely to exhibit upon failures. This can be accomplished by defining the manifestative

behaviour for each of the failure classes shown in Fig. 2.3, and listing the applicable failure classes to which each particular type of system component may be subjected. As such, a system characterised by the timed asynchronous distributed system model can be expected to exhibit the following properties:

1. Software components located on different hosts communicate by exchanging messages over an unreliable network datagram service with performance failure semantics. The datagram service may occasionally drop messages, resulting in an omission failure, or it may delay transmission such that messages do not arrive within the imposed timeout (if one is specified that is), which would translate into a late response failure.

2. A timeout service is available at application level, by means of which performance failures can be detected. It relies on the local hardware clock to which software components are granted access. Clocks on different network hosts have a bounded, though unspecified, drift rate [98].

3. "All services [in the system] are timed: specifications prescribe not only the outputs and state transitions that should occur in response to inputs, but also the time intervals within which [...] these outputs and transitions [may be expected to occur]" [98]. Software components in particular (including those related to the operating system and the middleware deployment environment) have content/performance failure semantics. If the underlying algorithm fails to produce a correct result, the service will suffer a content failure; a late response failure will occur if it fails to return a result within the imposed timeout. If the algorithm has crashed and has ceased execution, omission failures will emerge.

4. Components susceptible to performance failures may crash and persistently exhibit omission failures for a prolonged period of time. Affected components may subsequently recover, although this is not assumed to be the case throughout this thesis **(A05)**.

For the remainder of this chapter, however, we shall only consider the content, crash and performance failure classes as disturbances that may potentially affect the operations of individual versions. The network datagram service, the middleware deployment environment and the NVP composite itself are assumed to behave properly without any disturbances appearing **(A06)** — *cf.* (A04), properties 1 and 3. This implies that no disturbances can emerge from an improper configuration of the system or any of its components, and that no administrative mistakes are made by the people operating the system [100].

During the past 40 years, queuing models have been found to serve well as an effective tool to model and analyse computing systems and their processing capacity planning. As such, a suitable queuing policy was sought to model the admission of requests for processing at individual replicas, which best matches the internal characteristics of conventional service endpoint implementation technologies. Throughout this thesis, it is assumed requests are admitted for processing by a specific version based on a $G/G/c$ queuing model **(A07)**. The choice for this type of model reflects the properties defined in the Java™ Servlet Specification and its reference implementation supplied with the Oracle Java™ EE runtime [101] — refer

to Sect. 8.5.2 and 6.1.1, p. 193, resp. 91, for more detailed information on queuing models.

Many different application servers have been developed with the purpose of deploying and hosting (servlet-based) web service endpoints. It is common for these server programs to maintain a pool of standby threads to deal with incoming requests. For the sake of simplicity, let us assume that a single pool of $c \geq 1$ threads is maintained for each (version) endpoint, each of which can be used to serve a single request at a time. Each request runs independently and in complete isolation of any other; the queuing discipline is based on a first come, first served (FCFS) scheme, without support for processor sharing — *v.* (A07). Once a request has been completely processed, the allocated thread will be released, after which it can then be reallocated for serving another request — *v.* (A42).

The system properties described above imply that the processing capacity of each version is managed by a multiple-channel, single-phase queuing system, and confirm our claim of a $G/G/c$ model. The worker threads that were mentioned correspond to identical server instances (channels) that service requests in parallel. Once a request has been processed by a worker thread, it no longer requires further service (single-phase processing).

2.6 Software Failure Classes: Effects, Scope and Duration

In complex software systems, there always remain design faults which eluded detection despite rigorous and extensive testing and debugging. Whether or not a latent software design fault is activated, is entirely dependent on the execution path followed when a version is invoked to process a particular request. More specifically, it is the (initial) internal software state and the received input arguments that will affect branch conditions and determine how execution is to proceed and which blocks of programming logic are to be executed. As program execution progresses and the selected code blocks along the execution path are executed, the design faults hidden within will be activated. The activation of a software fault is assumed to directly result in the emergence of a disturbance, without any latency between the fault's activation and its manifestation **(A08)**. Moreover, the effects of the manifestation of some disturbance in the content or timing failure domain affecting an invocation $\langle \mathcal{C}, \ell, i \rangle$ are considered to instantaneously result in the temporary unavailability of the service provided by v_i with respect to $\langle \mathcal{C}, \ell, i \rangle$, and in complete isolation of any other request **(A09)**. For such types of failure classes, the potential disruptive effects on the response value acquired for an affected request $\langle \mathcal{C}, \ell, i \rangle$ are expected to have dissipated by the time the voting round $(\mathcal{C}, \ell)$ transitions into its `(c)` state **(A10)**. With the exception of crash failures, it is possible that a failure may occur before the effects of another failure that occurred earlier on have dissipated.

Among software defects, a distinction is commonly made between Bohrbugs and Heisenbugs, primarly from a debugging perspective [100]. It rests on the (different) types of disturbances that can materialise due to the activation of a design fault. The former type of design faults will "systematically [result in disturbances of the same failure class] in the presence of [identical] input conditions and initial state" [4]. Whereas Bohrbugs exhibit deterministic properties in terms of the failure behaviour they induce, the disturbances caused by Heisenbugs may span several failure classes; their materialisation usually "depend[s] on subtle combinations of the system state

and its environment" [4]. This section will now elaborate how software design faults can manifest as content or crash failures **(A11)**.

2.6.1 Response Value Content Failures

Whereas it would be expected that functionally-equivalent versions sharing a common specification would return the same response when provided with identical input, discrepancies between their response values may arise due to response value failures (RVFs) [102]. Such type of disturbances may find their origins in the activation of (i) Bohrbugs, examples of which are (implicit) type conversions that were overseen during development, or careless deployment of the software on a host with a different arithmetic unit, as well as (ii) Heisenbugs, *e.g.* race conditions.

Despite being classified as failures, the occurrence of RVF failures during system operations will usually not make the system appear to fail. Rather, the content of the response returned via the service interface is syntactically correct, though diverges from implementing the service's functional specification — *cf.* (A02).

It is assumed that RVF failures manifest only when a latent software design fault is activated along the current execution path followed whilst version v_i is processing request $\langle \mathcal{C}, \ell, i \rangle$ **(A12)**, *i.e.* when the request is in the `RequestProcessing` state (A01), as can be seen from the yellow hatched area in Fig. 2.1. In spite of failure activation, v_i will proceed along the current execution path until completion, after which a response will be returned and transmitted over the network. As shown by transition `RVF (1)` in Fig. 2.2, the occurrence of an RVF failure affecting a pending request being processed and not subject to any other failure having previously occurred, will result in the preemptive replacement of the previously scheduled `RequestCompletedNormally` event by a `RequestCompletedAbnormally` event, scheduled at the same time at which the version v_i was initially expected to complete processing of request $\langle \mathcal{C}, \ell, i \rangle$. The expected service processing time therefore remains unaffected.

Let $V_{rvf}^{(\mathcal{C},\ell)} \subseteq V^{(\mathcal{C},\ell)}$ be the set of versions that, during voting round $(\mathcal{C}, \ell)$, were affected by an RVF failure and that were not subject to any other type of failure of higher severity. The event sequence for an invocation $\langle \mathcal{C}, \ell, i \rangle$, with $v_i \in V_{rvf}^{(\mathcal{C},\ell)}$ will resume along the normal event sequence highlighted in Fig. 2.2, and will conclude by a `RequestHandled` event reporting a result that is syntactically valid but not conform v_i's functional specifications. Furthermore, let $V_{nf}^{(\mathcal{C},\ell)} \subseteq V^{(\mathcal{C},\ell)}$ denote the set of versions that were not affected by any failure at all; a `RequestHandled` event will result in a similar way for any invocation of a $v_i \in V_{nf}^{(\mathcal{C},\ell)}$, signaling the exact value has been returned, in line with (A03). During the voting procedure, a partition $\dot{\cup}_{j \in \{1,\ldots,k^{(\mathcal{C},\ell)}\}} P_j = \wp^{(\mathcal{C},\ell)} \setminus P_F^{(\mathcal{C},\ell)}$ is constructed for all versions in $V_{rvf}^{(\mathcal{C},\ell)} \cup V_{nf}^{(\mathcal{C},\ell)}$ that returned a syntactically valid response. This partition will hold equivalence classes $P_1, \ldots, P_{k^{(\mathcal{C},\ell)}}$, such that each of these sets contains those versions $v_i \in V^{(\mathcal{C},\ell)}$ which reported identical results **(A13)**. Ideally, in a situation without disturbances of any kind, *i.e.* unanimous consensus, only one class P_1 would need to be created to accommodate all $n^{(\mathcal{C},\ell)}$ versions in $V_{nf}^{(\mathcal{C},\ell)}$ such that $\wp^{(\mathcal{C},\ell)} \setminus P_F^{(\mathcal{C},\ell)} = \{P_1\} = V_{nf}^{(\mathcal{C},\ell)}$ and $P_F^{(\mathcal{C},\ell)} = V_{rvf}^{(\mathcal{C},\ell)} = \emptyset$. Contrarily, dissenting versions in $V_{rvf}^{(\mathcal{C},\ell)}$ require the creation of additional equivalence classes.

Let c_b denote the cardinality $|V_{rvf}^{(\mathcal{C},\ell)}|$ and $\mathcal{B}^{(\mathcal{C},\ell)} = \dot{\cup}_{j\in\{0,\ldots,c_b\}} b_j$ a generalised partition with $c_b + 1$ possibly empty consensus blocks. Equivalence class $b_0 \in \mathcal{B}^{(\mathcal{C},\ell)}$ is defined to hold all versions that were not affected by any failure and that returned the exact result as it was expected from (A03). The other blocks b_j will group the remaining versions in $V_{rvf}^{(\mathcal{C},\ell)}$ based on the equivalence of the responses returned. When $\mathcal{B}^{(\mathcal{C},\ell)}$ has been constructed, the final partition $\wp^{(\mathcal{C},\ell)} \setminus P_F^{(\mathcal{C},\ell)}$ can be established by removing all empty equivalence classes. Moreover, its cardinality $k^{(\mathcal{C},\ell)}$ is representative of how many different syntactically valid responses were returned by versions in $V^{(\mathcal{C},\ell)}$.

2.6.1.1 Generating Response Values

In order to compensate for the application-dependent nature of the range of valid response values, our simulation model was deliberately designed for abstraction of actual response values **(D02)**. Rather, it will employ advanced techniques to generate an applicable response value for each invocation $\langle \mathcal{C}, \ell, i\rangle$ affected by a content failure. This procedure will be called upon adjudication when the voting round $(\mathcal{C}, \ell)$ transitioned into state `(c)`, i.e. after all invocations $\langle \mathcal{C}, \ell, i\rangle$ for each $v_i \in V^{(\mathcal{C},\ell)}$ have had their `RequestHandled` events processed.

Let $Ran(X)$ be a finite range for some random variable $X : \Omega \mapsto \mathbb{R}$ that is representative of the response acquired for an invocation $\langle \mathcal{C}, \ell, i\rangle$. In addition, we define a function $h^{(\mathcal{C},\ell)} : V^{(\mathcal{C},\ell)} \mapsto (Ran(X) \cup \bot)$ such that $h(v_i)$ yields a random variate $x \in Ran(X)$ symbolising a syntactically valid response that was acquired for an invocation $\langle \mathcal{C}, \ell, i\rangle$, or $\bot$ if no such response could be acquired in a timely manner. Moreover, each block b_j is defined to cover some specific interval $X_{b_j} \subset Ran(X)$, given that $\bigcap_{j\in\{0,\ldots,c_b\}} X_{b_j} = \emptyset$ such that any two random variates of X are considered as equivalent if and only if they both fall within the coverage interval X_{b_j} of the same consensus block b_j **(A14)**.

By analogy with the formalised majority voting procedure defined in [7, Sect. 2.1], versions $v_i \in V^{(\mathcal{C},\ell)}$ can be classified using an inexact notion of equality between results acquired from the corresponding invocations, an equivalence relation that can be formalised as follows:

$$\mathscr{R}_d^{(\mathcal{C},\ell)} = \{\, (v_{i_1}, v_{i_2}) \in V^{(\mathcal{C},\ell)} \times V^{(\mathcal{C},\ell)} \mid$$

$$[\, h^{(\mathcal{C},\ell)}(v_{i_1}) = h^{(\mathcal{C},\ell)}(v_{i_2}) = \bot \,] \tag{2.1a}$$

$$\veebar\, [\, \exists! j : h^{(\mathcal{C},\ell)}(v_{i_1}), h^{(\mathcal{C},\ell)}(v_{i_2}) \in X_{b_j} \,] \,\} \tag{2.1b}$$

The first clause (2.1a) expresses consensus amongst all versions in $V^{(\mathcal{C},\ell)}$ for which, within the scope of the current voting round, a late or syntactically invalid response was received, or that suffered from an omission failure. Such versions are to be classified in $P_F^{(\mathcal{C},\ell)}$ accordingly — *cf.* Sect. 2.6.2 and 2.6.4. Valid response messages that can be retrieved in a timely manner for the remaining versions in $V_{rvf}^{(\mathcal{C},\ell)} \cup V_{nf}^{(\mathcal{C},\ell)}$ will be used to assess and detect consensus on the basis of equivalent results, as shown by clause (2.1b). More specifically, if two random variates corresponding to the responses secured for any two versions $v_{i_1}, v_{i_2} \in V^{(\mathcal{C},\ell)}$ lie within a common

coverage interval X_{b_j}, then v_{i_1} and v_{i_2} are thought to be in consent and both shall be classified in b_j. We will now present two alternative approaches for generating (deterministically reproducible) response values for versions in $V_{rvf}^{(\mathcal{C},\ell)}$. In doing so, the notion of consensus based on the equivalence of response values, as formalised by means of relation $\mathscr{R}_d^{(\mathcal{C},\ell)}$, will be complemented by redefining clause (2.1b) in terms of a two-argument distance function $d : Ran(X) \times Ran(X) \mapsto \mathbb{R}_0^+$ inspired by [7]. Generally speaking, $d(x,y) \leqslant \epsilon$ for any pair of versions v_{i_1}, v_{i_2} held within b_j, with x and y random variates drawn as response values $h^{(\mathcal{C},\ell)}(v_{i_1})$, respectively $h^{(\mathcal{C},\ell)}(v_{i_2})$, and $\epsilon \in \mathbb{R}_0^+$. In other words, two responses are considered to be equivalent when they differ by no more than ϵ (within the context of a specific voting round $(\mathcal{C},\ell)$).

From a Normally Distributed Variable The first approach is based on the probability density function of the normal distribution $\mathcal{N}(\mu,\sigma^2)$, with $\mu = 0$ and $\sigma = 1$. The rationale of using this distribution is that it is assumed that outlier response values are centred around the exact value, and that extreme outliers are less likely to occur than minor deviations from the exact value **(A15)** [103, pp. 292–299]. As shown in Fig. 2.4, we now delimit $Ran(X)$ as the discrete interval $[\mu - 3\sigma, \mu + 3\sigma]$; this interval supports the vast majority of the possible response values (random variates of the normally distributed variable X) — up to 99.73% [104]. We now proceed by subdividing $Ran(X)$ into $2(c_b + 1)$ equidistant subintervals of size $\epsilon = {}^{3\sigma}/_{c_b+1}$, as illustrated in Fig. 2.4. Any coverage interval X_{b_j} for $j \in \{0,\ldots,c_b\}$ is now represented as the merger of two such subintervals spaced at equal distance of and at both sides of μ, i.e.

$$X_{b_j} = \begin{cases} [\text{-}\epsilon, \epsilon] & j = 0 \qquad (2.2a) \\ \big[\text{-}(j+1)\epsilon, \text{-}j\epsilon\big] \cup \big]j\epsilon, (j+1)\epsilon\big] & j \in \{1,\ldots,c_b\} \qquad (2.2b) \end{cases}$$

such that each block b_j will accommodate the same amount of variability in the response values generated for the versions classified within, in comparison to the exact value μ **(A16)** — thereby demarcating a range of possible output values that are considered to be equivalent, in line with (A13) and (A14). As the absolute value of a real number may intuitively be thought of as its distance from $\mu = 0$ and thus the magnitude of the discrepancy with respect to the exact value, we can redefine the distance function as $d(x,y) = ||x| - |y||$. Hence, one can then observe from Eq. 2.2 that, within the scope of a voting round $(\mathcal{C},\ell)$, the corresponding random variates representing the response value acquired from an invocation $\langle \mathcal{C},\ell,i \rangle$ of any of the consentient versions v_i classified within the same b_j are guaranteed not to differ by more than ϵ. In particular, for any two random variates $x, y \in X_{b_j}$ drawn from X as the response values of a pair of versions $v_{i_1}, v_{i_2} \in V^{(\mathcal{C},\ell)}$, clause (2.1b) of $\mathscr{R}_d^{(\mathcal{C},\ell)}$ holds if and only if $d(x,y) < \epsilon$ in case $v_{i_1}, v_{i_2} \in V_{rvf}^{(\mathcal{C},\ell)}$ are classified in b_j for $j \in \{1,\ldots,c_b\}$, or if $d(x,y) \leqslant \epsilon$ for versions $v_{i_1}, v_{i_2} \in V_{nf}^{(\mathcal{C},\ell)}$ classified in b_0.

In order to classify a version $v \in V_{rvf}^{(\mathcal{C},\ell)}$, we will draw a random variate x from the normally distributed variable X until we have a value that lies within some interval X_{b_j} other than X_{b_0}. The version under consideration will then be added to

the corresponding block b_j associated to the relevant coverage interval. Mapping a random variate x to some $b_j \in \{\mathcal{B}^{(\mathbb{C},\ell)} \setminus b_0\}$ is a two-stage procedure. First, an intermediate natural number is computed using the ancillary mapping function $f : Ran(X) \mapsto \mathbb{N}_0$:

$$f(x) = \left\lceil \frac{|x| - \epsilon}{\epsilon} \right\rceil \tag{2.3}$$

From this value $y = f(x)$, the destination block $b_j = g(f(x))$ can now be obtained by means of the function $g : \mathbb{N}_{[0,\ldots,c_b]} \mapsto \mathcal{B}^{(\mathbb{C},\ell)}$ such that $g(f(x)) = b_{f(x)}$. Classification of some version $v \in V^{(\mathbb{C},\ell)}_{rvf}$ entails the repeated process of drawing random variates x from a normally distributed random variable X until it was found that $g(f(x)) \in \{\mathcal{B}^{(\mathbb{C},\ell)} \setminus b_0\}$, implying the disposal of variates for which $f(x) = 0$ or $f(x) > c_b$ holds — *cf.* the constrained domain for which function g was just defined.

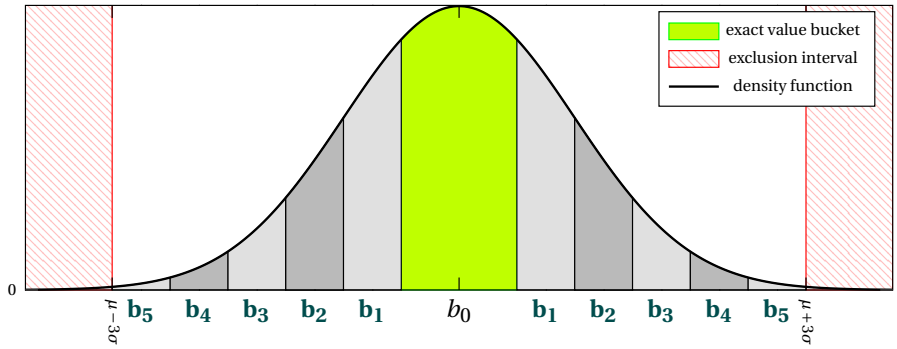


Figure 2.4: RVF classification system based on the probability density function of the normal distribution, assuming $c_b = 5$. Symmetrical coverage interval mergers in accordance with (A16).

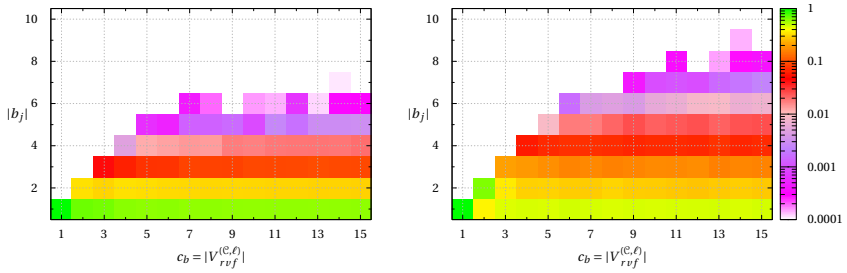
From a Uniformly Distributed Variable A second approach that can be taken is by sampling from a random variable X distributed according to a continuous uniform distribution $\mathcal{U}(a,b)$. However, assumption (A15) does not apply here; response values are equally likely to take any value within the interval $[a,b] = Ran(X)$ with probability $^{(b-a)}/_k$ **(A17)**. When choosing minimum and maximum values $a = 0$, respectively $b = c_b$, the common coverage interval width $\epsilon = 1$, and each coverage interval X_{b_j} will cover the random variates that lie within the range $[j-1, j]$, for $j \in \{1,\ldots,c_b\}$. Note that in contrast with the normal RVF manifestation model, no specific subinterval X_{b_0} of the considered range for X is considered for block b_0. In order to classify a version $v \in V^{(\mathbb{C},\ell)}_{rvf}$, we will draw a random variate x from the uniformly distributed variable X until we have a value that lies within $[a,b]$, after which v can be classified into some block b_j corresponding to the applicable interval X_{b_j}. Note that a Bernouilli trial with $p = 0.5$ will be used to decide which block to assign a version to when the variate x equals a common interval boundary, as is the case when $c_b > 1$ and $x \in \{1,\ldots,c_b - 1\}$.

The partial consensus relation between any two versions in $V^{(\mathbb{C},\ell)}_{rvf}$ classified within the same b_j was stipulated in clause (2.1b) of $\mathscr{R}^{(\mathbb{C},\ell)}_d$. It can now be formalised

by means of response equivalence in that the corresponding random variates $x, y \in X_{b_j}$ drawn from X are guaranteed to differ by no more than ϵ, *i.e.* $d(x,y) \leqslant \epsilon$ with the distance function redefined as $d(x,y) = |x-y|$.

Ramifications of Generation Approach In the previous two sections, the notion of consensus among versions was defined in terms of an equivalence relation applied on random variates drawn from a random variable X representative of the response acquired from an invocation of these versions. This relation was formalised by means of the distance function d and an upper bound ϵ imposed as the maximum discrepancy between any two random variates $x, y \in X_{b_j}$ drawn for any pair of versions in $V_{rvf}^{(\mathcal{C},\ell)} \cup V_{nf}^{(\mathcal{C},\ell)}$ classified in the same consensus block b_j during the adjudication procedure at the end of voting round $(\mathcal{C},\ell)$. Moreover, the equivalence relation was defined on specific coverage intervals X_{b_j} such that the properties associated to the mathematical definition of equivalence hold under both definitions of the distance function. Let $x, y, z \in X_{b_j}$. Unsurprisingly, when sampling from $\mathcal{N}(\mu,\sigma^2)$ and for j $\geqslant$1, reflexivity and symmetry show from $d(x,x) = 0 < \epsilon$, respectively $d(x,y) = d(y,x) < \epsilon$. Transitivity applies as well, as can be observed from the definitions of the coverage intervals in Eq. 2.2 and Sect. 16 that $d(x,z) < d(x,y) + d(y,z)$. Note that all properties still hold for j = 0, or when sampling from $\mathcal{U}(0,c_b)$, though the strict inequalities should be replaced by their corresponding non-strict counterpart. This remark is also applicable to strict voting, for which an exact notion of equality is used, *i.e.* $d(x,y) = x - y$, and $\epsilon = 0$, and clause (2.1b) would be equivalent to $h^{(\mathcal{C},\ell)}(v_{i_1}) = h^{(\mathcal{C},\ell)}(v_{i_2})$.

Though inspired by the formalised majority voting procedure as elucidated in [7, Sect. 2.1], the construction of the partition $\mathcal{B}^{(\mathcal{C},\ell)}$ differs in that versions are classified into consensus blocks b_j deterministically, depending on the order in which random variates are drawn from the variable X. This order intuitively corresponds to the order in which responses are acquired from the corresponding invocations $\langle \mathcal{C}, \ell, i \rangle$ issued within the scope of a given voting round $(\mathcal{C},\ell)$ (increasing response times) **(A18)**. Indeed, without the classification procedures defined in the previous two sections, several eligible partitions could emerge based on the equivalence of response values in $Ran(X)$ instead of individual coverage intervals X_{b_j}, an issue



(a) Sampling from $\mathcal{U}(0,c_b)$ (b) Sampling from $\mathcal{N}(0,1)$

Figure 2.5: Probability mass function f_A indicative of the cardinality of individual consensus blocks $b_j \in \mathcal{B}^{(\mathcal{C},\ell)} \setminus b_0$.

extensively pointed out in [7]. Consequently, consensus blocks $b_j \in \mathcal{B}^{(\mathcal{C},\ell)}$ are not necessarily maximal with respect to the property that $d(x,y) \leqslant \epsilon$, for any $x, y \in Ran(X)$, as is the case in formalised majority voting; versions will be partitioned according to the discrepancies observed between their (generated) response values instead.

The implications of choosing either of the suggested approaches for generating response values relate particularly to the generalised partition $\mathcal{B}^{(\mathcal{C},\ell)} \setminus b_0$ of $V_{rvf}^{(\mathcal{C},\ell)}$ constructed during the partitioning procedure. Recall that this partition is iteratively constructed, and that an update is called for whenever a new response has been secured for versions $v \in V^{(\mathcal{C},\ell)}$ participating in round $(\mathcal{C},\ell)$. More specifically, the chosen model will affect two inherently related properties of $\mathcal{B}^{(\mathcal{C},\ell)} \setminus b_0$, *viz.* the number of non-empty consensus blocks b_j held within, and their cardinality $|b_j|$. Let us denote this latter property by the discrete random variable A. Within the scope of a single voting round $(\mathcal{C},\ell)$ affected by $c_b = |V_{rvf}^{(\mathcal{C},\ell)}|$ RVF failures, this variable effectively defines a relation $\Omega \mapsto \mathbb{N}_{[1,\ldots,c_b]}$.

A comparison of the cardinality of generated consensus blocks for classifying versions affected by RVF failures under the two alternative approaches for response value generation is shown in Fig. 2.5: it depicts the probability mass function f_A of the discrete random variable A, which has been computed for eligible values in $Ran(A)$ (shown on the vertical axis) in view of the corresponding abscissae c_b (shown on the horizontal axis). The abscissae represent how many versions affected by RVF failures should be classified, effectively delimiting $Ran(A)$ (vertical axis) as $\mathbb{N}_{[1,\ldots,c_b]}$. For each value along the horizontal axis, frequency data was obtained from 1000 sample runs, during each of which response values were generated and c_b versions were classified accordingly in terms of the equivalence relation defined hereabove. A characteristic shared by both approaches is that consensus blocks of larger cardinality are less likely to materialise, which is reflected in the corresponding confidence intervals plotted for A in Fig. 2.6. Lower, shaded series of intervals relate to $\mathcal{U}(0,c_b)$, upper series to $\mathcal{N}(0,1)$. Note how $Ran(A)$ reduces to $\{1\}$ for $c_b = 1$.

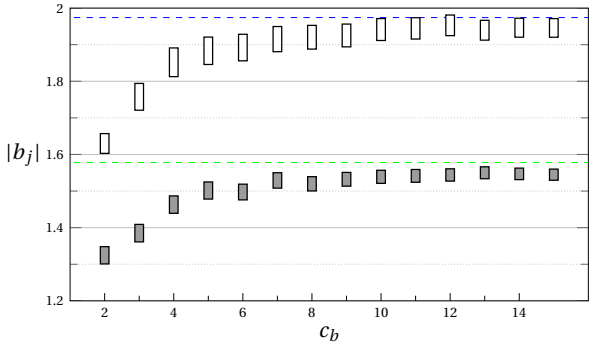


Figure 2.6: Confidence intervals computed on the observational mean of the random variates in $Ran(A)$ collected over 1000 sample runs (95% confidence level).

One can however clearly observe that sampling response values from $\mathcal{N}(0,1)$ is more likely to result in consensus blocks of larger cardinality than had $\mathcal{U}(0,c_b)$ been used. This observation was anticipated because of the increased probability of generating minor discrepancies that tend to centre around the mean μ, as stated

by (A15) and illustrated in Fig. 2.4 — *cf.* (A17). As the generation approach based on the normal distribution results in an increased cardinality of consensus blocks, it logically follows that its application will generally reflect in the construction of partitions comprising fewer blocks when compared to the alternative based on a uniform distribution. Moreover, the data plotted in Fig. 2.6 seems to suggest that, as c_b increases, the centres of the confidence intervals converge towards a common value, while the corresponding radii exhibit a gradual decline. This observation has been corroborated by additional experiments in which c_b is varied up to 1000 with results showing how averaging the obtained centres yields 1.974 for $\mathcal{N}(0,1)$ with a standard deviation as little as 0.015, or 1.578, respectively 0.013 for $\mathcal{U}(0,c_b)$. Furthermore, preliminary experimentation has shown that, in the long run, similar results with only little divergence would emerge had different parameters been chosen for the underlying sampling distribution used.

2.6.2 Erroneous Value Content Failures

The activation of a latent software design fault may also cause the normal flow of execution to be interrupted abruptly by means of an exception being thrown, a situation characterising the manifestation behaviour of what will be referred to as erroneous value failures (EVFs). Note that the exceptional behaviour caused by the occurrence of an EVF failure is transient and will disappear immediately such that it will only affect the relevant pending requests that were being serviced at the time the failure occurred **(A19)** — *cf.* (A02), (A09) and (A10). This implies that any relevant exceptions raised are *caught* and dealt with internally — inside the software unit itself, that is — so that it can gracefully recover and avoid corruption of its internal state. Falling within the content failure domain, a syntactically invalid response message that was not covered by the functional specifications will be returned by the middleware environment for the affected invocation, containing the serialised exception thrown. As exceptions are fully contained within the software unit from which they originate, they cannot propagate to the deployment environment, nor can they affect any other component deployed on top of it.

It follows from (A01) that EVF failures emerging from a design fault in version v_i can only affect a pending request $\langle \mathcal{C}, \ell, i \rangle$ whilst it is in the `RequestProcessing` state **(A20)**, as can be seen from the yellow hatched area in Fig. 2.1. The occurrence of an EVF failure affecting a pending request being processed and not subject to any other failure having previously occurred, will result in the replacement of the formerly scheduled `RequestCompletedNormally` event by a `RequestFailedExceptionally` event (see Fig. 2.2, transition `EVF (2)`). Unlike RVF failures, the occurrence of an EVF failure will result in the immediate scheduling of the replacement event, as the version will not pursue processing along the currently followed execution path. The `RequestFailedExceptionally` event will then immediately issue a failure message to be returned and transmitted over the network, resulting in transition $\mathtt{Q}_{\mathtt{r,out}}$ to be fired in the version invocation state transition model as shown in Fig. 2.1.

Let $V_{evf}^{(\mathcal{C},\ell)} \subseteq V^{(\mathcal{C},\ell)}$ be the set of versions that, during voting round $(\mathcal{C},\ell)$, were affected by an EVF failure and that were not subject to any other type of failure of higher severity. The event sequence for an invocation $\langle \mathcal{C}, \ell, i \rangle$, with $v_i \in V_{evf}^{(\mathcal{C},\ell)}$ will rejoin the normal event sequence highlighted in Fig. 2.2 by a `ResponseSent` event, and will conclude by a `RequestHandled` event reporting a syntactically invalid

result. When the voting round $(\mathcal{C},\ell)$ has transitioned into state (c), the partitioning procedure will classify all versions $v \in V_{evf}^{(\mathcal{C},\ell)}$ into $P_F^{(\mathcal{C},\ell)}$.

2.6.3 Crash Failures

A version v_i struck by a crash failure, *e.g.* because of a deadlock, infinite loop, or uncaught exceptions, has become permanently unresponsive: it will continuously exhibit omission failures for any request it was servicing at the time of failure occurrence, as well as subsequent requests **(A21)**. The service v_i is expected to provide has become permanently unavailable, thereby violating (A09) and (A10) [4]. Execution is perceived to have stopped, *i.e.* it "does not take the next step of its algorithm and has lost all its previous state" [105]. It is assumed that crash failures can only arise as the result of the activation of a latent software design fault along the current execution path followed whilst version v_i is processing a request $\langle \mathcal{C},\ell,i\rangle$ **(A22)**, *i.e.* when the request is in the RequestProcessing (2) state, in line with (A01), and illustrated by the yellow hatched area in Fig. 2.1.

As a crash failure effectively halts any processing or activities of the affected version v_i, its activation will affect the version invocation state transition model in two possible ways. Firstly, any pending request $\langle \mathcal{C},\ell,i\rangle$ being serviced at the time a crash failure affected v_i, will never transition to state (3). The processing capacity that had previously been allocated for servicing these requests will therefore never be relinquished. Secondly, all other requests destined for execution by v_i that did not already transition to state (2) will never transition to the RequestProcessing (2) state. Note that crash failures were explicitly defined to originate from a design fault in the software implementation of a version; the activation of such failures is assumed to affect only the availability of the version, and does not affect the behaviour of the deployment environment **(A23)**. Consequentially, any request $\langle \mathcal{C},\ell,i\rangle$ arriving at the host at which v_i was deployed will be accepted and queued, but will become stuck in the RequestReceived (1) state, awaiting processing indefinitely. Requests that already transitioned into state (3) prior to the emergence of the crash failure will pursue along the normal transition path and have their response sent out for delivery (transition $\mathtt{Q}_{\mathtt{r,out}}$).

For requests susceptible to a crash failure having occurred, the associated omission failure will manifest by means of a performance failure **(A24)**, as described in Sect. 2.6.4. As can be seen from the discrete event model in Fig. 2.2, for each request being serviced, when not subject to any other failure having previously occurred, the occurrence of a crash failure will result in the replacement of the previously scheduled RequestCompletedNormally event by a RequestFailedComponentCrash event, scheduled at time $+\infty$. Stalled requests $\langle \mathcal{C},\ell,i\rangle$ awaiting processing will result in the RequestProcessing event to be replaced similarly.

2.6.4 Performance Failures

As can be seen in Fig. 2.1, not all requests $\langle \mathcal{C},\ell,i\rangle$ take an equal amount of time to return their response. Let t_{max} represent the largest permissible response time that an NVP composite can afford for any request $\langle \mathcal{C},\ell,i\rangle$ to complete. This parameter is of particular interest as it is used to detect late timing failures, often referred to as performance failures — any version $v_i \in V^{(\mathcal{C},\ell)}$ failing to produce and have its

response returned within the time constraints imposed would obviously translate into such type of failure. It is assumed that a performance failure manifests only when a timely response for a request is not available **(A25)** by the time the decision algorithm is expected to adjudicate the outcome for the current voting round $(\mathcal{C}, \ell)$, as can be seen from the hatched area below state `(c)` in Fig. 2.1. Note that a performance failure does not directly affect the progress of the request itself. Moreover, its emergence is not necessarily due to the activation of a software design fault affecting the version servicing the request, as it can materialise from the interplay of several endogenous and exogenous conditions as well **(A26)** [106]. For instance, the complexity of an implementation (version) — an endogenous condition — may exceed the computational capacity of the underlying hardware platform — an exogenous condition — such that the application's timing characteristics as described in the specifications can hardly ever be met. Other examples of exogenous conditions include, *e.g.* network transmission delays, and queuing delays or overhead originating from the deployment platform.

As such, two types of performance failure can be discerned. Firstly, a performance failure can effectively detect omission failures when the request was affected by a crash failure — *cf.* Sect. 2.6.3. Secondly, any other type of request failing to deliver a response within the specified time constraints, and that was not affected by a crash failure in the meantime, is said to suffer from a late response failure (LRF). Such type of failure usually occurs due to inadequate capacity planning, where the system — version, in this case — would be observed to perform subnormally, since the assumptions on the expected load prove to be unrealistic, or not in line with current demand. In describing the state transition path for a request, a clear distinction needs to be made between these two types of performance failures.

As for omission failures due to a crash failure affecting v_i, Sect. 2.6.3 already enlarged on the possible paths followed by a request $\langle \mathcal{C}, \ell, i \rangle$ in the state transition diagram shown in Fig. 2.1. Because the version has become permanently unresponsive, an event of type `RequestFailedComponentCrash` had already been scheduled at time $+\infty$ — *cf.* (A09) and (A10). If request $\langle \mathcal{C}, \ell, i \rangle$ failed to return its response to the NVP composite before the t_{max} timeout has lapsed since its initialisation, the `PerformanceFailure` event will be processed and replaced by an event of type `FailureReceived` to be instantaneously processed. Finally, the replacement event will cancel the previously scheduled `RequestFailedComponentCrash` event and hand over the response (failure message) for voting, after which the response acquisition procedure has been completed and the request has been completely handled. Note how a `RequestHandled` event emerges from a `FailureReceived` event being successively replaced, complemented by a preemptive replacement of a `RequestFailedComponentCrash` event, as it is defined by the join convergence element contained within path `(c)`.

When a performance failure emerges due to a request $\langle \mathcal{C}, \ell, i \rangle$ suffering from an LRF failure, a correct response may eventually be delivered but will be discarded, for the request will continue in the background, following the normal path in the version invocation state transition model, during which failures may still affect its course. Such type of failure occurs in isolation of other requests in the system, adhering to (A02), (A09) and (A10). Note that the request may already have been affected by content failures by the time its response was due. After the request has timed out, *i.e.* no event of type `ResponseReceived` had occurred in the request's preliminary event

sequence during a t_{max} time span since its initialisation, the `PerformanceFailure` event will be processed and a `FailureReceived` event will subsequently be immediately scheduled, signaling the lack of a timely response. Finally, as it can be observed from the convergence element held within path `(b)`, when both `FailureReceived` and `ResponseReceived` events are available for successive, respectively preemptive replacement, the final `RequestHandled` replacement event will be scheduled.

Note that when a request is initialised, a specific `PerformanceFailure` event will automatically be scheduled for processing when a period equal to t_{max} has lapsed, apart from a `RequestInitialised` event. The replacement procedure formalised by the convergence elements in paths `(b)` and `(c)` shows how the `RequestHandled` event emerges as the successor for the `FailureReceived` event, whereas the other input events, *i.e.* `RequestFailedComponentCrash`, respectively `ResponseReceived`, are preemptively replaced and therefore discarded. Having detected a performance failure for invocation $\langle \mathcal{C}, \ell, i \rangle$, the version v_i will be classified in $P_F^{(\mathcal{C},\ell)}$, for the `RequestHandled` event signaled a performance failure as the result for $\langle \mathcal{C}, \ell, i \rangle$.

2.6.4.1 Injecting Late Response Failures

When running simulations, one may occasionally wish to inject performance failures into the system and investigate to what extent they may degrade its performance. As will be explained in Sect. 6.4, our simulation environment has been designed to support multiple failure injection mechanisms. What all of these mechanisms have in common is that they allow the user to specify predefined or custom-made failure activation events which are to be scheduled automatically so as to affect the desirable replicas and/or voting rounds. Additional information that can be specified include, *e.g.*, the conditionalities that should hold before scheduling this type of events, and the resulting manifestative behaviour — failure class, that is. Whereas omission failures would spontaneously emerge after injecting a crash failure affecting the desirable replica, the injection of LRF failures is somewhat more complex. In what follows, a two-phase procedure is suggested for injecting an LRF failure within the scope of a newly initiated version invocation.

	current state	scheduled state	stance	variate
(1)	`RequestInitialised`	`RequestSent`	client overhead	0
(2)	`RequestSent`	`RequestReceived`	RTT (incoming)	1
(3)	`RequestReceived`	`RequestProcessing`	queuing overhead	$\perp$
(4)	`RequestProcessing`	`RequestCompleted` `(Ab)Normally`	RPT (overall)	2, 3
(5)	`RequestCompleted` `(Ab)Normally / Exceptionally`	`ResponseSent`	replica overhead	5
(6)	`ResponseSent`	`ResponseReceived`	RTT (outgoing)	4

Table 2.1: State transitions in the version invocation discrete event model during which an activation of an LRF failure may occur.

Any failures to be injected so as to affect a given request $\langle \mathcal{C}, \ell, i \rangle$ should only be activated at a relevant stage during its execution, depending on the susceptibility of its current state to the corresponding types of disturbances. As this applies to

LRF failures as well, the first step of the suggested procedure involves designating which transition in the version invocation discrete event model is to be affected, *i.e.* which transition should be postponed so as to trigger a performance failure — *cf.* Fig. 2.2. Table 2.1 lists the state transitions that could potentially trigger an LRF failure: the current, actual state is shown in the second column; the next state, which corresponds to the currently scheduled event in our discrete event model, is shown in the third column. If, at any time during a request's execution, the currently pending transition is one of the listed eligible transitions, and if it fails to fire before the t_{max} timeout has lapsed, an LRF failure would emerge, which would be caught by means of a performance failure[6]. The last column in the table shows the random variates that correspond to specific state transitions where the LRF should be injected — with variates sampled from a random variable distributed according to the suggested binomial distribution $\mathscr{B}(n,p)$, as per Fig. 2.7.

For the injection of LRF failures, one should not consider extending transition `(3)` from `RequestReceived` to `RequestProcessing` though; this transition should remain under the control of the queuing mechanism that is utilised by the applicable replica v_i for managing and admitting requests for processing. Tampering with the duration of this transition would almost certainly result in intervals during which the system may be in an inconsistent state — *e.g.* an empty waiting queue during (part of) the additional delay.

The total simulated execution time of a request is the result of several constituents, each of which quantifies the time required to deal with the request from a specific perspective. Each of these consituents corresponds to the time it takes for a state transition listed in Table 2.1 to fire[7]. Obviously, a significant part of the overall execution time is the request's servicing time, which is composed of the request processing time (RPT) and the time the request is kept waiting until processing capacity becomes available — transitions `(4)` and `(3)`, respectively. Another significant part would be the time required for transferring the request and response messages to/from the targeted replica v_i, *i.e.* the round-trip time (RTT): transitions `(2)` and `(6)` from an incoming, respectively outgoing perspective with respect to v_i. The remaining transitions `(1)` and `(6)` account for the internal overhead of the outgoing network transmission queue at the client and replica and the minor delays that may ensue thereof. Due to the way in which the manifestative behaviour of EVF failures was defined in Sect. 2.6.2, LRF failures cannot emerge during the RPT of a request that has already been affected by an EVF failure. Indeed, an EVF failure occurrence essentially cancels out events of type `RequestCompleted(Ab)Normally`, and the replacement event of type `RequestCompletedExceptionally` that is scheduled for immediate execution will in itself be replaced by its successor event `ResponseSent` in due course — *cf.* transition `(5)`.

[6] Recall that in a discrete event simulation, a state transition $p \rightarrow q$ between the states p and q is identified by scheduling only an event corresponding to the next state q; the current state p is not explicitly represented. The transition is said to fire when this event is actually processed, which corresponds to the notion of successive event replacement introduced in Sect. 2.3 on p. 43. When emphasis is placed on the time prior to the firing of a transition, the notion of state transitions shall be used; otherwise, event q will be referred to as the currently scheduled event, and a reference will be made to the applicable event replacement procedure.

[7] All other transitions shown in the normal event sequence of the version invocation discrete event model in Fig. 2.2 were defined to occur instantaneously, without introducing any further delay.

Let us consider a random variable $Y : \Omega \mapsto \mathbb{N}_0$ that follows the binomial distribution $\mathscr{B}(n,p)$ with parameters $n = 5$ and $p = 0.5$. The probability density function of such distribution is plotted in Fig. 2.7 (units along vertical axes equal $1/32$). In order to determine which transition to prolong so as to inject an LRF failure for a given request, a random variate $y \in [0,n]$ is drawn from Y. This sample is drawn once for each request in the initial state — *cf.* Fig. 2.2. The value obtained from y identifies the state transition that should be postponed; this mapping is shown in column 5 in Table 2.1.

The rationale for sampling from this distribution is that the impact of the RPT on the execution time of requests is assumed to outweigh the impact of the RTT, which is usually true in case these requests are issued to replicas hosted within a local area network (LAN) — located within the same administrative network unit, that is. Furthermore, the significance of the RTT will account for invocations of versions hosted across a wide area network (WAN), in which network transmission delays are expected to exhibit greater magnitude and variability.

Having selected the state transition to prolong, the second step in the LRF failure injection procedure is to prolong the time span before this transition is fired. Immediately after each subsequent event replacement during the simulation of the request's execution, the system will check if the scheduled replacement event is eligible for LRF failure injection, *i.e.*, it corresponds to one of the state transitions listed in Table 2.1. Rescheduling the currently scheduled event so as to occur at a later point in time will then lead to a prolongement of the current state transition. Let $t_{init} \in \mathbb{R}$ represent a time stamp in simulation time at which the request has been initialised, and q a real number such that $q > 0$. It is easy to see how an LRF failure will spontaneously emerge by rescheduling the currently scheduled event to occur after an additional delay of $(1+q) \cdot t_{max} - t_{init}$ time units, and this with respect to the time it had initially been scheduled to occur.

Suitable values for q could be sampled, *e.g.*, from a random variable Q that is distributed according to a gamma distribution $\Gamma(k,\theta)$, with shape parameter $k > 1$. As this distribution is positively skewed and bounded at 0, the right tail of its probability density function is considerably longer than the left tail, and thus

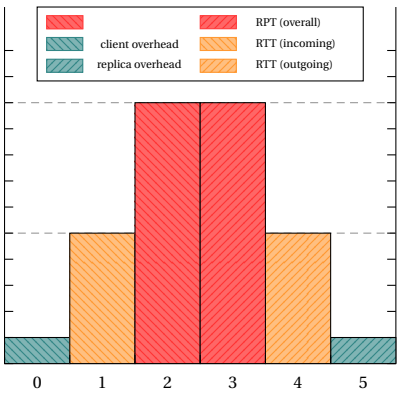


Figure 2.7: Probability mass function of $\mathscr{B}(n,p)$, with $n = 5$, and $p = 0.5$, used to determine which state transition to defer when injecting LRF failure.

random variates obtained for q are more likely to be of moderate size [107, p. 88–89]. This is a desirable property, since we are not simulating omission failures in which some request (or response message) is lost, and such values ensure the t_{max} timeout is not exceeded to an unrealistic extent.

As a final remark, it should be pointed out that the injection of LRF failures affect only the scheduling of events appertaining to the execution of requests; failure activation events remain untouched and can still affect the further course of the request's execution.

2.6.5 Failure Class Severity Hierarchy

Throughout the previous sections, we elaborated on the conditions that could lead to the manifestation of a particular failure class, and the ensuing repercussions on the transition path in the version invocation state transition and discrete event models (Fig. 2.1, resp. 2.2). So far, the impact of a single occurrence of some failure class was described, in complete isolation of any other occurrence of (another) failure class. During the course of its lifetime, a request $\langle \mathcal{C}, \ell, i \rangle$ may, however, be affected by multiple failures **(A27)**; it is the combination of failure class manifestations, and especially the sequence in which they occur and affect a request, that will completely determine the request transition path followed, and whether or not a timely or valid response will be available at the time of voting.

It is noteworthy to point out that, in the context of (the simulation of) a single request $\langle \mathcal{C}, \ell, i \rangle$, multiple (scheduled) occurrences of a single failure class affecting the request in question, possibly activated by different software design faults, are assumed to be idempotent: only the first occurrence will result in a state transition; subsequent occurrences will be withheld[8] **(A28)**. Idempotence does not apply to performance failures though, for at most one such failure can occur for a given request.

Furthermore, as was already pointed out in Sect. 2.6.4, a request affected by an LRF (performance) failure will continue its execution in the background, during which it may be affected by other failure classes. In case its further course was not subject to failures other than content failures, a response will eventually be delivered, though it will be discarded as the voting procedure had already been initialised, immediately after the manifestation of the performance failure[9]. A crash failure affecting the request will result in a silent, unattended omission failure.

We will now elaborate on all possible sequences of failure occurrences of different failure classes and how these may affect a request, by means of the discrete event model in Fig. 2.2. In doing so, an intuitive notion of severity will be associated to the different failure classes introduced in the previous sections.

[8] Note that these failure occurrences are withheld only with respect to the request under consideration, and may affect other pending requests, depending on their current state.

[9] At the time the performance failure for the request under consideration occurred, *i.e.* after a time span of t_{max} since the state `(b)` was reached in the transition path of voting round $(\mathcal{C}, \ell)$, which is represented by the area hatched in orange below state `(c)` in Fig. 2.1, the NVP composite is expected to retrieve a response for each of the requests previously spawned. Late response or omission failures will be assumed and caught by performance failures for those $v_i \in V^{(\mathcal{C},\ell)}$ for which no response was previously acquired. The decision algorithm will then be called to adjudicate the outcome for $(\mathcal{C}, \ell)$ based on this preliminary digest of responses.

Let $\langle \mathcal{C}, \ell, i \rangle$ be a pending request still being serviced and already affected by an RVF failure, *i.e.* its current scheduled event is `RequestCompletedAbnormally`. The occurrence of an EVF failure will overrule the previous RVF failure, resulting in the transition `RVF` $\rightsquigarrow$ `EVF (3)` being fired, such that the current event will be superseded by a `RequestCompletedExceptionally` event, as described in Sect. 2.6.2. Alternatively, a crash failure affecting v_i would result in the replacement of the current `RequestCompletedAbnormally` event by a `RequestFailedComponentCrashed` event, the latter which will be scheduled at time $+\infty$, similar to the procedure outlined in Sect. 2.6.3. Regardless of the failure classes having affected $\langle \mathcal{C}, \ell, i \rangle$, omission and LRF failures will always be caught by performance failures.

In case a pending request $\langle \mathcal{C}, \ell, i \rangle$ still being serviced is hit by an EVF failure, *i.e.* its currently scheduled event is of type `RequestCompletedExceptionally`, it cannot be affected by any other failure originating from a software design fault affecting v_i — *cf.* assumptions (A12), (A20) and (A22). This can be motivated by the fact that after EVF failure occurrence, the request is no longer being serviced; a fault message has been handed over to the deployment environment, queued and awaiting transmission. At this stage, only LRF failures caught by a performance failure may supersede the EVF failure when the response would not arrive in a timely manner.

Obviously, because a crash failure hitting a version v_i will persist in affecting any new and pending request $\langle \mathcal{C}, \ell, i \rangle$, content failures are not expected to emerge, as the crash caused all processing by v_i to be halted, and therefore design faults that can only be activated from particular execution flows cannot be triggered anymore — *cf.* assumptions (A21), (A12) and (A20). Such requests will eventually exhibit an omission failure, caught by a performance failure, a procedure described in Sect. 2.6.3 and 2.6.4.

In conclusion, there exists a total order between failure classes by considering the severity of their manifestation affecting the response for a request $\langle \mathcal{C}, \ell, i \rangle$ at the revelant stages in its lifecycle, which is shown in Fig. 2.3 as the order: first performance failures, then EVF and finally RVF failures.

2.6.6 Software Failure Activation

As it can be seen from the dashed arcs in the version invocation discrete event model presented in Fig. 2.2, the manifestation of software failures during the relevant life span of an invocation, *i.e.* the yellow hatched area in Fig. 2.1 in which the request is in the `RequestProcessing (3)` state, will result in one or more previously scheduled events to be preemptively replaced. For this type of event replacement procedure to be initiated, additional external failure activation events are required — *cf.* the replacement procedures defined in Sect. 2.3. As such, we define a set of dedicated failure activation events whose sole purpose is to model the occurrence of some software failure type, and to enact its manifestative behaviour on the invocation event model. Precisely how this type of events should be scheduled will be discussed in the scope of Sect. 6.4, which will elaborate on advanced techniques for failure injection. Such activation events are defined for each failure class included in the total order introduced in Sect. 2.6.5, *i.e.* the `ResponseValueFailure`, `ErroneousValueFailure` and `CrashFailure` event types. Upon occurrence, such activation event may immediately initiate a preemptive replacement procedure,

adhering to (A08). The activation event `PerformanceFailure` does not abide this approach though, for its successor `FailureReceived` event will be scheduled during the processing ensuing from its occurrence, as it can be seen from Fig. 2.2.

Failure activation events will only initiate a preemptive replacement procedure if the invocation is susceptible to that type of failure class at the time the failure emerges — *cf.* (A08). We therefore introduce a flexible enactment mechanism, in which the predefined events from the model in Fig. 2.2 may be composed of several marker interfaces, and that is compliant to the failure class severity hierarchy stated in the previous section [108, pp. 179–180]. Three such marker interfaces are defined for each of the software failure classes, *viz.* `AffectedByRVF`, `AffectedByEVF` and `AffectedByCrashFailure`. A preemptive replacement procedure will be initiated for any event in the set of all scheduled, unprocessed events for an invocation matching the marker interface corresponsing to the failure activation event currently being processed. More specifically, an event of type `RequestCompletedNormally` is composed of all three marker interfaces; the `AffectedByEVF` interface is applicable for events of type `RequestCompletedAbnormally`[10].

Additionally, the `AffectedByLateResponseFailure` marker interface has been defined for the purpose of injecting LRF failures. Note how this marker interface is used not to pre-emptively replace the currently scheduled event by an event of another type, but to reschedule it to occur at a later point in time, in order to trigger a performance failure. More information on this procedure and the further flow of execution can be found in Sect. 2.6.4.

The annotated UML class diagram in Fig. 2.8 illustrates the interplay between failure activation events and scheduled events for pending invocations, and highlights the underlying role of a sound inheritance tree of marker interfaces to enforce the failure severity hierarchy. Marker interfaces prefixed by `AffectedBy` are used to drive the preemptive event replacement procedure described in Sect. 2.3. Events eligible for replacement must extend the appropriate interfaces so as to enforce the failure severity hierarchy introduced in Sect. 2.6.5. Observe how the scheduled states of the state transitions susceptible to the injection of LRF failures, as listed in Table 2.1, adopt the `AffectedByLateResponseFailure` marker interface.

The version invocation discrete event model was explicitly designed for extensibility, in that additional failure classes can be defined and included in the failure severity hierarchy. As such, failures emerging from disturbances affecting the service that other system components intend to provide can be modelled and simulated as well. This includes, *e.g.*, the network datagram service, the underlying hardware, or the middleware deployment environment — *cf.* (A04) and (A06). It is sufficient to add specific marker interfaces for new failure classes, inserting these interfaces at a suitable position in the inheritance tree shown in Fig. 2.8, and decorating the susceptible events in the discrete event model accordingly. In doing so, one should make sure to correctly decorate events pertaining to request invocations so as to preclude violations of assumption (A28). Additional information regarding the proposed design, its implementation and its purpose with respect to a simulation

[10] From Fig. 2.2, the `RequestProcessing` event type can also be observed as eligible for preemptive replacement due to a crash failure having previously occurred, *i.e.* it exposes the `AffectedByCrashFailure` interface **(D03)**. This design decision is implementation-specific, as a `CrashFailure` activation event will be automatically injected so as to enforce (A21).

platform designed to assess the effectiveness of (NVP-based) redundancy schemata can be found in Chapt. 6.

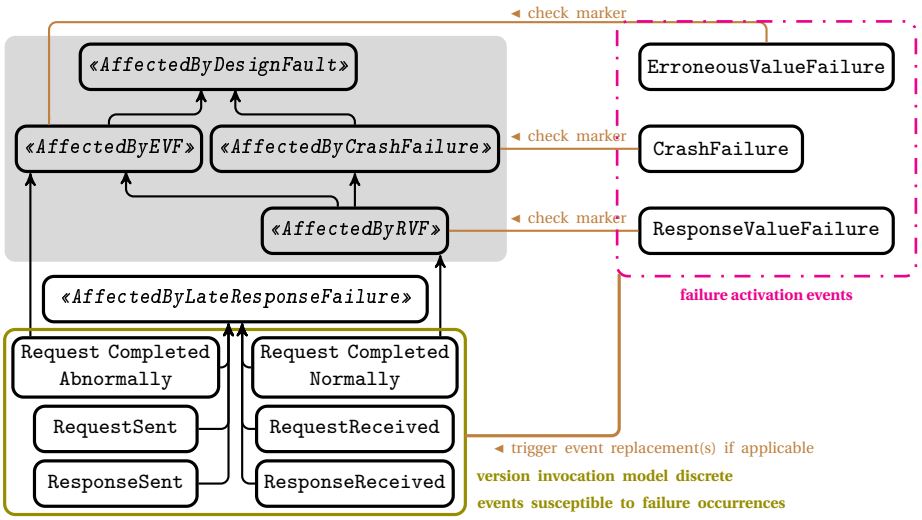


Figure 2.8: Preemptive event replacement and the inheritance tree of marker interfaces are effectively used to realise the defined failure severity hierarchy.

2.7 Counter Update Discrete Event Model

The counter update discrete event model, shown as the event sequence below the normal invocation event sequence in Fig. 2.2, encompasses a series of predefined events that enable to model the impact of a version invocation $\langle \mathcal{C}, \ell, i \rangle$ on the load statistics maintained for v_i by the deployment environment (middleware) of the host on which this version is deployed. Its design was inspired by the `NumberOfRequests`, `NumberOfFailedRequests` and `NumberOfSuccessfulRequests` counter metrics defined in the operation manageability capability of the Management of Web services (MOWS) specification [109][11]. Furthermore, the model describes how changes in these statistics are conveyed to the NVP composite — *v.* Sect. 8.3.

When a new counter update is initiated from the lower branch of the parallel gateway divergence elements following the replacement of either a `RequestReceived` or `ResponseSent` event in the course of an invocation $\langle \mathcal{C}, \ell, i \rangle$, a new `CounterUpdate-Issued` event will be scheduled to be instantaneously processed, and configured appropriately so as to relay relevant load-related information pertaining to v_i. A `CounterUpdateIssued` event ensuing from the replacement of a `RequestReceived` event serves the purpose of incrementing the `NumberOfRequests` counter, reflecting the arrival of a new request at the deployment environment hosting v_i, which took place when the invocation transitioned into state `(1)` in Fig. 2.1. Conversely, when

[11] Throughout this dissertation, unless explicitly stated otherwise, it was assumed that a version v_i exposes only a single operation, such that one set of the aforementioned counters is maintained for v_i **(A29)**. Refer to Chapt. 8 for more information.

the invocation has been handled by v_i and a response message is being returned over the network, *i.e.* state (4), the CounterUpdateIssued event emerging from the replacement of the ResponseSent event will increment either of the counters NumberOfSuccessfulRequests or NumberOfFailedRequests, depending on the event sequence(s) followed and whether the invocation had been affected by a content failure during its processing. More specifically, the syntactically invalid fault message returned for invocations that were not were subject to any other failure class of higher severity than an EVF failure, will be reflected in the value of the NumberOfFailed-Requests counter. Alternatively, the CounterUpdateIssued event will update the NumberOfSuccessfulRequests counter if a syntactically valid response value is returned, such as it is the case for invocations unaffected by any type of disturbance, or those that were only affected by RVF failures. In order to enforce consistency, each counter update will convey the values of all three counters.

The occurrence of crash failures may inhibit the emergence of new CounterUpdateIssued events. Such events will continue to be issued by the replacement procedure of the RequestReceived event for new requests $\langle \mathcal{C}, \ell, i \rangle$ arriving at the host at which v_i was deployed, even though v_i had been halted due to a crash failure. The replacement RequestProcessing event will be preemptively replaced though, for no actual processing will occur, as the invocation has become stuck in the RequestReceived (1) state depicted in Fig. 2.1. Consequently, the replacement procedure of the ResponseSent event is disabled and a final counter update therefore cannot be issued — which is also relevant for invocations $\langle \mathcal{C}, \ell, i \rangle$ that were being serviced at the time the crash failure affecting v_i was activated. Note that the counter update discrete event model itself is not susceptible to failures, *cf.* (A04), property 3 and (A06).

When an issued CounterUpdateIssued event is processed, it will automatically schedule its CounterUpdateSent successor replacement event to occur after some delay comparable to $\mathtt{Q}_{\mathtt{r,out}}$, during which the counter update is enqueued, awaiting network transmission. Finally, the arrival of the counter update message at the NVP composite is signaled by the occurrence of the CounterUpdateReceived event, which was scheduled to replace its predecessor CounterUpdateSent event after a simulation time equal to the network transmission delay.

Any version v_i used for the processing of an invocation $\langle \mathcal{C}, \ell, i \rangle$ on behalf of the NVP composite cannot reasonably be assumed to be a dedicated resource, for v_i may also be used by a third party, resulting in additional external load. Such additional invocations $\langle \perp, \perp, i \rangle$ are modelled so as to follow the same discrete event model, including the issuance of counter updates at the proper stages during its progression. Their execution is performed independently of any pending voting round $(\mathcal{C}, \ell)$, *i.e.* the results held within their scheduled RequestHandled event are simply discarded. They may, however, lead to the occurrence of a crash failure affecting v_i which may have repercussions on invocations $\langle \mathcal{C}, \ell, i \rangle$ for which the result should still be acquired, *i.e.* a ResponseSent event had not been scheduled by the time the crash failure was activated — *cf.* (A22).

Note how in distributed computing systems the dissemination of counter updates would usually take place by means of asynchronous publish-and-subscribe messaging models [109]. The proposed counter update discrete event model in Fig. 2.2 was designed so as to mimic the potential timing overhead of such messaging models, though considers only a single subscriber — the NVP composite — to be notified of counter updates **(D04)**.

CHAPTER 3

Capturing the Effectiveness of Redundancy Configurations

In this chapter, we argue that the effectiveness of a redundancy scheme is largely determined by the redundancy configuration used within, and its ability to counterbalance the disturbances ensuing from the environment in which it operates and to which it is subject. A set of ancillary metrics is introduced, which allow to capture contextual information by assessing the effectiveness of any given redundancy configuration at the end of a specific voting round, and which enable the assessment of the proximity of hazardous situations that may require its adjustment. The contents of this chapter have been disseminated to the public through the publications [83] and [84]. Related research question(s): RQ-1 and RQ-2.

Redundancy-based fault-tolerant strategies have long been used as a means to avoid a disruption in the service provided by the system in spite of the occurrence of failures in the underlying components. Adopting these fault-tolerance strategies in highly dynamic distributed computing systems, in which components often suffer from long response times or temporary unavailability due to the manifestation of disturbances, does not necessarily result in the anticipated improvement in dependability.

The effectiveness of a fault-tolerant redundancy scheme such as NVP is largely determined by the redundancy configuration used within, *i.e.* the amount of redundancy used and, accordingly, a selection of functionally-equivalent software components, and its ability to counterbalance the disturbances ensuing from the environment in which it operates and to which it is subject.

Statically predefined redundancy configurations encompassing a fixed amount of redundancy and an immutable set of versions have traditionally been employed within many classic dependability strategies. Such redundancy configurations are, however, context-agnostic, *i.e.* they do not take account of changes in the operational status of any of the components contained within the redundancy scheme, and may prove to be inadequate to maintain the effectiveness of the fault-tolerant unit from the following three angles.

3.1 From a Dependability Perspective

"Whether or not the availability [of the scheme] is improved depends on the amount of redundancy employed and the availability of the software components used to construct the system" [3, 110]. The amount of redundancy, in conjunction with the voting algorithm, controls how many simultaneously failing versions the NVP composite can tolerate whilst continuing to provide the user with the expected service. For instance, an NVP scheme, *e.g.* one based on majority voting, can mask disturbances affecting up to a minority of its versions — a function of the amount of redundancy indeed. Furthermore, the dependability of any NVP composite is determined by the dependability of the versions employed within. As elucidated in [3, Sect. 4.3.3], the use of replicas of poor reliability can result in a system tolerant of faults but with poor reliability. Likewise, versions exhibiting low availability may in time lead to a more rapid exhaustion of the available redundancy, or result in a failure of the scheme when the amount of redundancy becomes insufficient to mask the ensuing failures. It is therefore of paramount importance to construct fault-tolerant systems using highly dependable software components, and to avoid the use of resources that do not significantly contribute to an increase of dependability.

The minimal amount of redundancy the NVP composite would require during the operational time interval for a voting round $(\mathcal{C},\ell)$ in order to stay dependable and have the decision algorithm return the correct result in spite of being challenged by a given number of disturbances $e^{(\mathcal{C},\ell)}$ is expressed by its contextual redundancy, by means of the function $cr:\mathbb{N}_0\mapsto\mathbb{N}^+$ such that $cr(e^{(\mathcal{C},\ell)})=2\cdot e^{(\mathcal{C},\ell)}+1$ [106]. An example of a reactive redundancy adjustment procedure is shown in Fig. 3.1: red regions represent periods during which composite $\mathcal{C}$ fails because of an insufficient amount of redundancy; green regions indicate voting rounds $(\mathcal{C},\ell)$ for which disturbances are successfully counterbalanced — *i.e.* $n^{(\mathcal{C},\ell)}\geq cr(e^{(\mathcal{C},\ell)})$ — *cf.* (A30).

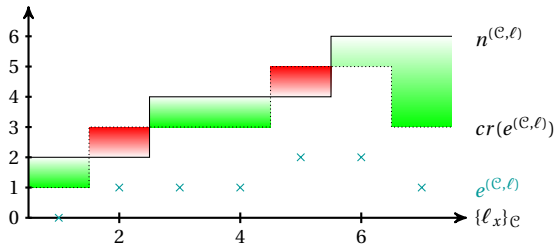


Figure 3.1: Example of a reactive redundancy adjustment procedure. Redundancy undershooting regions marked in red; overshooting regions in green.

Let $P^{(\mathcal{C},\ell)}$ be the set of largest cardinality in the generated partition $\wp^{(\mathcal{C},\ell)}\setminus P_F^{(\mathcal{C},\ell)}$, or $\emptyset$ if no such set exists, *i.e.* when this partition is the empty family and therefore $P_F^{(\mathcal{C},\ell)}=V^{(\mathcal{C},\ell)}$. This generalised partition $\wp^{(\mathcal{C},\ell)}$ materialises at the end of voting round $(\mathcal{C},\ell)$, *i.e.* when a result has been acquired for each $v\in V^{(\mathcal{C},\ell)}$ — *cf.* state `(c)` in Fig. 2.1. Then $c_{max}^{(\mathcal{C},\ell)}=|P^{(\mathcal{C},\ell)}|$ represents the largest consent found between the versions in $V^{(\mathcal{C},\ell)}$ within the scope of $(\mathcal{C},\ell)$. In order for the majority voting procedure to be able to adjudicate a result o of the scheme, there should be a consensus amongst an absolute majority of the $n^{(\mathcal{C},\ell)}$ versions, *i.e.* $c_{max}^{(\mathcal{C},\ell)}\geqslant m^{(\mathcal{C},\ell)}$,

with $m^{(\mathcal{C},\ell)}$ defined as

$$m^{(\mathcal{C},\ell)} = \left\lceil \frac{n^{(\mathcal{C},\ell)}+1}{2} \right\rceil \tag{3.1}$$

Put differently, $m^{(\mathcal{C},\ell)}$ is indicative of the smallest degree of consent needed for a consensus block $P^{(\mathcal{C},\ell)}$ to qualify for the equivalence class $[o]$ — *cf.* (A13) and (A14). Conversely, if $c_{max}^{(\mathcal{C},\ell)} < m^{(\mathcal{C},\ell)}$, the voting procedure will not be able to determine a result.

It is worth recalling that the exact result is assumed to be properly returned for versions unaffected by any type of disturbance — *cf.* (A03). Even though $P^{(\mathcal{C},\ell)}$ is uniquely identified by a single consensus block in $\wp^{(\mathcal{C},\ell)} \setminus P_F^{(\mathcal{C},\ell)}$ if $c_{max}^{(\mathcal{C},\ell)} \geqslant m^{(\mathcal{C},\ell)}$, it does not necessarily correspond to the consensus block $b_0 \in \mathcal{B}^{(\mathcal{C},\ell)}$ constituted by those versions in $V_{nf}^{(\mathcal{C},\ell)}$ that were found to be equivalent with respect to this exact value, if such block exists. Indeed, the decision algorithm will only be able to guarantee the correct result o if $n^{(\mathcal{C},\ell)} - e^{(\mathcal{C},\ell)} = |V_{nf}^{(\mathcal{C},\ell)}| \geqslant m^{(\mathcal{C},\ell)}$, *i.e.* when $n^{(\mathcal{C},\ell)} \geqslant cr(e^{(\mathcal{C},\ell)})$ and therefore $[o] = P^{(\mathcal{C},\ell)} = b_0$ **(A30)**. Another situation may arise when a unique consensus block $P^{(\mathcal{C},\ell)} \neq b_0$ exists, such that $e^{(\mathcal{C},\ell)} \geqslant |V_{rvf}^{(\mathcal{C},\ell)}| \geqslant m^{(\mathcal{C},\ell)}$, in which case the voting procedure shall not be able to adjudicate the correct result and an incorrect value will be returned, resulting in the (temporary) unavailability of the scheme. Should there exist multiple eligible consensus blocks with a common maximum cardinality less than $m^{(\mathcal{C},\ell)}$, one of these blocks may be selected nondeterministically and assigned as $P^{(\mathcal{C},\ell)}$. Note how $m^{(\mathcal{C},\ell)} - 1$ represents the largest possible degree of consent within a consensus block that constitutes a plurality, though not an absolute majority. Consequently, a scheme is resilient to withstand disturbances affecting at most a minority $n^{(\mathcal{C},\ell)} - m^{(\mathcal{C},\ell)}$ of the $n^{(\mathcal{C},\ell)}$ versions used throughout a voting round $(\mathcal{C},\ell)$.

3.1.1 Hazard Proximity Identification

The *dtof* metric, initially announced in [78], was meant to provide a measure of the proximity of hazardous situations that may necessitate the adjustment of the currently employed redundancy configuration so as to ensure the availability of the composite's service. For a specific voting round $(\mathcal{C},\ell)$, function $dtof : \wp^{(\mathcal{C},\ell)} \mapsto \mathbb{N}_0$ is defined as

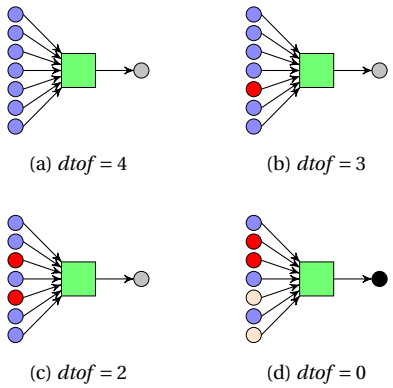
$$dtof^{(\mathcal{C},\ell)} = \begin{cases} 0 & c_{max}^{(\mathcal{C},\ell)} < m^{(\mathcal{C},\ell)} & \text{(3.2a)} \\ m^{(\mathcal{C},\ell)} - d^{(\mathcal{C},\ell)} & c_{max}^{(\mathcal{C},\ell)} \geqslant m^{(\mathcal{C},\ell)} \wedge n^{(\mathcal{C},\ell)} \text{ odd} & \text{(3.2b)} \\ m^{(\mathcal{C},\ell)} - d^{(\mathcal{C},\ell)} - 1 & c_{max}^{(\mathcal{C},\ell)} \geqslant m^{(\mathcal{C},\ell)} \wedge n^{(\mathcal{C},\ell)} \text{ even} & \text{(3.2c)} \end{cases}$$

where $d^{(\mathcal{C},\ell)}$ in Eq. (3.2b) and (3.2c) represents $n^{(\mathcal{C},\ell)} - c_{max}^{(\mathcal{C},\ell)}$, *i.e.* the number of versions that are either faulty or that returned a vote that differs from the majority, if any such majority exists[1]. As can be easily seen, *dtof* returns an integer in $[0, m^{(\mathcal{C},\ell)}]$ for any odd $n^{(\mathcal{C},\ell)}$ or in $[0, m^{(\mathcal{C},\ell)} - 1]$ for any even $n^{(\mathcal{C},\ell)}$. This integer represents how close we were to failure at the end of voting round $(\mathcal{C},\ell)$.

[1] For the sake of brevity, we say that the faulty versions in $P_F^{(\mathcal{C},\ell)}$ are in dissent with the responses returned by any of the versions classified within $P_1 \ldots P_{k^{(\mathcal{C},\ell)}}$.

Unaware of the disturbances that affected any of the invocations $\langle \mathcal{C}, \ell, i \rangle$ and the repercussions thereof on the partitioning procedure, *dtof* is unable to discern whether the consensus block identified as a majority corresponds to b_0. As such, the result o adjudicated from a consensus block $P^{(\mathcal{C},\ell)}$ constituting a majority will be assumed to be the exact result; the impossibility to establish a majority, however, will be regarded as a failure of the scheme **(A31)** — *cf.* (A30). Considering the random variable A and its probability mass function f_A defined in Sect. 2.6.1.1, the probability of violating the former clause of this assumption can be quantified as $p \cdot Pr[A \geqslant m^{(\mathcal{C},\ell)}]$, where the multiplier equals the complement of the cumulative distribution function $F_A(m^{(\mathcal{C},\ell)} - 1)$ and p is the probability that during round $(\mathcal{C}, \ell)$ at least $m^{(\mathcal{C},\ell)}$ versions are affected by an RVF failure. As the mass of f_A is largely situated in the probabilities associated with the random variates 1 and 2, the adjudication of an erroneous outcome is quite likely for $n^{(\mathcal{C},\ell)} \leqslant 3$, although this effect may be mitigated by the value taken by p. The use of higher levels of redundancy will result in markedly smaller multiplier values — *cf.* Fig. 2.5.

The maximum distance is reached when there is full consensus among the replicas, *i.e.* $\wp^{(\mathcal{C},\ell)} \setminus P^{(\mathcal{C},\ell)} = \emptyset$, therefore $V^{(\mathcal{C},\ell)} = P^{(\mathcal{C},\ell)}$ and accordingly $c_{max}^{(\mathcal{C},\ell)} = n^{(\mathcal{C},\ell)}$. Conversely, the larger the dissent, the smaller is the value returned by *dtof*, and the closer we are to the failure of the voting scheme. A critically low value $dtof^{(\mathcal{C},\ell)} = 1$ represents a situation for which the majority was attained by only $m^{(\mathcal{C},\ell)}$ versions. During this voting round $(\mathcal{C}, \ell)$, the available redundancy $n^{(\mathcal{C},\ell)}$ was equal to $cr(e^{(\mathcal{C},\ell)})$ and was completely exhausted to counterbalance the maximal number of disturbances the scheme could tolerate, *i.e.* $e^{(\mathcal{C},\ell)} = n^{(\mathcal{C},\ell)} - m^{(\mathcal{C},\ell)}$. If the scheme would have been subjected to additional disturbances affecting any of the versions $v \in P^{(\mathcal{C},\ell)}$, the scheme would have failed to reach a majority and *dtof* returns 0. Fig. 3.2 depicts some examples when the number of replicas n is 7. In (a), unanimous consensus is reached, which corresponds to the farthest "distance" to failure. For scenarios (b) and (c), an increasing number of votes dissent from the majority (red and yellow circles) and correspondingly the distance shrinks. In (d), no majority can be found — thus, failure is reached.



(a) $dtof = 4$ (b) $dtof = 3$

(c) $dtof = 2$ (d) $dtof = 0$

Figure 3.2: Distance-to-failure in an NVP/MV scheme with $n = 7$ replicas, for varying levels of consensus.

3.1.2 Nett Redundancy: Abundance or Shortfall

Intuitively, $dtof^{(\mathcal{C},\ell)} = 1$ corresponds to the existence of a consent between precisely $m^{(\mathcal{C},\ell)}$ versions, given that $n^{(\mathcal{C},\ell)}$ versions were involved during the current voting round $(\mathcal{C},\ell)$. Accordingly, for any $dtof^{(\mathcal{C},\ell)} > 0$, one can observe that

$$c_{max}^{(\mathcal{C},\ell)} = m^{(\mathcal{C},\ell)} + \left(dtof^{(\mathcal{C},\ell)} - 1\right) \tag{3.3}$$

Based on this observation, $c_{max}^{(\mathcal{C},\ell)} - m^{(\mathcal{C},\ell)}$ essentially defines a measure providing a quantitative estimation of how closely the currently allocated amount and selection of resources within an NVP/MV composite matched the observed disturbances — by shortcoming or excess. Having defined $c_{max}^{(\mathcal{C},\ell)}$ as a natural number in $[0, n^{(\mathcal{C},\ell)}]$, one can easily see that $Ran(c_{max}^{(\mathcal{C},\ell)} - m^{(\mathcal{C},\ell)})$ lies in $[-m^{(\mathcal{C},\ell)}, n^{(\mathcal{C},\ell)} - m^{(\mathcal{C},\ell)}]$. In other words, $c_{max}^{(\mathcal{C},\ell)} - m^{(\mathcal{C},\ell)}$ provides an indirect estimation of the shortage or abundance of redundancy with respect to the disturbances that affected round $(\mathcal{C},\ell)$: a positive value essentially quantifies how many versions there exist in excess of the mandatory $m^{(\mathcal{C},\ell)}$ versions that collectively constitute the majority for round $(\mathcal{C},\ell)$[2]. For negative values, the absolute value $|c_{max}^{(\mathcal{C},\ell)} - m^{(\mathcal{C},\ell)}|$ represents the lack of consent relative to $P^{(\mathcal{C},\ell)}$ that would be required so as to constitute a majority. Such a value is interpreted as a symptom that the currently experienced disturbances cannot be successfully counterbalanced by the redundancy configuration used, *i.e.* $n^{(\mathcal{C},\ell)} < cr(e^{(\mathcal{C},\ell)})$, and the scheme would fail to guarantee the availability of the service it seeks to provide despite its fault-tolerant nature.

3.1.3 Contextual Redundancy and Dependability

Fig. 3.1 shows how two regions of interest may emerge when the employed level of redundancy diverges from the contextual redundancy that is required to systematically tolerate disturbances [106].

An "overshooting region" represents operational intervals of sustained availability of the service the composite is sought to provide. Such type of region encompasses one or more non-adjacent subsequences of $\{\ell_x\}_{\mathcal{C}}$ (highlighted in green in Fig. 3.1) for which the employed degree of redundancy can safely guarantee the delivery of the correct result, though may show to be overabundant with respect to the currently experienced threat. More specifically, for each voting round $(\mathcal{C},\ell)$ lying within this region, $n^{(\mathcal{C},\ell)} \geqslant cr(e^{(\mathcal{C},\ell)})$. By reason of (A30) and (A31), we shall say that a redundancy configuration for which $dtof^{(\mathcal{C},\ell)} > 0$ is cr-dependable with respect to round $(\mathcal{C},\ell)$, as clearly $n^{(\mathcal{C},\ell)} \geqslant cr(e^{(\mathcal{C},\ell)})$ holds true, and the configuration could have withstood an additional $\lfloor\left(n^{(\mathcal{C},\ell)} - cr(e^{(\mathcal{C},\ell)})\right)/2\rfloor = c_{max}^{(\mathcal{C},\ell)} - m^{(\mathcal{C},\ell)}$ disturbances — *v.* theorem A.1, App. A.

Similarly, an "undershooting region" (the red aggregate in Fig. 3.1) represents intervals of voting rounds $(\mathcal{C},\ell)$ during which the foreseen redundancy $n^{(\mathcal{C},\ell)}$ proves to be insufficient to counteract the current environmental disturbances $e^{(\mathcal{C},\ell)}$. The

[2] $P^{(\mathcal{C},\ell)}$ was previously used to denote the set of all versions in $V^{(\mathcal{C},\ell)}$ that contributed to the majority found at the end of round $(\mathcal{C},\ell)$. Let $P_{m^{(\mathcal{C},\ell)}} \subseteq P^{(\mathcal{C},\ell)}$ such that $|P_{m^{(\mathcal{C},\ell)}}| = m^{(\mathcal{C},\ell)}$. If $dtof^{(\mathcal{C},\ell)} = 1$, $P_{m^{(\mathcal{C},\ell)}} = P^{(\mathcal{C},\ell)}$. For $dtof^{(\mathcal{C},\ell)} > 1$, the majority is reconfirmed by $dtof^{(\mathcal{C},\ell)}-1$ surplus versions, *i.e.* $|P^{(\mathcal{C},\ell)} \setminus P_{m^{(\mathcal{C},\ell)}}| = dtof^{(\mathcal{C},\ell)}-1$.

availability of the service the composite is expected to sustain will be temporarily compromised, for the adjudication procedure will fail to produce the correct result for voting rounds lying within this region — *cf.* (A30).

3.2 From a Timeliness Perspective

A second challenge has its origins in the fact that remotely deployed software components may occasionally suffer from long response times, which is mainly to be attributed to any network latency as the result of message exchanges and, to a lesser extent, to excessive concurrency demands ensuing from periods of elevated load; the latter which may result in requests to be temporarily stalled upon their arrival at the target host — *cf.* state `(1)` in Fig. 2.1. For time-critical applications in which the timely availability of results is of paramount importance, any additional delay in the response time of a resource involved in a redundancy scheme may impact the scheme's effectiveness to deliver an outcome within the imposed time constraints [31, 102].

NVP voting schemes can be designed to retrieve a reply for each invocation $\langle \mathcal{C}, \ell, i \rangle$ of a version $v_i \in V^{(\mathcal{C},\ell)}$ within a guaranteed time slot t_{max}, a procedure formalised in Sect. 2.6.4. Any version failing to produce its response within the time constraints imposed by the voting system would obviously translate into a performance failure and, as such, have a detrimental impact on the effectiveness of the redundancy configuration [4, 102]. The time required for a voting round $(\mathcal{C}, \ell)$ to retrieve a result for each $v_i \in V^{(\mathcal{C},\ell)}$, *i.e.* the time it takes for $(\mathcal{C}, \ell)$ to complete the transition from state `(b)` into state `(c)`, is consequently guaranteed not to exceed t_{max}, and will equal the maximum time required to complete the transition from the initial `(0)` to the terminal `(5)` state for any invocation $\langle \mathcal{C}, \ell, i \rangle$ if none of these were affected by a performance failure.

3.3 From a Resource Expenditure Perspective

The application of redundancy schemata clearly brings with it some tangible impacts, the foremost of which are a significantly higher development cost and associated, increased infrastructural requirements for the development and deployment of additional software components. A predetermined degree of redundancy may, therefore, prove to be cost ineffective in that it inhibits to economise on resource consumption in case the actual number of disturbances could be successfully overcome by a lesser amount of redundancy, *i.e.* a value $dtof \geqslant 1$.

3.4 Application-Agnostic Context Properties

Having motivated the deficiencies inherently connected to the use of a statically predefined redundancy configuration with respect to the effectiveness of the fault-tolerant unit within which it is hardwired, there is thence an urgent need for adaptive software fault-tolerant solutions, encompassing sophisticated context-aware redundancy management.

The characteristic of context-awareness refers to the fact that a redundancy scheme is aware of the surrounding environment (*i.e.* the context) in which it

operates — *cf.* Sect. 1.2. This environment was shown to have "an exceedingly powerful impact on [the scheme] either because the latter needs to adapt in response to changing external conditions or because it relies on resources whose availability is subject to continuous change" — disturbances that may very well put the scheme's effectiveness into jeopardy had it held on to a static redundancy configuration [102, 111]. Examples of contextual information include, but are not limited to, properties like the amount of redundancy currently employed, to what extent this amount was capable of tolerating disturbances (*i.e. dtof*), the evolution of voting outcomes, and the operational status of each of the available resources $v \in V$. The operational status of a version v comprises a set of attributes encompassing statistics on response time, the number of pending requests[3], and a measure for reliability approximation that will be introduced in Chapt. 4 shortly.

Triggered by changes in the context, such adaptive fault-tolerant strategies may autonomously tune the amount of redundancy and the selection of functionally-equivalent resources employed within the redundancy scheme so as to sustain its effectiveness. Such an adaptive dependability strategy is introduced for NVP-based redundancy schemata in Sect. 5, in which the redundancy configuration is dynamically constructed in view of the knowledge obtained from the context properties mentioned hereabove.

[3] Considering the non-negative integer values of the counter metrics defined in Sect. 2.7, the number of pending requests can easily be determined as `NumberOfRequests` – (`NumberOfFailedRequests` + `NumberOfSuccessfulRequests`).

CHAPTER 4

Approximating Reliability

In order to allow for the autonomous adjustment of the employed redundancy configuration in view of potentially changing environmental behaviour, it is imperative that the system be able to approximate the operational status of individual resources. Expanding on its previous announcements in [83] and [84], the key contribution of this chapter is to be found in the introduction of a mathematically defined structure that is capable of efficiently capturing how a specific software component — version, that is — has affected the reliability of the fault-tolerant composite throughout its operational life span. The advantage of this normalised dissent metric is that an extremely small memory footprint will suffice to store this type of contextual information. Related research question(s): RQ-1.

It was already pointed out that the dependability of any NVP composite is affected by the dependability of the components integrated within. Controversial opinions exist on whether it is meaningful to use probabilistic measures of dependability, most of which are based on an analogy of traditional hardware dependability, to evaluate the quality of software. In particular, many people have questioned the adequacy of software reliability to quantify the operational profile of a software system.

A first major objection that has frequently been put forth is that, in spite of the proliferation of software reliability models that have been developed since the early 1970s, only few of these models seem to be able to capture and quantify a satisfying amount of complexity without excessive limitations [112]. Failing to adequately quantify the reliability of a software component inhibits the application of commonly used analytical combinatorial techniques for reliability analysis of hardware redundancy schemata to equivalent schemes involving diversely designed functionally-equivalent software components [3, Chapt. 4].

Moreover, it is hard to determine a quantitative approximation of the overall failure rate for a given software component. Apart from residual design faults, in a distributed computing environment, the failure rate of a software component may be influenced by the emergence of disturbances as failures in the underlying deployment platform or hardware, in any required external resource or network connectivity failures [102, 113] — *cf.* (A06).

As an alternative to a probabilistic measure for the reliability of a software component, we now define a generic property to measure the suitability of a particular software component (version) within a given NVP/MV redundancy scheme. Whereas *dtof* is a valuable metric for capturing the instantaneous effectiveness of a given redundancy configuration used throughout the life span of a single completed voting round $(\mathcal{C}, \ell)$ from a dependability perspective, it fails to assess the impact of a particular version on the scheme over time. We therefore define a measure to quantify the historical and relative impact of any version $v \in V$ on the redundancy scheme $\mathcal{C}$ — the normalised dissent $D(\mathcal{C}, v)$. Inspired by the $\boldsymbol{\alpha}$-count approach, penalties and rewards are repeatedly issued for individual versions depending on whether or not versions are perceived to be subject to a failure [77, 114] [5, Chapt. 3]. Our approach differs, however, in that such updates reflect the operational context in which the appurtenant versions have been operating, *i.e.* one or more completed voting rounds, rather than issuing updates of a constant magnitude.

A single value $D(\mathcal{C}, v)$ is maintained at the NVP composite $\mathcal{C}$ for each $v \in V$, and is updated whenever the transition from the `(c)` to the terminal state `(d)` is fired after the the partitioning and adjudication procedure for a voting round $(\mathcal{C}, \ell)$ has been completed (Fig. 2.1). Let $\{y_z\}_{\mathcal{C}}$ be a monotonically increasing sequence of strictly positive integer indices $y_z = z$ in $Y = \mathbb{N}^+$, such that each consecutive completion of some voting round $(\mathcal{C}, \ell)$ originating from an invocation of $\mathcal{C}$ is uniquely identified by the next element y in $\{y_z\}_{\mathcal{C}}$. Note how the bijective mapping function $b_{\mathcal{C}} : Y \mapsto L$ defines the correlation between the terms in either of the sequences $\{y_z\}_{\mathcal{C}}$ and $\{\ell_x\}_{\mathcal{C}}$[1]. Furthermore, an indicator random variable $E^{(\mathcal{C},\ell)}(v)$ is defined for all $v \in V$

$$E^{(\mathcal{C},\ell)}(v) = \begin{cases} 0 & v \in V \setminus V^{(\mathcal{C},\ell)} \quad (4.1a) \\ 1 & v \in V^{(\mathcal{C},\ell)} \quad (4.1b) \end{cases}$$

and can be used to discriminate between idling versions and versions that are engaged in the redundancy configuration used for a voting round $(\mathcal{C}, \ell)$. Considering the number z of voting rounds appertaining to $\mathcal{C}$ that completed since its initialisation, the last known value $D(\mathcal{C}, v)$ of the normalised dissent for a version $v \in V$ is given by $D^{(z)}(\mathcal{C}, v)$ as follows:

$$D^{(z)}(\mathcal{C}, v) = \begin{cases} 0 & z = 0 \quad (4.2a) \\ D^{(z-1)}(\mathcal{C}, v) + p^{(\mathcal{C},\ell)}(v) & E^{(\mathcal{C},\ell)}(v) = 1 \wedge dtof^{(\mathcal{C},\ell)} > 0 \wedge v \notin P^{(\mathcal{C},\ell)} \quad (4.2b) \\ & E^{(\mathcal{C},\ell)}(v) = 1 \wedge dtof^{(\mathcal{C},\ell)} = 0 \quad (4.2c) \\ D^{(z-1)}(\mathcal{C}, v) \times r^{(\mathcal{C},\ell)}(v) & E^{(\mathcal{C},\ell)}(v) = 0 \quad (4.2d) \\ & E^{(\mathcal{C},\ell)}(v) = 1 \wedge dtof^{(\mathcal{C},\ell)} > 0 \wedge v \in P^{(\mathcal{C},\ell)} \quad (4.2e) \end{cases}$$

As can be seen from Eq. (4.2a), the initial value $D^{(0)}(\mathcal{C}, v)$ of the normalised dissent for some version $v \in V$ is set to be 0. Then each update subsequently issued on $D(\mathcal{C}, v)$ as the result of the completion of a voting round $(\mathcal{C}, \ell)$ with $\ell = b_{\mathcal{C}}(y_z)$

[1] Note that the order in which voting rounds complete, *i.e.* the sequence $\{y_z\}_{\mathcal{C}}$, is not necessarily the order in which these voting rounds have been initialised, as represented by the sequence $\{\ell_x\}_{\mathcal{C}}$. Put differently, it is unlikely that $b_{\mathcal{C}}(y_z) = \ell_x$ for $z = x$ (as exemplified in Fig. 5.1). Indeed, in large-scale distributed computing environments, one may expect a significant amount of variability in the response time of an invocation on the NVP composite, which may be due to the dynamically changing redundancy configuration used for different voting rounds. More information on timeliness issues can be found in Sect. 2.6.4 and 3.2.

and $y_z = z \geqslant 1$ in $\{y_z\}_{\mathcal{C}}$ will depend on the information accrued on the effectiveness of the employed redundancy configuration $V^{(\mathcal{C},\ell)}$, and the potential role of v therein. The rationale is that a penalty $p^{(\mathcal{C},\ell)}(v) \in]0,1]$ is fined for any engaged version in dissent with the majority that resulted at the end of voting round $(\mathcal{C},\ell)$, or when simply no majority was found, which corresponds respectively to Eq. (4.2b) and (4.2c). A version v that repeatedly failed to provide a useful contribution to the voting scheme $\mathcal{C}$ will therefore translate into a higher value $D(\mathcal{C},v)$. Inversely, a reward $r^{(\mathcal{C},\ell)}(v) \in]0,1[$ will weigh down previously accumulated penalties as v is observed to sustain the availability of the composite throughout the life span of round $(\mathcal{C},\ell)$, or when it was not engaged in the redundancy configuration used — *v.* Eq. (4.2d) and (4.2e). Both penalisation and reward mechanisms are presented in greater detail hereafter.

4.1 Acquiring Context Information

A substantial characteristic of both models is that the penalty addends and the reward factors they generate aim to capture the robustness of the NVP/MV voting scheme. For a given voting round $(\mathcal{C},\ell)$ during which a majority could be found, *i.e.* $dtof^{(\mathcal{C},\ell)} > 0$, let

$$w_e^{(\mathcal{C},\ell)} = \begin{cases} 1 & 0 < n^{(\mathcal{C},\ell)} \leqslant 2 \quad (4.3a) \\ 1 - \dfrac{dtof^{(\mathcal{C},\ell)} - 1}{n^{(\mathcal{C},\ell)} - m^{(\mathcal{C},\ell)}} & n^{(\mathcal{C},\ell)} > 2 \quad (4.3b) \end{cases}$$

The above definition takes advantage of the *dtof* metric as defined in Eq. (3.2) to acquire information on the effectiveness of the redundancy configuration employed during the voting round $(\mathcal{C},\ell)$. The fraction involved in Eq. (4.3b) was designed so as to provide insight into the robustness of the redundancy configuration in face of the disturbances encountered. Specifically, the numerator can be regarded as the number of additional disturbances the redundancy configuration could have withstood during round $(\mathcal{C},\ell)$ — *cf.* Eq. (3.3). Conversely, the denominator represents the maximum number of disturbances that the scheme can withstand, given the available amount of redundancy, $n^{(\mathcal{C},\ell)}$. As such, $w_e^{(\mathcal{C},\ell)}$ provides an estimation of how close a given redundancy configuration was to exhausting the available amount of redundancy whilst it tried to counterbalance the disturbances experienced during round $(\mathcal{C},\ell)$. Considering the premise that $dtof^{(\mathcal{C},\ell)} > 0$, it can be seen from Eq. (4.3) that $w_e^{(\mathcal{C},\ell)}$ is a real number contained within the interval $[0,1]$. A critically low value $dtof^{(\mathcal{C},\ell)} = 1$, *i.e.* $w_e^{(\mathcal{C},\ell)} = 1$, represents a situation for which the majority was attained by only $m^{(\mathcal{C},\ell)}$ versions. During this voting round $(\mathcal{C},\ell)$, the available redundancy $n^{(\mathcal{C},\ell)}$ was completely exhausted to counterbalance the maximal number of disturbances the scheme could tolerate, *i.e.* $n^{(\mathcal{C},\ell)} - m^{(\mathcal{C},\ell)}$. Similarly, for $n^{(\mathcal{C},\ell)} > 2$, a value $w_e^{(\mathcal{C},\ell)} = 0$ corresponds to a voting round with full unanimity, *i.e.* $c_{max}^{(\mathcal{C},\ell)} = n^{(\mathcal{C},\ell)}$. Such additional consent contributes to the robustness of the scheme and its redundancy configuration, for it is resilient to withstand up to $n^{(\mathcal{C},\ell)} - m^{(\mathcal{C},\ell)}$ disturbances.

Furthermore, for $v \in V^{(\mathcal{C},\ell)}$, we define an ancillary function

$$c^{(\mathcal{C},\ell)}(v) = \begin{cases} |P_j| & v \in P_j \wedge P_j \in \{\wp^{(\mathcal{C},\ell)} \setminus P_F^{(\mathcal{C},\ell)}\} \quad (4.4a) \\ \perp & \text{otherwise} \quad (4.4b) \end{cases}$$

which allows to obtain the amount of versions that reported the same result as v at the end of round $(\mathcal{C},\ell)$ — *cf.* (A13). It can easily be seen that the range of this function is $[1, n^{(\mathcal{C},\ell)}]$.

4.2 Penalisation Mechanism

We now characterise the penalisation mechanism used in Eq. (4.2b) and (4.2c) for a subset of engaged versions $V^{(\mathcal{C},\ell)} \subseteq V$ — that is, a set of versions $v \in V^{(\mathcal{C},\ell)}$ for which $E^{(\mathcal{C},\ell)}(v) = 1$:

$$p^{(\mathcal{C},\ell)}(v) = \begin{cases} s^{(\mathcal{C},\ell)}(v) \times w_e^{(\mathcal{C},\ell)} & v \notin P_F^{(\mathcal{C},\ell)} \wedge dtof^{(\mathcal{C},\ell)} > 0 \quad (4.5a) \\ \dfrac{m^{(\mathcal{C},\ell)} - c^{(\mathcal{C},\ell)}(v)}{m^{(\mathcal{C},\ell)} - 1} & v \notin P_F^{(\mathcal{C},\ell)} \wedge dtof^{(\mathcal{C},\ell)} = 0 \quad (4.5b) \\ 1 & v \in P_F^{(\mathcal{C},\ell)} \quad (4.5c) \end{cases}$$

The penalty $p^{(\mathcal{C},\ell)}(v)$ inflicted on an engaged version $v \in V^{(\mathcal{C},\ell)}$ in dissent with the majority found at the end of voting round $(\mathcal{C},\ell)$ is given by Eq. (4.5a). The idea behind the multiplier $w_e^{(\mathcal{C},\ell)}$ is that a replica disagreeing with the majority during round $(\mathcal{C},\ell)$ should be penalised relatively to the detrimental impact it may have on the robustness of the currently selected redundancy configuration — *cf.* Eq. (4.3). The closer round $(\mathcal{C},\ell)$ was to failure (that is, the closer to $dtof^{(\mathcal{C},\ell)} = 0$), the stronger the multiplier shall penalise the dissentient replica. The further away from failure, the less we penalise as the excess degree of consent enhances the robustness of the redundancy configuration such that it is capable of tolerating additional disturbances. Note how the above multiplier cannot evaluate to 0 for at least v is in dissent for round $(\mathcal{C},\ell)$, and therefore full consensus, *i.e.* the maximum value for $dtof^{(\mathcal{C},\ell)}$ as defined in Eq. (3.2), can never be reached — *cf.* Eq. (4.2b). The range of $w_e^{(\mathcal{C},\ell)}$, which was previously defined as $[0,1]$ in Eq. (4.3), will therefore be confined to the interval $]0,1]$.

The multiplicand $s^{(\mathcal{C},\ell)}(v)$ will then scale the intermediate penalty obtained using $w_e^{(\mathcal{C},\ell)}$ inversely proportional to the amount of consent between a minority of engaged versions, including v

$$s^{(\mathcal{C},\ell)}(v) = 1 - \frac{c^{(\mathcal{C},\ell)}(v)}{m^{(\mathcal{C},\ell)}} \quad (4.6)$$

Indeed, any version v in dissent with the majority found is part of a minority equivalence class in $\wp^{(\mathcal{C},\ell)} \setminus \{P^{(\mathcal{C},\ell)}, P_F^{(\mathcal{C},\ell)}\}$. As the range of the previously defined function $c^{(\mathcal{C},\ell)}(v)$ will consequently narrow to $[1, n^{(\mathcal{C},\ell)} - m^{(\mathcal{C},\ell)}]$, one can observe from Eq. (4.6) that the values obtained for $s^{(\mathcal{C},\ell)}(v)$ lie in $]0,1[$.

Having defined the maximum plurality that is not an absolute majority as $m^{(\mathcal{C},\ell)} - 1$ in Sect. 3.1, the penalty for any of the versions involved in voting round $(\mathcal{C},\ell)$ for which no majority could be determined, can be found using Eq. (4.5b). A version v will be attributed the maximum penalty if its result is unique and in dissent with all the other versions, *i.e.* $c^{(\mathcal{C},\ell)}(v) = 1$. On the contrary, should there exist a minority of consentient active versions with cardinality equal to $m^{(\mathcal{C},\ell)} - 1$, each of the versions

would be penalised in the most gentle way. In other words, the more isolated the case, the heavier the penalty; the larger the cardinality of the minority to which a given version belongs, the less each of the versions that constitute the minority will be penalised.

Finally, faulty replicas that did not return a meaningful response are assigned the maximum penalty 1 — *cf.* Eq. (4.5c).

4.3 Reward Model

Whenever a version $v \in V^{(\mathcal{C},\ell)}$ produces a response that complies with the majority determined at the end of voting round $(\mathcal{C},\ell)$, a reward should compensate for any penalties that may have been imposed in previously completed voting rounds and consequently result in the gradual decline of its normalised dissent $D(\mathcal{C},v)$ — *cf.* Eq. (4.2e). Unlike the penalisation mechanism, which is only applicable to engaged versions, the reward model is also used for idle replicas that are not currently involved in the redundancy configuration for a given voting round $(\mathcal{C},\ell)$ but that may have been used in previously completed or pending voting rounds — *cf.* Eq. (4.2d).

Let $0 < k_2 < k_1 < k_{max} < 1$. We now define the reward factor $r^{(\mathcal{C},\ell)}(v)$ for a version $v \in V$ as

$$r^{(\mathcal{C},\ell)}(v) = \begin{cases} k_1 + \left((k_{max} - k_1) \times w_i^{(\mathcal{C},\ell)}(v)\right) & E^{(\mathcal{C},\ell)}(v) = 0 \quad (4.7a) \\ k_2 + \left((k_1 - k_2) \times w_e^{(\mathcal{C},\ell)}\right) & E^{(\mathcal{C},\ell)}(v) = 1 \quad (4.7b) \end{cases}$$

For any version v, a smaller reward factor $r^{(\mathcal{C},\ell)}(v)$ will result in a steeper decline of its normalised dissent $D(\mathcal{C},v)$, whereas a larger factor would result in a more gradual decline. The number of voting rounds emerging from the invocation of $\mathcal{C}$ in which $v \in V$ was actively engaged, up until and including the last completed voting round $(\mathcal{C},\ell)$, is denoted by $\#rounds(\mathcal{C},v)$. In addition, we define $\#consent(\mathcal{C},v)$ as the number of those voting rounds which were accounted for in $\#rounds(\mathcal{C},v)$ for which v contributed to the majority. Consequently, $\#rounds(\mathcal{C},v) - \#consent(\mathcal{C},v)$ corresponds to those voting rounds in which v had been engaged, such that either v was in dissent with the majority, or no majority was found at all. Note how these counters are updated along with the updates issued for the corresponding $D(\mathcal{C},v)$ value.

$$w_i^{(\mathcal{C},\ell)}(v) = \begin{cases} 0 & \#rounds(\mathcal{C},v) = 0 \quad (4.8a) \\ \dfrac{\#rounds(\mathcal{C},v) - \#consent(\mathcal{C},v)}{\#rounds(\mathcal{C},v)} & \#rounds(\mathcal{C},v) > 0 \quad (4.8b) \end{cases}$$

With $w_i^{(\mathcal{C},\ell)}(v)$ defined as a real number in $[0,1]$, Eq. (4.7a) shows how the reward factor is determined for an idle version $v \in V \setminus V^{(\mathcal{C},\ell)}$ that is not involved in the current voting round $(\mathcal{C},\ell)$. It follows that $r^{(\mathcal{C},\ell)}(v)$ is contained within $[k_1, k_{max}]$. The upper endpoint of the range, k_{max}, is defined to be close to, but less than 1. This is motivated by the fact that, if k_{max} were equal to 1, a value $r^{(\mathcal{C},\ell)}(v) = 1$ would not be able to ensure that penalties accumulated during previous voting rounds are weighed down over time — *cf.* Eq. (4.2d) and (4.2e). The smallest reward

$r^{(\mathcal{C},\ell)}(v)$ is equal to k_1 and corresponds to the case when v did not participate in any voting round so far, *i.e.* Eq. (4.8a), or when the replica contributed to the majority for every voting round it was previously engaged in, *i.e.* Eq. (4.8b) when $\#rounds(\mathcal{C}, v) = \#consent(\mathcal{C}, v)$. Larger reward values will be obtained for versions v, up to a maximum of k_{max}, proportional to the relative amount of voting rounds for which an engaged version v previously failed to support the voting scheme and was subsequently penalised, *i.e.* $w_i^{(\mathcal{C},\ell)}(v)$.

The reward procedure for engaged versions that were in consent with the outcome of the current voting round $(\mathcal{C}, \ell)$ is described in Eq. (4.7b). Having defined $w_e^{(\mathcal{C},\ell)}$ as a real number contained within the interval $[0, 1]$ in Eq. (4.3), it can be seen the range of $r^{(\mathcal{C},\ell)}(v)$ is delimited by $[k_2, k_1]$ for any version v engaged during round $(\mathcal{C}, \ell)$. As it can be seen in Eq. (4.7b), larger values for $w_e^{(\mathcal{C},\ell)}$, *i.e.* $dtof^{(\mathcal{C},\ell)} - 1$ approaches 0, lead to a larger reward factor $r^{(\mathcal{C},\ell)}(v)$, up to the maximum value k_1. Contrariwise, more robust redundancy configurations translate into smaller values for $w_e^{(\mathcal{C},\ell)}$ and will be allotted smaller values for $r^{(\mathcal{C},\ell)}(v)$ accordingly. This allows to counterbalance and rectify a situation where v was undeservedly penalised in any preceding voting rounds it participated in, *i.e.* v did produce a correct result, but it was penalised because of an inadequate selection $V^{(\mathcal{C},\ell)}$.

As a final remark, we would like to point out that it was a deliberate design decision to define the reward model for idle versions in a separate range $[k_1, k_2]$, resulting in reward factors of comparatively greater magnitude, so as to ensure a more gradual decline in normalised dissent when compared to engaged replicas.

CHAPTER 5

An Adaptive Context-Aware Fault-Tolerant Strategy

Having motivated the desire to adjust a redundancy scheme's internally used redundancy configuration in order to safeguard the scheme's dependability objectives, or to avoid excessive resource consumption, we will now proceed by outlining the core design principles of a novel NVP-based fault-tolerant strategy, and elaborate on its support for advanced redundancy management. The key contribution of this chapter lies in an adaptive dependability strategy in which an autonomous context-aware adjustment process of the redundancy configuration is configurable by means of various resource allocation policies, and which is driven by the set of application-agnostic context properties suggested in Sect. 3.4. The contents of this chapter have been disseminated to the public through the publications [83] and [84]. Related research question(s): RQ-2 and RQ-5.

In this section, we introduce our adaptive NVP-based fault-tolerant strategy and elaborate on the advanced redundancy management it supports. Aiming to autonomously tune its internal configuration in view of changes in context, it was designed to dynamically find the optimal redundancy configuration. Our context-aware reformulation of the classical NVP/MV system structure encompasses two complementary parameterised models that jointly determine the redundancy configuration to be used throughout a newly initialised voting round $(\mathcal{C}, \ell)$, with ℓ the next element in the sequence $\{\ell_x\}_{\mathcal{C}}$. The retrieval procedure of the redundancy configuration takes place whilst preparing to fire the transition from state `(a)` to state `(b)` as shown in the voting round state transition model in Fig. 2.1 **(D05)**.

During the first stage of this procedure, the redundancy dimensioning model, which will be explained shortly in Sect. 5.2, will select the appropriate degree of redundancy $n^{(\mathcal{C},\ell)}$ to be employed in function of the disturbances experienced in previous voting rounds. In doing so, it will attempt to economise on resource expenditure whenever it can be argued safe to do so. Next, the replica selection model will establish which replicas $v \in V$ are most appropriate to constitute $V^{(\mathcal{C},\ell)}$. This second stage, which will be elaborated upon in Sect. 5.3, was designed to

enrol those replicas targeting an optimal trade-off between the context properties introduced in Sect. 3.4 and Chapt. 4, *i.e.* normalised dissent, response time and pending load.

5.1 Application-Specific Requirements

The optimal redundancy configuration is, however, not only determined by the quantitative assessment in terms of the context properties introduced in Sect. 3.4 and Chapt. 4, but also by the characteristics of the application itself, or the environment in which it operates. For instance, some applications may be latency-sensitive, whereas others may operate in a resource-constrained environment. The A-NVP/MV algorithm was conceived to take these application-specific intricacies into account, in that the redundancy dimensioning and replica selection models can be configured by means of a set of user-defined parameters.

Our A-NVP/MV algorithm has been designed primarily to maximise the redundancy scheme's dependability, and secondarily, it may be configured to target other application objectives such as time constraints as well as load balancing. User-defined weights $w_D^{\mathcal{C}}$, $w_T^{\mathcal{C}}$ and $w_L^{\mathcal{C}}$ for each of the three respective application objectives listed, can be used to configure the replica selection model such that it will engage the most appropriate replicas so as to maximise the overall effectiveness of the voting scheme. It is assumed that

$$\sum_{X \in \{D,T,L\}} w_X^{\mathcal{C}} = 1 \tag{5.1}$$

Furthermore, an optional user-defined parameter t_{max} represents the largest permissible response time that an NVP composite $\mathcal{C}$ can afford for any invocation $\langle \mathcal{C}, \ell, i \rangle$ to complete, with $v_i \in V^{(\mathcal{C},\ell)}$. A smaller value represents more stringent requirements on the scheme's response time, implicitly indicating that the application is more latency-sensitive. The t_{max} parameter is of particular interest as it is used to detect omission and late response failures: if a version $v_i \in V^{(\mathcal{C},\ell)}$ failed to return its response to the NVP composite before the t_{max} time-out has lapsed since the initialisation of the corresponding invocation $\langle \mathcal{C}, \ell, i \rangle$, v_i will be classified in $P_F^{(\mathcal{C},\ell)}$ and penalised accordingly as described in Sect. 4.2, 2.6.4 and 3.2.

Finally, applications deployed in resource-constrained environments may benefit from the parameter $n_{max} \geqslant 1$ to set an upper bound on the number of replicas to be used in parallel, which may result in the utilisation of fewer computing and networking resources. This parameter may affect the degree of redundancy $n^{(\mathcal{C},\ell)}$ as determined by the redundancy dimensioning model, possibly at the expense of a significantly higher risk of failure of the voting scheme. Contrariwise, the parameter $n_{min} \geqslant 1$ can be used to set a lower bound on the degree of redundancy to be used, such that the scheme is guaranteed to be resilient to withstand at least $\lfloor (n_{min}-1)/2 \rfloor$ disturbances. Lastly, a parameter n_{init} will set the default degree of redundancy to be initially used, *i.e.* $n^{(\mathcal{C},\ell)} = n_{init}$ for $\ell = 1$ and $n_{min} \leqslant n_{init} \leqslant n_{max}$.

5.2 Redundancy Dimensioning Model

Given the set V of available functionally-equivalent versions in the system, our redundancy dimensioning model is responsible for autonomously adjusting the

degree of redundancy employed such that it closely follows the evolution of the observed disturbances. In the absence of exceptional disturbances, the scheme should scale down its use of redundant replicas so as to avoid the unnecessary expenditure of resources. Contrarily, when the foreseen amount of redundancy is not enough to compensate for the currently experienced disturbances, it would be beneficial to dynamically revise that amount and enrol additional resources — if available.

Let n_c denote a function defining a relation $\mathscr{C} \times L \mapsto \mathbb{N}^+$ designed to return an integer representing the degree of redundancy $n^{(\mathcal{C},\ell)}$ to be employed for round $(\mathcal{C},\ell)$, with $\mathcal{C} \in \mathscr{C}$, $\ell \in \{\ell_x\}_{\mathcal{C}} \subseteq L$ and $\mathscr{C}$ the set of NVP-based redundancy schemata deployed. Ideally, the redundancy configuration used for a round $(\mathcal{C},\ell)$ would involve a number $n^{(\mathcal{C},\ell)}$ of versions that is consistently greater than or equal but close to $cr(e^{(\mathcal{C},\ell)})$, with $e^{(\mathcal{C},\ell)}$ the number of disturbances that challenge the scheme during the operational life span of round $(\mathcal{C},\ell)$ — *cf.* Sect. 3.1. Maintaining such levels of redundancy would allow the system to exhibit resilience with minimal overshooting and to balance both the design goal of reliability persistence and reducing the operational costs of the system [106]. Nonetheless, the robustness of the system relies upon the quality of n_c, and in particular on its ability to effectively track the evolution of the $e^{(\mathcal{C},\ell)}$. As the number of disturbances affecting the course of a specific voting round is not precisely known[1], the task of estimating $cr(e^{(\mathcal{C},\ell)})$ upfront it is quite arduous when relying solely on contextual information acquired from previously completed rounds. Even if a meaningful trend could have been caught, the issue is further aggravated by missing information from pending requests and the potential intermittent whimsicality of the environment. For example, the redundancy adjustment procedure depicted in Fig. 3.1 clearly fails to anticipate changes in the number of disturbances affecting subsequent voting rounds, resulting in occasional breaches of the targeted dependability. A solution to this issue avoiding disruptions of the availability of the expected service may be to allocate an adequate amount of additional redundancy.

The redundancy dimensioning model is expected to determine $n^{(\mathcal{C},\ell)}$ upon initialisation of the voting round $(\mathcal{C},\ell)$, abiding the premise that $n_{min} \leqslant n^{(\mathcal{C},\ell)} \leqslant \min(|V|, n_{max})$. Note that the behaviour of the system is undefined in case $|V| < n_{min}$, or when the optimal degree of redundancy as inferred by the model exceeds n_{max} or falls beneath n_{min}. Depending on the application domain, the A-NVP/MV scheme could simply report failure, or it could proceed with the suboptimal redundancy currently supported.

5.2.1 Window of Context Information

In order to make an informed decision on the amount of redundancy $n^{(\mathcal{C},\ell)}$ to be used, the model will consider the course of the amount of redundancy used for previously completed voting rounds and whether or not the selected redundancy proved to be sufficient to guarantee the scheme's availability. Recalling that z was defined in Chapt. 4 to denote the number of completed voting rounds appertaining to the scheme $\mathcal{C}$, the system shall therefore maintain a data structure to store this

[1] Recall from (A31) that an estimation of $e^{(\mathcal{C},\ell)}$ cannot be deduced until the time $(\mathcal{C},\ell)$ has completed the transition into state `(c)` — *cf.* Fig. 2.1.

type of contextual information for each y in the subsequence $\{y_{\min(1,z-r_d+1)},\ldots,y_z\}_{\mathcal{C}}$ of $\{y_z\}_{\mathcal{C}}$. More specifically, for each completed voting round y lying within the window constituted by this subsequence, the context properties of interest are: the corresponding identifier $\ell_x = b_{\mathcal{C}}(y)$, the amount of redundancy $n^{(\mathcal{C},\ell_x)}$ that was employed throughout its execution, and the extent to which this redundancy was found to be capable of masking disturbances such that a majority could be adjudicated. This last context property was defined in Sect. 3.1 as $c_{max}^{(\mathcal{C},\ell)} - m^{(\mathcal{C},\ell)}$ and provides an indirect estimation of the shortage or abundance of redundancy with respect to the currently experienced disturbances threatening the successful completion of the voting round $(\mathcal{C},\ell)$ — *cf.* (A31).

Let r_d, r_u and r_f be natural numbers in $\mathbb{N}^+$ such that $1 \leqslant r_f \leqslant r_u < r_d$. The former number r_d represents the minimum number of consecutive successful voting round completions before contemplating scaling down the current level of redundancy. In line with (A31), a given voting round $(\mathcal{C},\ell)$ is observed to have completed successfully if a sufficiently large degree of consent could be found between the responses acquired from the subordinate invocations $\langle\mathcal{C},\ell,i\rangle$ of the involved versions $v_i \in V^{(\mathcal{C},\ell)}$ such that a majority could be found **(A32)**, *i.e.* $c_{max}^{(\mathcal{C},\ell)} - m^{(\mathcal{C},\ell)} \geqslant c_{sm}$, with a discretionary safety margin c_{sm} defined as a natural number in $[0, n^{(\mathcal{C},\ell)} - m^{(\mathcal{C},\ell)}]$ — *cf.* Eq. 3.3. Note how r_d imposes an upper bound on the maximum window capacity maintained by the data structure, as shown in Fig. 5.1. This data structure is maintained by context manager located within the A-NVP/MV composite — *v.* Fig. 5.2, p. 87. View on system state before $n^{(\mathcal{C},\ell)}$ is set for use throughout a newly initialised voting round $(\mathcal{C},\ell)$.
It is assumed that shorter window lengths may result in an incautious downscaling of the redundancy, which in itself might lead to failure of the voting scheme in subsequent voting rounds. Contrariwise, one may reasonably expect that the redundancy scheme is less likely to fail due to the downscaling of the employed degree of redundancy for larger values of r_d, at the expense of postponing the relinquishment of excess redundancy **(A33)**.

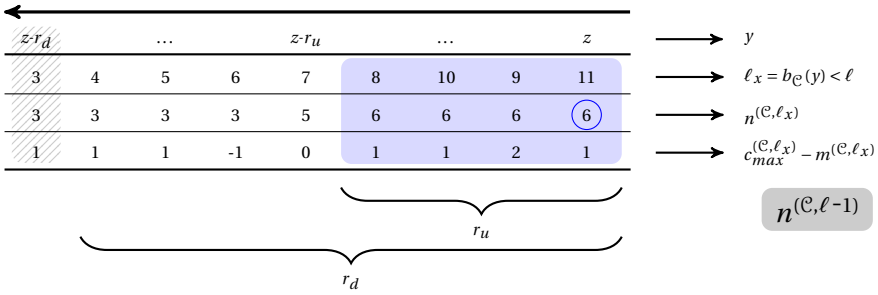


Figure 5.1: Window-based data structure holding context information regarding the employed degree of redundancy and its ability to sustain the scheme's availability throughout the last r_d completed voting rounds.

The number r_u expresses the maximum number of successive voting round completions that failed to meet the criterion of success as defined hereabove, before responding by considering the use of additional redundancy. On the one hand, such scenario would involve voting rounds for which a result o could be adjudicated, yet the cardinality $c_{max}^{(\mathcal{C},\ell)}$ of the corresponding consensus block $[o]$ did

not comprise an ample number of consentient replicas so as to suit the imposed safety margin c_{sm}. On the other hand, it encompasses voting rounds for which the decision algorithm failed to adjudicate a result, *i.e.* $dtof^{(\mathcal{C},\ell)} = 0$ — a case the model will endure at most r_f times within the observation window constituted by the corresponding subsequence $\{y_{\min(1,z-r_u+1)},\ldots,y_z\}_{\mathcal{C}}$ of $\{y_z\}_{\mathcal{C}}$[2]. At risk of prolonging the scheme's unavailability, temporarily refraining from increasing the employed degree of redundancy after observing a potentially hazardous situation might allow the replica selection model to regain the scheme's intended dependability by substituting poorly performing versions by more reliable, idling versions. Smaller values for r_u (and r_f) would usually enable potentially hazardous situations to be detected more rapidly. This would come at the price of a more aggressive upscaling strategy though, in which system resources would be allocated rather lavishly **(A34)**.

The data structure shown in Fig. 5.1 is updated upon each subsequent completion y of a voting round, as represented by the corresponding element in the sequence $\{y_z\}_{\mathcal{C}}$. Each column characterises an observational sample pertaining to a single voting round $\ell = b_{\mathcal{C}}(y)$ and holds information regarding the redundancy $n^{(\mathcal{C},\ell)}$ employed and its effectiveness to counterbalance the disturbances $e^{(\mathcal{C},\ell)}$ to which it was subject, *i.e.* $c_{max}^{(\mathcal{C},\ell)} - m^{(\mathcal{C},\ell)}$. Information pertaining to round $b_{\mathcal{C}}(y_{z-r_d})$ will be discarded for values $z > r_d$, as exemplified at the left in Fig. 5.1.

It follows from (D5) that the function n_c used to determine the degree of redundancy $n^{(\mathcal{C},\ell)}$ to be used for a voting round ℓ in $\{\ell_x\}_{\mathcal{C}}$ takes the shape of a piecewise constant function $\{\ell_x\}_{\mathcal{C}} \mapsto \mathbb{N}^+$. Each piece of the function delineates a certain disjoint subsequence of the function's domain $\{\ell_x\}_{\mathcal{C}}$, such that the same amount of redundancy was used for each of the voting rounds represented by the elements contained within the subsequence. Observe how step discontinuities in between any two adjoining such subsequences emerge because of the adjustment of the degree of redundancy. The redundancy dimensioning model has been designed so as to determine, whenever desirable, the extent to which the currently employed degree of redundancy should be adjusted, *i.e.* the oscillation of the discontinuities is applied to the value the function took in the previous subsequence. As such, the system will maintain the level of redundancy currently employed within $\mathcal{C}$. Having determined the amount of redundancy $n^{(\mathcal{C},\ell)}$ to be used for a newly initialised voting round ℓ, ℓ being the next element in the sequence $\{\ell_x\}_{\mathcal{C}}$, the system will store this newly computed value, effectively overriding the current level of redundancy that was used before, *i.e.* $n^{(\mathcal{C},\ell-1)}$.

5.2.2 Window Semantics

We will now elaborate on the procedure used to determine the amount of redundancy to be used throughout a newly initialised voting round ℓ. In doing so, we use the abstract notion of window semantics $\mathcal{S}_{\mathcal{C}}$ to epitomise the specific conditionalities and correlational techniques that enable the redundancy dimensioning model to deduce the optimum degree of redundancy matching the scheme's operational context from the stored information. In this capacity, $\mathcal{S}_{\mathcal{C}}$ defines two ancillary functions describing a relation $L \mapsto \mathbb{N}^+$. More specifically, the upscaling function

[2]Note how all observations with a nett redundancy larger than or equal to 0 are held within the green redundancy overshooting region as exemplified in Fig. 3.1. On the contrary, rounds for with a negative value is reported, *i.e.* $dtof = 0$, are part of the red undershooting region.

$f_u^{(\mathcal{C},\ell)}$ is responsible for determining if and to what extent the current level of redundancy $n^{(\mathcal{C},\ell\text{-}1)}$ should increase, whereas the downscaling function $f_d^{(\mathcal{C},\ell)}$ quantifies the extent by which $n^{(\mathcal{C},\ell\text{-}1)}$ should be lowered. The final degree of redundancy $n^{(\mathcal{C},\ell)}$ to be used for the continuation of ℓ is then resolved as follows:

$$n^{(\mathcal{C},\ell)} = \begin{cases} \min(\min(n_{max}, |V|), n^{(\mathcal{C},l\text{-}1)} + f_u^{(\mathcal{C},\ell)}) & cnt_u \geqslant r_u \text{ or } cnt_f \geqslant r_f \quad (5.2a) \\ \max(n_{min}, n^{(\mathcal{C},\ell\text{-}1)} - f_d^{(\mathcal{C},\ell)}) & cnt_d = r_d \quad (5.2b) \\ n^{(\mathcal{C},\ell\text{-}1)} & \text{otherwise} \quad (5.2c) \end{cases}$$

The above Eq. (5.2a) and (5.2b) formalise the upscaling, respectively downscaling procedure for $\ell > 1$. Observe how the adjustment of the redundancy is constrained by the application-specific parameters n_{min} and n_{max}, as well as by the amount $|V|$ of system resources available. Discontinuities in the piecewise constant function n_c depicting the evolution of the employed degree of redundancy throughout the domain $\{\ell_x\}_{\mathcal{C}}$ will only emerge for values $n^{(\mathcal{C},\ell)} - n^{(\mathcal{C},\ell\text{-}1)} \neq 0$, effectively portraying the appurtenant oscillations. Indeed, other scenarios would reduce to Eq. (5.2c), indicating that the same level of redundancy is maintained, therefore prolonging the applicable subsequence to include ℓ.

The value of the optional safety margin c_{sm} expressing the amount of consent supplementary to the mandatory $m^{(\mathcal{C},\ell)}$ required for the successful adjudication of a result o is set at the discretion of the chosen window semantics $\mathcal{S}_{\mathcal{C}}$. It serves as a parameter to the redundancy dimensioning model, primarily aiming to reduce the likelihood that the downscaling procedure itself would result in failure of the scheme in the first few subsequent voting rounds. Moreover, such safety margin could anticipate a shortfall in redundancy when the effectiveness of the employed redundancy is observed to exhibit a decreasing trend and proactively trigger the upscaling procedure. Either way, the underlying rationale for maintaining a slightly higher degree of redundancy stems from the assumption that the environment behaves unpredictably and the number of disturbances $e^{(\mathcal{C},\ell)}$ it brings about affecting ongoing voting rounds ℓ, therefore, may vary considerably, exhibiting a trend most whimsical **(A35)**. In sharp constrast, the redundancy dimensioning model was designed to gradually adjust the used degree of redundancy downwards, targeting the contextual redundancy $cr(e^{(\mathcal{C},\ell)})$, in line with the trend perceived from the data held within the observation window. The safety margin can therefore intuitively be seen as the maximum aberration in terms of additional disturbances that the scheme can tolerate as compared with the observed trend.

The internals of any realisation of a specific window semantics $\mathcal{S}_{\mathcal{C}}$ encompass the auxiliary counters cnt_d, cnt_u and cnt_f and stipulate the conditionalities under which these counters are to be updated for successive voting round completions y_z in the sequence $\{y_z\}_{\mathcal{C}}$. In particular, the basic criterion of success defined in (A32) can be restricted by imposing additional constraints. Counters may be updated and reset immediately before the actual insertion of the observational sample holding the contextual information acquired from a completed voting round y_z. The insertion of a new sample matching the imposed constraints will result in counter cnt_d to be incremented, unless there exists a previously inserted sample pertaining to round

$b_{\mathcal{C}}(y_{z-r_d})$ that had been accounted for as well[3]. Whether or not cnt_d is reset after a downward adjustment of the degree of redundancy, remains undefined and depends on the implementation of the chosen window semantics. Samples failing to comply with the (constrained) success criterion will cause the counter cnt_d to be reset and the value held in cnt_u to be incremented, as well as the value in cnt_f if $dtof^{(\mathcal{C},\ell)} = 0$ for $\ell_x = b_{\mathcal{C}}(y_z)$. As the window $\{y_{(z-r_u+1)}, \ldots, y_z\}_{\mathcal{C}}$ is shifted so as to accommodate y_z for $z > r_u$ and the element y_{z-r_u} no longer lies within the scope of the observed window, these counters will be decremented if their value was affected at the time y_{z-r_u} was inserted.

Furthermore, the correlational techniques applied in order to make an informed decision on how the currently employed degree of redundancy should be adjusted, are held within the implementation of the upscaling and downscaling functions $f_u^{(\mathcal{C},\ell)}$, respectively $f_d^{(\mathcal{C},\ell)}$. Which function, if any, will be called upon the initialisation of each successive voting round ℓ in $\{\ell_x\}_{\mathcal{C}}$, is dependent on the state of the aforementioned counters at the time the model attempts to settle for $n^{(\mathcal{C},\ell)}$. In case $cnt_u \geqslant r_u$, or $cnt_f \geqslant r_f$, the upscaling function will be called, and the adjusted degree of redundancy to be employed will be determined as per Eq. (5.2a). If such upward adjustment is caused because of a state in which $cnt_f \geqslant r_f$, counter cnt_u may or may not be reset (cnt_f will always be). Whether or not this counter is reset remains undefined and depends on the implementation of the chosen window semantics. In case $cnt_d = r_d$, the degree of redundancy will be adjusted downwards, as shown in Eq. (5.2b). If neither of the previous two cases holds true, the system will proceed with the degree of redundancy as it was previously employed — *v.* Eq. (5.2c).

5.3 Replica Selection Model

Having established the degree of redundancy $n^{(\mathcal{C},\ell)}$ to be employed throughout round $(\mathcal{C},\ell)$, the replica selection model will then determine a selection of versions $V^{(\mathcal{C},\ell)}$ to be used by the redundancy scheme $\mathcal{C}$, such that $|V^{(\mathcal{C},\ell)}| = n^{(\mathcal{C},\ell)}$. The proposed model has been designed so as to achieve an optimal trade-off between dependability as well as performance-related objectives such as load balancing and timeliness, respectively represented as the $w_D^{\mathcal{C}}$, $w_L^{\mathcal{C}}$ and $w_T^{\mathcal{C}}$ application-specific configuration parameters. More specifically, its purpose is to mitigate the adverse effects of employing inapt resources that consistently perform poorly in terms of the envisaged effectiveness objectives and that, consequentially, may threaten the effectiveness of the overall redundancy scheme. If the degree of redundancy $n^{(\mathcal{C},\ell)}$ to be utilised follows a constant or decreasing trend, then, depending on the availability of eligible versions that can be used as a substitute, the model will be successful in excluding such inapt replicas.

The suitability of a particular version $v \in V$ within an NVP/MV scheme can now be assessed quantitatively, leveraging the context properties introduced in Sect. 3.4 and Chapt. 4. More specifically, this assessment is made by computing a score $s(\mathcal{C}, v)$ in which all relevant contextual information accrued during previously completed voting rounds is taken into account.

[3] Note how this scenario can only materialise when $z > r_d$, for the maximum window capacity r_d is entirely used, necessitating the disposal of sample y_{z-r_d} so as to free sufficient capacity prior to the insertion of sample y_z.

Let us denote the last known values of the normalised dissent, the number of pending requests and the average response time for a version $v \in V$ by $D(\mathcal{C}, v)$, $L(\mathcal{C}, v)$ and $T(\mathcal{C}, v)$ respectively. If no such value was previously reported, all variables will hold the value 0. The process of determining a trade-off between the different application objectives can now be facilitated by normalising the context properties, which were defined without any upper bounds, to the same range. We therefore define $\delta_D^{\mathcal{C}}$ as the maximum value $D(\mathcal{C}, v)$ for all versions $v \in V$. $\delta_L^{\mathcal{C}}$ and $\delta_T^{\mathcal{C}}$ are defined analogously as the maximum value of $L(\mathcal{C}, v)$ and $T(\mathcal{C}, v)$, respectively. Whereas $D(\mathcal{C}, v)$, $L(\mathcal{C}, v)$ and $T(\mathcal{C}, v)$ are initialised to 0, the thresholds $\delta_D^{\mathcal{C}}$, $\delta_L^{\mathcal{C}}$ and $\delta_T^{\mathcal{C}}$ will be initialised to 1. Subsequently, the values for these context properties can now be normalised to a real number over the interval $[0, 1]$:

$$X_N(\mathcal{C}, v) = \frac{\delta_X^{\mathcal{C}} - X(\mathcal{C}, v)}{\delta_X^{\mathcal{C}}} \text{ for } v \in V \tag{5.3}$$

where $X \in \{D, L, T\}$ stands for any of the three context properties normalised dissent, pending load and response time. Practically speaking, a larger value $X(\mathcal{C}, v)$ for any of the three properties under consideration is representative of a worse impact of the replica v on the redundancy scheme. Accordingly, larger values of the normalised value $X_N(\mathcal{C}, v)$ signal versions more suitable to support the redundancy scheme. After the context property values were normalised onto a common range $[0, 1]$, one can now determine the score $s(\mathcal{C}, v)$ for each version $v \in V$ as follows:

$$s(\mathcal{C}, v) = w_D^{\mathcal{C}} \times D_N(\mathcal{C}, v) + w_L^{\mathcal{C}} \times L_N(\mathcal{C}, v) + w_T^{\mathcal{C}} \times T_N(\mathcal{C}, v) \tag{5.4}$$

The replica selection procedure is then reduced to a mere sorting problem, in which the versions are ranked by descending values of $s(\mathcal{C}, v)$. At this stage, all information regarding the redundancy configuration is available, and the execution of the voting round $(\mathcal{C}, \ell)$ can proceed using the first $n^{(\mathcal{C},\ell)}$ versions.

It was already pointed out in Sect. 5.1 that the optional user-defined parameter t_{max} is used to enable the detection of performance failures. Whenever a replica v is detected to be affected by such a type of failure throughout the course of a voting round $(\mathcal{C}, \ell)$, the stalled invocation request should promptly be abandoned, and a predefined internal failure message will be issued as the response message. Version v will consequently be classified in $P_F^{(\mathcal{C},\ell)}$, and penalised as described in Sect. 4.2, directly affecting the version's normalised dissent value.

The use of the t_{max} configuration parameter will also have repercussions on the $T(\mathcal{C}, v)$ context property. As one can see in Eq. (5.4), if some context property value $X(\mathcal{C}, v)$ for a specific replica v was not updated after its initialisation, *i.e.* $X(\mathcal{C}, v) = 0$ and therefore $X_N(\mathcal{C}, v) = 1$, the version is tacitly assumed to contribute to the success of the scheme in terms of the application objective associated with that property — *cf.* (A39). We have therefore chosen to report t_{max} as the response time of versions that fail to return their response within the imposed time constraint, such that the system can guarantee that $T(\mathcal{C}, v) \leqslant t_{max}$.

5.4 Context Manager

Obviously, an important prerequisite to obtain an accurate resource selection $V^{(\mathcal{C},\ell)}$ is to have the required contextual information instantly available. As shown in

Fig. 5.2, the A-NVP composite contains a context manager component that is responsible for continuously monitoring any changes in the operational status of the available resources, *i.c.* the context properties introduced in Sect. 3.4 and Chapt. 4 for each of the functionally-equivalent versions $v \in V$ available in the system. It also serves the purpose of maintaining the data structure depicted in Fig. 5.1.

When new information regarding one or more context properties is reported, the context manager will update its internal data structures accordingly, enforcing appropriate synchronisation mechanisms so as to ensure data consistency. As such, any update of a context property $X(\mathcal{C}, v)$ for a version v will instantaneously be reflected in the value of the corresponding $X_N(\mathcal{C}, v)$. Property updates may account for internally deduced information, *e.g.* the $c_{max}^{(\mathcal{C},\ell)} - m^{(\mathcal{C},\ell)}$, normalised dissent and version invocation response time metrics, which are harvested by the A-NVP/MV scheme at the end of each voting round $\ell = b_{\mathcal{C}}(y)$ for successive elements y in $\{y_z\}_{\mathcal{C}}$ — *cf.* Chapt. 4. Other metrics such as pending load may, however, be externally provided; updates to their values may be subjected to delays due to the issuance, transmission and processing of notification messages, a procedure which was formalised in the counter update discrete event model introduced in Sect. 2.7.

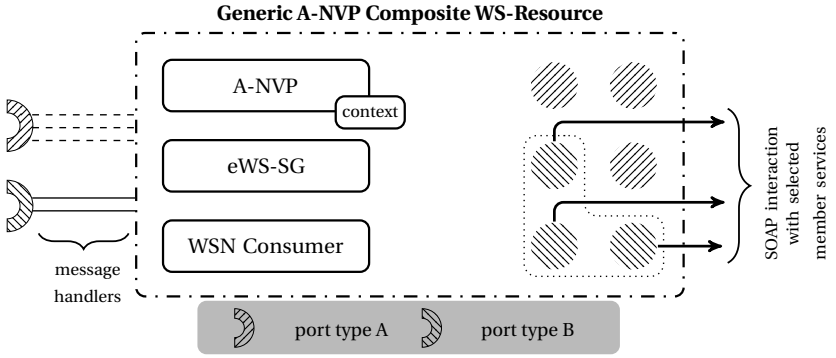


Figure 5.2: An A-NVP-based redundancy scheme designed as a WSDM-enabled WS-Resource. Includes manageability capability implementing this algorithm, with dedicated context manager component.

CHAPTER 6

Simulation Tools for Conducting Performance Analyses

One of the key contributions of the research reported throughout this dissertation is the design of a comprehensive discrete event simulation framework to scrutinise the behaviour of (fault-tolerant) system models, and to conduct extensive performance analyses so as to assess the effectiveness of various types of policies for advanced redundancy management (research question RQ-3). In this chapter, we elaborate on the many predefined measures that are available out of the box, and that allow to report on the operational behaviour of system components, to analyse the performance of redundancy schemata from various perspectives, and, as such, provide insight on the system-environment fit or mismatch. We then highlight the key means that support the designer in modelling the behaviour and properties of the system itself, the entities held within, and the environment in which it is planned to operate. Other related research question(s): RQ-5.

The principal properties that contribute to system dependability were already listed and defined in Chapt. 1. But to what extent is a system really dependable in practice? How can we quantify how well a system is operating? Does it make sense to consider these properties in isolation? This section provides an overview of some useful measures and metrics that can be used to report and provide insight on the system-environment fit or mismatch, and on the performance of the system, from various perspectives, including dependability and timeliness.

In conducting rigorous performance analyses using discrete event simulations, the designer requires adequate tools to quantitatively assess and judge the observed behaviour while zooming in on specific system and/or environmental traits. Such assessments are usually made by *measuring* several directly observable values or performance aspects of interest — **measures**, that is — at specific stages of the simulation process. However, it rarely makes sense to consider these measures in isolation, for some are inextricably linked and may, considered together, ease the interpretation of simulation results.

While a measure expresses a way to obtain numbers or quantities, it is the associated **metric** that will put these values in perspective and give an actual meaning to the values obtained from measuring. Typical metrics in DES simulations are a (discrete) time unit, a number of resources (simulation entities and/or objects), a percentage or a real number where a higher value is indicative of a better score, and classic longitudinal and monetary measurements. A measure obtained for a specific performance attribute is specific to the environment in which the system was operating, and how that system was actually used over time. A change in environmental properties (including deviating load pattern) is very likely to yield another measure [14]. Furthermore, measurements can be used to ascertain the current system-environment fit. Indeed, the violation of one or more assumptions about the system's behaviour and properties, or about the environment in which it is operating, are very likely to cause some (but usually not all) measurements that can be flagged as outliers with respect to some predefined reference interval.

In this section, an overview is given of the various metrics that are available to the designer, as well as a description of how measurements are actually computed for each. The framework automatically keeps track of the evolution of these measures during simulation runs, and comes with facilities for automatically ranking system resources (*e.g.* replicas) and/or simulation runs and batches based on numerous combinations of metrics that the designer desires to monitor. The metrics defined here are merely a glimpse of those that are available out of the box as the simulation framework is shipped, to which the designer can easily add purpose-built metrics that may accommodate specific requirements.

Any type of information can be measured, both application/domain-agnostic and -specific properties. Measurements are **instantaneous** observations, and are computed and recorded based on the state taken by the system and the environment at a specific point in simulation time. Further correlation and analysis during or after the simulation is usually required, and multiple measures are typically recorded for the same set of metrics at various stages of the simulation. It may be useful to apply simple summation (*e.g.* to obtain the total number of voting rounds) or descriptive statistics (*e.g.* the average degree of redundancy being used). At times, specific algorithms can be used, an example of which can be found in the normalised dissent that was presented in chapter 4. Some may even perform correlation taking the simulation time into account when the individual measurements were recorded.

6.1 Measuring How Well Replicas Perform

Having pointed out the compositional nature of fault-tolerant redundancy schemata in Sect. 1.3.2.1, and motivated how their effectiveness is largely determined by the performance of the underlying versions (replicas) in Chapt. 3, we will start by covering some useful replica-specific metrics/measures that aid in the performance analysis of individual replicas.

6.1.1 Requesting Service: Decomposing the End-to-End Response Time

The recorded end-to-end **response time** of a request for service is the result of several constituents, each of which quantifies the time required to deal with the request from a specific perspective.

On the Role of the Queuing Model and Quality of Service A version is a network-accessible software component that is destined to provide a specific service and therefore placed in production by deploying it to a (corporate) IT infrastructure — *v.* (A41). Such infrastructure refers to all network and connectivity services, hardware appliances and middleware solutions required to realise the service that the deployed entity is expected to deliver. Whereas hardware appliances and message-oriented middleware provide the technical underpinning that is needed to realise the solution logic held within the software component, the service that is brought about should be accessible to the anticipated audience, hence the need for network connectivity services — *v.* (A06). Holding the actual solution logic, a version can be seen as a **processing facility**, where — much like a traditional **queuing system** — requests arrive, where they receive service, after which they depart — *v.* (A07) and (A42).

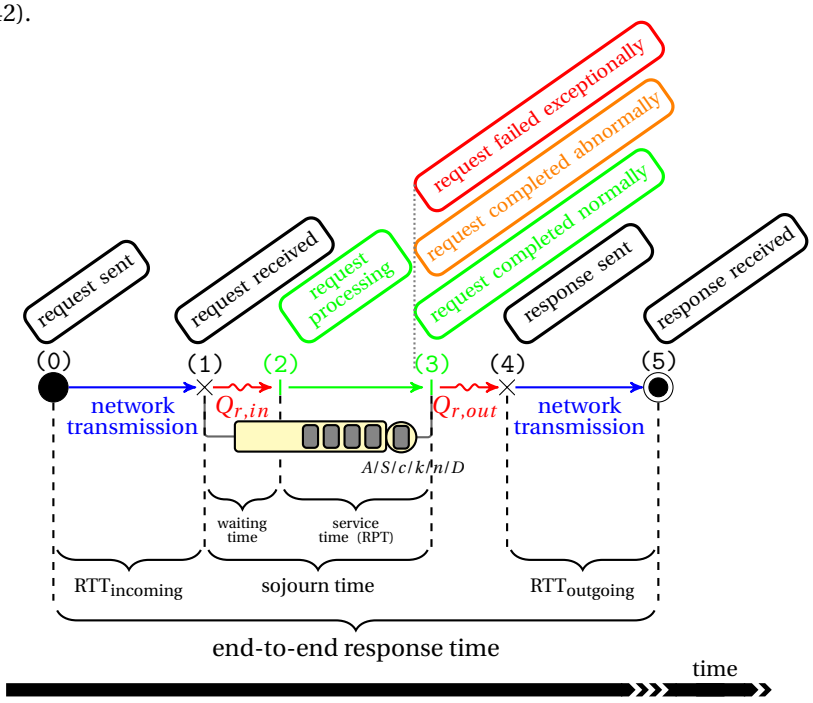


Figure 6.1: Decomposing the end-to-end response time for service invocation. Customisation of [45, p. 69 Fig. 3.1]. Above: the corresponding events in the discrete event simulation model shown in Fig. 2.2.

Queuing Models and the Kendall Notation Over the past 50 years, a lot of research has been done in the area of queuing theory, and many different types of queuing models with widely varying properties have been reported in the literature. The Kendall notation is a commonly used abstract notation that is available to theoretically classify queuing models, and to denote their analytical properties as a tuple $A/S/c/k/n/D$ [115, Chapt. 1]. More specifically, the notation allows to characterise the input or **arrival process** A, the **service time distribution** S, the number of available **server instances** c, the **buffer size** k, the size of the **calling population** n, as well as the **scheduling discipline** D.

A queuing model is essentially a processing facility, using a processing resource at its core to handle service requests. The processing facility corresponds to the actual system being analysed. Requests for service *arrive* at random times, eventually receive service, after which they are said to *leave* or *depart* from the facility. The (inter-)arrival times between successive request messages are characterised by the input or **arrival process** A, whereas the service (processing) times are characterised by the **service time distribution** S. In reality, request inter-arrival and service processing times are rarely constant, and various probability distributions have been tried for A and S in an attempt to accurately approximate realistic values [116]. At any time, at most c requests can be simultaneously serviced by the processing resource. Given this limited processing capacity, the facility includes a buffer — the **waiting queue** — in which at most k incoming requests may be temporarily kept on hold until a sufficient share of the processing facility's capacity becomes available. When the processing facility becomes available, the **scheduling discipline** D is responsible for deciding which deferred request will be popped from the waiting queue to be served next. Examples are a simple first come, first served (FCFS) scheduling policy, in which requests will be serviced in the order they were stalled and put on hold — *v.* (A07). When requests need to be serviced with different SLA levels, a priority-based scheduling discipline (PNPN) may prove useful.
Finally, the **calling population** n expresses how many potential software entities or actors may be requesting service, or — put differently — the cardinality of the set of all types of requests permissible in the facility.
Each parameter in the model characterises a trait of how the service is rendered, taken its deployment environment into account, and how it is exposed to and consumed by various actors and systems. No specific model is targeted throughout this chapter and the generic representation will be used as is.

Many contemporary distributed computing systems rely on message-oriented middleware for exposing the solution logic encapsulated within a software component to the outside world — *v.* Sect. 8.1. Such middleware solutions are often responsible for dealing with incoming request and outgoing response messages, for (de)serialising messages/data objects, and for scheduling and queuing these messages. Since it is not common to embed scheduling and queuing logic within the actual solution logic (separation of concerns), the queuing parameters k and D are generally determined by the middleware itself, its configuration and/or its feature set. More information about the role and type of queuing models in contemporary distributed computing systems can be found in Sect. 8.5.2.

When crafting the simulation model, the designer should select a suitable queuing model, for it will be decisive in the way the modelled resource — version, that is — will be dealing with requests.

The Constituents of the End-to-End Response Time As shown in Fig. 6.1, the end-to-end response time recorded for a single invocation of a specific version can be decomposed in several precisely quantifiable types of delays, resulting from the actual processing of the request, and overhead resulting from messaging and queuing activities. Having pointed out that a version is a network-accessible software component whose service is exposed and managed using a queuing model, we can now proceed by decomposing and defining the different terms that make up the actual end-to-end response time as it is perceived by the service requestor (QoS):

– Throughout this dissertation, the **round-trip time** is considered to be the time required for transferring the request and response messages to/from the processing facility. This time accounts for the time to transfer the initial request originating from the service requestor to the processing facility ($RTT_{incoming}$), as well as for the time to send a response message back ($RTT_{outgoing}$). Here, we assume that the service is accessible as an operation — a software routine — exhibiting a request-response message invocation pattern **(A36)** [117]. The RTT does *not* include the RPT at the processing facility, *nor* potential waiting times.

– Within the scope of a single service invocation request, the **sojourn time** is the time interval starting when the request message has arrived at the processing facility, up until the response is handed over to the message-oriented middleware and system runtime, the response message has been built, and is all set to be despatched to the initial service requestor. It is composed of two parts:

  + If all processing capacity is taken upon the arrival of a new request message, that request will be temporarily stalled, and kept waiting in a queue. It will be left in there until processing capacity becomes available and the scheduling policy D allows to resume the processing of the request at hand [118]. The **inbound waiting time** — denoted as $\mathtt{Q}_{\mathtt{r,in}}$ in Fig. 6.1 — is the time that the request has spent waiting in the queue before processing capacity was allocated to process it. At any one time, at most k requests can be kept on hold. When all buffering capacity is used, additional requests coming in are typically rejected.
    The time spent in the waiting state is determined by the interplay of several factors. Apart from exogenous factors like the influx of load, the main endogenous factors can be found in parameters that affect the total processing capacity being allocated. The number of **available server instances** c, as it is called, plays a part here. It is usually determined by the threading model applied in the middleware runtime or the solution logic. Many modern systems maintain a thread pool of a specific configurable size, in which each thread can handle a single request. This is covered in more detail in Sect. 8.5. Obviously, the longer the average service time, given a fixed amount of processing threads c, the less frequent a single processing thread will be relinquished given a constant arrival rate of requests, resulting in longer waiting times.

  + The **request processing time (RPT)**: the duration during which the request is actually being serviced by the processing facility. In discrete event simulations, RPT values are commonly generated by drawing random variates from the service time distribution S, and are therefore commonly referred to as request **service times**.
    The RPT may vary, and is determined by various endogenous and exogenous conditions. It is the immediate result of the parameters S and c, and the indirect result of the parameters A and n. It is dependent on the service time distribution S, which is directly affected by the computational complexity of the solution logic itself and the algorithms used to realise it, and the processing capacity of the hardware infrastructure on which the software component is deployed. Concurrent processing of multiple requests is

expected to result in longer service times. In addition to the limited capacity of the hardware, typical concurrency issues may occasionally cause idling, examples of which can be found in resource sharing and synchronisation overhead. Though the processing facility itself is shielded and protected by the waiting queue buffer, it are exogenous factors such as the arrival rate A and the number of service requestors n that will determine how many requests will need to be handled in parallel. In addition, this influx of requests is curbed by the inbound queue's capacity k.

– The inbound waiting time is the result from buffering newly arrived requests until part of the processing capacity has been relinquished to actually start processing them. Apart from this incoming queuing delay, we can also observe an **outbound waiting time** that accounts for all the overhead and delays to (i) serialise the result obtained from the solution logic, (ii) to construct a proper and syntactically correct response message, embedding the serialised response value, and (iii) to initiate the actual despatch to the intended recipient being the service requestor. These tasks are typically the responsibility of message-oriented middleware solutions. Due to the fact that they are highly optimised to achieve extreme levels of performance in dealing with these messaging-related tasks, this constituent of the response time is assumed to be negligible, and will not be taken into further account **(A37)**. It is depicted in Fig. 6.1 though and is marked as $\mathtt{Q}_{\mathtt{r,out}}$.

Client Overhead In considering the overall time required to handle a specific request, several client-specific preparatory and concluding activities will result in additional, albeit negligible, overhead to add to the end-to-end response time. Such overhead is also applicable to redundancy schemata, which take the role of client when requesting service from one or more versions:

– The **outbound client overhead** includes the overhead resulting from the activities needed to construct the request message, to serialise it and have it sent out. It is the time required to handle all of these activities before the request moves into state `(0)`, which precedes the end-to-end response time itself (Fig. 6.1). This can also be seen as the time it takes before an event of type `RequestSent` can be activated after the corresponding `RequestInitialised` event (Fig. 2.2).

– The **inbound client overhead** accounts for the overhead resulting from the deserialisation and parsing of the response message and its processing. In the case of an NVP redundancy scheme, this would include the voting and partitioning algorithm. Starting when the request has transitioned into state `(5)`, it succeeds the end-to-end response time, and its duration corresponds to the time needed before an event of type `RequestHandled` is activated after the corresponding `RequestSent` event (Fig. 6.1 and 2.2).

Throughout this dissertation, it is assumed the outbound and inbound client overhead is negligble **(A38)**. The proposed simulation framework does allow to apply different models though, and, in doing so, assess their impact on the performance of various types of redundancy schemata.

6.1.2 How to Measure Metrics in Discrete Event Simulations?

One of the core principles in discrete event simulations is that an event is scheduled to occur at a distinct (discrete) point in simulation time. By doing so, real time delays that would emerge from conducting experiments using emulation — using a software system that is actually put in production — can be avoided when conducting similar experiments using discrete event simulations. This advantage is commonly referred to as the principle of next-time advance, as elucidated in Sect. 1.4. Since the behaviour of the system and the environment in which it operates is modelled as a set of events, and a single event will cause the model to move from one state into another *instantaneously*, there is no need to actually wait for a transition to complete. Instead, it is sufficient to take note of the (virtual) simulation time at which the event was handled, and the delay that would be observed in the actual real-world system can be skipped over. That delay — essentially the time it takes for a transition to fire — can be approximated by computing the difference in simulation time based on the time stamps that were recorded for successive/relevant events.

The approach described here above is applicable to all **time-based measures**; values that express a duration relative to a base unit of time — be it (milli)seconds, minutes, hours or days. It is applicable in particular, but not limited to, the measures defined in this section. In Fig. 6.1, the relevant events that determine the constituents of the end-to-end response time are labeled `(1)` to `(5)`, each of which corresponds to an event of the model shown in Fig. 2.2 and described in Sect. 2.2.

For other types of metrics, which apply a base unit different than time, a specific measurement is computed and recorded for each event in addition to the time at which it was handled. This type of metrics requires to compare the recorded measurements, and maybe feed them into some algorithm for further analysis, rather than the time stamps associated to the relevant events. The *dtof* and normalised dissent are examples of measures that are computed using an algorithm — *v.* Chapt. 3 and 4. Less complex examples are measures for reporting, for instance, on the amount of redundancy used throughout a specific voting round. Such measures usually do not require specific algorithms to extract information from the system state, and can usually be deduced by counting a specific number of resources in the simulation model.

6.1.3 Time-based Dependability Measures

"Suppose that once a system becomes operational, it will take a certain time to fail again. The average time for the system to fail is called ***mean time to failure (MTTF)****. Once the system fails, it will take a certain time to recover from the failure and return to an operational state. The average time it takes for the system to recover is called* ***mean time to repair (MTTR)****" [45, Sect. 11.4]. Once recovered, the "system [...] will operate [, on the average,] for a time corresponding to the MTTF before encountering its next failure. The time between two failures is the sum of the MTTF and the MTTR and is the* ***mean time between failures (MTBF)****" [3, Sect. 4.2.5].*

Software entities are often perceived to be in a so-called *up* or *down* state [119]. Indeed they can be seen to alternate between periods of normalcy, during which they remain fully operational in line with the intended (non-)functional requirements,

and periods of abnormalcy during which they behave anomalously. Transitions between either type of period are triggered by the emergence of disturbances, and the dissipation of their effects:

- When an entity is placed into operation, it is assumed to initially perform correctly **(A39)**. Software is commonly assumed to age throughout its operational life span, and, by doing so, to become increasingly vulnerable to disturbances.
- when an entity is (has become) fully operational (again), the first emergence of a disturbance indicates the start of a period of anomalous behaviour;
- contrarily, when an entity is in a period of anomalous behaviour, it will resume its normal behaviour as soon as the effects of all disturbances have dissipated.

As a side note, the author would like to mention **software rejuvenation**, a proactive technique for recovery-oriented computing which, building on assumption (A39), has been identified as a cost-effective solution to overcome the effects of **software aging**. It does so by "periodically restarting software modules to flush out latent errors" [44]. Although this technique is not explicitly considered throughout this thesis, it may be applied whenever the proposed A-NVP algorithm indicates that a specific component performs inadequately, and that the component should (temporarily) be taken out of service.

A dual-state model effectively captures how the "life of the system is perceived by its users [...] with respect to the specified service" [12]: the system is said to be *up* when the service is being delivered as specified (**service accomplishment**), whereas it is said to be *down* whenever it deviates from what was specified (**service interruption**, or **degradation**). Quantifying the accomplishment-interruption alternations (transitions) in this model is helpful in measuring the system's effectiveness from a dependability perspective. Apart from time-based measures like the **mean time to failure (MTTF)**, **mean time to repair (MTTR)** and **mean time between failures (MTBF)**, it also supports key dependability measures like availability and reliability, which are expressed as a dimensionless, conditional probability or a percentage. In line with their definition on p. 9, the property of availability can be seen as "a measure of the [*current*] service accomplishment with respect to the alternation of accomplishment and interruption", and that of reliability as "a measure of the *continuous* service accomplishment (or, equivalently, of the time to failure) from a reference initial instant" [12].

The stochastic dual-state model is summarised in the upper right corner of the diagram below. It shows how two parameters λ and μ, which have commonly been used in the literature to approximate dependability-related measures, determine when the system goes into either of the *up* or *down* state: "the system fails, *i.e.*, goes from *up* to *down*, with a rate λ and gets repaired, *i.e.*, goes from *down* to *up*, with a rate μ" [45, Chapt. 11]. Here, the overall **failure rate** λ is to be seen as the expected number of disturbances that would manifest and affect the device or system and the service it is rendering per given time period. The overall **repair rate** μ corresponds to "the average number of **repairs** that occur per time" unit — essentially an approximation of how quickly the effects of a disturbance are expected to disappear [3, Sect. 4.2]. It is, however, quite difficult to obtain accurate approximations for these parameters, especially when dealing with complex failure occurrence and recovery/repair patterns as they are usually observed in distributed computing systems. And even if such approximations would be available, they fail

to indicate how the rate at which failures emerge and their effects dematerialise — *i.e.*, the system recovers or repairs from their effects — evolves throughout the system's **operational life**.

We will now clarify the relationship between the concepts of MTBF, MTTR and MTTF. In Fig. 6.2, the horizontal time line represents the operational life of a software entity; the origin at the left corresponds to the time the entity is placed into operation.

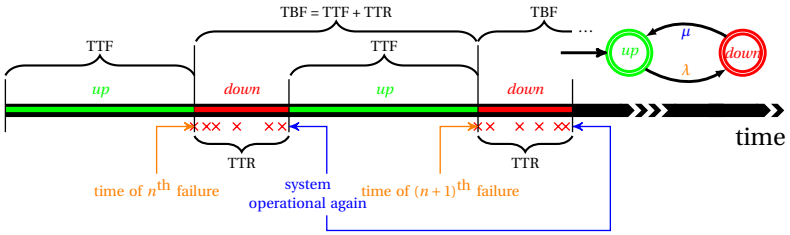


Figure 6.2: Relationship between MTBF, MTTR and MTTF. Reproduction of [3, Fig. 4.2] and [45, Fig. 11.2–11.3].

Mean Time to/between Failure/Repair For now, our primary interest is in the MTTF, MTBF and MTTR measures. Even though these measures have been found to serve well for quantitatively evaluating a system's dependability attributes, they have typically been defined with an implicit focus on mission-critical hardware appliances and digital circuitry, assuming that the system will be taken out of service as soon as the first failure has occurred, and that it will only be brought back into operation once faulty components have been replaced. Clearly, this is not in line with the focus of this dissertation, which is primarily on software systems and design faults. Disturbances emerging from such type of defects do not necessarily cease operation/execution indefinitely, and it is not always feasible to retrace a disturbance to one particular originating design fault, since its manifestation is usually also determined by various environmental factors. As such, for mission-critical (distributed) software systems leveraging redundancy schemata, faulty software components are deliberately kept running in the assumption that disturbances may be of transient or intermittent nature, and that a faulty component can recover and resume normal behaviour **(A40)**. This is supported by the expectation that no mission-cricital system will ever be placed in production, unless it is found to be sufficiently mature, and has been submitted to rigorous testing routines to detect and remove as many defects as possible.

Operational Life Once a software entity has been deployed, the operational life is the total time during which the entity is running and accepting requests for processing, starting from the moment the entity is launched, *i.e.* it is instructed to start its operations, until the system is shut down, *i.e.* it is explicitly instructed to cease its operations.

Furthermore, the system can also be brought to a halt implicitly, whenever a crash failure materialises. This will bring the system down and cause it to become permanently unresponsive and continuously exhibit omission failures — *v.* (A21).

From this point of view, the operational life has been commonly used as the time until the very first crash failure (**time to failure**) — in line with assumption (A08) that states that a fault immediately translates into a failure upon activation. Note how the system remains live despite potential transient and/or intermittent periods of degraded service that may result from other types of disturbances, *viz.* RVF, EVF and LRF failures.

Mean Time between Failure Occurrence (MTBFO) The **time between failure occurrence** is defined as the time between *any two consecutive* failure occurrences, both of which result in a disturbance affecting the availability of the software entity under consideration. Different from the computation of MTTF and MTBF values, every activation of a failure is accounted for, regardless of the state the entity is in at the time of activation (up or down) — *v.* (A08). The metric is useful in estimating how frequent latent software faults are triggered that remain within the implementation logic of the software entity, and in assessing the risk of service perturbation.

Mean Time to Failure (MTTF) The MTTF can be defined as the average duration of all periods throughout the system's operational life during which its availability is sustained in full and without interruption, *i.e.* it behaves in line with its (non-) functional specifications. In other words: the MTTF represents the average amount of time that the system has countinuously spent in the *up* state. It is approximated by taking the arithmetic mean of individual measurements, each of which represents the time that has passed between the start of a period of normalcy, until the very first failure occurrence that will make the system go *down* again. Note how only a subset of all potential failure occurrences is considered, as opposed to the MTBFO metric. In case of a burst of failures, only the first may affect the MTTF, and subsequent failures are typically assumed to be endured by the system as part of the MTTR.

Although one may find many references in the literature where the MTTF is approximated as $1/\lambda$, one should realise that such approximation is correct only when the system behaviour obeys the **exponential failure law**, which in itself assumes that failures occur at an approximately constant average rate λ [3, Sect. 4.2] [45, Sect. 11.4]. Put differently, the time between subsequent failure occurrences is exponentially distributed. Even though this assumption has commonly been used due to its simplicity and desirable properties, and to great success for the analysis of digital circuitry, it is inadequate to support detailed analysis of software systems, whose behaviour can be more accurately modelled using a time-varying failure rate function. The lognormal distribution has been found to capture well the overall behaviour of distributed software systems [82, 116, 120–126]. Furthermore, the failure rate can also depend on the type of fault, and its properties in terms of duration and recurrence. For intermittent failures in particular, these properties can be precisely approximated using a Weibull distribution [127, Chapt. 2] [112, Sect. 3.4] [128, Chapt. 13] [129].

During simulation runs, the described discrete event simulation framework will collect all information that is needed in order to compute exact measures for the MTTF. This allows the designer to use whatever desired combination of distributions (if any) to inject disturbances into the system.

Mean Time to Repair (MTTR) The MTTR can be quantified as the average duration of all periods throughout the system's operational life, each of which is characterised by a prolongment of unavailability, as the system continuously fails to meet the objectives defined in its (non-)functional specifications, resulting from the occurrence of disturbances. Put differently, it is the average time delay, starting from the moment the system becomes unavailable, to the earliest point in time when the system resumes it operations in full. This corresponds to the time it takes for the system to recover, as can be quantified by the time between two successive transitions from *up* to *down*, and vice versa.

As long as the system stays *down*, it will fail to successfully process and handle requests; due to disturbances, all of them will treated in such a way that the service will exhibit faulty behaviour, either by violating the functional requirements, or by violating the non-functional requirements — *cf.* Sect. 2.6 [1]. Since a software system cannot subsequently recover from crash failures, the occurrence of such type of disturbance essentially characterises the end of a system's operational life — *v.* (A21). Since the MTTR was defined with respect to a system's operational life, only measurements indicating the actual recovery latency are taken into account.

In practice, the MTTR depends on various endogenous and exogenous factors. It depends on the fault recovery mechanism used by the system itself, and operational matters like the maintenance schedule, monitoring systems, the deployment infrastructure, and the availability and location of spare components (on-site *vs* off-site) [2, Sect. 3.3] — *v.* Sect. 1.3.

The MTTR is usually approximated as $1/\mu$, given an overall repair rate μ. It is, however, "extremely difficult to estimate, and is often determined experimentally" to ensure accuracy [3, Sect. 4.2]. The described simulation framework has been implemented in line with this recommendation.

Mean Time between Failure (MTBF) The MTBF is the average duration of all periods throughout the system's operational life, each of which starts whenever the system suddenly becomes unavailable, extending beyond the resumption of its normal operations, before it ultimately stops when it is about to become unavailable once more. Each such period is immediately proceeded by a period during which the system behaved correctly and the availability of the service it seeks to provide is sustained in full. Each such period includes "any time required to repair the system and place it back into an operational status" [3, Sect. 4.2]. In other words, the MTBF is calculated by dividing the operational life by the number of transitions from the *up* to the *down* state that can be observed throughout. Or, more intuitively, within the operational life, it is expected that MTBF = MTBF + MTTR, as is illustrated in Fig. 6.2 — *cf.* assumptions (A09) and (A10).

[1] Despite assumptions (A09) and (A10) that state that — with the exception of crash failures — all effects of a disturbance immediately dissipate, the corrupted results of affected requests may be delivered to the requesting party with some reasonable delay. This is because of middleware messaging and queuing overhead, and network transmission delays — *cf.* Fig. 2.1 and 2.2. In case the requesting party is a redundancy scheme, it may still perceive anomalous behaviour.

6.2 Redundancy Schemata & Performance Measures

As was illustrated in Fig. 1.5, a knowledge source is to be found at the core of any autonomic computing software system. Being autonomic software units *in se*, such architectural model is also applicable to adaptive redundancy schemata, which rely on a dedicated context-aware management component — *v.* Fig. 5.2. Such type of component takes the role of knowledge source and is responsible for collecting and managing runtime data that is used to compute various types of metrics. Precisely how well the software entity will perform depends on the selection of metrics that are monitored, and the quality of the collected measurements.

Furthermore, adaptive redundancy schemata aim to autonomously adjust the employed redundancy configuration by allocating the available redundancy capacity in the system only in part, and selecting only those versions that are expected to contribute most to the scheme's overall objectives — *cf.* Sect. 5.1. This allocation is controlled by specific redundancy dimensioning and selection models into which context-aware information is fed — *cf.* Sect. 5.2 and 5.3. Unlike classic predefined redundancy schemata, in which a static set of versions is used, dynamic redundancy configurations may vary throughout time, and so can the selection of the versions available in the system.

6.2.1 General Measures

Regardless of whether a single instance of a context management component is used for multiple redundancy schemata, or multiple, dedicated instances are used for each individual scheme, all collected measurements are linked to a specific operational context: a specific fault-tolerant composite $\mathcal{C}$. Even though they may not seem very significant, the following basic measures should always be monitored, for they will put various other types of measures in perspective, thereby facilitating the analysis of how well adaptive redundancy schemata perform, primarily from a dependability and resource expenditure perspective:

– The **total number of voting rounds** $\ell^{\mathcal{C}}_{total}$ that have been simulated and analysed throughout a particular simulation run. This indicates how many times service was requested from the redundancy scheme. The emphasis is placed on the fault-tolerant composite $\mathcal{C}$ itself rather than the underlying versions.

– The total number of voting rounds #**rounds**$(\mathcal{C}, \mathbf{v})$ in which a specific version was part of the redundancy configuration and, as such, contributing to the overall outcome of a specific scheme $\mathcal{C}$ — *v.* p. 77. Measurements are maintained for each version in the system. Absolute values collected for this metric are often put in perspective and expressed as values relative to the total number of voting rounds $\ell^{\mathcal{C}}_{total}$. Obviously, $\#rounds(\mathcal{C}, v) \leq \ell^{\mathcal{C}}_{total}$.

– When analysing the performance of a redundancy scheme using a redundancy dimensioning model in which a safety margin c_{sm} is enforced, the simulation framework will compute $\ell_{\mathbf{suspect}}$: this measure corresponds to the number of voting rounds for which an outcome could be adjudicated, though the degree of consent in addition to the required absolute majority did not exceed the imposed margin, *i.e.* $0 \leq c^{(\mathcal{C},\ell)}_{max} - m^{(\mathcal{C},\ell)} < c_{sm}$ — *v.* p. 82 and 84. Applying a safety margin is

useful for applications that should exhibit highly dependable characteristics, and measurements recorded for $\ell_{suspect}$ are indicative of intervals in the operational life of the investigated scheme in which the degree of consent is suspiciously close to the minimum degree of consent $c_{max}^{(\mathcal{C},\ell)}$ required for the system to survive. The measure is usually compared to the total number of voting rounds $\ell_{total}^{\mathcal{C}}$.

The proposed A-NVP algorithm relies on the availability of context information. These data structures that are typically maintained by a dedicated context manager component included in the NVP FCU — *cf.* Sect. 5.2.1 and 5.4. The data held within is analysed and used to adjust the redundancy configuration, and is essential for the functioning of the redundancy dimensioning and replica selection models — *cf.* Sect. 5.2 and 5.3. Throughout the operational life of a given redundancy scheme, several measures can be recorded based on the information held within this so-called **window of context information**, which may provide further useful insight in the overall performance of the proposed algorithm:

– To be able to make an informed decision and adequately dimension the optimum level of redundancy requires the availability of high-quality and complete context information. The data structure shown in Fig. 5.1 is updated upon each subsequent completion y of a voting round $\ell = b_{\mathcal{C}}(y)$; only then the information for updating the context knowledge can be deduced. Whereas the data held within this data structure is updated as soon as voting rounds have completed, the decision as to what level of redundancy is to be applied for subsequent voting rounds is determined based on the collected information available at the time these new rounds are initialised. The order in which voting rounds complete does not necessarily correspond to the order in which they have been initialised. Such mismatch is the result of the arrival pattern by which new requests hit the redundancy scheme and the degree of variance that is noted with respect to the time required to handle them (which in turn is mainly determined by the end-to-end response time of the subordinate version invocations).
  Partially incomplete context information may result in suboptimal redundancy dimensioning. Such situation is referred to as a corrupted state of the window of context information. The total number of voting rounds during which the level of redundancy was determined based on incomplete context information is denoted as $\square_{\textbf{corrupt}}$. The measure is commonly expressed as a percentage relative to the total number of voting rounds $\ell_{total}^{\mathcal{C}}$.
  Window corruption is verified at the start of each voting round. To do so, a technique based on triangular numbers can be applied, with the n^{th} tringular number T_n defined as the sum of all natural numbers $k \in \mathbb{N}^+$ such that $T_n = \sum_{k=1}^{n} k = {}^{n(n+1)}/_{2}$. Consider the state of the cached context information when a new voting round $(\mathcal{C},\ell)$ is being initiated, and let ℓ_{min} and ℓ_{max} represent the smallest, respectively largest voting round identifier value. Then all context information is present in the window if $T_{\ell_{max}} - T_{\ell_{min}} = \sum_{k=max(1,z-r_d+1)}^{z} b_{\mathcal{C}}(y_k)$. More intuitively, this condition will verify that a sample of context information is cached for all of the last $min(z, r_d)$ voting rounds whose initialisation preceded that of round $(\mathcal{C},\ell)$. In doing so, the insertion (round completion) order of the samples does not matter. Even if the data structure were sorted based on the voting round identifiers, *e.g.* using a priority queue, gaps may remain for rounds

that are pending.
Note that applying a sequential arrival pattern for the arrival of requests at the fault-tolerant composite — or, in other words, voting round initialisations — where voting rounds are handled one after another, will always correspond to $\square_{corrupt} = 0$ — *cf.* p. 109.

– Another useful measure is $\square_{\mathbf{fill}}$, denoting how many samples were present at the end of each voting round. It is used to analyse how quickly the maximum window capacity r_d is actually allocated. The closer measurements would approximate r_d, the more information is available to make an informed decision on the degree of redundancy to be used in subsequent voting rounds. The default context manager implementation will collect measurements and report mimimum and maximum values, which should always lie within the interval $[1, r_d]$. It will also report the average value, the observed standard deviation as well as information about the confidence interval.
Eventually, for values $\ell^{\mathcal{C}}_{total} \geq r_d$, $\square_{fill} = r_d$. However, if the pattern in which requests arrive at the fault-tolerant composite would lead to many voting rounds being concurrently handled, in particular during the early phases of the scheme's operational life, it may take longer before the total window capacity is completely allocated — which the designer can observe from the reported metric data.

– A variation of the previous measure is used to zoom in on the size of the upscaling view maintained on the current state of the context window. This view comprises the last $\{y_{min(1,z-r_u+1)}, ..., y_z\}$ samples of the overall context window, and corresponds to periods during which the scheme is temporarily deferring an anticipated upward adjustment of the currently applied level of redundancy. Such scenario would typically occur upon the insertion of an observational sample holding the contextual information acquired from a completed voting round for which an outcome could be adjudicated, yet in spite of a qualified majority, the consentient ballots to spare fell below the imposed safety margin c_{sm}. Another scenario would be a voting round for which a majority could not be established. The following statistical data is collected and available: upper and lower values observed, arithmetic mean and standard deviation, as well as confidence intervals.

6.2.2 From a Dependability Perspective

Often expressed as a percentage of the total number of simulated voting rounds $\ell^{\mathcal{C}}_{total}$, the total number of voting rounds for which the scheme failed to adjudicate an outcome is denoted as $\ell^{\mathcal{C}}_{\mathbf{failure}}$. In n-version programming, such situation would occur when an insufficient level of congruence between the acquired ballots inhibits the adjudication of a majority. More formally: whenever $c^{(\mathcal{C},\ell)}_{max} - m^{(\mathcal{C},\ell)} \leq 0$ — *cf.* Sect. 3.1.2. The measure gives an estimation of how frequently the redundancy scheme has been unavailable, and failed to successfully render the service it was expected to.

In addition, one may discern the following replica-specific measures:

– The (relative) number of those voting rounds which were accounted for in $\#rounds(\mathcal{C}, v)$ for which v contributed to the majority is denoted as $\#\mathbf{consent}(\mathcal{C}, \mathbf{v})$ — *cf.* Sect. 4.3. Obviously, $\#consent(\mathcal{C}, v) \leq \ell^{\mathcal{C}}_{total}$.

– Within the scope of a particular voting round ℓ, whenever an engaged version v_i is affected by some type of disturbance while processing a service request $\langle \mathcal{C}, \ell, i \rangle$, that version is expected to be in dissent with the majority established for that round, if one can be found at all — *cf.* Sect. 4.3 and (A31). The number of voting rounds during which a specific version v did not contribute to the adjudication of an outcome of the redundancy scheme can be computed as $\#rounds(\mathcal{C}, v) - \#consent(\mathcal{C}, v)$, and should not exceed $\ell^{\mathcal{C}}_{total}$. This number is often expressed as a value relative to the total number of voting rounds $\ell^{\mathcal{C}}_{total}$, or relative to the number of voting rounds in which the version was included in the scheme's redundancy configuration — $\#rounds(\mathcal{C}, v)$, that is.

– The **normalised dissent** $D(\mathcal{C}, v)$ was introduced in Chapt. 4 as a replica-specific metric that represents how well a given version v contributed to the sustained availability of a redundancy scheme $\mathcal{C}$ throughout the scheme's operational life span. Measurements are recorded for all versions V every time a voting round has completed (to be precise: when it has transitioned into state `(d)` — *v.* Fig. 2.1). The metric is used as an alternative to approximate the actual reliability, and the evolution of the measurements recorded for a specific version should match the version's susceptibility to disturbances.

All of the time-based dependability measures introduced in Sect. 6.1.3 are collected for the fault-tolerant composite itself, as well as for all of the versions used by the underlying redundancy configuration. These measures are usually expressed as real-world (continuous) time intervals. However, as the emphasis is placed on the sustained availability of the fault-tolerant composite as a whole, the proposed simulation framework will compute measurements expressed as a number of consecutive voting round invocations, rather than expressing them in units of discrete simulation time. Individual updates of measurements can be obtained at the end of each voting round, while honouring the order in which voting rounds have been initialised. That being said,

– The **time to failure (TTF)** of a given redundancy scheme corresponds to the total number of voting rounds until the scheme (first) failed to produce/adjudicate an outcome — *cf.* (A31). Similarly, the TTF for a specific version corresponds to the total number of voting rounds during which that version sustained the schema's availability in full, until it was affected by some disturbance that caused it to behave anomalously — *cf.* (A30).

– In addition to that, the framework is shipped with support to compute the XTTF, or the time until the x^{th} failure, where x can be set at will. Whereas basic TTF measurements fail to represent the reliability of the scheme and its underlying versions during the early phases of its operational life, the XTTF may provide more insight in the degree of intermittence between various failure occurrences.

– The **mean time between failures (MTBF)** was defined before as the average duration of all periods throughout a software entity's operational life, each of which starts whenever the system suddenly becomes unavailable, extending beyond the resumption of its normal operations, before it ultimately stops when it is about to become unavailable once more. Inituitively, this can be seen as the

average amount of time that a software entity has spent in the *up* state throughout its operational life. Similarly, the **mean time to repair (MTTR)** was defined as the average duration in between two successive periods of continuous fault-free behaviour, or the average amount of time the entity spent in the *down* state. Discrete values can be used to indicate the extent of the theoretical, time-based measurements. Values can be determined for each replica that is part of the available redundancy capacity, or for the redundancy scheme itself:

+ For a given redundancy scheme, consider a chart in which the horizontal axis represents a timeline in which voting rounds are represented in the order in which they have been initialised. Then all of those rounds are marked (plotted) for which the given version was affected by one or more disturbances. For that given version, the MTBF can then be approximated as the average number of voting rounds during which it exhibited fault-free behaviour in between two successive rounds during which it exhibited faulty behaviour. Or, more formally, by averaging the overall length of all the applicable subsequences in $\{\ell_x\}_{\mathcal{C}}$.
  Similarly, the MTTR can be approximated by computing the average length of windows of transient unavailability. Or, put differently, the repair period corresponds to the number of successive voting rounds during which the version uninterruptedly exhibited faulty behaviour.
  These replica-specific measures are computed based on the failures that are actually observed; there is no relation with the contribution of the version — its ballots — to the actual outcomes adjudicated by the decision algorithm.
+ For a given redundancy scheme, such estimation can be obtained as the cumulative number of voting rounds covered by all intervals during which the scheme's availability was sustained in full (MTBF), or during which it showed to be unavailable for a longer period, albeit transient in nature (MTTR). As criterion to determine if the scheme is (un)available, it is sufficient to check that the underlying decision algorithm was able to adjudicate an outcome at the end of a particular voting round, *i.e.* $dtof^{(\mathcal{C},\ell)} > 0$ — *cf.* (A32).
  Measurements are computed in two different ways: honouring the initialisation order in which voting rounds have been initiated, as well as the order in which voting rounds have completed. The latter mechanism was introduced since, depending on the request arrival rates and the variance of the overall RPT, voting rounds do not necessarily complete in the same order they were initialised — *v.* p. 74. It actually makes sense to do so, since both measures primarily intend to capture information with respect to (the detection of) failure occurrences, and failures will only be detected at the end of specific voting rounds — *cf.* (A06).

6.2.3 From a Timeliness Perspective

Latency-sensitive applications typically come with the non-functional requirement of being able to render a service within a predefined time limit — *cf.* Sect. 5.1. Such requirement should be taken into account while determining an appropriate redundancy configuration, and a balance should be found between a fair distribution of the load across the available redundancy (versions) and the delay in acquiring an outcome. For adaptive redundancy schemata to be able and to attempt to

meet such type of application-specific objective requires an indication on the responsiveness of the individual versions available in the system. The proposed discrete event simulation framework is shipped with a default context manager implementation that is capable of monitoring the **end-to-end round-trip time** of version invocation requests and all of its constituents as listed in Sect. 6.1.1. Outbound as well as inbound client overhead can, optionally, be modelled and monitored as well. All of these metrics are collected specifically for all versions operating in a given redundancy scheme, and the mean as well as the standard deviation are computed based on the recorded version-specific measurements.

6.2.4 From a Resource Expenditure Perspective

In Sect. 1.5, we argued that static redundancy configurations, or the use of a higher degree of redundancy, do not necessarily yield a higher degree of dependability. Even though a sufficiently high degree of redundancy is, without doubt, an essential requisite to maximally sustain the availability of redundancy schemata, the available redundancy should be managed carefully and well-consideredly, and be sparingly allocated. This way, part of the additional cost in using excessive levels of redundancy could be economised — *cf.* Sect. 3.3. The following metrics are measured throughout the simulation of voting rounds:

- Honouring the constraints imposed on the applied degree of redundancy consumed within a given redundancy scheme $\mathcal{C}$, where $n_{min} \leq n^{(\mathcal{C},\ell)} \leq n_{max}$, the **total amount of redundancy** units consumed over a simulation run is denoted as $\sum n^{(\mathcal{C},\ell)}$, $\forall \ell \in \{\ell_x\}_{\mathcal{C}}$ — *cf.* Sect. 5.1. Intuitively, a single unit of redundancy can be seen as a single invocation $\langle \mathcal{C}, \ell, i \rangle$ of a single version v_i. Measurements will give an indication how extensively the available redundancy — that is the finite set V of functionally-equivalent versions available in the system — was actually used.
- Within the context of a given voting round $(\mathcal{C}, \ell)$, the **contextual redundancy** was defined in Sect. 3.1.3 as the minimum degree of redundancy required to counterbalance the effects caused by a given number of disturbances $e^{(\mathcal{C},\ell)}$. The measure can be used to determine if the employed degree of redundancy $n^{(\mathcal{C},\ell)}$ was (in)sufficient, and to indicate redundancy under- or overshooting — *cf.* Sect. 3.1.2. Accumulating these measurements for all rounds $\ell \in \{\ell_x\}_{\mathcal{C}}$ gives a clear indication on the bare minimum of redundancy (units) that would have been required to guarantee the scheme's availability under comparable conditions (identical types of disturbances with a similar failure occurrence pattern).
- Although the result is surely not indicative of a scheme's overall reliability, it is useful to put the total redundancy consumption $\sum n^{(\mathcal{C},\ell)}$ in perspective. Nonetheless, both measures should be compared bearing into account the number of voting rounds $\ell^{\mathcal{C}}_{failure}$ during which disturbances could not be successfully masked. This is because a poor redundancy dimensioning strategy could result in redundancy undershooting, which might result in a total amount of redundancy $\sum n^{(\mathcal{C},\ell)}$ very close to the cumulative contextual redundancy $\sum cr(e^{(\mathcal{C},\ell)})$[2].

[2] When computing contextual redundancy measurements, the designer may choose whether or not to take strict LRF failures into consideration. Strict LRF failures are disturbances that affect a specific scope — version invocation $\langle \mathcal{C}, \ell, i \rangle$, that is — that is not affected by other, more severe types of disturbances — *cf.* Sect. 2.3.

In order to maximally safeguard the dependability of a redundancy scheme, versions that are affected by disturbances while processing requests originating from the scheme should be marked and considered as suspicious, and should be closely monitored. Ideally, any suspicious version is directly taken out of service, keeping only those other versions in service that exhibit near-perfect behaviour as specified. Obviously, this is not always possible, particularly in case the available redundancy is limited, as it usually is. After all, each additional implementation incurs a significant cost for its design and development, as well as increased infrastructural and operational costs. Furthermore, a limited redundancy might necessitate to recover previously excluded versions when there is a sudden need to scale the redundancy up. The algorithm introduced in Chapt. 5 was designed with these concerns in mind. It is used to manage the redundancy configuration employed within redundancy schemata, and to remove or add versions as appropriate, either taking suspicious, poorly performing replicas out of service, or integrating previously suspected replicas as soon as it is deemed safe to do so. But how long will we monitor a version marked as suspicious before acting and taking it (temporarily) out of service? And how much damage can it cause during this interval? How frequent do we reach out to previously disabled redundancy to bridge periods in which an unusually high number of disturbances are observed? The following metrics may help to understand. They are collected specifically for all versions operating within the context of a given redundancy scheme, and the mean as well as the standard deviation are computed based on the recorded version-specific measurements:

– The term **exclusion latency** is used to denote a period corresponding to a specific subsequence of $\{\ell_x\}_{\mathcal{C}}$, starting from a voting round during which the version was first affected by a disturbance since it was used in the scheme's redundancy configuration, and ending with the last round in this sequence, followed by a round during which the version was immediately excluded. When the version is hit by a burst of disturbances, the first will indicate the start of the interval, and all subsequent disturbances that occur before the first exclusion will be considered as exclusion failures. The exclusion latency indicates how long a faulty replica that was marked as suspicious, was kept in service, before it was actually excluded. It is expressed as a number of voting rounds and is indicative of the uncertainty that the continued use of the faulty version causes, as it may have a negative effect on the effectiveness of the employed redundancy configuration. The latency depends on the fault detection effectiveness and how severely replicas are penalised: higher penalty values may be expected to result in shorter latency windows — *cf.* Chapt. 4.

– During the exclusion latency, the applicable version is kept in service, but it may suffer from additional disturbances. That would incur penalties to be inflicted in updating the corresponding normalised dissent measurements, as it clearly threatens the reliability of the redundancy scheme as a whole — *cf.* Sect. 4.2. An **exclusion failure** is a failure of a given version during a specific voting round that is part of the exclusion latency. Since only a single type of disturbance — the most severe type — is considered to affect a version within the scope of a specific voting round, the exclusion failures cannot exceed the exclusion latency itself.

– The **re-integration latency** is the delay, expressed as the number of consecutive voting rounds, between the moment when a specific version had been excluded

from the scheme's redundancy configuration, until it was once again selected and engaged during a subsequent voting round. Throughout this period, the version was not used in the scheme's redundancy configuration. How quickly an excluded version will be re-enabled in a redundancy configuration depends on various factors. In case the version was taken out of service because it had been affected by one or more disturbances, there is a direct relationship with the applied reward factor (lower values would translate in shorter latencies) — *cf.* Chapt. 4. However, the recorded latencies will also depend on how well or poorly the other versions in the system have been performing, and to what extent their observed performance is in line with the scheme's endeavoured objectives.

– When a specific replica is properly functioning and contributing to the sustained availability of the redundancy scheme, there is no specific need to exclude it from the scheme's redundancy configuration (although it could be in case the used level of redundancy is reduced, and that version shows to be performing relatively poorly with respect to the other versions participating in the redundancy configuration). A **successive usage period** is a sequence of consecutive voting rounds during with a specific version was engaged and actively used within a given scheme's redundancy configuration. Put differently, a single such period corresponds to a specific subsequence of $\{\ell_x\}_{\mathcal{C}}$, and ends at the last round in this sequence, followed by a round during which the version is excluded.

Apart from the above, the context manager will also maintain replica-specific **load statistics**. Based on the metrics defined in the MoWS specification, the context manager will keep track of three attributes that help to characterise throughput levels: `NumberOfRequests`, `NumberOfFailedRequests` and `NumberOfSuccessful Requests`. More information on this specification can be found in Sect. 8.2.4. Measurements are acquired after the receipt of notification messages, which imply a potential delay before the metrics are updated — *cf.* Sect. 2.7 and 3.4, and (D04). Subject to these delays, the currently known **pending load** $L(\mathcal{C}, v)$ associated with a version v is determined as `NumberOfRequests` – (`NumberOfFailed`– `Requests` + `NumberOfSuccessfulRequests`), which should approximate the current number of pending requests. Note how update notifications are issued at various stages of the version invocation state transition model; measurements are expected to be updated in reasonable time — *cf.* Fig. 2.2.

Finally, the degree of redundancy employed within adaptive and autonomous redundancy schemata is likely to vary throughout the scheme's operational life — *cf.* Fig. 3.1. Each adjustment to the level of redundancy being used corresponds to what is referred to as a **redundancy event**. For dynamic redundancy configurations, it is useful to compute the following two measures:

– The total number of events during which the degree of redundancy was increased (scaled up) and how many of these events resulted in a redundancy level (configuration) that proved sufficient to sustain the scheme's availability. Whereas the former is denoted as $\triangle_{\textbf{total}}$, the latter is denoted as $\triangle_{\textbf{success}}$.

– Similarly, $\triangledown_{\textbf{total}}$ denotes the total number of events during which the redundancy was brought/scaled down, and how many of these events resulted in a redundancy configuration that proved to be inadequate to sustain the scheme's availability and thus resulted in one or more failures ($\triangledown_{\textbf{failure}}$).

– Another useful measure is the number of times $\triangle_{\textbf{bump}}$ the system intended to scale the current level of redundancy up, but failed to do so because of the imposed upper limit n_{max}. Measurements may help to assess if the total redundancy capacity is adequate to support bursts of failures. Assuming that the available redundancy is efficiently allocated and used, it is likely that each such type of redundancy event corresponds with a failure of the redundancy scheme itself.

All of the above are often expressed as a percentage of the total number of redundancy events observed for each of the directions in which the redundancy level can be adjusted (up or down, respectively $\triangle_{total}$ and $\triangledown_{total}$). They provide an indication as to how effective the redunancy dimensioning and replica selection models were to proactively reach out for spare redundancy capacity in case (there is a risk) of redundancy undershooting, or economise on using excess redundancy in case of redundancy overshooting. Obviously, as no such type of events would be observed for traditional, static redundancy configurations, both measure will evaluate to 0.

6.3 Modelling the Environment

Another thing that is needed are tools for the designer to exert control over the environment in which the system is operating, and the effect it may have on the system, its behaviour and performance. At times, one may wish to analyse the essential characteristics of the system under normal operating conditions, in order to assess if and how the performance of the system can be improved. Such type of analysis often calls for the ability to (temporarily) suppress any undesired behaviour that could result from disturbances affecting the system itself, or the infrastructure it relies on (in particular network connectivity services and hardware and middleware components). If not, it would be very useful if the designer were able to analyse and/or reproduce specific conditions, in particular when catastrophic failures need to be examined ex-post. This would help to comprehend and reveal the exact root cause, required to harden the system and improve its resilience. Discrete event simulations are particularly helpful to support the designer in deterministically reproducing a specific scenario of correct or erroneous behaviour.

Apart from being able to control the occurrence of disturbances — a subject that will be touched in the next section — the following parameters play a key part in ensuring the determinism of discrete event simulations:

– Being processing facilities *in se*, the characteristics of a software entity can easily be denoted in Kendall notation. One of the key attributes included in such type of representation is the **arrival process** A: a property that is used to characterise the influx of new requests arriving at the facility. This is no different when considering fault-tolerant redundancy schemata. However, a distinction needs to be made between the actual scheme itself, and the underlying versions:

  + When considering an **individual version** v_i, the load imposed upon it is the result of two types of requests. One can observe requests that were instantiated within the context of a specific voting round. This type of requests are denoted as $\langle \mathcal{C}, \ell, i \rangle$, where $\mathcal{C}$ represents the redundancy scheme (the client, *v.* p. 6) that issued the request, and this within the scope of a specific voting round ℓ. A

version can also be requested to deliver its service by third-party components and/or system actors other than the redundancy scheme itself. This second category of requests — collectively referred to as **external load** and denoted as $\langle \perp, \perp, i \rangle$ — are issued independently of the redundancy scheme in whose redundancy configuration the version may be engaged – *v.* p. 64.

+ The simulation framework allows the designer to plug in a specific model that reflects a desired arrival process of external load. Such model can optionally be defined for a specific version, and is used to inject stand-alone requests in the background as the simulation progresses. The mechanism relies on the generation of inter-arrival times, so that requests can be easily injected sequentially — *v.* p. 43.
  Subordinate requests will arrive in function of the creation and handling of voting rounds; no further configuration is possible at replica-level.
+ When a request arrives at an NVP-based **redundancy scheme**, it will lead to the instantiation of a new voting round. The simulation framework allows the designer to plug in a specific model that reflects the desired arrival process. The designer can choose to define a time-based inter-arrival model, or start new voting rounds in sequence. This allows to control whether different voting rounds may be handled (partly) in parallel.

– The second most important parameter in Kendall notation is the **service time distribution** *S*. Such distribution is often used to sample RPT values. More formally, as illustrated in Fig. 6.1, this corresponds to the time during which a request stayed in state `(2)` before transitioning into state `(3)`.
  The simulation framework allows the designer to plug in a **version-specific model** that reflects a desired service time distribution. No specific model is considered for **redundancy schemata**, since the actual RPT of voting rounds is largely determined by the RTT values recorded for the subordinate version invocations.

– The author believes there is little use in modelling specific **queuing overhead** for the A-NVP composite and/or individual versions, in view of assumptions (A07) and (A38). Indeed, in handling subordinate version invocations, the end-to-end response time is mainly composed of the RTT and RPT, and the (outbound) **waiting time** and **inbound/outbound client overhead** usually do not significantly contribute.
  Furthermore, when performing simulations, usually only a single redundancy scheme is instantiated, and it does not make sense to consider additional queuing overhead. This is mainly motivated because the end-to-end response time for a specific voting round is largely determined by the end-to-end response times recorded for the subordinate version invocations. Additional latencies that may result from queuing-specific complexity at the level of the fault-tolerant composite are usually negligible or irrelevant for the objectives that one hopes to achieve by means of simulation. Nevertheless, should the designer disagree, specific properties of the environment on which these software entities have been deployed can easily be configured by associating a specific model (implementation logic) to the relevant entities — *cf.* Fig. 6.1.
  At the level of the redundancy scheme, the outbound client overhead can be

modelled — *v.* p. 94.
For versions, one can define the number of server instances c as well as buffer size k, the scheduling discipline D. The simulation framework will (re)schedule the relevant events, thereby applying the defined queuing model. Optional version-specific configuration allows to specify the extent of the overhead resulting from the enveloping, serialising and requesting transmission of (i) syntactically valid response messages (no EVF failures occurred), (ii) error messages (in case of EVF failures or omission failures), or (iii) counter update messages — *cf.* Sect. 2.7[3].

– Adequate models used to sample network transmission delays can be specified for individual version instances. The reason this type of models are associated to versions is that the network topology can be modelled in a modular way, and the latencies resulting from the route to/from the (single) instantiated redundancy scheme can be defined. These are used to determine the RTT for subordinate version invocations, encompassing the time required for transferring the request from the scheme to the applicable replica, and sending the resulting response back. The default implementation shipped with the simulation framework does not assume any specific network topology.

– Finally, versions may be affected by all sorts of faults during their operational life, and disturbances can therefore be observed while processing requests. Although the primary interest lies on software design faults, it may be useful to model the behaviour of the deployment environment too, in particular the reliability of the underlying hardware and middleware. In the context of discrete event simulations, contructing and applying suitable fault models is closely related to the technique of failure injection, which will be covered in the following section.

6.4 Failure Injection Mechanisms

Failure injection is a technique commonly used in software testing, and is typically used to improve "the coverage of a test by introducing faults to test code paths, [and to analyse and improve the effectivenes of] error handling [routines in particular] that might otherwise rarely be followed" [130]. When analysing the effectiveness of redundancy schemata using discrete event simulations, the technique is helpful, in that it allows the designer to exert control over the number and frequency of failure occurrences — the number of observed disturbances, that is.

Defining a suitable fault model is an intrinsic aspect of modelling the environment in which redundancy schemata operate — *cf.* Sect. 6.3. It allows to identify how failures should be injected, at what failure rate they should be, which replicas are susceptible to specific types of faults, and how a specific type of fault may materialise — *cf.* the various types of disturbances defined in Sect. 2.6. The proposed simulation framework allows to implement and configure a specific fault model and analyse its impact on the behaviour and effectiveness of redundancy schemata. To facilitate this,

[3] This model is used to identify the amount of simulation time between the occurrence of an event of type `request completed (ab)normally` and one of type `response sent` in case of (i); between the occurrence of an event of type `request completed exceptionally` and one of type `response sent` in case of (ii); and between the occurrence of an event of type `counter update issued` and one of type `counter update sent` in case of (iii) — *cf.* Fig. 2.2.

these models can be defined in two ways, depending on the designer's preferences and requirements:

(i) Specific fault behaviour can easily be modelled by defining how a specific (design) fault will affect and disrupt the normal operation of one or more versions. Using a special type of **auxiliary event**, the activation of faults can be deterministically simulated and failures can be injected accordingly. A single instance of such event will be scheduled for each fault at the start of a simulation run, which will be rescheduled using an event replacement procedure, reflecting inter-failure times that are typically sampled from the probability density function of some random variable — *cf.* the definition of event replacement (p. 43). Although the default implementation assumes that fault activation and failure occurrence concur, as per assumption (A08), the designer has the liberty to define manifestation latencies at will.

(ii) For each individually deployed instance of a version — which will be referred to as a **replica**[4] — the designer can optionally configure a specific fault model. If no such configuration is specified, the replica is assumed to be fault-free, although failures can still be explicitly injected using the previous approach (i). Such model allows to inject various types of disturbances that may result from the activation of design faults. It also allows to define policies to determine when failures should be injected, which allows to inject failures only when specific conditions apply, or during specific voting rounds. Using this approach, the designer can choose between two options:

   (a) The designer can choose to inject replica-specific failures by **sampling inter-failure times** from a random variable. This approach is similar to (i), yet differs in that this behaviour is encapsulated within a specific fault model that is specifically linked to one or more replicas. Both approaches can also be used to inject hardware and/or middleware failures.

   (b) A **trend-based failure injection mechanism** is also available for additional convenience, allowing the designer to define trends in which a given amount of failures can be injected within the scope of specific voting rounds. Considering a chart in which the abscissae correspond to the elements of $\{\ell_x\}_{\mathcal{C}}$ (in that order), and where the ordinates reflect the desired number of disturbances to be injected during the corresponding voting round, the designer can configure failure injection by defining one or more intervals (subsequences of voting rounds), and defining a simple linear curve. While the simulation is progressing, for each subsequent voting round, the system will inject a suitable number of disturbances that approximates the amount computed from the selected model. Trends can be described using simple XML fragments, that are validated against the XSD scheme shown in Lst. 6.1. Using such type of injection configuration, for a given voting round ℓ, the framework will select a random subset

[4] The distinction between the notions of version *vs* replica is only made throughout this section for the sake of convenience. This allows to simulate the same fault behaviour, but trigger different types of disturbances, which may be useful in case of multiple deployments — replicas — of the same version across different hosts.

from the set of available replicas V and inject the necessary disturbances, whereby injection can be steered according to specific conditions defined by the designer (*e.g.* targeting or sparing specific versions, trigger crash failures *etc.*).

Listing 6.1: trend-based failure injection requires an XML fragment that is valid as per the XSD schema listed below

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema targetNamespace="http://pats.ua.ac.be/adss/"
            xmlns:tns="http://pats.ua.ac.be/adss/"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema">

    <xsd:element name="rounds" type="xsd:integer"/>
    <xsd:element name="value" type="xsd:double"/>
    <xsd:element name="slope" type="xsd:double"/>

    <xsd:element name="scope">
        <xsd:complexType>
            <xsd:sequence>
                <!-- one or more subsequences of voting rounds within the current scope;
                     each with a specific failure trend -->
                <xsd:element ref="tns:interval" minOccurs="1" maxOccurs="unbounded"/>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>

    <xsd:element name="interval">
        <!-- characterised by a linear curve f(x) = ax + b -->
        <xsd:complexType>
            <xsd:sequence>
                <!-- how many voting rounds current interval applies to -->
                <xsd:element ref="tns:rounds"/>
                <!-- an optional start value; pick last known from previous interval
                     if not present -->
                <xsd:element ref="tns:value" minOccurs="0"/>
                <xsd:element ref="tns:slope"/>
                <!-- cap the maximum number of disturbances to be injected -->
                <xsd:element ref="tns:bound"/>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>

    <xsd:element name="failure_trend">
        <xsd:complexType>
            <xsd:sequence>
                <!-- one or more subsequences of voting rounds for independent
                     analysis; each is divided into intervals -->
                <xsd:element ref="tns:scope" minOccurs="1" maxOccurs="unbounded"/>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>

</xsd:schema>
```

6.5 Simulation Tools

When conducting experimentation through the use of discrete event simulations, it is crucial to be able and zoom in on specific behavioural characteristics of the modelled system, rather than having to cope with an overwhelming amount of complexity that results from the activity of processing significant amounts of *event* objects and effectuate the desired change in state — *v.* Sect. 1.4. Indeed, judging by the length of Chapt. 2, and by the number of assumptions that it revealed, one can easily see that it is extremely hard to interpret the results of simulations lacking adequate tooling to report on specific phenomena of interest.

Given the need to reduce complexity and to be able to zoom in on specific properties of the system's behaviour, our discrete event simulation framework has been designed to generate well-readable, customisable reports reporting on key measures, measurements and statistics that can visualise and clarify the environmental behaviour and how the model behaves in such environment.

6.5.1 Automated Analysis of Individual Simulation Runs

Once individual simulation runs are successfully completed, the framework will generate a number of graphs and tables that will prove valuable to understand when disturbances have occurred, if, how and when the level of redundancy has changed, and by what extent, during which rounds a specific version has been involved, and whether or not it contributed to the scheme's availability. Furthermore, based on statistics that are computed based on various measurements that were collected throughout the simulation run, an overview is given in which the effectiveness of individual versions is quantified, and in which these versions are automatically ranked so as to indicate the best system-environment fit.

Listing 6.2: example trend-based failure injection configuration, satisfying the syntactical structure defined in Lst. 6.1

```
<?xml version="1.0" encoding="UTF-8"?>
<adss:failure_trend xmlns:adss="http://pats.ua.ac.be/adss/"
                    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                    xsi:schemaLocation="http://pats.ua.ac.be/adss/ failure_trend.xsd">
    <adss:scope>
        <adss:interval>
            <adss:rounds>10</adss:rounds>
            <adss:slope>1.15</adss:slope>
        </adss:interval>
        <adss:interval>
            <adss:rounds>10</adss:rounds>
            <adss:value>2</adss:value>
            <adss:slope>-0.8</adss:slope>
            <adss:bound>3</adss:bound>
        </adss:interval>
        <adss:interval>
            <adss:rounds>10</adss:rounds>
            <adss:slope>0.67</adss:slope>
        </adss:interval>
        <adss:interval>
            <adss:rounds>10</adss:rounds>
            <adss:value>8</adss:value>
            <adss:slope>-2</adss:slope>
```

```
        </adss:interval>
        <adss:interval>
            <adss:rounds>10</adss:rounds>
            <adss:slope>0.05</adss:slope>
        </adss:interval>
        <adss:interval>
            <adss:rounds>10</adss:rounds>
            <adss:slope>0.25</adss:slope>
            <adss:bound>4</adss:bound>
        </adss:interval>
    </adss:scope>
</adss:failure_trend>
```

As an example, we have added the generated report for a simulation in which the behaviour of a traditional NVP redundancy scheme with a fixed degree of redundancy $n = 5$ is analysed when it would be operating in a specific environment that is configured as follows:

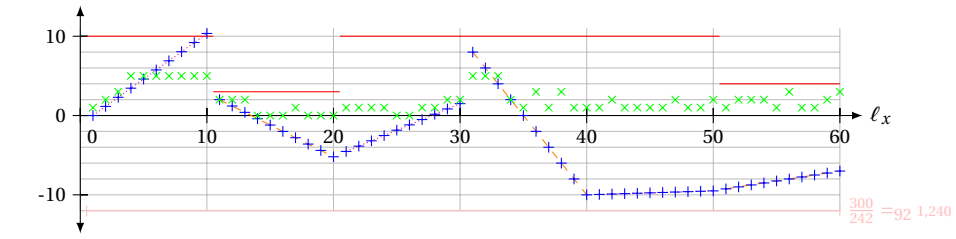
– Failures are injected using the trend-based failure injection mechanism — *v.* Sect. 6.4. The configuration is defined as an XML fragment, which can be found here above in Lst. 6.2. Trends are defined as linear equations for specific domain intervals; the corresponding curves are plotted in Fig. 6.3a (dashed and dotted curves in orange).

– All versions in the system are subject to the same fault model: there is a 78% chance that the fault will materialise as an RVF, a 21% chance that it will materialise as an EVF, and a mere 1% chance that it will materialise as a crash failure. RVF failures are sampled from a uniform distribution, as described in Sect. 2.6.1.1.

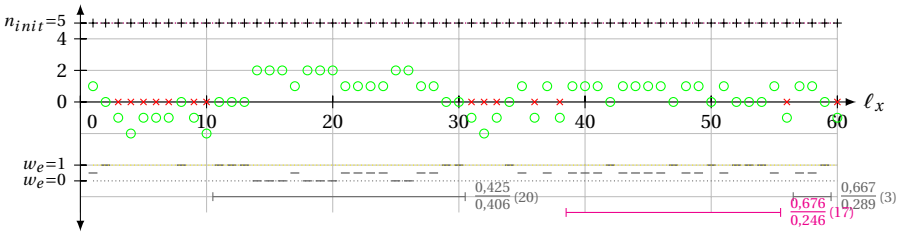
– As shown in Fig. 2.1 and 2.2, versions are processing facilities *in se*, where the available processing capacity is typically managed in a way that can easily be formally described as some type of queuing system. In this particular experiment, a simplistic model is applied, since its main purpose is to illustrate the interpretability of the output generated by the described reporting tools. Service response times are sampled from an exponentially distributed random variable with a rate λ set to 0.4. Versions will accept and process requests one by one, in the order in which they arrive, *i.e.* they apply an FCFS scheduling discipline. No specific arrival process is defined: no external load will be injected, and requests only originate from the (single) redundancy scheme under analysis. New voting rounds are injected into the system only — that is to say: requests *arrive* — as soon the previous have completed.

– For simplicity, constant network transmission times of 3.5 units of simulation time are defined for transferring the initial request to individual versions, and the response (or error) message back to the redundancy scheme. A timeout t_{max} of 12 units of simulation time is chosen.

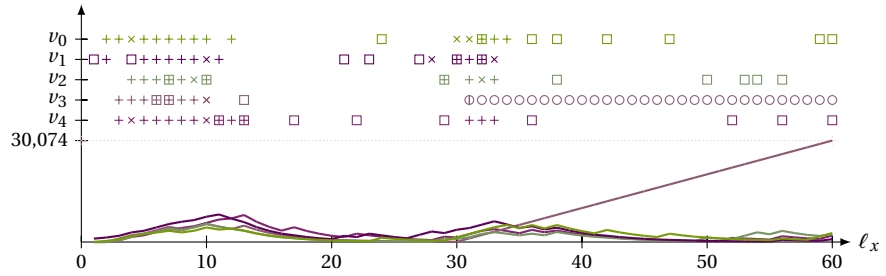
It was already pointed out in Sect. 3.1 that the amount of disturbances and the time at which they are injected directly affects the availability of the redundancy scheme. In this particular experiment, the primary application objective is dependability, *i.e.* $w_D^{\mathcal{C}} = 1$ — *cf.* Sect. 5.1. The correlation between the availability of the scheme throughout time, the injection/occurrence of failures and into what specific types of disturbances these translate, can be observed from Fig. 6.3a–6.3c:



(a) failures - overview



(b) redundancy - overview



(c) version-specific normalised dissent evolution

Figure 6.3: Automatically generated graphs for a single simulation run shed light on failure occurrence patterns, redundancy/resource consumption and sustained availability.

– The number of failures to be injected is indirectly configured by means of the XML fragment listed in Lst. 6.2, and is plotted as the orange dashed and dotted lines. Specific values for individual voting rounds are plotted as +. These reflect the maximum number of failures that should be injected into the system, thereby affecting a corresponding number of versions in the overall pool of available redundancy V.

– However, the actual number of injected failures that were injected during a given voting round $(\mathcal{C}, \ell)$ — indicated as × — depends on the number of available versions in the system $|V|$, the corresponding redundancy configuration that is being applied, and the system's internal state (*e.g.*, when one or more versions are affected by crash failures). Furthermore, the designer can optionally define an upper boundary to the number of failures to be injected for any specific voting

round $(\mathcal{C}, \ell)$ that falls within the span of a single trend interval. Such boundaries are visualised as red horizontal lines.

– Fig. 6.3c shows the exact type of disturbance by which a specific version was struck during a given voting round $(\mathcal{C}, \ell)$: crash failures and the resulting omission failures are marked as $|$, respectively $\circ$, RVF failures as $+$ and EVF failures as $\times$. In case of a dynamic redundancy configuration, where the replica selection may otherwise not be clear for individual voting rounds, fault-free versions are marked with pentagonal symbols. Note how LRF failures are marked as $\square$. The overview needs to be interpreted heedful of the failure class severity hierarchy defined in Sect. 2.6.5. A unique colour is used for each individual version, so as to easily spot the impact these failure occurrences have on the version's normalised dissent.
Note that omission failures — and performance failures in general — are not injected as such; they are automatically injected after a crash failure had previously been injected — *v.* Sect. 2.6.3. Furthermore, the emergence of LRF failures is not directly linked to software (design) faults, but rather to an inadequate system-environment fit, and is usually revealed by applying application-specific constraints and requirements — *v.* Sect. 2.6.4 and 5.1. Therefore, the occurrence of performance failures — be it omission or LRF failures — cannot be managed by the user-defined failure trend. Nor can their occurrence be suppressed by a boundary that is applied to constrain the total amount of disturbances for individual voting rounds. This is done to ensure the soundness of the simulation models defined in Chapt. 2.

– Furthermore, disturbances also have a direct effect on the partitioning procedure that underpins the majority voting adjudication mechanism. This can also be seen in Tables 6.2 and 6.3: failures are denoted by the symbol $\times$. Versions that remain fault-free for the entire duration during which they processed a subordinate invocation request $\langle \mathcal{C}, \ell, i \rangle$ are marked as $\surd$. Versions whose response corresponded to a ballot that contributed to the majority are marked accordingly[5].

n_{min}	5	$\ell^{\mathcal{C}}_{failure}\ (c^{(\mathcal{C},\ell)}_{max} - m^{(\mathcal{C},\ell)} < 0)$			23% (14 of 60)
n_{max}	5	$\sum n^{(\mathcal{C},\ell)} = 300$	$\sum cr(e^{(\mathcal{C},\ell)}) = 242(292)$	$\triangle_{success}$	0% (0 of 0)
n_{init}	5	$\sum(n^{(\mathcal{C},\ell)} - cr(e^{(\mathcal{C},\ell)})) = 58(8)$		$\nabla_{failure}$	0% (0 of 0)
t_{max}	12,000	$w_e^{(\mathcal{C},\ell)} \in_{46} [0,000, 1,000]$	$\mu = 0,576$	$\sigma = 0,349$	$ci_{90\%}$: (4.91E-1, 6.61E-1)
$\ell^{\mathcal{C}}_{total}$	60	$n^{(\mathcal{C},\ell)} \in [5,5]$	$\mu = 5,000$	$\sigma = 0,000$	$ci_{90\%}$: (5.00, 5.00)
TTF	2	$XTTF - 2 = 3$	$MTBF =_{13} 3,385$	$MTTR =_6 2,167$	

Table 6.1: General overview of redundancy scheme behaviour.

The report also generates resource consumption statistics:

– Whereas the essential statistics are displayed in Table 6.1, the actual level of redundancy used and its evolution throughout the scheme's operational life is plotted in Fig. 6.3b (+ marks).

[5] Observe how for the voting round $\ell = 8$ a majority was found despite the RVF failures that affected the consentient versions. Such conditions are acceptable behaviour, as per assumptions (A30) and (A31), and are signalled by the grey background in the *dtof* column. This is because NVP/MV redundancy schemata lack perceptional abilities to *directly* detect failures; they are only purposefully aware and rely on fault masking and adjudication algorithms [36].

– For each voting round $(\mathcal{C}, \ell)$, the nett redundancy is plotted using green ∘ marks — *cf.* Sect. 3.1.2. Measurements are indicative of the shortage or abundance of the number of versions with respect to the mandatory majority $m^{(\mathcal{C},\ell)}$. Voting round failures are highlighted by red × marks. Further details on the voting procedure can be found in Tables 6.2 and 6.3.

– Although not present in Fig. 6.3b, additional markings might indicate conditions where the redundancy dimensioning algorithm instructed to scale the level of redundancy up, but was not able to (for instance, when all resources in the system had been allocated, or when the parameter n_{max} disallows such upscaling action). This is only applicable for dynamic redundancy configurations; corresponding events would be denoted as pink △ marks. Likewise, the report will also report on the number of unsuccessful redundancy downscaling events, where the scheme would become unavailable as soon as the redundancy configuration would be adjusted to apply a lesser amount of redundancy.

– Table 6.1 also lists the applicable values for n_{init}, n_{min} and n_{max}. Furthermore, it shows the cumulative number of resource allocations $\sum n^{(\mathcal{C},\ell)}$ throughout the scheme's operational life. It also shows the cumulative contextual redundancy[6] $\sum cr(e^{(\mathcal{C},\ell)})$ so as to give an intuitive, yet overly simplified, indication of whether too many or to few resources have been used — *v.* p. 105. Similarly, it will mention how many redundancy upscaling events proved to be successfull in regaining or prolonging the scheme's availability.

– Finally, an overview of key dependability measures is given, including MTTF, MTBF and MTTR.

A key factor in determining correct values for the normalised dissent is the effectiveness factor $w_e^{(\mathcal{C},\ell)}$: this factor is used in particular to assess the instantaneous effectiveness of a specific redundancy configuration during a specific voting round $(\mathcal{C}, \ell)$, given the occurrence of disturbances $e^{(\mathcal{C},\ell)}$ — *cf.* Sect. 4.1. Measurements are collected at the end of each voting round throughout the simulation run, and are visualised in Fig. 6.3b; their exact values, together with the corresponding *dtof* measurements, are listed in Tables 6.2 and 6.3. Note how for this particular experiment the computation of the normalised dissent follows a configuration in which k_1 is set to 0.85, k_2 to 0.75 and k_{max} to 0.95 — *cf.* Sect. 4.3. For extended time to failure intervals, an indication of the effectiveness factor is given by means of an average as well as standard deviation, including the number of samples (Fig. 6.3b).

ℓ	$\wp^{(\mathcal{C},\ell)} \setminus P_F^{(\mathcal{C},\ell)}$	$P_F^{(\mathcal{C},\ell)}$	*dtof*	$w_e^{(\mathcal{C},\ell)}$	v_0	v_1	v_2	v_3	v_4
1	{4, 2, 3, 0}	{1}	2	0,500	$\surd_m$	×	$\surd_m$	$\surd_m$	$\surd_m$
2	{1, 0} {2, 3, 4}	∅	1	1,000	×	×	$\surd_m$	$\surd_m$	$\surd_m$
3	{0, 4} {3} {2, 1}	∅	0	-	×	√	√	×	×
4	{2} {3}	{0, 4, 1}	0	-	×	×	×	×	×

Table 6.2: Voting round summary: replica selection & partitioning.

[6] As can be seen in table 6.1, two measurements are collected: one without taking LRF failures into account, and another that does take this type of failures into account – the latter is shown in parentheses.

ℓ	$\wp^{(\mathbb{C},\ell)} \setminus P_F^{(\mathbb{C},\ell)}$	$P_F^{(\mathbb{C},\ell)}$	*dtof*	$w_e^{(\mathbb{C},\ell)}$	v_0	v_1	v_2	v_3	v_4
5	{2}{1,0}{4,3}	∅	0	-	×	×	×	×	×
6	{2,0}{1}{4}	{3}	0	-	×	×	×	×	×
7	{1}{0,4}	{3,2}	0	-	×	×	×	×	×
8	{2,0,3}{1}{4}	∅	1	1,000	$\times_m$	×	$\times_m$	$\times_m$	×
9	{1,0}{3,4}	{2}	0	-	×	×	×	×	×
10	{0}	{3,1,4,2}	0	-	×	×	×	×	×
11	{1}{0,3,2}	{4}	1	1,000	$\surd_m$	×	$\surd_m$	$\surd_m$	×
12	{4,0}{1,3,2}	∅	1	1,000	×	$\surd_m$	$\surd_m$	$\surd_m$	×
13	{2,1,0}	{3,4}	1	1,000	$\surd_m$	$\surd_m$	$\surd_m$	×	×
14	{3,0,1,2,4}	∅	3	0,000	$\surd_m$	$\surd_m$	$\surd_m$	$\surd_m$	$\surd_m$
15	{3,1,2,4,0}	∅	3	0,000	$\surd_m$	$\surd_m$	$\surd_m$	$\surd_m$	$\surd_m$
16	{1,2,4,0,3}	∅	3	0,000	$\surd_m$	$\surd_m$	$\surd_m$	$\surd_m$	$\surd_m$
17	{1,2,0,3}	{4}	2	0,500	$\surd_m$	$\surd_m$	$\surd_m$	$\surd_m$	×
18	{2,0,3,4,1}	∅	3	0,000	$\surd_m$	$\surd_m$	$\surd_m$	$\surd_m$	$\surd_m$
19	{1,4,2,0,3}	∅	3	0,000	$\surd_m$	$\surd_m$	$\surd_m$	$\surd_m$	$\surd_m$
20	{0,4,3,2,1}	∅	3	0,000	$\surd_m$	$\surd_m$	$\surd_m$	$\surd_m$	$\surd_m$
21	{0,4,2,3}	{1}	2	0,500	$\surd_m$	×	$\surd_m$	$\surd_m$	$\surd_m$
22	{1,3,0,2}	{4}	2	0,500	$\surd_m$	$\surd_m$	$\surd_m$	$\surd_m$	×
23	{2,4,0,3}	{1}	2	0,500	$\surd_m$	×	$\surd_m$	$\surd_m$	$\surd_m$
24	{2,3,1,4}	{0}	2	0,500	×	$\surd_m$	$\surd_m$	$\surd_m$	$\surd_m$
25	{0,4,1,2,3}	∅	3	0,000	$\surd_m$	$\surd_m$	$\surd_m$	$\surd_m$	$\surd_m$
26	{2,1,0,4,3}	∅	3	0,000	$\surd_m$	$\surd_m$	$\surd_m$	$\surd_m$	$\surd_m$
27	{2,0,3,4}	{1}	2	0,500	$\surd_m$	×	$\surd_m$	$\surd_m$	$\surd_m$
28	{2,3,0,4}	{1}	2	0,500	$\surd_m$	×	$\surd_m$	$\surd_m$	$\surd_m$
29	{1,0,3}	{2,4}	1	1,000	$\surd_m$	$\surd_m$	×	$\surd_m$	×
30	{2,3,4}	{0,1}	1	1,000	×	×	$\surd_m$	$\surd_m$	$\surd_m$
31	{4}{1,2}	{0,3}	0	-	×	×	×	×	×
32	{4}	{2,1,0,3}	0	-	×	×	×	×	×
33	{0}{4,2}	{1,3}	0	-	×	×	×	×	×
34	{0}{2,1,4}	{3}	1	1,000	×	$\surd_m$	$\surd_m$	×	$\surd_m$
35	{1,4,2,0}	{3}	2	0,500	$\surd_m$	$\surd_m$	$\surd_m$	×	$\surd_m$
36	{2,1}	{0,4,3}	0	-	×	$\surd$	$\surd$	×	×
37	{2,4,1,0}	{3}	2	0,500	$\surd_m$	$\surd_m$	$\surd_m$	×	$\surd_m$
38	{4,1}	{2,0,3}	0	-	×	$\surd$	×	×	$\surd$
39	{0,1,2,4}	{3}	2	0,500	$\surd_m$	$\surd_m$	$\surd_m$	×	$\surd_m$
40	{1,2,4,0}	{3}	2	0,500	$\surd_m$	$\surd_m$	$\surd_m$	×	$\surd_m$
41	{4,2,1,0}	{3}	2	0,500	$\surd_m$	$\surd_m$	$\surd_m$	×	$\surd_m$
42	{2,1,4}	{0,3}	1	1,000	×	$\surd_m$	$\surd_m$	×	$\surd_m$
43	{0,1,2,4}	{3}	2	0,500	$\surd_m$	$\surd_m$	$\surd_m$	×	$\surd_m$
44	{1,2,0,4}	{3}	2	0,500	$\surd_m$	$\surd_m$	$\surd_m$	×	$\surd_m$
45	{2,1,0,4}	{3}	2	0,500	$\surd_m$	$\surd_m$	$\surd_m$	×	$\surd_m$
46	{4,1,0,2}	{3}	2	0,500	$\surd_m$	$\surd_m$	$\surd_m$	×	$\surd_m$
47	{2,1,4}	{0,3}	1	1,000	×	$\surd_m$	$\surd_m$	×	$\surd_m$
48	{0,4,1,2}	{3}	2	0,500	$\surd_m$	$\surd_m$	$\surd_m$	×	$\surd_m$
49	{2,4,0,1}	{3}	2	0,500	$\surd_m$	$\surd_m$	$\surd_m$	×	$\surd_m$
50	{0,4,1}	{2,3}	1	1,000	$\surd_m$	$\surd_m$	×	×	$\surd_m$
51	{1,4,0,2}	{3}	2	0,500	$\surd_m$	$\surd_m$	$\surd_m$	×	$\surd_m$
52	{2,0,1}	{4,3}	1	1,000	$\surd_m$	$\surd_m$	$\surd_m$	×	×
53	{0,4,1}	{2,3}	1	1,000	$\surd_m$	$\surd_m$	×	×	$\surd_m$
54	{0,1,4}	{2,3}	1	1,000	$\surd_m$	$\surd_m$	×	×	$\surd_m$
55	{1,4,2,0}	{3}	2	0,500	$\surd_m$	$\surd_m$	$\surd_m$	×	$\surd_m$

Table 6.3: Voting round summary: replica selection & partitioning (continued).

The above table shows detailed information about how the injected disturbances affect particular replicas, and what impact these disturbances have on the rendered

service: it shows how deviations from the normal behaviour — resulting in ballots with a lesser degree of mutual equivalence — typically lead to partitions being generated by the applied adjudication algorithm with a lessened likelihood of finding a qualified majority.

Based on these recorded values, an overview is then given of which replicas best perform from various angles (Table 6.6). This may help to clarify the observed behaviour of the system under investigation — the fault-tolerant composite, that is — and may provide further insight into the impact of version-specific fault models on the scheme's overall performance.

In case the total available redundancy in the system V would exceed the level of redundancy applied in any specific redundancy configuration $n^{(\mathcal{C},\ell)}$, this overview will show the selection of replicas that were (not) involved — *cf.* Sect. 5.3. Detailed statistics are collected and reported for all version in the system; Tables 6.4 and 6.5 clearly list specific values, their inverse, and — where useful — indications of the average and standard deviation[7].

version	v_0	v_1	v_2	v_3	v_4
queuing model	null	null	null	null	null
outbound client overhead	$\mu_{60} = 0,000$ $\sigma = 0,000$	$\mu_{60} = 0,000$ $\sigma = 0,000$	$\mu_{60} = 0,000$ $\sigma = 0,000$	$\mu_{60} = 0,000$ $\sigma = 0,000$	$\mu_{60} = 0,000$ $\sigma = 0,000$
RTT (incoming)	$\mu_{60} = 3,500$ $\sigma = 0,000$	$\mu_{60} = 3,500$ $\sigma = 0,000$	$\mu_{60} = 3,500$ $\sigma = 0,000$	$\mu_{60} = 3,500$ $\sigma = 0,000$	$\mu_{60} = 3,500$ $\sigma = 0,000$
$\mathtt{Q}_{\mathtt{r,out}}$	$\mu_{60} = 0,000$ $\sigma = 0,000$	$\mu_{60} = 0,000$ $\sigma = 0,000$	$\mu_{60} = 0,000$ $\sigma = 0,000$	$\mu_{30} = 0,000$ $\sigma = 0,000$	$\mu_{60} = 0,000$ $\sigma = 0,000$
RTT (outgoing)	$\mu_{60} = 3,500$ $\sigma = 0,000$	$\mu_{60} = 3,500$ $\sigma = 0,000$	$\mu_{60} = 3,500$ $\sigma = 0,000$	$\mu_{30} = 3,500$ $\sigma = 0,000$	$\mu_{60} = 3,500$ $\sigma = 0,000$
inbound client overhead	$\mu_{60} = 0,750$ $\sigma = 0,000$	$\mu_{60} = 0,750$ $\sigma = 0,000$	$\mu_{60} = 0,750$ $\sigma = 0,000$	$\mu_{30} = 0,750$ $\sigma = 0,000$	$\mu_{60} = 0,750$ $\sigma = 0,000$
end-to-end response time	$\mu_{60} = 9,446$ $\sigma = 2,304$	$\mu_{60} = 9,485$ $\sigma = 3,133$	$\mu_{60} = 9,498$ $\sigma = 3,364$	$\mu_{30} = 9,654$ $\sigma = 2,252$	$\mu_{60} = 9,703$ $\sigma = 2,281$
$\mathtt{Q}_{\mathtt{r,in}}$	$\mu_{60} = 0,000$ $\sigma = 0,000$	$\mu_{60} = 0,000$ $\sigma = 0,000$	$\mu_{60} = 0,000$ $\sigma = 0,000$	$\mu_{30} = 0,000$ $\sigma = 0,000$	$\mu_{60} = 0,000$ $\sigma = 0,000$
RPT	$\mu_{60} = 2,446$ $\sigma = 2,304$	$\mu_{60} = 2,485$ $\sigma = 3,133$	$\mu_{60} = 2,498$ $\sigma = 3,364$	$\mu_{30} = 2,654$ $\sigma = 2,252$	$\mu_{60} = 2,703$ $\sigma = 2,281$
sojourn time	$\mu_{60} = 2,446$ $\sigma = 2,304$	$\mu_{60} = 2,485$ $\sigma = 3,133$	$\mu_{60} = 2,498$ $\sigma = 3,364$	$\mu_{30} = 2,654$ $\sigma = 2,252$	$\mu_{60} = 2,703$ $\sigma = 2,281$

Table 6.4: Replica configuration & statistics.

6.5.2 Automated Analysis of Simulation Batches

As soon as the system and its environment have been modelled and a first simulation run indicates a correct implementation, one then usually proceeds by running simulation batches. Batches are mainly helpful for further in-depth analysis of the system-environment fit:

[7] A note on Table 6.4 though: the subscript numbers represent amount of observations used in computing arithmetic mean and standard deviation, and represent the number of voting rounds the corresponding version was involved in. The values for v_3 reflect the occurrence of a crash failure. As the version has crashed, requests will still be accepted by the queuing system, but they will never start processing. Here, we assume the middleware on which the version is deployed will continue to function, regardless of v_3 having crashed. Hence the two different subscript values (30 *vs* 60).

– the designer might want to assess how different redundancy management policies and algorithms behave given similar environmental conditions;

– or (s)he may want to evaluate the effectivess of a specific policy when the model is subject to varying conditions.

As soon as all configured/queued simulation runs have completed, the proposed discrete event simulation framework will automatically rank all individual simulation runs, and this from various angles. By default, the ranking is generated based on the default redundancy scheme-specific metric measurements that have been collected — *cf.* Sect. 6.2.

The ranking logic has been designed in such a way that the designer can implement new metrics and define how measurements should be collected during individual simulation runs, and can define whether ranking should be based on ascending rather than descending values. Furthermore, ranking criteria can be defined as a combination of various metrics.

version	v_0	v_1	v_2	v_3	v_4
re-integration latency	n/a	n/a	n/a	n/a	n/a
	n/a	n/a	n/a	n/a	n/a
exclusion latency	n/a	n/a	n/a	n/a	n/a
	n/a	n/a	n/a	n/a	n/a
exclusion failures	n/a	n/a	n/a	n/a	n/a
	n/a	n/a	n/a	n/a	n/a
successive usage period	$\mu_1 = 60$	$\mu_1 = 60$	$\mu_1 = 60$	$\mu_1 = 60$	$\mu_1 = 60$
	$\sigma = 0$	$\sigma = 0$	$\sigma = 0$	$\sigma = 0$	$\sigma = 0$
$\#rounds(\mathcal{C}, v) - \#consent(\mathcal{C}, v)$	22	18	16	39	21
$\hookrightarrow \ell^{\mathcal{C}}_{total}$	37 %	30 %	27 %	65 %	35 %
$\hookrightarrow \#rounds(\mathcal{C}, v)$	37 %	30 %	27 %	65 %	35 %
$\#rounds(\mathcal{C}, v)$	60	60	60	60	60
$\hookrightarrow \ell^{\mathcal{C}}_{total}$	100 %	100 %	100 %	100 %	100 %
$\#consent(\mathcal{C}, v)$	38	37	42	22	38
$\hookrightarrow \#rounds(\mathcal{C}, v)$	63 %	62 %	70 %	37 %	63 %
normalised dissent	$\mu_{60} = 2,035$	$\mu_{60} = 2,604$	$\mu_{60} = 1,686$	$\mu_{60} = 8,711$	$\mu_{60} = 2,262$
	$\sigma = 1,426$	$\sigma = 2,112$	$\sigma = 1,303$	$\sigma = 9,357$	$\sigma = 1,820$
end-to-end response time	$\mu_{60} = 9,157$	$\mu_{60} = 8,908$	$\mu_{60} = 8,923$	$\mu_{60} = 10,681$	$\mu_{60} = 9,387$
	$\sigma = 1,653$	$\sigma = 1,544$	$\sigma = 1,666$	$\sigma = 1,747$	$\sigma = 1,621$
TTF	1	1	3	2	2
$\hookrightarrow$ inv	1,000	1,000	0,333	0,500	0,500
XTTF-2	2	4	4	3	3
$\hookrightarrow$ inv	0,500	0,250	0,250	0,333	0,333
MTBF	1,286	1,583	1,900	0,541	1,308
$\hookrightarrow$ inv	0,778	0,632	0,526	1,850	0,765
MTTR	5,000	3,250	3,667	8,000	7,000
$\hookrightarrow$ inv	0,200	0,308	0,273	0,125	0,143

Table 6.5: Failure injection replica statistics.

version	v_0	v_1	v_2	v_3	v_4
re-integration latency	-	-	-	-	-
exclusion latency	-	-	-	-	-
exclusion failures	-	-	-	-	-
successive usage period	-	-	-	-	-
$\#rounds(\mathcal{C}, v) - \#consent(\mathcal{C}, v)$	4	2	1	5	3
$\#rounds(\mathcal{C}, v)$	1	1	1	1	1
$\#consent(\mathcal{C}, v)$	2	3	1	4	2
normalised dissent	2	4	1	5	3
end-to-end response time	3	1	2	5	4
TTF	3	3	1	2	2
XTTF-2	3	1	1	2	2
MTBF	4	2	1	5	3
MTTR	3	1	2	5	4
outbound client overhead	1	1	1	1	1
RTT (incoming)	1	1	1	1	1
$\mathtt{Q}_{\mathtt{r,in}}$	1	1	1	1	1
RPT	1	2	3	4	5
sojourn time	1	2	3	4	5
$\mathtt{Q}_{\mathtt{r,out}}$	1	1	1	1	1
RTT (outgoing)	1	1	3	2	1
inbound client overhead	3	3	2	1	3

Table 6.6: Replica auto-ranking: the ability to automatically rank replicas based on specific performance attributes, may prove helpful in better comprehending the system-environment fit.

CHAPTER 7

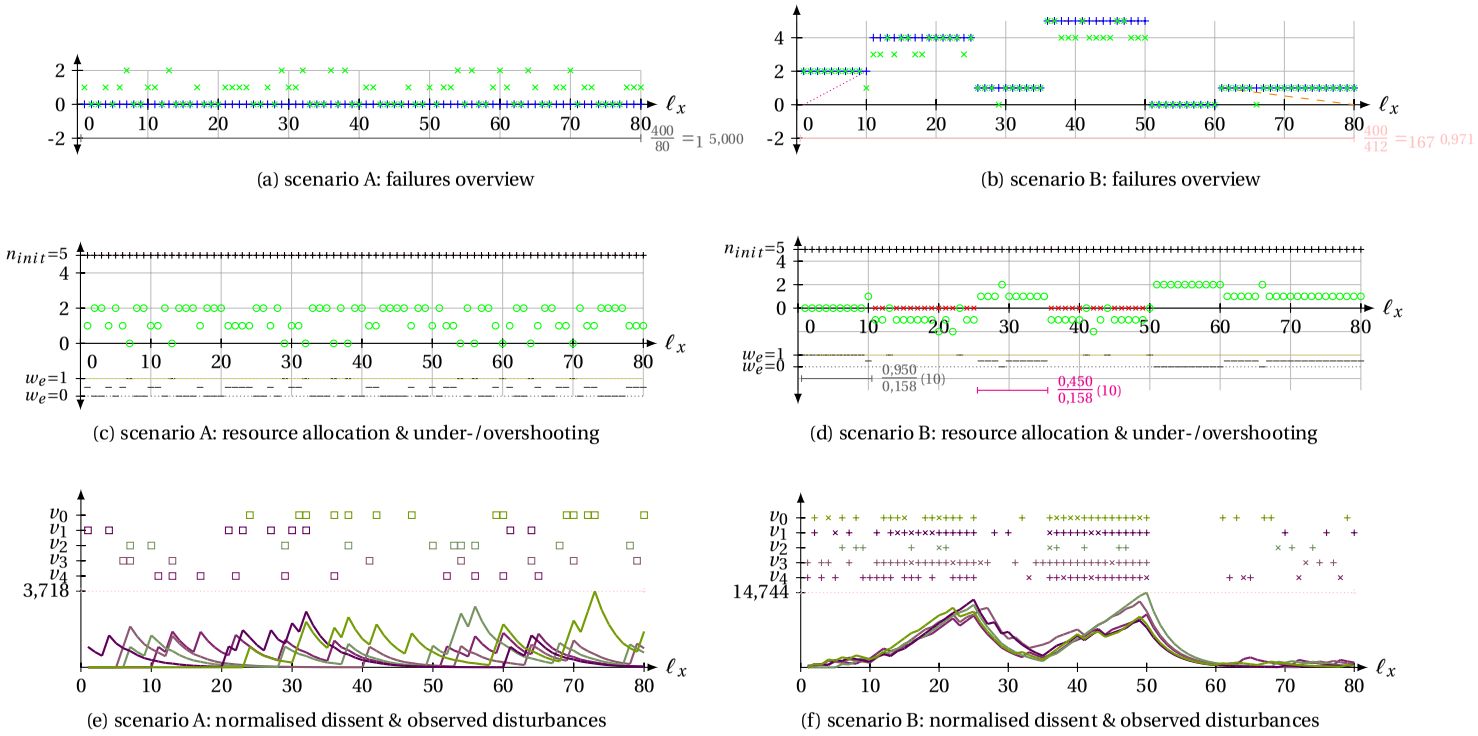
Performance Analyses

In this chapter, some exemplary policies will be suggested for implementing the abstract A-NVP/MV strategy presented in Chapt. 5. Designed (i) to sustain the availability of redundancy schemata and (ii) to increase the overall cost effectiveness by parsimoniously allocating system resources, each policy defines how the cached context information will be used to effectuate the desired change to the current redundancy configuration. First, we will show how such dynamic redundancy configurations prove to be effective in overcoming the limitations faced by applying static redundancy configurations. Next, we will zoom in into how such policies perform under various conditions, when and to what extent they improve the overall performance of the system, and when their use is not guaranteed to translate in performance gains. In doing so, we use the simulation framework that was described in the previous chapter. Related research question(s): RQ-2 and RQ-3.

7.1 Redundancy Configurations: How Dynamic Configurations can Overcome the Limitations of Static Configurations

The purpose of section is to illustrate the limitations inherent to the use of traditional, static redundancy configurations, and how dynamic redundancy configurations may be used to overcome these limitations to some extent. We shall primarily focus on the concern of dependability, where it is desirable to maximally sustain the scheme's availability (reliability), while applying some form of parsimony in the allocation of system resources so as to economise on resource expenditure. Whereas similar experimentation can be performed for assessing the impact of specific redundancy configuration on the scheme's effectiveness in terms of timeliness, *e.g.* by applying intelligent load balancing techniques to realise lower response times, this concern is kept out of scope[1].

[1] Throughout this dissertation, the emphasis is placed on dependability analysis rather than capacity management. Applying load balancing techniques to achieve lower response times typically requires detailed modelling of the deployment environment on which individual versions are running — more specifically the computational capacity of hardware and middleware infrastructure — as well as the



(a) scenario A: failures overview

(b) scenario B: failures overview

(c) scenario A: resource allocation & under-/overshooting

(d) scenario B: resource allocation & under-/overshooting

(e) scenario A: normalised dissent & observed disturbances

(f) scenario B: normalised dissent & observed disturbances

Figure 7.1: Experiment 7.1: the system-environment fit determines the effectiveness of the redundancy scheme. Scenarios A and B show the impact of different fault models on a similar environment.

Throughout this section, we will use the simulation framework that was described in Chapt. 6 to generate various graphs and tables that should help in analysing and comprehending the various limitations of traditional redundancy schemata, as well as the potential benefits of dynamically allocating the available redundancy.

7.1.1 The Deficiencies of Static Redundancy Configurations

The effectiveness of an NVP composite is largely determined by the dependability of the versions employed within. As elucidated in [3, Sect. 4.3.3], and demonstrated in experiment 7.1, the use of replicas of poor reliability can result in a system tolerant of faults but with poor reliability. It is therefore crucial for the system to continuously monitor the operational status of the available resources and avoid the use of resources that do not significantly contribute to an increase in dependability, or that may even jeopardise the schemes' overall effectiveness.

Experiment 7.1

In Fig.7.1, one can easily see how important it is to select versions that perform well in terms of dependability. The system-environment fit determines the effectiveness of the redundancy scheme. We compare two scenarios; both apply a static redundancy configuration using 5 versions.
As one may observe from Fig. 7.1c, the scheme is always able to tolerate the disturbances that sporadically affect the selected versions. In this scenario A, there is a good match between the system and its environment (the underlying versions, their operating conditions, *etc.*). In this environment, only disturbances in the timing domain materialise, as can be seen in Fig. 7.1e, and the applied degree of redundancy shows to be effective to counterbalance any occasional LRF failures that affect one or more of the underlying versions in use. Apart from these occasional LRF failures, no other disturbances are injected.
In scenario B, the redundancy scheme applies another static redundancy configuration: it applies the same degree of redundancy, yet another set of replicas. Another fault model applies, and only disturbances in the content domain — *i.e.* RVF and EVF failures — are applied. For versions affected by RVF failures, the corresponding ballots that will be used by the voting procedure are sampled from a uniform distribution — *v.* p. 51. As one can observe from Fig. 7.1d and 7.1f, the selected versions are quite unreliable, therefore the scheme itself becomes unreliable. Indeed, the applied degree of redundancy is not always sufficient to mask all failures (disturbances in the content domain). In fact, in terms of availability, the scheme performs worse than had a simplex system been used consisting only of version v_2: the simplex system would have been available for 66 out of 80 voting rounds (82.5%), whereas the scheme was available for 55 of the total of 80 voting rounds (a mere 68%). Note how the difference in system-environment fit is apparent from the normalised dissent measurements.

network topology and throughput — *cf.* Sect. 2.5 and 8.5.2. Such detailed modelling would also require to make environment-specific decisions as to where the individual resources are actually deployed and hosted, which goes beyond the intention of this dissertation, where we wish to make an abstraction of the whereabouts of individual versions (*e.g.* on a corporate LAN, a WAN, or on the Internet). As described in Sect. 6.3, our simulation framework includes the functionality that allows for basic modelling of the environment and all of its contituents. If needed, the reader can implement custom artefacts that better fit his/her specific needs.

In [36], the authors propose a classification to categorise fault-tolerant solutions by analysing resilient behaviour as the result of four properties: perception, awareness, planning and dynamicity. In line with this classification:

- ✖ Traditional NVP redundancy schemata simply rely on a voting algorithm to filter out any flawed results and, in doing so, adjudicate a correct outcome — a principle generally known as fault masking. There is no active monitoring of the versions on which the scheme relies, nor the deployment environment in which it is operating. Because of this, they fail to capture the specific nature of the disturbances that emerge: they are systems **lacking awareness**, having **little or no perceptional abilities**.

Experiment 7.2

This limitation is exemplified in Fig. 7.2e: given a static TMR configuration, the scheme would fail to detect that one of the underlying versions was struck by, *e.g.* a permanent failure (as is the case for version v_2 that has crashed just before or during voting round $\ell = 29$). As a consequence, the scheme would fail to deduce that — for voting round $\ell = 29$ and subsequent voting rounds — this particular version no longer contributes to sustain its availability and that its prolonged use consistently translated in omission failures. This can be observed by *dtof* or nett redundancy measurements being recorded that indicate the available redundancy is (nearly) exhausted (Fig. 7.2c) — *cf.* Sect. 3.1.1–3.1.2. The scheme would observe that it would no longer be able to withstand the total number of disturbances, where it would fail to fully mask the underlying failures, in spite of incidental conditions during which it could occasionally regain its availability (although by chance).
Versions were configured with an exponentially distributed service time S ($\lambda = 0.4$), no waiting time ($c = k = \infty$) and constant network transmission times set to 3.5 time units. RVF and EVF failures are injected with an 80%, resp. with 20% probability, with RVF failures sampled from a uniform distribution (refer to p. 51). $t_{max} = 30$ time units. Normalised dissent values are computed with parameters set as follows: $k_1 = 0.85$, $k_2 = 0.75$ and $k_{max} = 0.95$.
Had a dynamic redundancy scheme been applied as in Fig. 7.2f, the redundancy configuration could have been altered so that v_2 would (temporarily) be taken out of service, and would have been replaced by another version that was judged as more reliable[a] (scheme configured with $r_d = 5$, $r_u = r_f = 1$ and $c_{sm} = 0$ — *cf.* Chapt. 5).
When comparing scenarios A and B, one can observe that — in spite of the same cumulative number of resource allocations $\sum n^{(\mathcal{C},\ell)}$ — there is a significant improvement of the scheme's overall availability. Whereas a 91% availability is recorded for an operational life in which the scheme has handled $\ell^{\mathcal{C}}_{total} = 80$ voting rounds, only a 51% availability is recorded for scenario A (refer to Tables 7.1a and 7.1a)[b]. Furthermore, note that the application of a dynamic redundancy scheme has the potential to attain considerably higher MTBF and lower MTTR values.
The crash failure affecting version v_2 essentially renders it pretty much useless for further use in any redundancy scheme. Dynamically selecting the versions at runtime will remove poorly performing versions that do not (significantly) contribute to the scheme's effectiveness, and can successfully replace these with other, more reliable versions. In this particular case, because version v_2 will automatically be removed from the redundancy configuration, it will have considerably less negative impact on the redundancy scheme's operations: in scenario B, v_2 will participate to a mere

44% of the total number of voting rounds $\ell^{\mathcal{C}}_{total}$ (unlike 100% in scenario A). As a consequence, the number of disturbances that the scheme has to overcome because of the use of this particular version, decreases from 59 in scenario A to a mere 19 in scenario B (refer to Tables 7.1c and 7.1d)[c].

[a] For the unexploited share of redundancy, normalised dissent measurements may record (near-)optimal values. Even though such replicas may handle external traffic throughout the scheme's operational life span, and can thus suffer from any type of failure, such disturbances will not be accounted for in the context of this particular scheme's.

[b] We apply an intuitive notion of availability, that is obtained by subtracting the relative amount of voting rounds for which a failure was recorded from 1, given the total amount of redundancy $\ell^{\mathcal{C}}_{total}$.

[c] Note that after v_2 was excluded because of subsequent increases in normalised dissent measurements, v_2 is eventually re-introduced in the replica selection. This is due to the nature of the reward model defined in Sect. 4.3, that will gradually undo any penalty that was previously imposed to the version because of disturbances that might have affected the scheme's availability. The rationale of this approach is that versions that are affected by disturbances of transient nature, or by intermittent bursts of disturbances can still sustain the scheme's overall availability at relevant stages of its operational life. The pace with which previous penalties are undone can be configured according to application needs, by means of several parameters — *cf.* Eq. (4.2e), p. 74.

✖ The traits of **planning and dynamicity are absent**, as the redundancy configuration is static and determined at design time, and a specific, immutable selection of redundant resources in the system is used.

  – This does not allow to exclude poorly performing versions, nor does it allow to change the replica selection at runtime — *cf.* Sect. 5.3. Besides, there is no relevant contextual knowledge to justify such adaptation; the NVP composite simply is unaware of which versions perform better (in terms of reliability). This approach may result in suboptimal redundancy configurations being used over a prolonged period of time, with reliability below the anticipated levels, possibly resulting in catastrophic failure. Such scenario can be observed in Fig. 7.2e.

Experiment 7.3

In Fig. 7.3, we consider an environment in which two specific versions suffer from intermittent bursts of disturbances that emerge as EVF failures. This is a common situation in real-world distributed computing systems: whenever a remotely deployed version would not be reachable because of issues with the network communication infrastructure, any subsequent invocation of such version would typically result in EVF failures (a timeout at the client side, *i.e.* in the NVP composite's runtime), LRF failures (whenever the latency exceeds the imposed upper limit t_{max}), or even RVF failures (when the result to be transferred is corrupted due to, *e.g.*, electromagnetic interference).
As can be observed from Fig. 7.3e, version v_2 exhibits perfect behaviour, fully in line with its (non-)functional requirements and without any disturbance at all. Within the operational context of the analysed NVP composite, versions v_0 and v_1 are affected by bursts of EVF failures throughout the intervals with voting round identifiers $[1,10]$, $[28,40]$ and $[65,80]$, resp. $[18,40]$ and $[60,80]$. As one can see in Fig. 7.3c, because

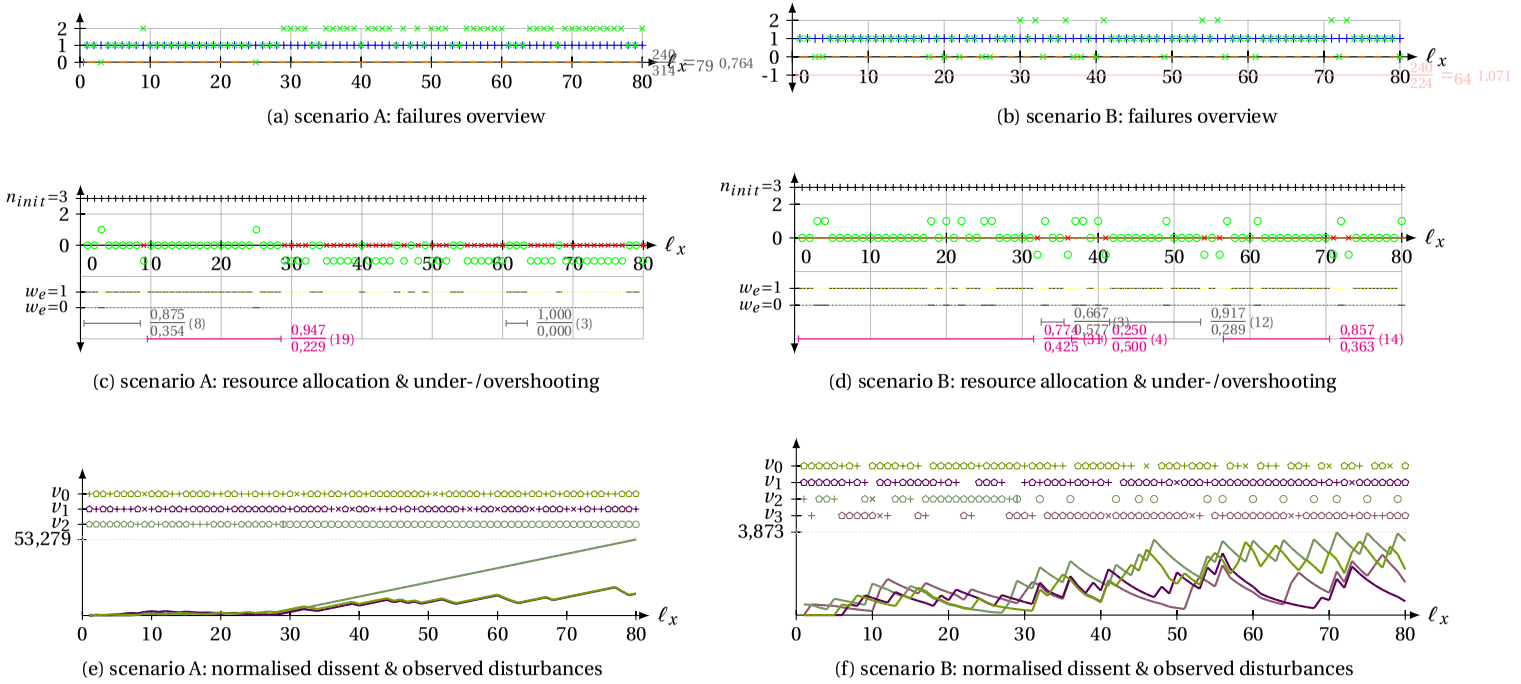


Figure 7.2: Experiment 7.2: triple-modular redundancy (TMR): using a static redundancy configuration (scenario A) *vs* using a dynamic redundancy configuration (scenario B). Version v_2 affected by a crash failure.

n_{min}	3	$\ell^{\mathcal{C}}_{failure}$ $(c^{(\mathcal{C},\ell)}_{max} - m^{(\mathcal{C},\ell)} < 0)$			49% (39 of 80)
n_{max}	3	$\sum n^{(\mathcal{C},\ell)} = 240$	$\sum cr(e^{(\mathcal{C},\ell)}) = 314(314)$	$\triangle_{success}$	0% (0 of 0)
n_{init}	3	$\sum(n^{(\mathcal{C},\ell)} - cr(e^{(\mathcal{C},\ell)})) = -74(-74)$		$\nabla_{failure}$	0% (0 of 0)
t_{max}	30,000	$w_e^{(\mathcal{C},\ell)} \in_{41} [0,000, 1,000]$	$\mu = 0,951$	$\sigma = 0,218$	$ci_{90\%}$: (8.95E-1, 1.01)
$\ell^{\mathcal{C}}_{total}$	80	$n^{(\mathcal{C},\ell)} \in [3,3]$	$\mu = 3,000$	$\sigma = 0,000$	$ci_{90\%}$: (3.00, 3.00)
TTF	8	$XTTF - 2 = 28$	$MTBF_{=38}$ 0,868	$MTTR_{=10}$ 3,800	

(a) scenario A: general overview for redundancy scheme

n_{min}	3	$\ell^{\mathcal{C}}_{failure}$ $(c^{(\mathcal{C},\ell)}_{max} - m^{(\mathcal{C},\ell)} < 0)$			9% (7 of 80)
n_{max}	3	$\sum n^{(\mathcal{C},\ell)} = 240$	$\sum cr(e^{(\mathcal{C},\ell)}) = 224(224)$	$\triangle_{success}$	0% (0 of 0)
n_{init}	3	$\sum(n^{(\mathcal{C},\ell)} - cr(e^{(\mathcal{C},\ell)})) = 16(16)$		$\nabla_{failure}$	0% (0 of 0)
t_{max}	30,000	$w_e^{(\mathcal{C},\ell)} \in_{73} [0,000, 1,000]$	$\mu = 0,781$	$\sigma = 0,417$	$ci_{90\%}$: (7.01E-1, 8.61E-1)
$\ell^{\mathcal{C}}_{total}$	80	$n^{(\mathcal{C},\ell)} \in [3,3]$	$\mu = 3,000$	$\sigma = 0,000$	$ci_{90\%}$: (3.00, 3.00)
r_d	5	$\square_{fill}{}^{d} \in_{80} [0,5]$	$\mu = 3,725$	$\sigma = 1,800$	$ci_{90\%}$: (3.39, 4.06)
		$\square_{reset}{}^{d} \in_{7} [1,5]$	$\mu = 3,429$	$\sigma = 1,813$	$ci_{90\%}$: (2.30, 4.56)
r_u	1	$\square_{fill}{}^{u} \in_{80} [0,1]$	$\mu = 0,087$	$\sigma = 0,284$	$ci_{90\%}$: (3.52E-2, 1.40E-1)
		$\square_{reset}{}^{u} \in_{7} [1,1]$	$\mu = 1,000$	$\sigma = 0,000$	$ci_{90\%}$: (1.00, 1.00)
r_f	1	$\square_{corrupt}$	0% (0 of 80)		
TTF	31	$XTTF - 2 = 35$	$MTBF_{=6}$ 5,833	$MTTR_{=7}$ 1,000	
c_{sm}	f	$c_{sm} = 0$	n/a	n/a	$ci_{90\%}$: (0.00, 0.00)
		$\ell_{suspect}$ $(0 \leq c^{(\mathcal{C},\ell)}_{max} - m^{(\mathcal{C},\ell)} < c_{sm})$			0% (0 of 80)

(b) scenario B: general overview for redundancy scheme

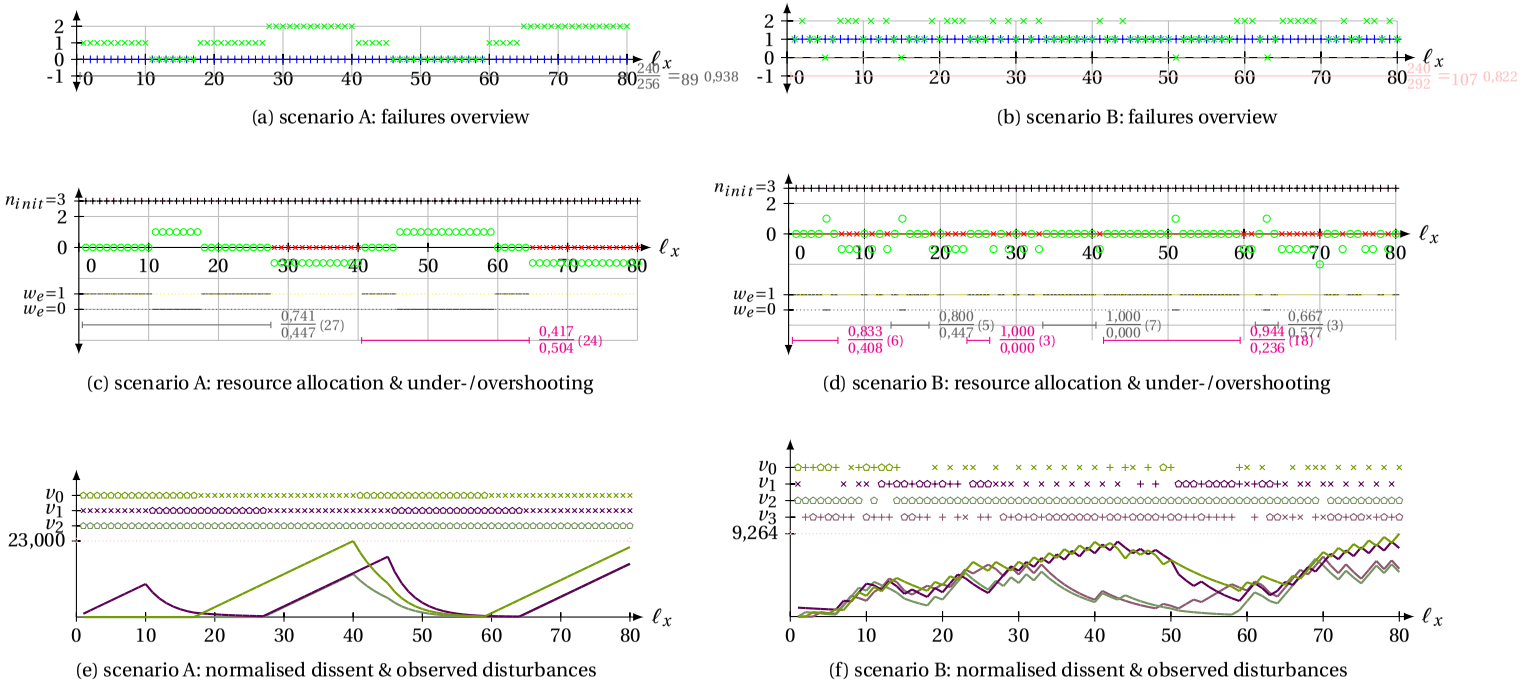
version	v_0	v_1	v_2
re-integration latency	n/a n/a	n/a n/a	n/a n/a
exclusion latency	n/a n/a	n/a n/a	n/a n/a
exclusion failures	n/a n/a	n/a n/a	n/a n/a
successive usage period	$\mu_1 = 80$ $\sigma = 0$	$\mu_1 = 80$ $\sigma = 0$	$\mu_1 = 80$ $\sigma = 0$
$\#rounds(\mathcal{C},v) - \#consent(\mathcal{C},v)$ $\hookrightarrow \ell^{\mathcal{C}}_{total}$ $\hookrightarrow \#rounds(\mathcal{C},v)$	28 35 % 35 %	30 38 % 38 %	59 74 % 74 %
$\#rounds(\mathcal{C},v)$ $\hookrightarrow \ell^{\mathcal{C}}_{total}$	80 100 %	80 100 %	80 100 %
$\#consent(\mathcal{C},v)$ $\hookrightarrow \#rounds(\mathcal{C},v)$	31 39 %	33 41 %	20 25 %
normalised dissent	$\mu_{80} = 7,671$ $\sigma = 5,581$	$\mu_{80} = 7,228$ $\sigma = 5,429$	$\mu_{80} = 18,433$ $\sigma = 17,684$
end-to-end response time	$\mu_{80} = 9,420$ $\sigma = 2,485$	$\mu_{80} = 9,523$ $\sigma = 2,974$	$\mu_{80} = 22,747$ $\sigma = 10,032$
TTF $\hookrightarrow$ inv	0 n/a	1 1,000	7 0,143
XTTF-2 $\hookrightarrow$ inv	3 0,333	4 0,250	10 0,100
MTBF $\hookrightarrow$ inv	1,815 0,551	1,690 0,592	2,684 0,373
MTTR $\hookrightarrow$ inv	1,474 0,679	1,706 0,586	1,429 0,700

(c) scenario A: version-specific measurements

version	v_0	v_1	v_2	v_3
re-integration latency	$\mu_{12} = 1,000$ $\sigma = 0,000$	$\mu_4 = 1,500$ $\sigma = 0,577$	$\mu_{17} = 2,588$ $\sigma = 1,372$	$\mu_6 = 2,667$ $\sigma = 1,366$
exclusion latency	$\mu_{12} = 2,833$ $\sigma = 3,129$	$\mu_4 = 6,250$ $\sigma = 5,123$	$\mu_{18} = 1,056$ $\sigma = 0,236$	$\mu_6 = 3,167$ $\sigma = 4,834$
exclusion failures	$\mu_{12} = 1,750$ $\sigma = 0,866$	$\mu_4 = 2,750$ $\sigma = 1,708$	$\mu_{18} = 1,056$ $\sigma = 0,236$	$\mu_6 = 1,500$ $\sigma = 0,837$
successive usage period	$\mu_{12} = 5,583$ $\sigma = 4,680$	$\mu_6 = 6,000$ $\sigma = 5,404$	$\mu_{45} = 0,778$ $\sigma = 2,010$	$\mu_{16} = 2,312$ $\sigma = 5,351$
$\#rounds(\mathcal{C},v) - \#consent(\mathcal{C},v)$ $\hookrightarrow \ell^{\mathcal{C}}_{total}$ $\hookrightarrow \#rounds(\mathcal{C},v)$	21 26 % 31 %	17 21 % 23 %	19 24 % 54 %	15 19 % 24 %
$\#rounds(\mathcal{C},v)$ $\hookrightarrow \ell^{\mathcal{C}}_{total}$	68 85 %	74 93 %	35 44 %	63 79 %
$\#consent(\mathcal{C},v)$ $\hookrightarrow \#rounds(\mathcal{C},v)$	44 65 %	57 77 %	16 46 %	45 71 %
normalised dissent	$\mu_{80} = 1,534$ $\sigma = 1,026$	$\mu_{80} = 1,034$ $\sigma = 0,601$	$\mu_{80} = 1,946$ $\sigma = 1,137$	$\mu_{80} = 1,145$ $\sigma = 0,608$
end-to-end response time	$\mu_{68} = 9,445$ $\sigma = 2,497$	$\mu_{74} = 9,022$ $\sigma = 1,728$	$\mu_{35} = 17,148$ $\sigma = 10,715$	$\mu_{63} = 10,044$ $\sigma = 3,433$
TTF $\hookrightarrow$ inv	5 0,200	6 0,167	0 n/a	1 1,000
XTTF-2 $\hookrightarrow$ inv	7 0,143	8 0,125	4 0,250	10 0,100
MTBF $\hookrightarrow$ inv	2,600 0,385	3,125 0,320	7,222 0,138	4,357 0,230
MTTR $\hookrightarrow$ inv	2,062 0,485	1,438 0,696	3,625 0,276	2,818 0,355

(d) scenario B: version-specific measurements

Table 7.1: Experiment 7.2: redundancy configuration/allocation and failure statistics.



(a) scenario A: failures overview

(b) scenario B: failures overview

(c) scenario A: resource allocation & under-/overshooting

(d) scenario B: resource allocation & under-/overshooting

(e) scenario A: normalised dissent & observed disturbances

(f) scenario B: normalised dissent & observed disturbances

Figure 7.3: Experiment 7.3: triple-modular redundancy (TMR): using a static redundancy configuration (scenario A) *vs* using a dynamic redundancy configuration (scenario B). Version v_2 exhibits fault-free behaviour.

n_{min}	3	$\ell^{\mathcal{C}}_{failure}\ (c^{(\mathcal{C},\ell)}_{max} - m^{(\mathcal{C},\ell)} < 0)$		36% (29 of 80)
n_{max}	3	$\sum n^{(\mathcal{C},\ell)} = 240$	$\sum cr(e^{(\mathcal{C},\ell)}) = 256(256)$	$\triangle_{success}$	0% (0 of 0)
n_{init}	3	$\sum(n^{(\mathcal{C},\ell)} - cr(e^{(\mathcal{C},\ell)})) = -16(-16)$		$\nabla_{failure}$	0% (0 of 0)
t_{max}	30,000	$w_e^{(\mathcal{C},\ell)} \in_{51} [0{,}000, 1{,}000]$	$\mu = 0{,}588$	$\sigma = 0{,}497$	$ci_{90\%}$: (4.74E-1, 7.03E-1)
$\ell^{\mathcal{C}}_{total}$	80	$n^{(\mathcal{C},\ell)} \in [3,3]$	$\mu = 3{,}000$	$\sigma = 0{,}000$	$ci_{90\%}$: (3.00, 3.00)
TTF	27	$XTTF - 2 = 28$	$MTBF_{=28}$ 0,857	$MTTR_{=2}$ 14,500	

(a) scenario A: failures overview

n_{min}	3	$\ell^{\mathcal{C}}_{failure}\ (c^{(\mathcal{C},\ell)}_{max} - m^{(\mathcal{C},\ell)} < 0)$		33% (26 of 80)
n_{max}	3	$\sum n^{(\mathcal{C},\ell)} = 240$	$\sum cr(e^{(\mathcal{C},\ell)}) = 292(292)$	$\triangle_{success}$	0% (0 of 0)
n_{init}	3	$\sum(n^{(\mathcal{C},\ell)} - cr(e^{(\mathcal{C},\ell)})) = -52(-52)$		$\nabla_{failure}$	0% (0 of 0)
t_{max}	30,000	$w_e^{(\mathcal{C},\ell)} \in_{54} [0{,}000, 1{,}000]$	$\mu = 0{,}926$	$\sigma = 0{,}264$	$ci_{90\%}$: (8.67E-1, 9.85E-1)
$\ell^{\mathcal{C}}_{total}$	80	$n^{(\mathcal{C},\ell)} \in [3,3]$	$\mu = 3{,}000$	$\sigma = 0{,}000$	$ci_{90\%}$: (3.00, 3.00)
r_d	5	$\square_{fill}{}^{d} \in_{80} [0,5]$	$\mu = 2{,}075$	$\sigma = 2{,}017$	$ci_{90\%}$: (1.70, 2.45)
		$\square_{reset}{}^{d} \in_{15} [1,5]$	$\mu = 2{,}467$	$\sigma = 1{,}727$	$ci_{90\%}$: (1.73, 3.20)
r_u	1	$\square_{fill}{}^{u} \in_{80} [0,1]$	$\mu = 0{,}325$	$\sigma = 0{,}471$	$ci_{90\%}$: (2.38E-1, 4.12E-1)
		$\square_{reset}{}^{u} \in_{15} [1,1]$	$\mu = 1{,}000$	$\sigma = 0{,}000$	$ci_{90\%}$: (1.00, 1.00)
r_f	1	$\square_{corrupt}$	0% (0 of 80)		
TTF	6	$XTTF - 2 = 7$	$MTBF_{=25}$ 1,880	$MTTR_{=15}$ 1,733	
c_{sm}	f	$c_{sm} = 0$	n/a	n/a	$ci_{90\%}$: (0.00, 0.00)
		$\ell_{suspect}\ (0 \le c^{(\mathcal{C},\ell)}_{max} - m^{(\mathcal{C},\ell)} < c_{sm})$			0% (0 of 80)

(b) scenario B: failures overview

version	v_0	v_1	v_2
re-integration latency	n/a n/a	n/a n/a	n/a n/a
exclusion latency	n/a n/a	n/a n/a	n/a n/a
exclusion failures	n/a n/a	n/a n/a	n/a n/a
successive usage period	$\mu_1 = 80$ $\sigma = 0$	$\mu_1 = 80$ $\sigma = 0$	$\mu_1 = 80$ $\sigma = 0$
$\#rounds(\mathcal{C},v) - \#consent(\mathcal{C},v)$ $\hookrightarrow \ell^{\mathcal{C}}_{total}$ $\hookrightarrow \#rounds(\mathcal{C},v)$	44 55 % 55 %	44 55 % 55 %	n/a n/a n/a
$\#rounds(\mathcal{C},v)$ $\hookrightarrow \ell^{\mathcal{C}}_{total}$	80 100 %	80 100 %	80 100 %
$\#consent(\mathcal{C},v)$ $\hookrightarrow \#rounds(\mathcal{C},v)$	36 45 %	36 45 %	51 64 %
normalised dissent	$\mu_{80} = 7{,}667$ $\sigma = 7{,}340$	$\mu_{80} = 5{,}699$ $\sigma = 5{,}133$	$\mu_{80} = 3{,}575$ $\sigma = 4{,}640$
end-to-end response time	$\mu_{80} = 8{,}113$ $\sigma = 1{,}969$	$\mu_{80} = 8{,}318$ $\sigma = 2{,}535$	$\mu_{80} = 9{,}386$ $\sigma = 2{,}059$
TTF $\hookrightarrow$ inv	17 0,059	0 n/a	n/a n/a
XTTF-2 $\hookrightarrow$ inv	18 0,056	1 1,000	n/a n/a
MTBF $\hookrightarrow$ inv	0,442 2,263	0,837 1,194	n/a n/a
MTTR $\hookrightarrow$ inv	23,000 0,043	14,000 0,071	n/a n/a

(c) scenario A: resource allocation & under-/overshooting

version	v_0	v_1	v_2	v_3
re-integration latency	$\mu_{24} = 1{,}583$ $\sigma = 1{,}558$	$\mu_{20} = 1{,}300$ $\sigma = 0{,}923$	$\mu_3 = 1{,}333$ $\sigma = 0{,}577$	$\mu_9 = 1{,}111$ $\sigma = 0{,}333$
exclusion latency	$\mu_{24} = 1{,}625$ $\sigma = 1{,}469$	$\mu_{21} = 2{,}190$ $\sigma = 2{,}994$	n/a n/a	$\mu_9 = 5{,}667$ $\sigma = 9{,}301$
exclusion failures	$\mu_{24} = 1{,}417$ $\sigma = 0{,}830$	$\mu_{21} = 1{,}667$ $\sigma = 1{,}390$	n/a n/a	$\mu_9 = 3{,}333$ $\sigma = 3{,}391$
successive usage period	$\mu_{38} = 1{,}079$ $\sigma = 1{,}496$	$\mu_{27} = 1{,}963$ $\sigma = 3{,}425$	$\mu_4 = 16{,}500$ $\sigma = 26{,}640$	$\mu_{10} = 5{,}700$ $\sigma = 9{,}141$
$\#rounds(\mathcal{C},v) - \#consent(\mathcal{C},v)$ $\hookrightarrow \ell^{\mathcal{C}}_{total}$ $\hookrightarrow \#rounds(\mathcal{C},v)$	35 44 % 83 %	35 44 % 66 %	n/a n/a n/a	36 45 % 52 %
$\#rounds(\mathcal{C},v)$ $\hookrightarrow \ell^{\mathcal{C}}_{total}$	42 53 %	53 66 %	76 95 %	69 86 %
$\#consent(\mathcal{C},v)$ $\hookrightarrow \#rounds(\mathcal{C},v)$	9 21 %	19 36 %	49 64 %	35 51 %
normalised dissent	$\mu_{80} = 5{,}115$ $\sigma = 2{,}302$	$\mu_{80} = 4{,}655$ $\sigma = 2{,}235$	$\mu_{80} = 2{,}677$ $\sigma = 1{,}684$	$\mu_{80} = 3{,}315$ $\sigma = 1{,}751$
end-to-end response time	$\mu_{42} = 8{,}309$ $\sigma = 2{,}487$	$\mu_{53} = 8{,}002$ $\sigma = 1{,}431$	$\mu_{76} = 9{,}208$ $\sigma = 2{,}211$	$\mu_{69} = 9{,}865$ $\sigma = 3{,}176$
TTF $\hookrightarrow$ inv	1 1,000	0 n/a	n/a n/a	1 1,000
XTTF-2 $\hookrightarrow$ inv	2 0,500	6 0,167	n/a n/a	3 0,333
MTBF $\hookrightarrow$ inv	1,294 0,773	1,294 0,773	n/a n/a	1,200 0,833
MTTR $\hookrightarrow$ inv	10,500 0,095	5,625 0,178	n/a n/a	2,706 0,370

(d) scenario B: resource allocation & under-/overshooting

Table 7.2: Experiment 7.3: redundancy configuration/allocation and failure statistics.

the replica selection is static and immutable (scenario A), the overlapping regions of both sets of intervals will result in redundancy undershooting and thus interrupted service of the redundancy scheme.
Versions are configured with an exponentially distributed service time S ($\lambda = 0.4$), no waiting time ($c = k = \infty$) and constant network transmission times set to 3.5 time units. Except for version v_2, in scenario B, RVF and EVF failures are injected with an 80%, resp. with 20% probability, with RVF failures sampled from a uniform distribution (refer to p. 51). Failures injected using trend-based definitions; additional intermittent intervals/bursts of EVF failures are injected for versions v_0 and v_1. $t_{max} = 30$ time units. Normalised dissent values are computed with parameters set as follows: $k_1 = 0.85$, $k_2 = 0.75$ and $k_{max} = 0.95$.
Had a dynamic redundancy configuration been used (scenario B), where an additional standby version v_3 could be called in as a temporary replacement for a faulty versions v_0 and/or v_1, this would have lead to a slightly higher availability of the NVP composite: 67% as opposed to 64% in scenario A (compare Tables 7.2b with 7.2a). Note that in addition to the burst of failures injected for versions v_0 and v_1, we are injecting (on average) an additional error for each voting round, which may not necessarily match reality. If fewer additional disturbances were injected, this difference would become more recognisable. Furthermore, one can also see a better spreading of the unavailability of the redundancy scheme itself, where the use of a dynamic redundancy configuration (dynamic replica selection, yet constant degree of redundancy) results in higher MTBF and lower MTTR measurements being recorded.
Finally, comparison of Tables 7.2c and 7.2d clearly shows how an A-NVP scheme is more aware of the environment in which it is operating, and its ability to steer the redundancy configuration accordingly (planning), resulting in versions v_0 and v_1 being used in far fewer voting rounds than would have been the case for a static redundancy configuration.

  - A predetermined degree of redundancy is, however, cost ineffective in that it inhibits to economise on resource consumption in case the actual number of disturbances could be successfully overcome by a lesser amount of redundancy. Reversely, when the foreseen amount of redundancy is not enough to compensate for the currently experienced disturbances, the inclusion of additional resources (if available) may prevent further service disruption. Refer to experiment 7.4 on p. 133.

7.1.2 How Dynamic Redundancy Configurations may Address the Shortcomings of Static Configurations

Unlike traditional NVP, our A-NVP algorithm is responsible for maintaining a dynamic redundancy configuration. The classification defined in [36] can help to categorise the resilient behaviour of our solution:

- ✔ Although our A-NVP algorithm is categorised as a resilient software system **lacking perceptional abilities** to *directly* detect environmental change, it is believed to possess the property of **awareness**. Relevant information about the scheme as well as the underlying redundant resources is harvested during the scheme's operational life and is cached in memory — Sect. 5.4. This contextual

information can be used to deduce information about the environmental conditions in which the scheme is operating, *e.g.* what versions did or did not contribute to the scheme's dependability, and to what extent, how much time did it take to acquire a ballot from a specific version, *etc.*

✔ Such contextual information is then used to analyse if the scheme might benefit from adjusting the current redundancy configuration. Such approach of **purposefully planning** allows to optimise what part of the available redundancy is to be allocated, either in terms of replica selection or redundancy level, resulting in truly **dynamic** redundancy configurations.

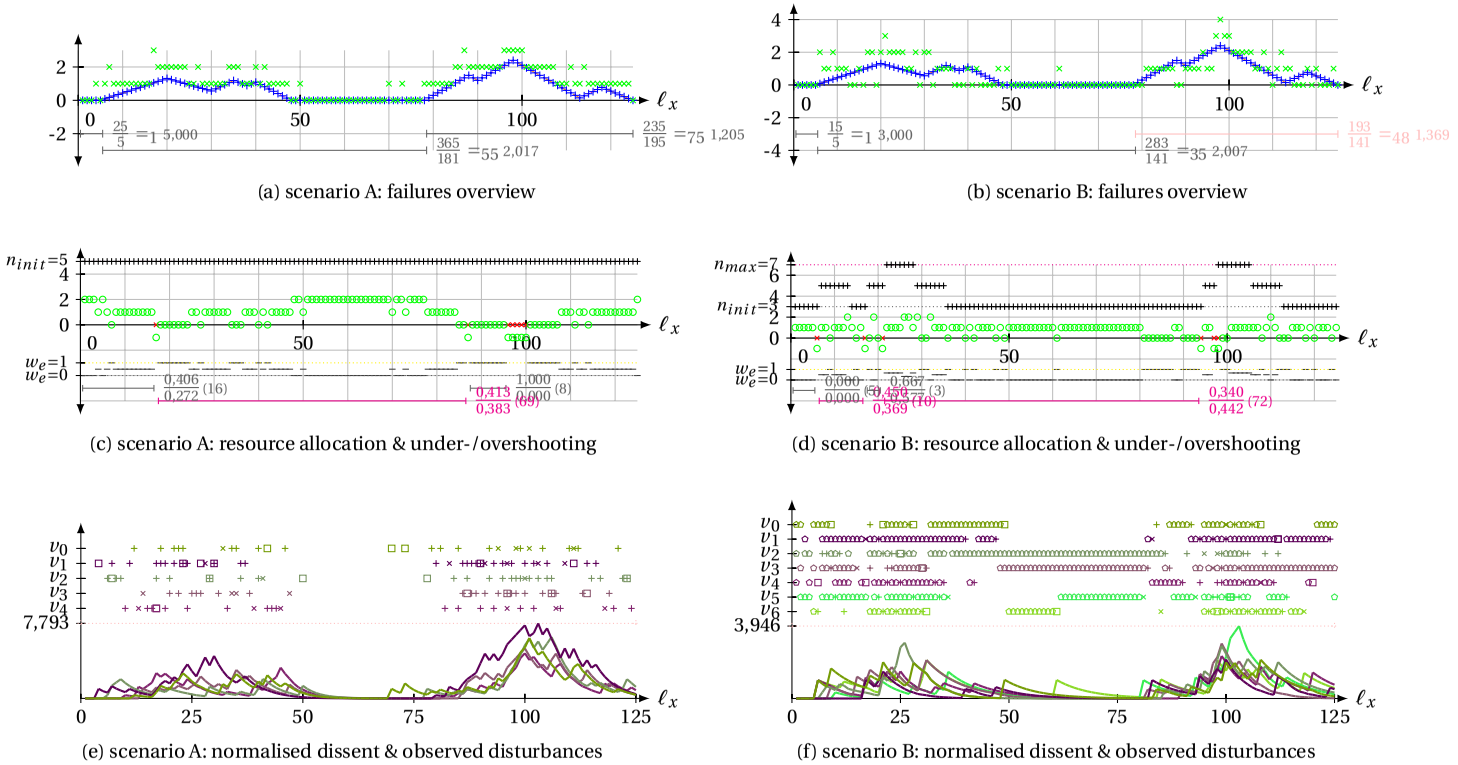
In case of redundancy undershooting, it can be beneficial to increase the degree of redundancy and to substitute poorly performing replicas for alternatives.
Redundancy undershooting is the event in which, for a particular redundancy configuration/level and during a given voting round, the maximum tolerable number of disturbances was exceeded — *cf.* Sect. 3.1.3 and Fig. 3.1 p. 66. Oftentimes, in classic redundancy schemata, this upper limit is defined before the system was placed in production. Although such systems are resilient in the sense that they will tolerate up to that level of disturbances, they will fail in case environmental change would result in additional versions being struck by failure (as illustrated in Fig.7.1d and 7.1f on p. 124).

Whereas static redundancy configurations would essentially be defenseless against such situations where bursts of failures would emerge, dynamic configurations might avoid failure by engaging additional versions, thereby increasing the degree of redundancy, or by removing poorly performing ones from the initial selection.

Experiments 7.2 and 7.3 illustrate how selecting the optimum set of versions proves effective to realise improved levels of availability. When adding dynamic redundancy scaling, additional benefits can be realised, including a reduction in resource expenditure (experiment 7.4).

Experiment 7.4

In Fig. 7.4, two fault-tolerant strategies are applied to the same pool of versions. On the left, a traditional NVP scheme is applied, with $n = 5$ (scenario A). On the right, an A-NVP scheme is applied, where the chosen degree of redundancy can vary between in $[3, 7]$ with increments or decrements of two (scenario B, an application of Strategy A, Variant 2, as defined in Sect. 7.3.1.2). The redundancy scheme is initialised such that it is capable of tolerating up to one failure, hence $n_{init} = 3$. The redundancy dimensioning and replica selection algorithm are defined using the model outlined in Chapt. 5, with $r_d = 7$, and $r_u = r_f = 1$. Such redundancy management approach is in line with what was published in [78, Sect. 3.3][a], and is *reactive* in nature, as redundancy is upscaled only when near-zero *dtof* measurements are recorded. If the voting scheme failed to find consensus amonst a majority of the replicas involved during the round $n^{(\mathcal{C},\ell\text{-}1)}$, the model will increase the number of redundant replicas to be used in the next voting round, to the extent that $n^{(\mathcal{C},\ell)} = n^{(\mathcal{C},\ell\text{-}1)} + 2$, provided that $n^{(\mathcal{C},\ell\text{-}1)} < |V|$. Conversely, when the scheme was able to produce an outcome with a given amount of redundancy for a certain amount r_d of consecutive voting rounds, a lower degree of redundancy shall be used for the next voting round $(\mathcal{C}, \ell)$, involving $n^{(\mathcal{C},\ell)} = max(3, ((\mathcal{C}, \ell)^{(\ell\text{-}1} + 2))$ replicas.



(a) scenario A: failures overview

(b) scenario B: failures overview

(c) scenario A: resource allocation & under-/overshooting

(d) scenario B: resource allocation & under-/overshooting

(e) scenario A: normalised dissent & observed disturbances

(f) scenario B: normalised dissent & observed disturbances

Figure 7.4: Experiment 7.4: using a static redundancy configuration on the left (scenario A) *vs* using a dynamic redundancy configuration on the right (scenario B). Scenario B is successful in tolerating a similar series of failure with significantly less resource expenditure.

n_{min}	5	$\ell^{\mathcal{C}}_{failure}\,(c^{(\mathcal{C},\ell)}_{max} - m^{(\mathcal{C},\ell)} < 0)$			6% (7 of 125)
n_{max}	5	$\sum n^{(\mathcal{C},\ell)} = 625$	$\sum cr(e^{(\mathcal{C},\ell)}) = 381(405)$	$\triangle_{success}$	0% (0 of 0)
n_{init}	5	$\sum(n^{(\mathcal{C},\ell)} - cr(e^{(\mathcal{C},\ell)})) = 244(220)$		$\nabla_{failure}$	0% (0 of 0)
t_{max}	16,000	$w_e^{(\mathcal{C},\ell)} \in_{118} [0,000, 1,000]$	$\mu = 0{,}504$	$\sigma = 0{,}373$	$ci_{90\%}$: (4.48E-1, 5.61E-1)
$\ell^{\mathcal{C}}_{total}$	125	$n^{(\mathcal{C},\ell)} \in [5, 5]$	$\mu = 5{,}000$	$\sigma = 0{,}000$	$ci_{90\%}$: (5.00, 5.00)
TTF	16	$XTTF-2 = 86$	$MTBF =_6 12{,}833$	$MTTR =_3 2{,}333$	

(a) scenario A: failures overview

n_{min}	3	$\ell^{\mathcal{C}}_{failure}\,(c^{(\mathcal{C},\ell)}_{max} - m^{(\mathcal{C},\ell)} < 0)$			5% (6 of 125)
n_{max}	7	$\sum n^{(\mathcal{C},\ell)} = 491$	$\sum cr(e^{(\mathcal{C},\ell)}) = 287(315)$	$\triangle_{success}$	80% (4 of 5)
n_{init}	3	$\sum(n^{(\mathcal{C},\ell)} - cr(e^{(\mathcal{C},\ell)})) = 204(176)$		$\nabla_{failure}$	0% (0 of 5)
t_{max}	16,000	$w_e^{(\mathcal{C},\ell)} \in_{119} [0,000, 1,000]$	$\mu = 0{,}389$	$\sigma = 0{,}433$	$ci_{90\%}$: (3.24E-1, 4.55E-1)
$\ell^{\mathcal{C}}_{total}$	125	$n^{(\mathcal{C},\ell)} \in [3, 7]$	$\mu = 3{,}928$	$\sigma = 1{,}404$	$ci_{90\%}$: (3.72, 4.13)
r_d	7	$\square_{fill}{}^{d} \in_{125} [0, 7]$	$\mu = 5{,}000$	$\sigma = 2{,}436$	$ci_{90\%}$: (4.64, 5.36)
		$\square_{reset}{}^{d} \in_{10} [2, 7]$	$\mu = 5{,}500$	$\sigma = 2{,}068$	$ci_{90\%}$: (4.42, 6.58)
r_u	1	$\square_{fill}{}^{u} \in_{125} [0, 1]$	$\mu = 0{,}048$	$\sigma = 0{,}215$	$ci_{90\%}$: (1.64E-2, 7.96E-2)
		$\square_{reset}{}^{u} \in_{6} [1, 1]$	$\mu = 1{,}000$	$\sigma = 0{,}000$	$ci_{90\%}$: (1.00, 1.00)
r_f	1	$\square_{corrupt}$	0% (0 of 125)		
TTF	5	$XTTF-2 = 16$	$MTBF =_5 17{,}400$	$MTTR =_5 1{,}200$	

(b) scenario B: failures overview

version	v_0	v_1	v_2	v_3	v_4
re-integration latency	n/a n/a	n/a n/a	n/a n/a	n/a n/a	n/A n/a
exclusion latency	n/a n/a	n/a n/a	n/a n/a	n/a n/a	n/a n/a
exclusion failures	n/a n/a	n/a n/a	n/a n/a	n/a n/a	n/a n/a
successive usage period	$\mu_1 = 125$ $\sigma = 0$	$\mu_1 = 125$ $\sigma = 0$	$\mu_1 = 125$ $\sigma = 0$	$\mu_1 = 125$ $\sigma = 0$	$\mu_1 = 125$ $\sigma = 0$
$\#rounds(\mathcal{C}, v) - \#consent(\mathcal{C}, v)$	26	34	28	27	25
$\hookrightarrow \ell^{\mathcal{C}}_{total}$	21 %	27 %	22 %	22 %	20 %
$\hookrightarrow \#rounds(\mathcal{C}, v)$	21 %	27 %	22 %	22 %	20 %
$\#rounds(\mathcal{C}, v)$	125	125	125	125	125
$\hookrightarrow \ell^{\mathcal{C}}_{total}$	100 %	100 %	100 %	100 %	100 %
$\#consent(\mathcal{C}, v)$	94	89	96	96	96
$\hookrightarrow \#rounds(\mathcal{C}, v)$	75 %	71 %	77 %	77 %	77 %
normalised dissent	$\mu_{125} = 1{,}236$ $\sigma = 1{,}294$	$\mu_{125} = 1{,}909$ $\sigma = 2{,}116$	$\mu_{125} = 1{,}161$ $\sigma = 1{,}455$	$\mu_{125} = 1{,}191$ $\sigma = 1{,}280$	$\mu_{125} = 1{,}068$ $\sigma = 1{,}112$
end-to-end response time	$\mu_{125} = 9{,}335$ $\sigma = 2{,}290$	$\mu_{125} = 9{,}247$ $\sigma = 2{,}372$	$\mu_{125} = 9{,}318$ $\sigma = 2{,}248$	$\mu_{125} = 9{,}404$ $\sigma = 2{,}471$	$\mu_{125} = 9{,}379$ $\sigma = 2{,}189$
TTF	11	6	5	13	9
$\hookrightarrow$ inv	0,091	0,167	0,200	0,077	0,111
XTTF-2	17	10	7	19	12
$\hookrightarrow$ inv	0,059	0,100	0,143	0,053	0,083
MTBF	3,955	2,633	3,875	3,375	3,957
$\hookrightarrow$ inv	0,253	0,380	0,258	0,296	0,253
MTTR	1,353	1,632	1,471	1,389	1,263
$\hookrightarrow$ inv	0,739	0,613	0,680	0,720	0,792

(c) scenario A: resource allocation & under-/overshooting

version	v_0	v_1	v_2	v_3	v_4	v_5	v_6
re-integration latency	$\mu_8 = 8{,}125$ $\sigma = 11{,}090$	$\mu_5 = 9{,}600$ $\sigma = 13{,}903$	$\mu_{11} = 2{,}364$ $\sigma = 1{,}120$	$\mu_8 = 4{,}125$ $\sigma = 4{,}970$	$\mu_{10} = 6{,}700$ $\sigma = 11{,}842$	$\mu_8 = 6{,}250$ $\sigma = 8{,}207$	$\mu_6 = 10{,}167$ $\sigma = 8{,}542$
exclusion latency	$\mu_5 = 2{,}800$ $\sigma = 3{,}493$	$\mu_5 = 9{,}200$ $\sigma = 11{,}367$	$\mu_8 = 3{,}250$ $\sigma = 3{,}240$	$\mu_5 = 4{,}000$ $\sigma = 4{,}243$	$\mu_5 = 6{,}200$ $\sigma = 7{,}120$	$\mu_7 = 4{,}286$ $\sigma = 4{,}112$	$\mu_6 = 5{,}500$ $\sigma = 7{,}314$
exclusion failures	$\mu_5 = 1{,}400$ $\sigma = 0{,}894$	$\mu_5 = 1{,}800$ $\sigma = 1{,}095$	$\mu_8 = 2{,}000$ $\sigma = 1{,}414$	$\mu_5 = 1{,}600$ $\sigma = 0{,}894$	$\mu_5 = 2{,}400$ $\sigma = 1{,}949$	$\mu_7 = 2{,}000$ $\sigma = 1{,}915$	$\mu_6 = 2{,}000$ $\sigma = 1{,}673$
successive usage period	$\mu_{65} = 0{,}846$ $\sigma = 3{,}073$	$\mu_{49} = 1{,}510$ $\sigma = 5{,}839$	$\mu_{39} = 2{,}205$ $\sigma = 8{,}186$	$\mu_{33} = 1{,}848$ $\sigma = 6{,}384$	$\mu_{72} = 0{,}736$ $\sigma = 2{,}763$	$\mu_{50} = 1{,}460$ $\sigma = 4{,}127$	$\mu_{68} = 0{,}779$ $\sigma = 3{,}190$
$\#rounds(\mathcal{C}, v) - \#consent(\mathcal{C}, v)$	12	10	17	12	15	14	15
$\hookrightarrow \ell^{\mathcal{C}}_{total}$	10 %	8 %	14 %	10 %	12 %	11 %	12 %
$\hookrightarrow \#rounds(\mathcal{C}, v)$	20 %	14 %	20 %	13 %	28 %	19 %	28 %
$\#rounds(\mathcal{C}, v)$	60	74	86	91	53	74	53
$\hookrightarrow \ell^{\mathcal{C}}_{total}$	48 %	59 %	69 %	73 %	42 %	59 %	42 %
$\#consent(\mathcal{C}, v)$	45	62	69	77	37	59	37
$\hookrightarrow \#rounds(\mathcal{C}, v)$	75 %	84 %	80 %	85 %	70 %	80 %	70 %
normalised dissent	$\mu_{125} = 0{,}621$ $\sigma = 0{,}573$	$\mu_{125} = 0{,}385$ $\sigma = 0{,}445$	$\mu_{125} = 0{,}627$ $\sigma = 0{,}757$	$\mu_{125} = 0{,}560$ $\sigma = 0{,}660$	$\mu_{125} = 0{,}562$ $\sigma = 0{,}518$	$\mu_{125} = 0{,}595$ $\sigma = 0{,}758$	$\mu_{125} = 0{,}674$ $\sigma = 0{,}510$
end-to-end response time	$\mu_{60} = 9{,}894$ $\sigma = 2{,}682$	$\mu_{74} = 9{,}034$ $\sigma = 2{,}019$	$\mu_{86} = 9{,}405$ $\sigma = 2{,}240$	$\mu_{91} = 9{,}209$ $\sigma = 2{,}010$	$\mu_{53} = 9{,}654$ $\sigma = 2{,}564$	$\mu_{74} = 9{,}190$ $\sigma = 2{,}184$	$\mu_{53} = 9{,}553$ $\sigma = 2{,}384$
TTF	17	16	6	8	21	9	5
$\hookrightarrow$ inv	0,059	0,062	0,167	0,125	0,048	0,111	0,200
XTTF-2	26	19	7	14	24	19	11
$\hookrightarrow$ inv	0,038	0,053	0,143	0,071	0,042	0,053	0,091
MTBF	13,667	12,375	6,000	8,400	7,091	7,000	9,182
$\hookrightarrow$ inv	0,073	0,081	0,167	0,119	0,141	0,143	0,109
MTTR	1,857	6,500	3,667	5,000	7,778	5,700	3,100
$\hookrightarrow$ inv	0,538	0,154	0,273	0,200	0,129	0,175	0,323

(d) scenario B: resource allocation & under-/overshooting

Table 7.3: Experiment 7.4: redundancy configuration/allocation and failure statistics.

Versions are configured with an exponentially distributed service time S ($\lambda = 0.4$), no waiting time ($c = k = \infty$) and constant network transmission times set to 3.5 time units. RVF and EVF failures are injected with an 80%, resp. with 20% probability, with RVF failures sampled from a uniform distribution (refer to p. 51). Failures injected using trend-based definitions. $t_{max} = 16$ time units. Normalised dissent values are computed with parameters set as follows: $k_1 = 0.85$, $k_2 = 0.75$ and $k_{max} = 0.95$.
In scenario A, the designer expected a maximum of two disturbances affecting the composite's underlying resources, hence the predetermined degree of redundancy $n = 5$. In scenario B, resources are allocated parsimoniously: the designer takes the risk to start with a lower degree of redundancy, knowing that more resources can be called in whenever needed (within a predefined limit of total available redundancy $|V| = 7$).
From the statistics listed in Table 7.3, one can see that a dynamic redundancy scheme has the potential to realise a significant reduction in resource expenditure, while better sustaining the scheme's availability:

– For most of its operational life, the scheme in scenario A is plagued by **redundancy overshooting**. In scenario B, whenever the system deems it safe or opportune to do so, it will attemt to downscale the currently applied degree of redundancy. The statistics recorded clearly show that scenario B is successful in tolerating a similar series of failure with a significantly lower resource expenditure: whereas an average resource consumpion of 5 redundancy units (versions) is recorded per voting round for scenario A, the average number of allocated resources per voting round in scenario B is a mere $\sum n^{(\mathcal{C},\ell)}/125 = 3.925$ — a reduction of 27.38%!

– In scenario B, when the scheme's environment suffers from more than the initially anticipated number of disturbances, the system will reach out to additional resources/redundancy, notwithstanding the fact that versions are dynamically selected at runtime. Trying to parsimoniously allocate system resources, when defining a static redundacy configuration, the designer has no other option than to make a trade-off between the risk appetite and the cost resulting from involving additional redundancy. Moreover, as exemplified by experiment 7.1, when hardwiring poorly performing versions, the additional cost may prove to be pointless.
One of the advantages of using an adaptive redundancy allocation mechanism is that it allows to temporarily go beyond the redundancy level that was initially judged appropriate, in case of **redundancy undershooting**. Such situation would occur when (i) replicas perform worse than expected, or (ii) when more disturbances than initially foreseen would simultaneously affect the scheme's underlying resources (due to, *e.g.* exogenous factors that may lead to additional LRF and/or EVF failures). As we can see from Fig. 7.4a and 7.4c and 7.4b and 7.4d, scenario B would perform better to counterbalance such peaks, given the availability of some remaining unallocated redundancy, and that those additional resources would perform well. By consequence, this could result in a slightly improved overall availability of the redundancy scheme: 95 and 94% for scenarios B, resp. A — *cf.* Tables 7.3a and 7.3b.
The applied algorithm in scenario B was, however, configured to work reactively. Had it been configured to apply, *e.g.*, an additional safety margin c_{sm} to proactively detect an emerging or upward failure trend, it would have resulted in an even higher availability of the redundancy scheme. Alternatively, a hybrid constellation could be devised in which an RB-like retry mechanism would be triggered to recover from failed voting rounds (using the same versions selected before, or

alternative ones). Both solutions would improve the scheme's dependability, at the expense of a slighly higher redundancy consumption (and, obviously, response times), though still lower than the cumulative resource consumption of the static configuration depicted in scenario A[b].

In line with previous experiments, a dynamic redundancy configuration can effectively realise lower MTTR and higher MTBF measurements than a static redundancy configuration. Note that — given a specific environment (failure model/trend) — the allocation of less redundancy will result in less disturbances to be masked by the redundancy scheme[c]. This can be observed in Tables 7.3c and 7.3d, by adding all voting rounds for which each specific engaged version was affected by one or more disturbances (140 in scenario A as opposed to 95 in scenario B). Quite convincing results for a simple A-NVP configuration that is mainly reactive and only upscales after undershooting. Capturing and analysing contextual data at runtime can help to act in a more proactive manner. As a final note, the differences in the way both approaches perform would obviously depend on the nature of the disturbances that materialise, the pace at which they occur, and their effect on the underlying versions. In short: all is determined by the system-environment fit.

[a] Even though it was applied on redundant data structures, it can readily be reused for dynamically determining the redundancy level. In line with traditional NVP, it will only report an odd degree of redundancy. Note that we have combined our dynamic replica selection algorithm with the initial algorithm, which in its original form selects versions at random from the available pool of resources. Had we not done so, the results would have shown less positive results, at the expense of a lower availability of the scheme.

[b] 6 failures times use of all available redundancy, or 42, plus 491 units already used, resulting in 533 units — still a saving of approximately 15%.

[c] The actual number of injected disturbances for any specific voting round is eventually determined by the amount of redundancy that is actually used.

Dynamically optimising the redundancy configuration can help to realise shorter average MTTR and prolonged time to failure.

Because of the previous proposition, if a dynamic redundancy configuration would be more resilient to overcome unforeseen bursts of disturbances, that would obviously translate in less failures of the NVP composite as a whole. Such lower probability should normally reflect in time between failure measurements of greater magnitude being recorded, as well as lower time to repair values. This seems to be corroborated by experiments 7.1–7.4.

In some cases, a dynamic redundancy configuration can realise a reduction in resource consumption/expenditure. In traditional NVP, the maximum number of tolerable disturbances that a given redundancy scheme can tolerate at a time is determined upfront, based on estimations, assumptions and analyses of the environment, and the available versions in the system. As dependability is the primary objective of any fault-tolerant system, the redundancy configuration to be used (and the degree of redundancy in particular) is typically chosen so that the system will for sure be resilient to withstand the worst anticipated scenario. However, in reality, this worst-case scenario hardly occurs, and during most of the scheme's operational life, an excessively large share of the available redundancy will therefore be allocated. Reducing the degree of redundancy is likely to result in a saving in

resource expenditure, without necessarily degrading the scheme's dependability characteristics.

In case of disturbances emerging due to exogenous factors, *e.g.* congestion on specific network links that may result in LRF failures, there might be a short period during which a clearly detectable trend announces an exceptionally high level of disturbances that the currently used redundancy configuration may soon no longer be able to tolerate. A-NVP is able to detect such trends, and to act accordingly. It can update the redundancy configuration by involving more resources (redundancy) to temporarily overcome this exceptional state, thereby safeguarding the scheme's resilience, and allocating redundancy on a just-in-time basis. As soon as the system considers it safe to relinquish the additional redundancy, it can gradually do so, and return to normal operations. This behaviour is exemplified by the simulations presented and discussed in experiment 7.4.

To conclude this section, a few notes though:

– In all of the experiments shown above, whenever a dynamic redundancy configuration was applied, replicas were selected using the model defined in Sect. 5.3, which was configured to target sustained availability only. More specifically, $w_D^{\mathcal{C}} = 1$ and $w_T^{\mathcal{C}} = w_L^{\mathcal{C}} = 0$ — *v.* Sect. 5.1.

– The above experiments, the environmental conditions and runtime parameters of the A-NVP algorithm were specifically chosen for demonstration purposes so as to enable the model's effectiveness to be concisely captured, and to corroborate the statements made with respect to the potentials gains and advantages in using dynamic redundancy schemata. Even though the plots in previous figures are of great help in analysing, comprehending and comparing the behaviour and performance of traditional *vs* adaptive NVP redundancy schemata, it is difficult to include them in printed form for longer operational life spans without affecting readability. The behaviour of redundancy configurations — be it static or dynamic — should be analysed over longer operational intervals, especially since the system-environment fit may evolve over time (like in experiment 7.3, for instance).

– Even though our discrete event simulation toolbox provides a multitude of failure injection mechanisms, a simplistic, trend-based injection mechanism was chosen for visualisation purposes, as illustrated in Fig. 7.1–7.4. Despite its convenience and simplicity, the trend-based failure injection approach does introduce some form of randomness that may complicate detailed analysis. By default, once the replicas were selected that will be used throughout a specific voting round, this type of failure injection will select, usually at random, a subset of these replicas for which disturbances will be injected — *v.* Sect. 6.5.1. This behaviour can be overridden though, as we did in experiments 7.2 and 7.3. A typical case would be when a specific (sub)set of versions should perform badly while analysing the impact on the NVP composite's behaviour.

7.2 Detecting Faulty Replicas, Removing them from Service, and Re-integrating them once they Recover

The main conclusion to be drawn when analysing experiment 7.1 is that faulty versions jeopardise the availability of the redundancy scheme. However, a specific version may only be periodically affected by disturbances — intervals during which use of the version should be avoided. Contrarily, when that version is functioning as expected, it will contribute in sustaining the scheme's overall availability.

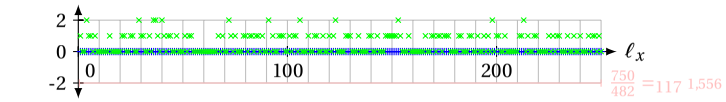
It is therefore useful to see to what extent the replica selection algorithm proposed in Sect. 5.3 is able to act decisively and (temporarily) take versions that are suspected to have failed out of service, and if it is capable of re-allocating them when it is safe to do so. To illustrate this, we will compare various failure occurrence patterns and assess the different impact on the **replica selection procedure**.

– When faulty versions recover and return to their normal functioning, if the risk that they will fail once more is low to moderate, it makes sense to use them again. After all, the are very likely to further support the availability of the redundancy scheme. The **re-integration latency** represents the delay between the moment a particular version is taken out of service, until that decision is undone, and the version is allocated again.

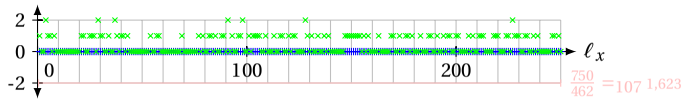
– Contrarily, once a version becomes faulty, it should no longer be part of the redundancy configuration, for its continued use is likely to put at risk the scheme's overall availability. By **exclusion latency**, we refer to the delay between the moment a failure occurs, or a burst of failures starts, until the time when this is detected/perceived and the system acts accordingly by removing it from the current redundancy configuration.

Both measures were already briefly touched in Sect. 6.2.4. As we pointed out in Sect. 5.1, the particular application requirements may require to select versions carefully, whereby a suitable trade-off should be found between the different potential application objectives: dependability, timeliness and load balancing. The environmental context in which the scheme is operating, together with the specific objectives, will determine what contextual information is used by the A-NVP algorithm to select (and remove) versions from the redundancy configuration. A full investigation is out of scope of this dissertation, and we will focus primarily on the dependability objective (as argued in note 1, p. 125).

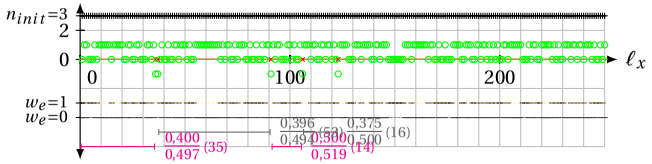
In Chapt. 4, we defined the normalised dissent metric as a mathematical structure to approximate the reliability of individual versions. This measure lies at the heart of our A-NVP algorithm, and it can be explicitly configured so as to allow steering the way in which the redundancy configuration may change — *cf.* Sect. 5.3. In particular, the designer can set the parameters $0 < k_2 < k_1 < k_{max} < 1$ of the reward model — *cf.* Sect. 4.3. In doing so, (s)he could choose to apply a more aggressive allocation policy for versions that were suspected (or found) to have failed. The values chosen for these configuration parameters will therefore have an effect on the re-integration latencies.



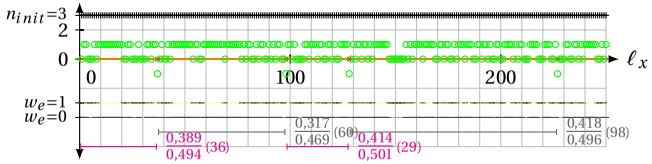
(a) scenario A: failures overview



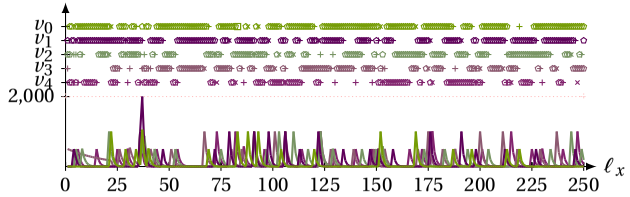
(b) scenario B: failures overview



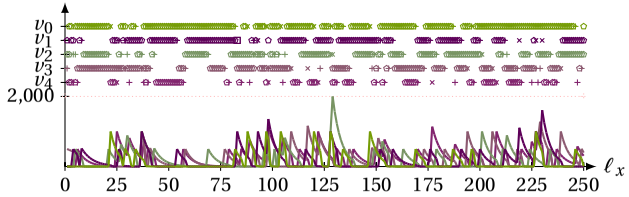
(c) scenario A: resource allocation & under-/overshooting



(d) scenario B: resource allocation & under-/overshooting



(e) scenario A: normalised dissent & observed disturbances



(f) scenario B: normalised dissent & observed disturbances

Figure 7.5: Experiment 7.5: the configuration parameters of the reward model affect the version re-integration latency, which in itself poses risk and may put at risk the overall redundancy scheme dependability.

n_{min}	3	$\ell^{\mathcal{C}}_{failure}\,(c^{(\mathcal{C},\ell)}_{max} - m^{(\mathcal{C},\ell)} < 0)$			2% (5 of 250)
n_{max}	3	$\sum n^{(\mathcal{C},\ell)} = 750$	$\sum cr(e^{(\mathcal{C},\ell)}) = 482(484)$	$\triangle_{success}$	0% (0 of 0)
n_{init}	3	$\sum(n^{(\mathcal{C},\ell)} - cr(e^{(\mathcal{C},\ell)})) = 268(266)$		$\triangledown_{failure}$	0% (0 of 0)
t_{max}	25,000	$w_e^{(\mathcal{C},\ell)} \in_{245} [0,000, 1,000]$	$\mu = 0,408$	$\sigma = 0,492$	$ci_{90\%}$: (3.56E-1, 4.60E-1)
$\ell^{\mathcal{C}}_{total}$	250	$n^{(\mathcal{C},\ell)} \in [3,3]$	$\mu = 3,000$	$\sigma = 0,000$	$ci_{90\%}$: (3.00, 3.00)
r_d	3	$\square_{fill}{}^{d} \in_{250} [0,3]$	$\mu = 2,880$	$\sigma = 0,516$	$ci_{90\%}$: (2.83, 2.93)
		$\square_{reset}{}^{d} \in_{4} [3,3]$	$\mu = 3,000$	$\sigma = 0,000$	$ci_{90\%}$: (3.00, 3.00)
r_u	1	$\square_{fill}{}^{u} \in_{250} [0,1]$	$\mu = 0,020$	$\sigma = 0,140$	$ci_{90\%}$: (5.41E-3, 3.46E-2)
		$\square_{reset}{}^{u} \in_{4} [1,1]$	$\mu = 1,000$	$\sigma = 0,000$	$ci_{90\%}$: (1.00, 1.00)
r_f	1	$\square_{corrupt}$	0% (0 of 250)		
TTF	35	$XTTF - 2 = 36$	$MTBF =_4 20,750$	$MTTR =_4 1,250$	
c_{sm}	f	$c_{sm} = 0$	n/a	n/a	$ci_{90\%}$: (0.00, 0.00)
		$\ell_{suspect}\,(0 \le c^{(\mathcal{C},\ell)}_{max} - m^{(\mathcal{C},\ell)} < c_{sm})$			0% (0 of 250)

(a) scenario A: failures overview

n_{min}	3	$\ell^{\mathcal{C}}_{failure}\,(c^{(\mathcal{C},\ell)}_{max} - m^{(\mathcal{C},\ell)} < 0)$			2% (4 of 250)
n_{max}	3	$\sum n^{(\mathcal{C},\ell)} = 750$	$\sum cr(e^{(\mathcal{C},\ell)}) = 462(464)$	$\triangle_{success}$	0% (0 of 0)
n_{init}	3	$\sum(n^{(\mathcal{C},\ell)} - cr(e^{(\mathcal{C},\ell)})) = 288(286)$		$\triangledown_{failure}$	0% (0 of 0)
t_{max}	25,000	$w_e^{(\mathcal{C},\ell)} \in_{246} [0,000, 1,000]$	$\mu = 0,390$	$\sigma = 0,489$	$ci_{90\%}$: (3.39E-1, 4.42E-1)
$\ell^{\mathcal{C}}_{total}$	250	$n^{(\mathcal{C},\ell)} \in [3,3]$	$\mu = 3,000$	$\sigma = 0,000$	$ci_{90\%}$: (3.00, 3.00)
r_d	3	$\square_{fill}{}^{d} \in_{250} [0,3]$	$\mu = 2,892$	$\sigma = 0,483$	$ci_{90\%}$: (2.84, 2.94)
		$\square_{reset}{}^{d} \in_{4} [3,3]$	$\mu = 3,000$	$\sigma = 0,000$	$ci_{90\%}$: (3.00, 3.00)
r_u	1	$\square_{fill}{}^{u} \in_{250} [0,1]$	$\mu = 0,016$	$\sigma = 0,126$	$ci_{90\%}$: (2.92E-3, 2.91E-2)
		$\square_{reset}{}^{u} \in_{4} [1,1]$	$\mu = 1,000$	$\sigma = 0,000$	$ci_{90\%}$: (1.00, 1.00)
r_f	1	$\square_{corrupt}$	0% (0 of 250)		
TTF	36	$XTTF - 2 = 97$	$MTBF =_3 62,333$	$MTTR =_4 1,000$	
c_{sm}	f	$c_{sm} = 0$	n/a	n/a	$ci_{90\%}$: (0.00, 0.00)
		$\ell_{suspect}\,(0 \le c^{(\mathcal{C},\ell)}_{max} - m^{(\mathcal{C},\ell)} < c_{sm})$			0% (0 of 250)

(b) scenario B: failures overview

version	v_0	v_1	v_2	v_3	v_4
re-integration latency	$\mu_{18} = 3,056$ $\sigma = 1,305$	$\mu_{21} = 3,667$ $\sigma = 2,309$	$\mu_{24} = 4,500$ $\sigma = 2,687$	$\mu_{24} = 5,208$ $\sigma = 4,128$	$\mu_{24} = 5,375$ $\sigma = 3,033$
exclusion latency	$\mu_{12} = 1,000$ $\sigma = 0,000$	$\mu_{15} = 1,533$ $\sigma = 2,066$	$\mu_{23} = 2,652$ $\sigma = 4,638$	$\mu_{24} = 1,292$ $\sigma = 0,908$	$\mu_{25} = 2,320$ $\sigma = 3,051$
exclusion failures	$\mu_{12} = 1,000$ $\sigma = 0,000$	$\mu_{15} = 1,067$ $\sigma = 0,258$	$\mu_{23} = 1,217$ $\sigma = 0,422$	$\mu_{24} = 1,167$ $\sigma = 0,482$	$\mu_{25} = 1,280$ $\sigma = 0,458$
successive usage period	$\mu_{55} = 3,073$ $\sigma = 6,602$	$\mu_{77} = 2,234$ $\sigma = 5,224$	$\mu_{109} = 1,294$ $\sigma = 3,505$	$\mu_{125} = 0,952$ $\sigma = 2,498$	$\mu_{132} = 0,886$ $\sigma = 2,789$
$\#rounds(\mathcal{C},v) - \#consent(\mathcal{C},v)$ $\hookrightarrow \ell^{\mathcal{C}}_{total}$ $\hookrightarrow \#rounds(\mathcal{C},v)$	13 5 % 7 %	16 6 % 9 %	28 11 % 20 %	28 11 % 22 %	32 13 % 27 %
$\#rounds(\mathcal{C},v)$ $\hookrightarrow \ell^{\mathcal{C}}_{total}$	194 78 %	173 69 %	141 56 %	125 50 %	117 47 %
$\#consent(\mathcal{C},v)$ $\hookrightarrow \#rounds(\mathcal{C},v)$	177 91 %	152 88 %	116 82 %	99 79 %	91 78 %
normalised dissent	$\mu_{250} = 0,070$ $\sigma = 0,196$	$\mu_{250} = 0,092$ $\sigma = 0,242$	$\mu_{250} = 0,104$ $\sigma = 0,215$	$\mu_{250} = 0,142$ $\sigma = 0,235$	$\mu_{250} = 0,112$ $\sigma = 0,218$
end-to-end response time	$\mu_{194} = 9,715$ $\sigma = 2,971$	$\mu_{173} = 9,523$ $\sigma = 2,709$	$\mu_{141} = 9,270$ $\sigma = 2,475$	$\mu_{125} = 9,140$ $\sigma = 2,211$	$\mu_{117} = 9,280$ $\sigma = 2,306$
TTF $\hookrightarrow$ inv	21 0,048	28 0,036	3 0,333	0 n/a	3 0,333
XTTF-2 $\hookrightarrow$ inv	33 0,030	36 0,028	4 0,250	25 0,040	5 0,200
MTBF $\hookrightarrow$ inv	16,909 0,059	13,467 0,074	8,074 0,124	7,741 0,129	6,839 0,146
MTTR $\hookrightarrow$ inv	4,333 0,231	4,938 0,203	5,864 0,171	6,652 0,150	6,652 0,150

(c) scenario A: resource allocation & under-/overshooting

version	v_0	v_1	v_2	v_3	v_4
re-integration latency	$\mu_{16} = 2,625$ $\sigma = 1,147$	$\mu_{20} = 4,800$ $\sigma = 3,381$	$\mu_{21} = 4,667$ $\sigma = 3,440$	$\mu_{20} = 5,150$ $\sigma = 3,281$	$\mu_{26} = 5,962$ $\sigma = 4,209$
exclusion latency	$\mu_{14} = 1,000$ $\sigma = 0,000$	$\mu_{15} = 1,667$ $\sigma = 2,093$	$\mu_{20} = 1,450$ $\sigma = 1,191$	$\mu_{20} = 2,150$ $\sigma = 3,543$	$\mu_{27} = 1,148$ $\sigma = 0,534$
exclusion failures	$\mu_{14} = 1,000$ $\sigma = 0,000$	$\mu_{15} = 1,067$ $\sigma = 0,258$	$\mu_{20} = 1,200$ $\sigma = 0,410$	$\mu_{20} = 1,150$ $\sigma = 0,489$	$\mu_{27} = 1,074$ $\sigma = 0,267$
successive usage period	$\mu_{42} = 4,905$ $\sigma = 10,745$	$\mu_{96} = 1,490$ $\sigma = 4,230$	$\mu_{98} = 1,418$ $\sigma = 3,744$	$\mu_{104} = 1,404$ $\sigma = 4,486$	$\mu_{158} = 0,576$ $\sigma = 1,731$
$\#rounds(\mathcal{C},v) - \#consent(\mathcal{C},v)$ $\hookrightarrow \ell^{\mathcal{C}}_{total}$ $\hookrightarrow \#rounds(\mathcal{C},v)$	14 6 % 7 %	17 7 % 11 %	24 10 % 16 %	23 9 % 16 %	29 12 % 32 %
$\#rounds(\mathcal{C},v)$ $\hookrightarrow \ell^{\mathcal{C}}_{total}$	207 83 %	154 62 %	152 61 %	146 58 %	91 36 %
$\#consent(\mathcal{C},v)$ $\hookrightarrow \#rounds(\mathcal{C},v)$	191 92 %	135 88 %	130 86 %	123 84 %	63 69 %
normalised dissent	$\mu_{250} = 0,131$ $\sigma = 0,254$	$\mu_{250} = 0,201$ $\sigma = 0,295$	$\mu_{250} = 0,202$ $\sigma = 0,290$	$\mu_{250} = 0,218$ $\sigma = 0,283$	$\mu_{250} = 0,278$ $\sigma = 0,263$
end-to-end response time	$\mu_{207} = 9,325$ $\sigma = 2,591$	$\mu_{154} = 9,726$ $\sigma = 2,819$	$\mu_{152} = 9,515$ $\sigma = 2,719$	$\mu_{146} = 9,396$ $\sigma = 2,415$	$\mu_{91} = 9,107$ $\sigma = 2,312$
TTF $\hookrightarrow$ inv	21 0,048	7 0,143	3 0,333	0 n/a	3 0,333
XTTF-2 $\hookrightarrow$ inv	33 0,030	25 0,040	4 0,250	28 0,036	5 0,200
MTBF $\hookrightarrow$ inv	16,231 0,062	13,800 0,072	9,000 0,111	10,273 0,097	7,679 0,130
MTTR $\hookrightarrow$ inv	3,786 0,264	6,571 0,152	6,368 0,157	6,833 0,146	9,105 0,110

(d) scenario B: resource allocation & under-/overshooting

Table 7.4: Experiment 7.5: redundancy configuration/allocation and failure statistics.

Experiment 7.5

In Fig. 7.5, we consider an environment in which five functionally-equivalent versions operate. The environment is modelled such that participation of a version v_i in the context of a voting round $(\mathbb{C}, \ell)$ is likely to be affected by a disturbance for $(i+1)\cdot 5\%$. For version v_4, this probability was set to 30%. A disturbance will materialise as RVF or EVF failures with 75, resp. 25%. The response values to be used as ballots for the voting procedure are sampled from a uniform distribution. t_{max} was set to 25 discrete time units.
For this particular experiment, we have configured the algorithm to apply a fixed redundancy level of $n^{(\mathbb{C},\ell)} = 3$: replicas will be dynamically selected, but the degree of redundancy will not vary, even though all contextual knowledge is collected to be able to do so. In scenario A, we have chosen $k_1 = 0.25$, $k_2 = 0.05$ and $k_{max} = 0.95$. In scenario B, we changed k_1 to 0.75. Based on Eq. 4.2 and 4.7 on p. 74, resp. 77, we would expect that:

– For replicas that are idle — that is: not actively participating in the redundancy configuration for a specific voting round — normalised dissent measurements will decrease at a higher rate when the difference between k_{max} and k_1 is larger. This difference would be 0.75 in scenario A *vs* 0.20 in scenario B.
  Maximising the difference between these two values (and thus selecting a lower value for k_1) will result in a **more rapid re-integration** of previously penalised — thus faulty — versions in the redundancy configuration. This would translate in a more aggressive resource allocation policy, which is more eager on including versions that may have been suspected or have been found to fail before (increased risk appetite).
– Conversely, when the designer would prefer to act more cautiously in selecting versions, (s)he could decide to lower the value assigned to k_2, and to broaden the distance between k_1 and k_2. In doing so, more trust is placed in versions that continued to support the scheme's functioning and that contributed to the ability to adjudicate an outcome. Faulty versions will consequentially remain **excluded for relatively longer intervals**.

Accordingly, lower re-integration latencies are recorded in scenario A — *cf.* Tables 7.5c and 7.5d. Comparing Fig. 7.5e and 7.5f reveals how normalised dissent measurement show a sharper decline. Because the configuration in scenario A undoes any previously inflicted penalties more rapidly than in scenario B, the side effect is that the version performing worst — v_4, that is — is now being used more intensively throughout the redundancy scheme's operational life: 47% as opposed to 36%, resulting in a slightly worse availability of the scheme itself (5 instead of 4 failures — compare Tables 7.5a and 7.5b).
It is important to set a suitable value for t_{max}, since LRF failures may result in measurements that do not correlate to disturbances originating from design faults.

The other part of the reliability approximation model — the penalisation mechanism — cannot be configured; it processes contextual information that is deduced by the generic voting component at the end of every voting round — *cf.* Sect. 4.2. It therefore depends on the ability to find consensus among the ballots acquired for each version in the redundancy configuration, which in itself is influenced by failures in the content and/or timing domain — *v.* Fig. 2.3. Compared to other participating versions that appear to function better, versions that suffer from disturbances during

a relatively higher share of their involvement in the redundancy scheme's operational life may be taken out of service faster — resulting in lower exclusion latencies.
To conclude this section, a few notes though:

– If, during the operational life of the redundancy scheme, a version is struck by a crash failure, that version might still be re-introduced in the replica selection later on. This can clearly be observed from Fig. 7.2f (and 7.2e), and is the result of the algorithm not being capable of *directly* sensing the nature of disturbances. It does not have perceptional abilities, though it is purposefully aware: the occurrence of failures is detected through observation of the evolution of normalised dissent measurements, which in themselves reflect the degree of consensus found throughout the voting procedure. However, one might consider extending the algorithm with a **probing functionality**, in which it could retry the request several times (in the background), and where it would permanently exclude the faulty version when it was suspected to have crashed — *e.g.* after a predefined number of successive omission failures.

– When applying a dynamic redundancy configuration with a varying redundancy level, the act of redundancy downscaling/upscaling is very likely to result in additional exclusion, respectively re-integration measurements (latencies) to be recorded, although one cannot generally state that this would result in better/worse latencies.

– Due to the self-optimising nature of the A-NVP algorithm, the exclusion and/or re-integration of specific versions depends on how well individual versions are believed to behave compared to the other (participating) versions. The algorithm will be less successfull in dynamically adjusting the replica selection as replicas are being used less frequently, or in case of incomplete contextual knowledge (a situation referred to as corrupted window of contextual information in Sect. 6.2.1, p. 101). We will be zooming in on this matter in greater detail in Sect. 7.4.

7.3 Policies for Parsimonious Resource Allocation

The proposed A-NVP **redundancy dimensioning model** aims to autonomously tune the employed degree of redundancy in view of encountered disturbances, and is fitted with accompanying policies intent upon increasing the scheme's cost effectiveness without breaching its dependability objective — *cf.* Sect. 5.2. The model can be configured in line with the designer's preferences, thereby allowing him/her to define policies that will be used to allocate system resources to a greater or lesser degree of parsimony. In this section, we will analyse if and how the proposed model can be used to **effectively and safely reduce the allocated redundancy**. In this context, safety actually means the sustained availability of the scheme, with minimal service interruption — preferably none at all, *i.e.* reliability[2].

7.3.1 Lowering the Cumulative Amount of Allocated Redundancy

In this section, we will zoom in into how such policies may perform under various conditions, when and to what extent they improve the overall performance of the

[2] Hence we use the intuitive notion of safety here, as opposed to the formal definition stated on p. 9.

system, primarily in terms of dependability and resource consumption. We will define a number of promising resource allocation policies, and subject them to experimentation so as to assess their impact on the overall performance of the proposed dependability strategy. This analysis will be achieved by means of the discrete event simulation framework that is founded upon the discrete event and failure manifestation models defined in Chapt. 2, and that includes a reference implementation of the A-NVP/MV dependability strategy proposed in Chapt. 5.

7.3.1.1 Challenges to Overcome in Applying Dynamic Redundancy Configurations

The use of autonomous redundancy schemata that apply dynamic redundancy configurations does introduce a few challenges that may affect the scheme's dependability. The designer should be well aware of these, and should apply proper configuration to mitigate these risks.

– It is not wise to apply too much frugality in allocating resources, especially at the early stages of the scheme's operational life. The choice of the initial degree of redundancy is likely to determine the initial time to failure. This can be seen when comparing Fig. 7.4c with Fig. 7.4d (p. 134): when subject to similar environmental conditions, the initial level of redundancy in scenario B shows to be insuffient to avoid failure during voting round $(\mathcal{C},6)$. Of course, all depends on the actual fault model of the environment, and how and when it will cause disturbances to materialise. Ergo, it is better to initialise the redundancy configuration with a moderate (to high) degree of redundancy to be used, rather than to start from the bare minimum (TMR). See also: experiment 7.6.

Experiment 7.6

In Fig. 7.6, a simplistic A-NVP strategy was applied that maintains an odd level of redundancy — that is, $f_u^{(\mathcal{C},\ell)} = f_d^{(\mathcal{C},\ell)} = 2$. In doing so, we have essentially applied Strategy A, Variant 2 as defined in Sect. 7.3.1.2. Versions are selected at runtime based on the most recent normalised dissent measurements (only). These measurements are computed with parameters set as follows: $k_1 = 0.45$, $k_2 = 0.05$ and $k_{max} = 0.85$. In this particular experiment, only RVF failures are considered, hence t_{max} is left undefined. The response values for invocations that are affected by RVF failures — to be used as ballots for the voting procedure — are sampled from a normal distribution with a standard deviation $\sigma = 0.5$. The redundancy dimensioning model was configured as follows: $r_d = 15$, $r_u = r_f = 1$ — no safety margin c_{sm} has been imposed.

At the left side (scenario A), this redundancy strategy was configured to economise on resource expenditure, by setting a lower initial degree of redundancy to be used ($n_{init} = 3$). At the right (scenario B), that value was set to 5. Apart from this, both scenarios apply $n_{min} = 3$ and $n_{max} = 5$.

Although both scenarios are characterised by the same number of failures, the recorded TTF measurements are: 10 *vs* 16. Despite this experiment's simplicity, the key message to convey here is that it is — generally speaking — not wise to set the initial degree of redundancy too low (unless one would have a very accurate and precise view on the actual fault model). Apart from realising better TTF, temporarily starting with a slightly higher redundancy level will also ensure a more accurate view is obtained on the constituent versions underpinning the redundancy scheme.

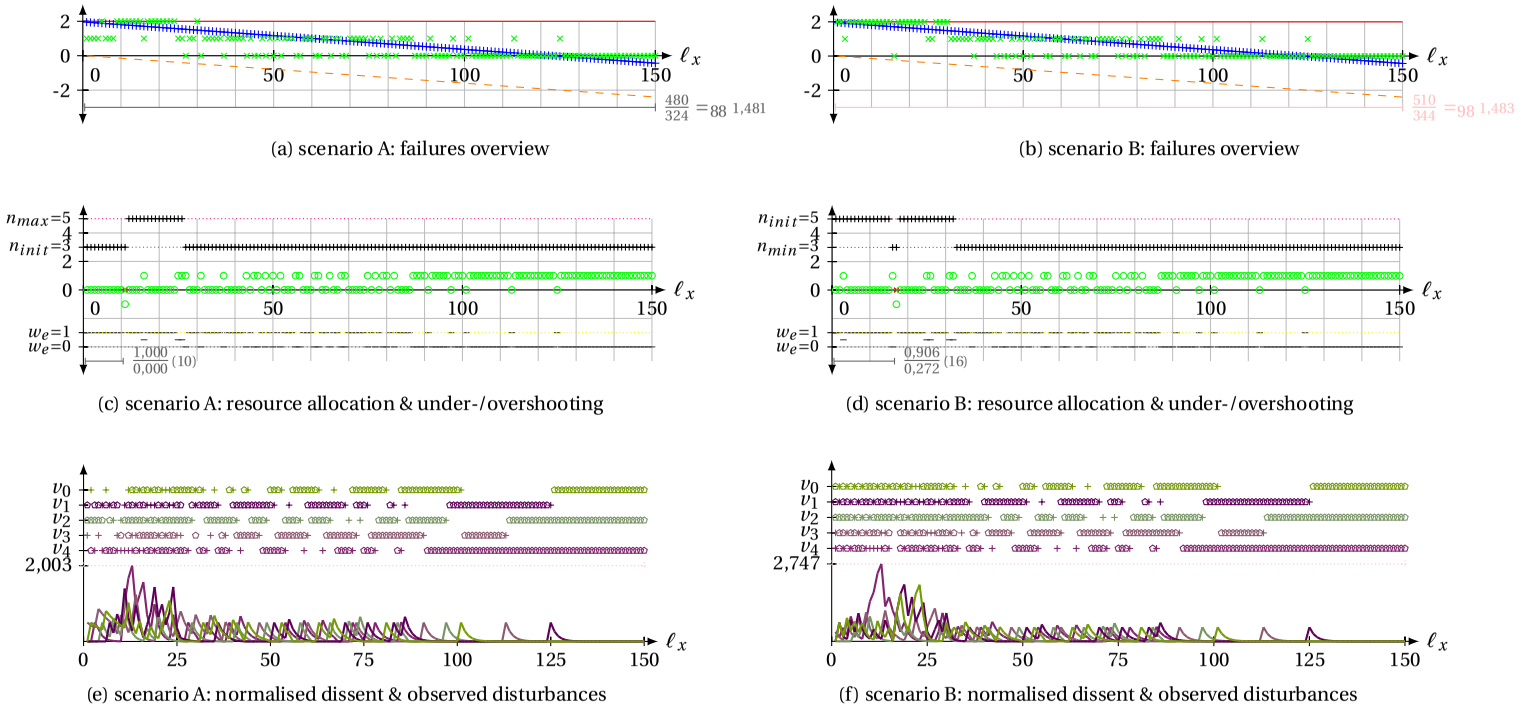
Usually, only few and relatively inaccurate contextual information is available at the start of the scheme's operational life, and active use of more system resources will generate more relevant contextual information that can be effectively used to steer the redundancy configuration.

– Probably the biggest risk would be found in the act of changing the degree of redundancy to be used. Both upscaling as well as downscaling introduce risk:

  + Upscaling to a moderate level of redundancy usually makes sense, but as soon as one includes too many versions, there may be so many disturbances $e^{(\mathcal{C},\ell)}$ that no qualified majority can be found. Indeed, some of the standby versions that will be included might have been previously excluded from the replica selection because they were found to underperform or not to contribute to the scheme's dependability at that time. Refer to experiment 7.10 for more information.
    A few useful metrics are the total amount of events at which the redundancy level to be used in the redundancy configuration was increased ($\triangle_{total}$), and how many of these events resulted from redundancy undershooting and lead to the immediate recovery of service availability ($\triangle_{success}$). Another one is the number of attempts to reach out to additional redundancy, with no remaining spare capacity ($\triangle_{bump}$) — a situation that would arise when $n^{(\mathcal{C},\ell)} = |V^{(\mathcal{C},\ell)}| = |V|$. The reactive nature of the applied dependability strategy can be observed from Fig. 7.4d on p. 134: the algorithm will attempt to increase the level of redundancy upon failure (at the start of the first voting round following round $(\mathcal{C},\ell)$, $\forall \ell \in \{6,17,21,94,97\}$). Mostly, it is successful in realising the immediate recovery of the scheme's availability, with the exception of $\ell = 98$, in which all available redundancy is allocated, and no spare capacity remains.
  + Similarly, downscaling in itself obviously introduces risk, as the redundancy configuration will rely on less versions to mask disturbances that may affect them. Unfortunately, even though a strategy may be capable of scaling down the utilisation of system resources, doing so might occasionally result in redundancy undershooting, even for relatively large values of r_d. A few useful metrics are the total number of events during which the redundancy was brought/scaled down (denoted by $\triangledown_{total}$), and how many of these events resulted in a redundancy configuration that proved to be inadequate to sustain the scheme's availability and thus resulted in one or more failures (denoted by $\triangledown_{failure}$).

    Experiment 7.7

    In this experiment, we analyse the behaviour of an A-NVP redundancy scheme, when it is configured using different instantiations of Strategy B, Variant 1 — *v.* Sect. 7.3.1.2. In the left half of Fig. 7.7 (scenario A), resources are allocated so as to attempt to mask and overcome any disturbance that may affect the scheme's availability. In the right half (scenario B), resources are relinquished more rapidly, thereby seeking to further reduce



(a) scenario A: failures overview

(b) scenario B: failures overview

(c) scenario A: resource allocation & under-/overshooting

(d) scenario B: resource allocation & under-/overshooting

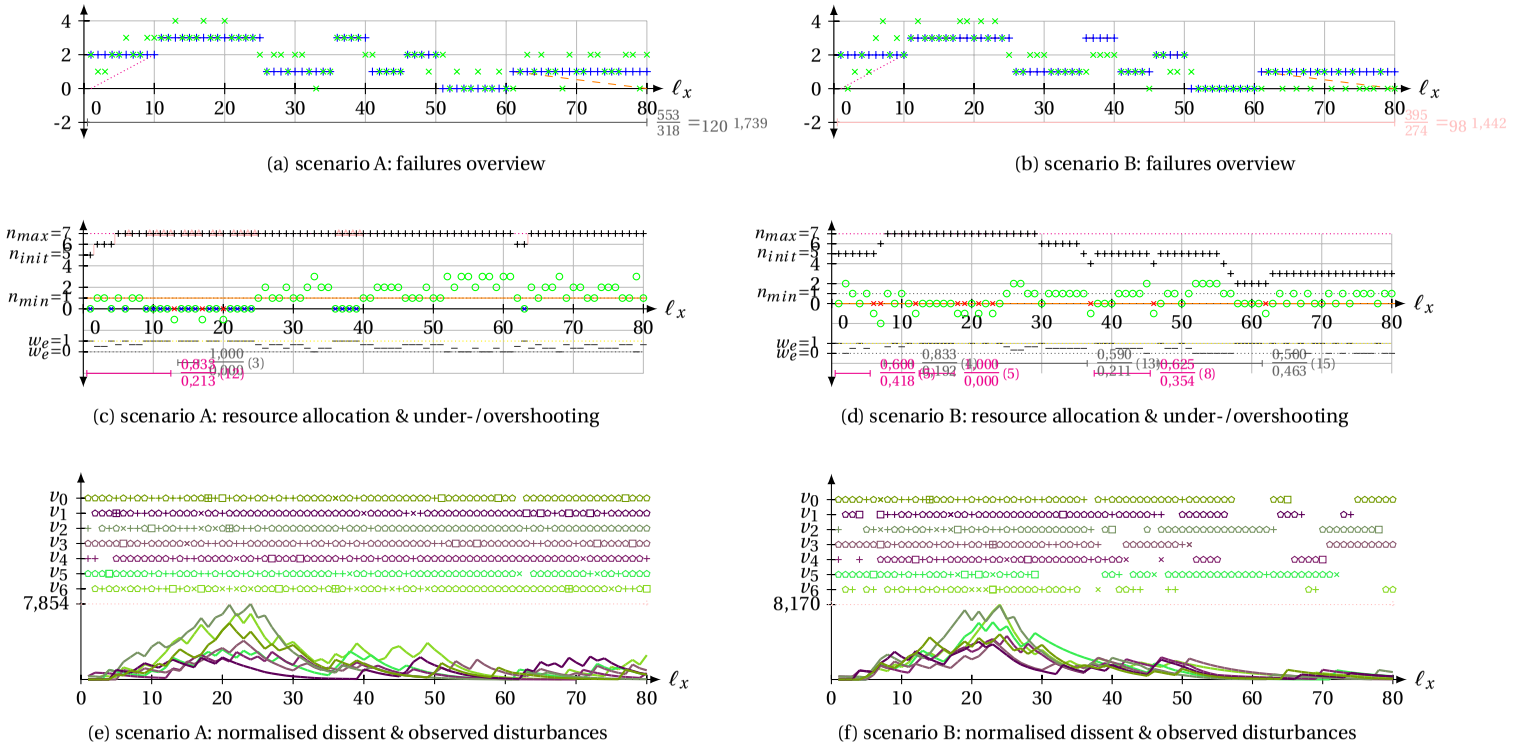
(e) scenario A: normalised dissent & observed disturbances

(f) scenario B: normalised dissent & observed disturbances

Figure 7.6: Experiment 7.6: the initial level of redundancy might determine the TTF. In scenario A, we start by using a low redundancy level; in scenario B, a higher level is used. It is apparent from the charts here above that the TTF will be worse for A.

resource expenditure. Versions are selected based on normalised dissent measurements. These measurements are cached at the end of each successive completed voting round, and are obtained with parameters set as follows: $k_1 = 0.85$, $k_2 = 0.75$ and $k_{max} = 0.95$. Performance failures are detected after a t_{max} timeout equal to 15 discrete time units have lapsed. Other types of disturbances will materialise with an 80 and 20% probability for RVF, resp. EVF failures (with response values for the former type of disturbance sampled from a uniform distribution). The redundancy dimensioning algorithm is configured with $r_u = r_f = 1$, $n_{init} = 5$, $n_{max} = 7$ and $n_{min} = 3$. The difference between scenarios A and B is to be found in that r_d is set to 10, resp. 5 voting rounds, and that a safety margin $c_{sm} = 1$ is applied in the former, whereas none is applied in the latter.
The more sparingly the available redundancy is allocated, the higher the risk that the redundancy scheme will be struck by failures. The act of downscaling can in itself introduce risk: as one can observe from Fig. 7.7d, there are $\bigtriangledown_{failure} = 2$ unsuccessful attempts to reduce the degree of redundancy, *i.c.* $\ell \in \{37, 46\}$ — *v.* Table 7.5. Furthermore, even though scenario B was able to reduce resource expenditure with 28.6% (compared to scenario B), this does come at the expense of a far inferior reliability, where 13% of downtime is recorded, in contrast to a mere 4% (expressed in view of the total number of voting round failures). One can also observe that there might be an impact on the TTF.
Shorter window lengths may result in a (slightly) more incautious downscaling of the redundancy, which in itself might lead to failure of the redundancy scheme in subsequent voting rounds. The general trend shows that the redundancy scheme is less likely to fail due to the downscaling of the employed degree of redundancy for larger values of r_d, at the expense of postponing the relinquishment of excess redundancy — a statement confirmed in [83, Sect. 4.2] and [78].

– When trying to economise on resource expenditure, every call (request) counts. To mask $e^{(\mathcal{C},\ell)}$ of disturbances, given a majority voting adjudication algorithm, at least $(\mathcal{C}, \ell) = cr(e^{(\mathcal{C},\ell)}) = 2 \cdot e^{(\mathcal{C},\ell)} + 1$ versions are required. This corresponds to an odd degree of redundancy, as it is commonly found in traditional NVP.
Odd levels of redundancy have long been used to avoid undetermined results at the end of the voting procedure, where one might observe only two distinct equivalence classes of the same cardinality — *cf.* Sect. 2.1. However, such reasoning is mainly applicable to plurality voting (PV) — *cf.* App. B. It is the author's belief that no such constraint should be adhered to when using $n^{(\mathcal{C},\ell)} \geq 3$. After all, when using majority voting, the qualified majority will decide, and there can only be one — *cf.* Eq. (3.1), p. 67. Furthermore, the ballots acquired from individual versions are usually validated and/or normalised before returning them to the voting mechanism. A common example would be to round numeric values to a specific number of significant decimal digits. Furthermore, the distance function $d(x, y)$ used within the voting mechanism will seek to find additional equivalence.
In the context of dynamic redundancy configurations, temporarily maintaining an even degree of redundancy — usually after downscaling the redundancy — may help to avoid redundancy undershooting, especially if the replica selection



(a) scenario A: failures overview

(b) scenario B: failures overview

(c) scenario A: resource allocation & under-/overshooting

(d) scenario B: resource allocation & under-/overshooting

(e) scenario A: normalised dissent & observed disturbances

(f) scenario B: normalised dissent & observed disturbances

Figure 7.7: Experiment 7.7: downscaling the degree of redundancy may result in redundancy undershooting.

n_{min}	1	$\ell^{\mathcal{C}}_{failure}\ (c^{(\mathcal{C},\ell)}_{max} - m^{(\mathcal{C},\ell)} < 0)$			4% (3 of 80)
n_{max}	7	$\sum n^{(\mathcal{C},\ell)} = 553$	$\sum cr(e^{(\mathcal{C},\ell)}) = 318(360)$	$\triangle_{success}$	100% (3 of 3)
n_{init}	5	$\sum(n^{(\mathcal{C},\ell)} - cr(e^{(\mathcal{C},\ell)})) = 235(193)$		$\nabla_{failure}$	0% (0 of 1)
t_{max}	15,000	$w_e^{(\mathcal{C},\ell)} \in_{77} [0{,}000, 1{,}000]$	$\mu = 0{,}574$	$\sigma = 0{,}335$	$ci_{90\%}$: (5.11E-1, 6.36E-1)
$\ell^{\mathcal{C}}_{total}$	80	$n^{(\mathcal{C},\ell)} \in [5,7]$	$\mu = 6{,}913$	$\sigma = 0{,}326$	$ci_{90\%}$: (6.85, 6.97)
r_d	10	$\square_{fill}{}^{d} \in_{80} [0,10]$	$\mu = 4{,}775$	$\sigma = 4{,}272$	$ci_{90\%}$: (3.99, 5.56)
		$\square_{reset}{}^{d} \in_{5} [1,10]$	$\mu = 5{,}000$	$\sigma = 4{,}583$	$ci_{90\%}$: (1.63, 8.37)
r_u	1	$\square_{fill}{}^{u} \in_{80} [0,1]$	$\mu = 0{,}300$	$\sigma = 0{,}461$	$ci_{90\%}$: (2.15E-1, 3.85E-1)
		$\square_{reset}{}^{u} \in_{6} [1,1]$	$\mu = 1{,}000$	$\sigma = 0{,}000$	$ci_{90\%}$: (1.00, 1.00)
r_f	1	$\square_{corrupt}$	0% (0 of 80)		
TTF	12	$XTTF - 2 = 16$	$MTBF =_2 2{,}500$	$MTTR =_3 1{,}000$	
c_{sm}	f	$c_{sm} = 1$	n/a	n/a	$ci_{90\%}$: (1.00, 1.00)
		$\ell_{suspect}\ (0 \le c^{(\mathcal{C},\ell)}_{max} - m^{(\mathcal{C},\ell)} < c_{sm})$			26% (21 of 80)

(a) scenario A: failures overview

n_{min}	1	$\ell^{\mathcal{C}}_{failure}\ (c^{(\mathcal{C},\ell)}_{max} - m^{(\mathcal{C},\ell)} < 0)$			13% (10 of 80)
n_{max}	7	$\sum n^{(\mathcal{C},\ell)} = 395$	$\sum cr(e^{(\mathcal{C},\ell)}) = 274(306)$	$\triangle_{success}$	80% (4 of 5)
n_{init}	5	$\sum(n^{(\mathcal{C},\ell)} - cr(e^{(\mathcal{C},\ell)})) = 121(89)$		$\nabla_{failure}$	29% (2 of 7)
t_{max}	15,000	$w_e^{(\mathcal{C},\ell)} \in_{70} [0{,}000, 1{,}000]$	$\mu = 0{,}593$	$\sigma = 0{,}410$	$ci_{90\%}$: (5.12E-1, 6.73E-1)
$\ell^{\mathcal{C}}_{total}$	80	$n^{(\mathcal{C},\ell)} \in [2,7]$	$\mu = 4{,}937$	$\sigma = 1{,}664$	$ci_{90\%}$: (4.63, 5.24)
r_d	8	$\square_{fill}{}^{d} \in_{80} [0,5]$	$\mu = 3{,}400$	$\sigma = 1{,}907$	$ci_{90\%}$: (3.05, 3.75)
		$\square_{reset}{}^{d} \in_{8} [1,5]$	$\mu = 3{,}875$	$\sigma = 1{,}808$	$ci_{90\%}$: (2.82, 4.93)
r_u	1	$\square_{fill}{}^{u} \in_{80} [0,1]$	$\mu = 0{,}125$	$\sigma = 0{,}333$	$ci_{90\%}$: (6.38E-2, 1.86E-1)
		$\square_{reset}{}^{u} \in_{8} [1,1]$	$\mu = 1{,}000$	$\sigma = 0{,}000$	$ci_{90\%}$: (1.00, 1.00)
r_f	1	$\square_{corrupt}$	0% (0 of 80)		
TTF	5	$XTTF - 2 = 6$	$MTBF =_9 5{,}222$	$MTTR =_8 1{,}250$	
c_{sm}	f	$c_{sm} = 0$	n/a	n/a	$ci_{90\%}$: (0.00, 0.00)
		$\ell_{suspect}\ (0 \le c^{(\mathcal{C},\ell)}_{max} - m^{(\mathcal{C},\ell)} < c_{sm})$			0% (0 of 80)

(b) scenario B: failures overview

version	v_0	v_1	v_2	v_3	v_4	v_5	v_6
re-integration latency	$\mu_1 = 1{,}000$	n/a	$\mu_1 = 1{,}000$	n/a	$\mu_1 = 2{,}000$	$\mu_1 = 1{,}000$	n/a
	-	n/a	-	n/a	-	-	n/a
exclusion latency	$\mu_1 = 57{,}000$	n/a	$\mu_1 = 1{,}000$	n/a	$\mu_1 = 2{,}000$	$\mu_1 = 53{,}000$	n/a
	-	n/a	-	n/a	-	-	n/a
exclusion failures	$\mu_1 = 16{,}000$	n/a	$\mu_1 = 1{,}000$	n/a	$\mu_1 = 2{,}000$	$\mu_1 = 15{,}000$	n/a
	-	n/a	-	n/a	-	-	n/a
successive usage period	$\mu_1 = 61{,}000$		$\mu_1 = 1{,}000$		$\mu_2 = 1{,}000$	$\mu_1 = 62{,}000$	
	-		-		$\sigma = 1{,}414$	-	
$\#rounds(\mathcal{C},v) - \#consent(\mathcal{C},v)$	20	16	20	21	15	19	29
$\hookrightarrow \ell^{\mathcal{C}}_{total}$	25 %	20 %	25 %	26 %	19 %	24 %	36 %
$\hookrightarrow \#rounds(\mathcal{C},v)$	25 %	20 %	25 %	26 %	19 %	24 %	37 %
$\#rounds(\mathcal{C},v)$	79	79	79	80	78	79	79
$\hookrightarrow \ell^{\mathcal{C}}_{total}$	99 %	99 %	99 %	100 %	98 %	99 %	99 %
$\#consent(\mathcal{C},v)$	58	62	57	56	61	60	50
$\hookrightarrow \#rounds(\mathcal{C},v)$	73 %	78 %	72 %	70 %	78 %	76 %	63 %
normalised dissent	$\mu_{80} = 1{,}425$	$\mu_{80} = 0{,}986$	$\mu_{80} = 1{,}860$	$\mu_{80} = 1{,}175$	$\mu_{80} = 0{,}907$	$\mu_{80} = 1{,}144$	$\mu_{80} = 2{,}225$
	$\sigma = 1{,}447$	$\sigma = 0{,}720$	$\sigma = 2{,}316$	$\sigma = 0{,}843$	$\sigma = 0{,}826$	$\sigma = 0{,}886$	$\sigma = 1{,}807$
end-to-end response time	$\mu_{79} = 9{,}548$	$\mu_{79} = 9{,}721$	$\mu_{79} = 9{,}482$	$\mu_{80} = 9{,}177$	$\mu_{78} = 8{,}958$	$\mu_{79} = 9{,}251$	$\mu_{79} = 9{,}122$
	$\sigma = 2{,}345$	$\sigma = 2{,}364$	$\sigma = 2{,}031$	$\sigma = 2{,}303$	$\sigma = 1{,}990$	$\sigma = 1{,}927$	$\sigma = 2{,}256$
TTF	4	4	0	8	0	9	2
$\hookrightarrow$ inv	0,250	0,250	n/a	0,125	n/a	0,111	0,500
XTTF-2	6	7	3	14	1	11	5
$\hookrightarrow$ inv	0,167	0,143	0,333	0,071	1,000	0,091	0,200
MTBF	2,000	5,273	2,611	3,067	5,583	2,882	2,125
$\hookrightarrow$ inv	0,500	0,190	0,383	0,326	0,179	0,347	0,471
MTTR	1,333	1,333	1,538	1,455	1,400	1,462	1,471
$\hookrightarrow$ inv	0,750	0,750	0,650	0,688	0,714	0,684	0,680

(c) scenario A: resource allocation & under-/overshooting

version	v_0	v_1	v_2	v_3	v_4	v_5	v_6
re-integration latency	$\mu_3 = 5{,}000$	$\mu_4 = 4{,}000$	$\mu_5 = 3{,}000$	$\mu_2 = 11{,}000$	$\mu_5 = 4{,}400$	$\mu_3 = 3{,}667$	$\mu_7 = 5{,}143$
	$\sigma = 4{,}000$	$\sigma = 2{,}449$	$\sigma = 2{,}121$	$\sigma = 11{,}314$	$\sigma = 3{,}286$	$\sigma = 4{,}619$	$\sigma = 6{,}309$
exclusion latency	$\mu_2 = 25{,}500$	$\mu_3 = 14{,}000$	$\mu_3 = 11{,}333$	$\mu_2 = 16{,}000$	$\mu_4 = 8{,}250$	$\mu_4 = 8{,}250$	$\mu_7 = 5{,}000$
	$\sigma = 9{,}192$	$\sigma = 22{,}517$	$\sigma = 17{,}898$	$\sigma = 19{,}799$	$\sigma = 14{,}500$	$\sigma = 9{,}142$	$\sigma = 9{,}713$
exclusion failures	$\mu_2 = 8{,}000$	$\mu_3 = 3{,}333$	$\mu_3 = 5{,}333$	$\mu_2 = 6{,}000$	$\mu_4 = 2{,}750$	$\mu_4 = 3{,}250$	$\mu_7 = 2{,}714$
	$\sigma = 8{,}485$	$\sigma = 4{,}041$	$\sigma = 7{,}506$	$\sigma = 5{,}657$	$\sigma = 3{,}500$	$\sigma = 3{,}862$	$\sigma = 3{,}684$
successive usage period	$\mu_{15} = 3{,}933$	$\mu_{22} = 2{,}591$	$\mu_{17} = 3{,}706$	$\mu_{22} = 2{,}182$	$\mu_{32} = 1{,}500$	$\mu_{19} = 3{,}211$	$\mu_{36} = 1{,}139$
	$\sigma = 10{,}257$	$\sigma = 8{,}760$	$\sigma = 8{,}773$	$\sigma = 8{,}279$	$\sigma = 6{,}396$	$\sigma = 8{,}483$	$\sigma = 4{,}691$
$\#rounds(\mathcal{C},v) - \#consent(\mathcal{C},v)$	17	13	19	13	14	17	20
$\hookrightarrow \ell^{\mathcal{C}}_{total}$	21 %	16 %	24 %	16 %	18 %	21 %	25 %
$\hookrightarrow \#rounds(\mathcal{C},v)$	26 %	23 %	30 %	22 %	29 %	28 %	47 %
$\#rounds(\mathcal{C},v)$	65	57	63	58	48	61	43
$\hookrightarrow \ell^{\mathcal{C}}_{total}$	81 %	71 %	79 %	73 %	60 %	76 %	54 %
$\#consent(\mathcal{C},v)$	44	39	42	40	29	43	21
$\hookrightarrow \#rounds(\mathcal{C},v)$	68 %	68 %	67 %	69 %	60 %	70 %	49 %
normalised dissent	$\mu_{80} = 1{,}454$	$\mu_{80} = 1{,}231$	$\mu_{80} = 1{,}898$	$\mu_{80} = 1{,}182$	$\mu_{80} = 1{,}485$	$\mu_{80} = 1{,}691$	$\mu_{80} = 1{,}805$
	$\sigma = 1{,}428$	$\sigma = 1{,}065$	$\sigma = 2{,}235$	$\sigma = 0{,}910$	$\sigma = 1{,}198$	$\sigma = 1{,}768$	$\sigma = 1{,}710$
end-to-end response time	$\mu_{65} = 9{,}416$	$\mu_{57} = 9{,}567$	$\mu_{63} = 9{,}047$	$\mu_{58} = 9{,}047$	$\mu_{48} = 8{,}958$	$\mu_{61} = 9{,}486$	$\mu_{43} = 9{,}495$
	$\sigma = 2{,}228$	$\sigma = 2{,}281$	$\sigma = 2{,}017$	$\sigma = 1{,}957$	$\sigma = 1{,}978$	$\sigma = 2{,}238$	$\sigma = 2{,}256$
TTF	4	7	0	8	0	9	2
$\hookrightarrow$ inv	0,250	0,143	n/a	0,125	n/a	0,111	0,500
XTTF-2	6	8	5	14	3	11	5
$\hookrightarrow$ inv	0,167	0,125	0,200	0,071	0,333	0,091	0,200
MTBF	2,000	6,333	3,133	2,818	3,600	4,167	2,667
$\hookrightarrow$ inv	0,500	0,158	0,319	0,355	0,278	0,240	0,375
MTTR	1,417	2,286	2,600	4,250	2,875	1,400	4,077
$\hookrightarrow$ inv	0,706	0,438	0,385	0,235	0,348	0,714	0,245

(d) scenario B: resource allocation & under-/overshooting

Table 7.5: Experiment 7.7: redundancy configuration/allocation and failure statistics.

also varies. Although one additional version is used more than what is — strictly speaking — necessary to mask $e^{(\mathcal{C},\ell)}$ of disturbances, keeping this additional resource in use can help to better steer the redundancy configuration towards the optimal replica selection[3].
Although a larger share of the redundancy will be allocated than what is — strictly speaking — required, temporarily maintaining an even degree of redundancy can partially mitigate the risk of downscaling/upscaling. Oftentimes, when a safety margin c_{sm} is defined, it would be economically beneficial to use an even degree of redundancy (otherwise even more resources would be used, since one would upscale to the nearest odd number).

– The system-environment fit may change over time, and various fault models may apply to different stages of the scheme's operational life. Even if the environmental behaviour would have been properly modelled, it would take time for the algorithm to rebalance the redundancy configuration, resulting in a transition period with suboptimal availability of the scheme itself. Combined with the challenges covered in Sect. 7.4, this may lead to situations in which a dynamic redundancy scheme is actually defenseless, and can only try to extract meaningful knowledge from the context information, thereby hoping to regain control and mitigate downtime.

Experiment 7.8

Both scenarios shown in Fig. 7.8 are exposed to similar environmental conditions, except that the fault model in scenario B on the right is more whimsical, in that the failure trends are more challenging to identify throughout the execution of voting rounds $\ell \in [60,180]$. To be more specific, for the interval $[60,105]$, the deviation from the failure trend is sampled from a uniformly distributed random variable with $a = -3$ and $b = 6$. Likewise, for $[105,179]$, $a = -2$ and $b = 2$. As one can observe in Fig. 7.8b, this behaviour clearly obfuscates the actual failure trend. Algorithm parameters set as follows: $k_1 = 0.85$, $k_2 = 0.75$ and $k_{max} = 0.95$. Performance failures are detected after a t_{max} timeout equal to 15 discrete time units have lapsed. Other types of disturbances will materialise with an 80 and 20% probability for RVF, resp. EVF failures (with response values for the former type of disturbance sampled from a uniform distribution). The redundancy dimensioning algorithm is configured with Strategy B, Variant 1 using the parameters $r_d = 5$, $r_u = 3$ and $r_f = 1$, $n_{init} = 5$, $n_{max} = 13$ and $n_{min} = 3$. A safety margin $c_{sm} = 1$ is applied in both scenarios.
Clearly, in scenario A, the failure trend is captured, albeit after a moderate delay, and the algorithm will adjust the redundancy configuration accordingly. The delay is mainly determined by the factor r_d, and — to a lesser extent — c_{sm} — *v.* Fig. 7.8a and 7.8c.
In scenario B, where the failure trend is blurred as of voting round $\ell \geq 60$, one can observe a markedly higher resource allocation, simply because the

[3]Trustworthy versions will be characterised with lower normalised dissent measurements being recorded. Recall that normalised dissent measurements are interpreted as an approximation of a version's reliability. Even though normalised dissent is updated at the end of each voting round — also for versions that have temporarily been taken out of service — the most accurate results would be available after continuous use. Indeed, that would mean the impact of the version has been actively and constantly monitored, by assessing if the ballots acquired from the version (if any) helped to adjudicate an outcome (majority).

redundancy scheme is sitting quite defenseless, hoping that upscaling the degree of redundancy will suffice to mask the disturbances that emerged and to resume and/or sustain the scheme's availability. Indeed, it is not capable of accurately anticipating the failure trend (model).

In summary: upscaling should be proactive, and should be context-aware to avoid calling underperforming versions. In general, one may expect that a properly chosen value c_{sm} matching the variability of the environment and the ensuing disturbances aids in intercepting the trend and may lead to a reduction of scheme failures due to more efficient proactive upscaling and less aggressive downscaling.

7.3.1.2 Some Exemplary Dependability Strategies

In this section, we will list a few policies that can be used to manage the redundancy configuration as part of an A-NVP redundancy scheme.

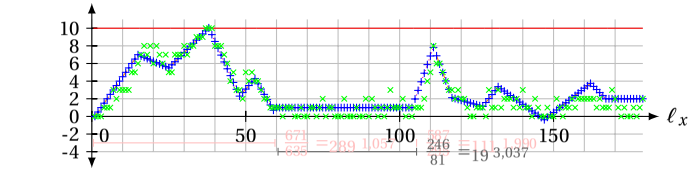
Strategy A The first policy is inspired by the strategy published in [78, Sect. 3.3], and was already used in previous experimentation to showcase the advantages of applying a dynamic — albeit reactive — redundancy scheme. In line with traditional NVP, it will only report an odd degree of redundancy. A simplex system is not allowed, hence $n_{min} \geq 3$. If the voting scheme failed to find consensus amongst a majority of the replicas involved during the last completed voting round, the model will increase the number of versions to be used in the next voting round, to the extent that $f_u^{(\mathbb{C},\ell)} = 2$. Conversely, when the scheme was able to produce an outcome with a given amount of redundancy for a certain amount r_d of consecutive voting round completions, a lower degree of redundancy shall be used for the next voting round, with $f_d^{(\mathbb{C},\ell)} = 2$.
The initial algorithm can be easily modelled as an A-NVP scheme: although initially observing only raw *dtof* measurements, without any correlation or trend analysis, it does fit the model, in which the success criterion could be used to detect success or failure.[4]. The parameters r_u and r_f are obviously set to 1; $n_{max} \geq n_{init} \geq 3$[5].
This strategy can be used in two possible ways:

Variant 1 In its basic form, the strategy selects versions arbitrarily from the available pool of resources. This corresponds with the model defined in [78].

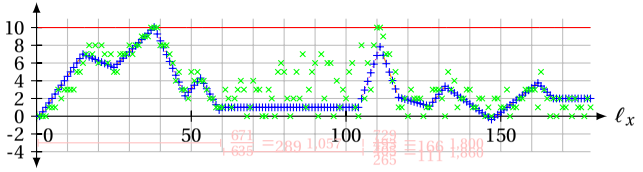
Variant 2 In an extended form, we have combined our dynamic replica selection algorithm with the initial algorithm: the scheme will rank the available versions and dynamically select the optimal

[4] Combination of the success criterion as it is defined on p. 82 with Eq. 3.3, p. 69 shows how a safety margin can be set to define the "critically low *dtof* value" mentioned in [78]. There is no mentioning of any precise value, and we presume *dtof* = 0 was used. The context manager will maintain a cache of recent nett redundancy measurements that can be used to support this model — *v.* Fig. 5.1, p. 82.

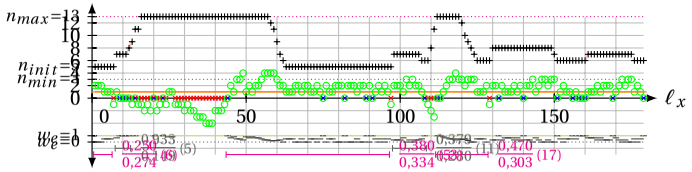
[5] Faithful to the original, the redundancy scheme is initialised such that it is capable of tolerating up to one failure, hence $r_f = 3$.



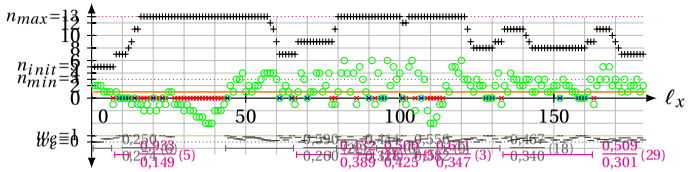
(a) scenario A: failures overview



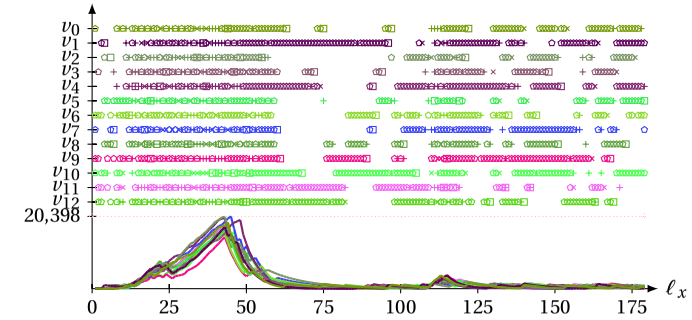
(b) scenario B: failures overview



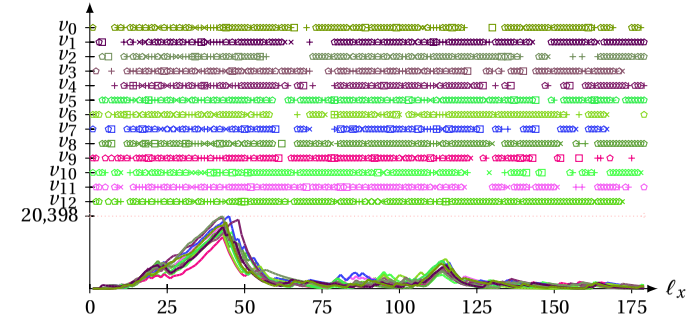
(c) scenario A: resource allocation & under-/overshooting



(d) scenario B: resource allocation & under-/overshooting



(e) scenario A: normalised dissent & observed disturbances



(f) scenario B: normalised dissent & observed disturbances

Figure 7.8: Experiment 7.8: A-NVP performance may temporarily deteriorate when the environmental conditions change. Once the conditions have evolved and stabilised to a new level, the redundancy scheme will eventually adjust the redundancy configuration to match the new situation.

n_{min}	3	$\ell^{\mathcal{C}}_{failure}(c^{(\mathcal{C},\ell)}_{max} - m^{(\mathcal{C},\ell)} < 0)$			18% (32 of 179)
n_{max}	13	$\sum n^{(\mathcal{C},\ell)} = 1504$	$\sum cr(e^{(\mathcal{C},\ell)}) = 1011(1105)$	$\triangle_{success}$	64% (7 of 11)
n_{init}	5	$\sum(n^{(\mathcal{C},\ell)} - cr(e^{(\mathcal{C},\ell)})) = 493(399)$		$\nabla_{failure}$	0% (0 of 16)
t_{max}	15,000	$w^{(\mathcal{C},\ell)}_{t} \in_{147} [0,000, 1,000]$	$\mu = 0,454$	$\sigma = 0,340$	$ci_{90\%} : (4.08E\text{-}1, 5.01E\text{-}1)$
$\ell^{\mathcal{C}}_{total}$	179	$n^{(\mathcal{C},\ell)} \in [5, 13]$	$\mu = 8,402$	$\sigma = 3,177$	$ci_{90\%} : (8.01, 8.79)$
r_d	5	$\square_{fill}{}^{d} \in_{179} [0, 5]$	$\mu = 2,620$	$\sigma = 2,178$	$ci_{90\%} : (2.35, 2.89)$
		$\square_{reset}{}^{d} \in_{17} [1, 5]$	$\mu = 3,882$	$\sigma = 1,691$	$ci_{90\%} : (3.21, 4.56)$
r_u	3	$\square_{fill}{}^{u} \in_{179} [0, 3]$	$\mu = 0,726$	$\sigma = 1,194$	$ci_{90\%} : (5.80E\text{-}1, 8.73E\text{-}1)$
		$\square_{reset}{}^{u} \in_{16} [1, 3]$	$\mu = 1,500$	$\sigma = 0,894$	$ci_{90\%} : (1.13, 1.87)$
r_f	1	$\square_{corrupt}$	0% (0 of 179)		
TTF	6	$XTTF - 2 = 12$	$MTBF =_{31} 2,935$	$MTTR =_{9} 3,556$	
c_{sm}	f	$c_{sm} = 1$	n/a	n/a	$ci_{90\%} : (1.00, 1.00)$
		$\ell_{suspect}\ (0 \leq c^{(\mathcal{C},\ell)}_{max} - m^{(\mathcal{C},\ell)} < c_{sm})$			13% (24 of 179)

(a) scenario A: failures overview

n_{min}	3	$\ell^{\mathcal{C}}_{failure}(c^{(\mathcal{C},\ell)}_{max} - m^{(\mathcal{C},\ell)} < 0)$			23% (42 of 179)
n_{max}	13	$\sum n^{(\mathcal{C},\ell)} = 1893$	$\sum cr(e^{(\mathcal{C},\ell)}) = 1305(1417)$	$\triangle_{success}$	77% (10 of 13)
n_{init}	5	$\sum(n^{(\mathcal{C},\ell)} - cr(e^{(\mathcal{C},\ell)})) = 588(476)$		$\nabla_{failure}$	0% (0 of 15)
t_{max}	15,000	$w^{(\mathcal{C},\ell)}_{e} \in_{137} [0, 000, 1, 000]$	$\mu = 0,506$	$\sigma = 0,330$	$ci_{90\%} : (4.60E\text{-}1, 5.53E\text{-}1)$
$\ell^{\mathcal{C}}_{total}$	179	$n^{(\mathcal{C},\ell)} \in [5, 13]$	$\mu = 10,575$	$\sigma = 2,570$	$ci_{90\%} : (10.26, 10.89)$
r_d	5	$\square_{fill}{}^{d} \in_{179} [0, 5]$	$\mu = 2,263$	$\sigma = 2,147$	$ci_{90\%} : (2.00, 2.53)$
		$\square_{reset}{}^{d} \in_{20} [1, 5]$	$\mu = 3,100$	$\sigma = 1,714$	$ci_{90\%} : (2.47, 3.73)$
r_u	3	$\square_{fill}{}^{u} \in_{179} [0, 3]$	$\mu = 0,816$	$\sigma = 1,211$	$ci_{90\%} : (6.67E\text{-}1, 9.64E\text{-}1)$
		$\square_{reset}{}^{u} \in_{20} [1, 3]$	$\mu = 1,700$	$\sigma = 0,865$	$ci_{90\%} : (1.38, 2.02)$
r_f	1	$\square_{corrupt}$	0% (0 of 179)		
TTF	6	$XTTF - 2 = 12$	$MTBF =_{41} 2,805$	$MTTR =_{15} 2,800$	
c_{sm}	f	$c_{sm} = 1$	n/a	n/a	$ci_{90\%} : (1.00, 1.00)$
		$\ell_{suspect}\ (0 \leq c^{(\mathcal{C},\ell)}_{max} - m^{(\mathcal{C},\ell)} < c_{sm})$			13% (23 of 179)

(b) scenario B: failures overview

version	s_0	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8	s_9	s_{10}	s_{11}	s_{12}
re-integration latency	$\mu_R = 6,000$	$\mu_6 = 4,111$	$\mu_{11} = 7,273$	$\mu_{10} = 7,100$	$\mu_9 = 5,375$	$\mu_{10} = 7,900$	$\mu_{14} = 4,857$	$\mu_R = 6,900$	$\mu_{10} = 6,900$	$\mu_6 = 4,333$	$\mu_{13} = 5,907$	$\mu_{11} = 5,000$	$\mu_9 = 4,917$
	$\sigma = 5,000$	$\sigma = 3,018$	$\sigma = 10,993$	$\sigma = 5,527$	$\sigma = 4,627$	$\sigma = 6,206$	$\sigma = 6,206$	$\sigma = 8,247$	$\sigma = 4,001$	$\sigma = 4,637$	$\sigma = 2,897$	$\sigma = 4,050$	$\sigma = 5,123$
exclusion latency	$\mu_3 = 9,857$	$\mu_3 = 16,143$	$\mu_4 = 6,111$	$\mu_6 = 9,250$	$\mu_6 = 15,500$	$\mu_6 = 7,000$	$\mu_4 = 9,125$	$\mu_3 = 10,571$	$\mu_3 = 9,429$	$\mu_4 = 17,000$	$\mu_3 = 8,273$	$\mu_R = 8,900$	$\mu_6 = 16,000$
	$\sigma = 17,009$	$\sigma = 29,977$	$\sigma = 13,523$	$\sigma = 16,602$	$\sigma = 24,203$	$\sigma = 15,174$	$\sigma = 15,100$	$\sigma = 17,934$	$\sigma = 16,652$	$\sigma = 16,769$	$\sigma = 15,193$	$\sigma = 14,813$	$\sigma = 28,420$
exclusion failures	$\mu_7 = 4,286$	$\mu_7 = 4,714$	$\mu_9 = 3,667$	$\mu_6 = 3,875$	$\mu_6 = 5,833$	$\mu_6 = 3,778$	$\mu_8 = 3,625$	$\mu_7 = 4,429$	$\mu_7 = 4,429$	$\mu_6 = 5,200$	$\mu_{11} = 2,909$	$\mu_R = 3,600$	$\mu_{11} = 6,000$
	$\sigma = 7,107$	$\sigma = 7,319$	$\sigma = 7,265$	$\sigma = 6,664$	$\sigma = 9,218$	$\sigma = 7,596$	$\sigma = 5,902$	$\sigma = 7,290$	$\sigma = 7,629$	$\sigma = 6,058$	$\sigma = 5,375$	$\sigma = 6,535$	$\sigma = 9,694$
successive usage period	$\mu_{42} = 1,672$	$\mu_{57} = 5,514$	$\mu_{31} = 1,120$	$\mu_{65} = 1,225$	$\mu_{43} = 2,707$	$\mu_{75} = 1,127$	$\mu_{63} = 1,508$	$\mu_{37} = 1,417$	$\mu_{65} = 1,553$	$\mu_{28} = 2,580$	$\mu_{44} = 2,041$	$\mu_{31} = 1,825$	$\mu_{52} = 2,423$
	$\sigma = 6,001$	$\sigma = 13,751$	$\sigma = 5,305$	$\sigma = 5,962$	$\sigma = 10,777$	$\sigma = 6,119$	$\sigma = 5,941$	$\sigma = 6,999$	$\sigma = 6,201$	$\sigma = 8,716$	$\sigma = 9,420$	$\sigma = 6,900$	$\sigma = 10,237$
$\#rounds(\mathcal{C},s) - \#evasive(\mathcal{C},s)$	34	35	36	35	39	37	32	36	37	30	36	38	38
$\rightsquigarrow \ell^{\mathcal{C}}_{total}$	19%	20%	20%	20%	22 %	21 %	18 %	20%	21 %	17%	20 %	21 %	21 %
$\rightsquigarrow \#rounds(\mathcal{C},s)$	30%	25%	29%	36%	30 %	38%	29 %	32%	35 %	23%	27 %	33%	30%
$\#rounds(\mathcal{C},s)$	112	140	93	98	129	97	111	112	107	129	135	115	126
$\rightsquigarrow \ell^{\mathcal{C}}_{total}$	63%	78%	52%	55%	72 %	54%	62 %	63%	60 %	72%	75 %	64%	70%
$\#evasive(\mathcal{C},s)$	67	95	47	51	81	50	63	65	60	84	90	67	77
$\rightsquigarrow \#rounds(\mathcal{C},s)$	60 %	68 %	51 %	52%	63 %	52%	57 %	58%	56 %	65%	67 %	58%	61 %
normalised dissent	$\mu_{174} = 2,579$	$\mu_{19} = 2,262$	$\mu_{13} = 3,157$	$\mu_{174} = 2,452$	$\mu_{13} = 5,113$	$\mu_{174} = 2,545$	$\mu_{13} = 2,379$	$\mu_{174} = 2,393$	$\mu_{13} = 2,692$	$\mu_{174} = 1,900$	$\mu_{13} = 2,360$	$\mu_{174} = 2,642$	$\mu_{13} = 5,333$
	$\sigma = 6,196$	$\sigma = 5,869$	$\sigma = 4,779$	$\sigma = 5,669$	$\sigma = 5,241$	$\sigma = 5,702$	$\sigma = 3,390$	$\sigma = 4,801$	$\sigma = 5,977$	$\sigma = 2,847$	$\sigma = 3,074$	$\sigma = 4,101$	$\sigma = 5,964$
end-to-end response time	$\mu_{112} = 9,360$	$\mu_{160} = 9,252$	$\mu_{88} = 9,116$	$\mu_{66} = 9,274$	$\mu_{120} = 9,245$	$\mu_{67} = 9,007$	$\mu_{11} = 9,261$	$\mu_{112} = 9,176$	$\mu_{102} = 9,526$	$\mu_{120} = 9,420$	$\mu_{136} = 9,374$	$\mu_{115} = 9,326$	$\mu_{136} = 9,251$
	$\sigma = 2,198$	$\sigma = 2,100$	$\sigma = 2,169$	$\sigma = 2,326$	$\sigma = 2,175$	$\sigma = 2,162$	$\sigma = 2,167$	$\sigma = 2,213$	$\sigma = 2,363$	$\sigma = 2,117$	$\sigma = 2,271$	$\sigma = 2,215$	$\sigma = 2,166$
TTF	8	10	11	6	7	12	7	12	14	10	4	8	8
$\rightsquigarrow$ inv	0,125	0,100	0,091	0,167	0,143	0,083	0,143	0,083	0,071	0,100	0,250	0,125	0,125
XTTF-2	10	13	15	12	11	13	11	14	15	12	9	9	9
$\rightsquigarrow$ inv	0,100	0,077	0,067	0,083	0,091	0,077	0,091	0,071	0,067	0,083	0,111	0,111	0,111
MTBF	4,020	3,701	4,125	4,433	3,886	3,727	4,079	4,233	3,933	5,040	4,161	3,629	3,467
$\rightsquigarrow$ inv	0,207	0,264	0,242	0,226	0,257	0,268	0,214	0,236	0,261	0,198	0,240	0,276	0,273
MTTR	2,562	2,789	5,750	2,350	4,000	4,190	2,621	2,889	3,053	2,111	3,389	4,000	2,619
$\rightsquigarrow$ inv	0,390	0,358	0,174	0,426	0,250	0,239	0,413	0,346	0,328	0,474	0,295	0,250	0,382

(c) scenario A: resource allocation & under-/overshooting

version	s_0	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8	s_9	s_{10}	s_{11}	s_{12}
re-integration latency	$\mu_6 = 3,625$	$\mu_9 = 3,111$	$\mu_6 = 4,750$	$\mu_9 = 2,889$	$\mu_{13} = 3,000$	$\mu_6 = 2,333$	$\mu_{10} = 3,500$	$\mu_9 = 4,444$	$\mu_6 = 2,750$	$\mu_{11} = 3,167$	$\mu_{11} = 5,091$	$\mu_{11} = 2,667$	$\mu_8 = 1,500$
	$\sigma = 2,504$	$\sigma = 2,571$	$\sigma = 4,268$	$\sigma = 1,691$	$\sigma = 2,582$	$\sigma = 1,366$	$\sigma = 3,308$	$\sigma = 1,944$	$\sigma = 1,282$	$\sigma = 2,082$	$\sigma = 2,456$	$\sigma = 2,229$	$\sigma = 0,577$
exclusion latency	$\mu_7 = 17,143$	$\mu_7 = 13,250$	$\mu_4 = 18,000$	$\mu_6 = 15,125$	$\mu_{11} = 10,545$	$\mu_7 = 26,000$	$\mu_4 = 15,750$	$\mu_7 = 14,857$	$\mu_4 = 20,000$	$\mu_7 = 15,714$	$\mu_6 = 12,889$	$\mu_R = 11,300$	$\mu_3 = 68,338$
	$\sigma = 21,416$	$\sigma = 20,324$	$\sigma = 27,085$	$\sigma = 19,313$	$\sigma = 16,571$	$\sigma = 31,032$	$\sigma = 20,886$	$\sigma = 22,653$	$\sigma = 22,370$	$\sigma = 21,498$	$\sigma = 34,923$	$\sigma = 21,397$	$\sigma = 46,004$
exclusion failures	$\mu_7 = 5,571$	$\mu_7 = 5,375$	$\mu_6 = 7,167$	$\mu_9 = 5,875$	$\mu_{13} = 4,091$	$\mu_9 = 9,000$	$\mu_8 = 5,250$	$\mu_7 = 6,143$	$\mu_6 = 6,500$	$\mu_7 = 5,000$	$\mu_9 = 6,667$	$\mu_{10} = 4,900$	$\mu_3 = 15,000$
	$\sigma = 7,706$	$\sigma = 7,920$	$\sigma = 9,806$	$\sigma = 7,259$	$\sigma = 7,077$	$\sigma = 11,036$	$\sigma = 6,561$	$\sigma = 6,295$	$\sigma = 8,313$	$\sigma = 6,272$	$\sigma = 10,630$	$\sigma = 7,992$	$\sigma = 11,350$
successive usage period	$\mu_{65} = 6,967$	$\mu_{28} = 6,821$	$\mu_{28} = 3,211$	$\mu_{13} = 4,394$	$\mu_{28} = 3,300$	$\mu_{14} = 11,071$	$\mu_{13} = 4,000$	$\mu_{22} = 2,442$	$\mu_{28} = 6,273$	$\mu_{42} = 3,262$	$\mu_{28} = 4,114$	$\mu_{22} = 4,531$	$\mu_{13} = 12,692$
	$\sigma = 13,808$	$\sigma = 13,100$	$\sigma = 12,937$	$\sigma = 11,907$	$\sigma = 10,709$	$\sigma = 23,799$	$\sigma = 14,821$	$\sigma = 9,440$	$\sigma = 14,748$	$\sigma = 11,530$	$\sigma = 16,217$	$\sigma = 15,000$	$\sigma = 31,719$
$\#rounds(\mathcal{C},s) - \#evasive(\mathcal{C},s)$	45	44	49	50	52	51	45	53	45	43	47	53	49
$\rightsquigarrow \ell^{\mathcal{C}}_{total}$	25 %	25 %	27 %	28 %	29%	28 %	24 %	30 %	24 %	24 %	26 %	30 %	26 %
$\rightsquigarrow \#rounds(\mathcal{C},s)$	39 %	39 %	36 %	34 %	39 %	31 %	30 %	42 %	28 %	31 %	33 %	36 %	28 %
$\#rounds(\mathcal{C},s)$	149	149	138	145	133	162	144	127	156	137	144	146	165
$\rightsquigarrow \ell^{\mathcal{C}}_{total}$	83 %	83 %	77 %	81 %	74 %	91 %	80 %	71 %	86 %	77 %	80 %	82 %	92 %
$\#evasive(\mathcal{C},s)$	85	91	75	78	70	97	81	60	97	76	81	78	104
$\rightsquigarrow \#rounds(\mathcal{C},s)$	57 %	61 %	54 %	54 %	53 %	60 %	56 %	47 %	63 %	55 %	56 %	53 %	63 %
normalised dissent	$\mu_{179} = 3,054$	$\mu_{179} = 2,950$	$\mu_{179} = 3,561$	$\mu_{179} = 2,960$	$\mu_{179} = 3,051$	$\mu_{179} = 2,980$	$\mu_{179} = 2,946$	$\mu_{179} = 3,067$	$\mu_{179} = 2,974$	$\mu_{179} = 2,407$	$\mu_{179} = 2,970$	$\mu_{179} = 3,159$	$\mu_{179} = 2,091$
	$\sigma = 4,122$	$\sigma = 3,737$	$\sigma = 4,633$	$\sigma = 3,512$	$\sigma = 5,035$	$\sigma = 3,669$	$\sigma = 4,548$	$\sigma = 3,907$	$\sigma = 2,715$	$\sigma = 3,948$	$\sigma = 3,751$	$\sigma = 3,948$	$\sigma = 3,915$
end-to-end response time	$\mu_{163} = 9,300$	$\mu_{48} = 9,210$	$\mu_{158} = 9,104$	$\mu_{163} = 9,199$	$\mu_{153} = 9,238$	$\mu_{102} = 9,222$	$\mu_{144} = 9,092$	$\mu_{137} = 9,393$	$\mu_{154} = 9,346$	$\mu_{137} = 9,467$	$\mu_{14} = 9,323$	$\mu_{146} = 9,308$	$\mu_{163} = 9,134$
	$\sigma = 2,291$	$\sigma = 2,273$	$\sigma = 2,054$	$\sigma = 2,140$	$\sigma = 2,218$	$\sigma = 2,112$	$\sigma = 1,934$	$\sigma = 2,452$	$\sigma = 2,203$	$\sigma = 2,320$	$\sigma = 2,251$	$\sigma = 2,148$	$\sigma = 2,013$
TTF	8	10	11	6	7	12	7	12	14	10	4	8	8
$\rightsquigarrow$ inv	0,125	0,100	0,091	0,167	0,143	0,083	0,143	0,083	0,071	0,100	0,250	0,125	0,125
XTTF-2	10	13	15	12	11	13	11	14	15	12	9	9	9
$\rightsquigarrow$ inv	0,100	0,077	0,067	0,083	0,091	0,077	0,091	0,071	0,067	0,083	0,111	0,111	0,111
MTBF	3,447	2,643	2,727	2,567	2,000	2,319	2,870	2,667	2,010	3,824	3,220	2,430	2,735
$\rightsquigarrow$ inv	0,290	0,370	0,367	0,397	0,357	0,431	0,347	0,375	0,355	0,262	0,311	0,410	0,370
MTTR	2,282	2,407	2,889	1,967	2,700	1,700	2,750	2,591	2,429	1,680	2,739	2,406	1,808
$\rightsquigarrow$ inv	0,438	0,415	0,346	0,508	0,369	0,588	0,364	0,386	0,412	0,595	0,365	0,416	0,553

(d) scenario B: resource allocation & under-/overshooting

Table 7.6: Experiment 7.8: redundancy configuration/allocation and failure statistics.

n replicas, based on the available contextual information. This variation was used in experiment 7.4, p. 133.

Strategy B A second strategy — that originally appeared in [84] — is a plain implementation of the mechanism defined in Sect. 5. When the degree of redundancy $n^{(\mathcal{C},\ell-1)}$ is found to be overabundant, the downscaling function will try to adjust downwards by an amount $f_d^{(\mathcal{C},\ell)} = (c_{max}^{(\mathcal{C},\ell_x)} - m^{(\mathcal{C},\ell_x)}) - c_{sm}$, with ℓ_x denoting the voting round in the observation window for which the smallest degree of consent was found. Undershooting the safety margin will cause $f_u^{(\mathcal{C},\ell)}$ to return $c_{sm} - |c_{max}^{(\mathcal{C},\ell_x)} - m^{(\mathcal{C},\ell_x)}|$, with ℓ_x the round in the observation window delimited by r_u exhibiting the largest deviation from the imposed security margin. The upscaling function $f_u^{(\mathcal{C},\ell)}$ will try to inflate the redundancy level by $|c_{max}^{(\mathcal{C},\ell_x)} - m^{(\mathcal{C},\ell_x)}| + c_{sm}$ in case of redundancy undershooting, with ℓ_x the eligible round with the smallest degree of consent.

Variant 1 In its basic form, the strategy will attempt to reach out to additional redundant capacity in case the scheme is found to underperform (or fail). The only limitation here is the total amount of redundant resources $|V|$ present in the system — *cf.* Sect.5.2.

Variant 2 Adding redundancy does not always translate in an improvement of dependability. To avoid aggressive upscaling, which in turn may trigger additional disturbances that the scheme's redundancy configuration cannot possibly overcome, this variation will maintain oversight of the normalised dissent measurements for all versions $v \in V$. In order to do so, it will rank all versions based on their latest normalised dissent measurement, and — if needed — override the redundancy dimensioning model's decision whenever the model would instruct to upscale in a situation where the previously unallocated redundancy was found to underperform. Having ranked the versions based on their normalised dissent measurements, those versions whose measurements fall in the upper quartile (Q_3) will be supposed to structurally underperform such that their inclusion in the redundancy configuration cannot be justified[6][7]. This will typically result in a reduced consumption of redundant resources. This variation was used in experiment 7.10.

Additional strategies can be added/defined as desired, with upscaling and downscaling functions usually taking into account cached contextual information. The designer can even decide to cache additional metric measurements, and act on these (possibly combined with the current set of supported metrics).

[6] The discrete event simulation framework can be configured for any specific quantile value. It's use is not limited to this exemplary configuration.

[7] A threshold cannot simply be applied to normalised dissent measurements, as measurements are relative to the current redundancy configuration, and indicate the extent to which specific versions outperform or underperform with respect to others. Considering all versions operate normally, although occasionally affected by a disturbance, with the exception of a specific subset of versions. For this last subset of versions, one will observe a steeper curve in the evolution of normalised dissent measurements being recorded. Unless behaviour is curbed, one will see that these measurements diverge from "normal" measurements that centre around some stable value.

7.3.2 Comparing the Effectiveness of Various Policies

We now compare the effectiveness of five redundancy dimensioning strategies, in addition to a traditional NVP scheme. Each of these strategies is assumed to be deployed within an A-NVP/MV scheme operating in an environment in which the same set V of versions had been deployed and that exhibits identical failure behaviour.

Experiment 7.9

With the exception of the voting rounds in interval [195, 205], no more than four failures are assumed to affect the versions in $V^{(\mathcal{C},\ell)}$. Such scenario could successfully be overcome by a static redundancy configuration with $n = 9^a$. Apart from such classic strategy, we will evaluate several other dependability strategies for parsimonious resource allocation, based on the formalism introduced in Chapt. 5:

- in scenarios B and C, Variant 1, resp. Variant 2 of strategy Strategy A are applied;
- scenarios D and E are examples are examples of Strategy B, Variant 1: in the former no safety margin is applied, whereas it is in the latter;
- scenario F is resemblant of scenario E, but will reach out to additional redundant resources.

A global value $t_{max} = 15$ is applied for all scenarios. $k_1 = 0.85$, $k_2 = 0.75$, and $k_{max} = 0.95$. For scenarios E and F, a discretionary safety margin $c_{sm} = 1$ is used; none is used in the other scenarios. Apart from scenario A, the employed redundancy can vary between $[3,9]$. In scenario F, $n_{max} = 11$. For scenarios B–F, at the expense of a more aggressive allocation strategy, we set $r_u = r_f = 1$, such that the availability of the scheme can swiftly be regained in case of redundancy undershooting. Whereas $r_d = 20$ is used in scenarios B–E , this value is reduced to 10 for scenario F. Injected failures are set to materialise as EVF and RVF content failures with a 20, respectively 80% probability, with response values being sampled from a uniformly distributed random variable — *v.* p. 51.

The following table gives a brief overview of the key findings at the end of simulation when each of these scenarios is subject to the same environment:

scenario	strategy	variant	r_d	c_{sm}	$\ell^{\mathcal{C}}_{failure}$	TTF	MTBF	MTTR	$\sum n^{(\mathcal{C},\ell)}$	saving
A	—	n/a	n/a	n/a	6%	9	11.733	2.667	2250	
B	A	1	20	n/a	8%	13	11.368	2.222	1992	-11.46%
C	A	2	20	n/a	8%	9	11.263	2.000	1984	-11.82%
D	B	1	20	0	12%	9	5.821	2.231	1775	-21.11%
E	B	1	20	1	7%	9	11.941	2.571	2164	-3.82%
F	B	1	10	1	3%	9	25.857	1.143	2193	-3.82%

As we can see from the above overview, the approach applying the greatest degree of parsimony (scenario D) results in a significantly worse overall availability of the redundancy scheme, in spite of an impressive reduction in resource allocation (in terms of versions being invoked). With a slightly worse overall availability, one can observe that more than 10% of version invocations can be avoided in scenarios B and C — that is, compared to the cumulative number of resource allocation that would be applicable for

the static redundancy configuration (scenario A). Even though the resource expenditure recorded is only of moderate size — less than 4% — scenarios E and F show the best results in terms of availability: both result in a an overall availability close to or better than what could be expected from a comparable static redundancy configuration[b]. For scenario E, a retry-based approach where the redundancy scheme would attempt to mask any faulty voting round, could result in even better performance in terms of availability, but with hardly any economisation of resource allocation[c]. Scenario F — in spite of reaching out to more functionally-equivalent versions — results in a significantly better dependability stance: a mere 3% of failure instead of 6%. And this result can be achieved by allocating, on average, less computational capacity (a lesser degree of version invocations). The key message to convey here is that — if one can afford the investment needed for the development of additional versions — one should consider this option for truly mission-critical applications. Also, when comparing scenarios B and/or C with E, the observation of E performing slightly better in terms of availability could have been anticipated, as B/C are reactive in nature and respond to actual failures rather than proactively adjusting the redundancy upwards were the agreed safety margin violated (scenario E).

[a] In Fig. 7.9, 7.10 and 7.11, one can occasionally observe additional disturbances: these result from performance failures or from RVF failure that emerge because of sampling response values (so-called ballots) from a specific distribution — *cf.* Sect. 2.6.1.1.

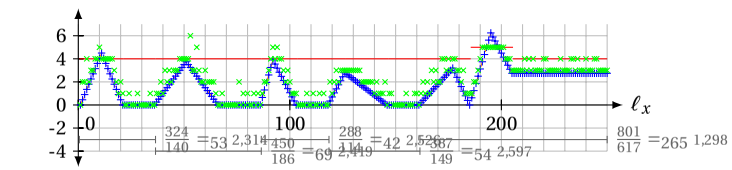
[b] When comparing with strategy D, observe the significant improvement with respect to the number of scheme failures when applying a small safety margin.

[c] Even if there would be no reduction in resource allocation at all, the amount of functionally-equivalent versions in the system would be the same, with identical development cost, yet the application of A-NVP would provide insight in how well each of these versions performs, and could be used for in-depth failure root cause analysis.

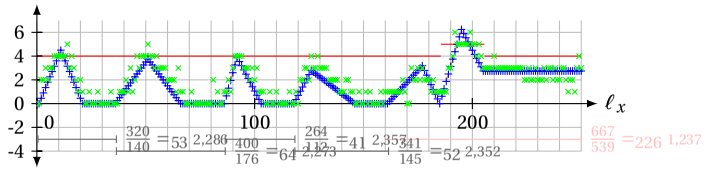
7.4 Identifying Scenarios in which the Approach does not Perform Well

Now that we have shown the benefits of using dynamic redundancy schemata and have shown that A-NVP dependability strategies can successfully realise these gains, we will explore the conditions under which their use is *not* guaranteed to translate in performance gains. After all, "a precise characterisation of the amount of [redundancy] necessary to deal with a certain [fault model] is not always easy or even possible" [78].

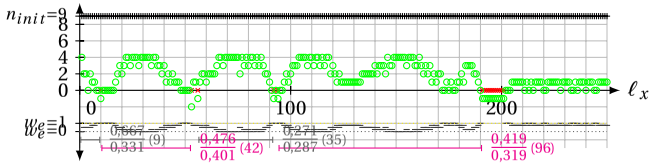
– Ideally, monitored versions should be continuously used. This will result in more accurate normalised dissent measurements. Even though measurements are generated for periods in which replicas are just sitting idle, the evolution of these measurements relies on the assumption that failures are transient in nature, and that temporarily taking a faulty version out of service may remove the detrimental effects of that version during a certain time interval, during which it may eventually recover. During that interval, the penalisation mechanism will gradually assess the version from seriously untrustworthy to moderately



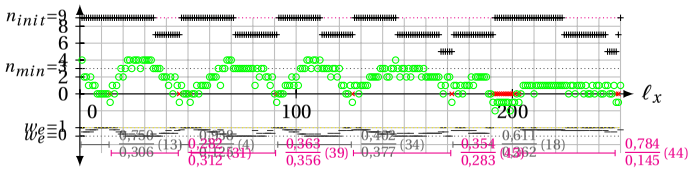
(a) scenario A: failures overview



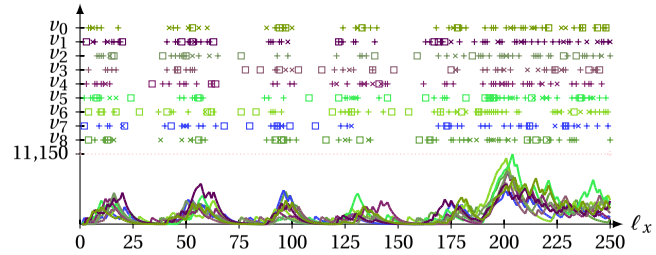
(b) scenario B: failures overview



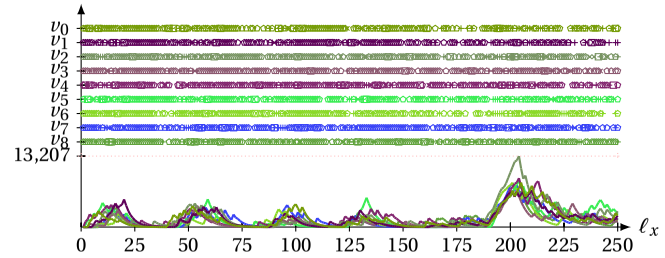
(c) scenario A: resource allocation & under-/overshooting



(d) scenario B: resource allocation & under-/overshooting



(e) scenario A: normalised dissent & observed disturbances



(f) scenario B: normalised dissent & observed disturbances

Figure 7.9: Experiment 7.9: comparison of various redundancy management strategies (scenario A — *i.e.* traditional NVP — *vs.* scenario B).

n_{min}	9	$\ell^{\mathbb{C}}_{failure}\ (c^{(\mathbb{C},\ell)}_{max} - m^{(\mathbb{C},\ell)} < 0)$			6% (16 of 250)
n_{max}	9	$\sum n^{(\mathbb{C},\ell)} = 2250$	$\sum cr(e^{(\mathbb{C},\ell)}) = 1206(1362)$	$\triangle_{success}$	0% (0 of 0)
n_{init}	9	$\sum(n^{(\mathbb{C},\ell)} - cr(e^{(\mathbb{C},\ell)})) = 1044(888)$		$\nabla_{failure}$	0% (0 of 0)
t_{max}	15,000	$w^{(\mathbb{C},\ell)}_{e} \in_{234} [0,000, 1,000]$	$\mu = 0,507$	$\sigma = 0,352$	$ci_{90\%}$: (4.70E-1, 5.45E-1)
$\ell^{\mathbb{C}}_{total}$	250	$n^{(\mathbb{C},\ell)} \in [9,9]$	$\mu = 9,000$	$\sigma = 0,000$	$ci_{90\%}$: (9.00, 9.00)
TTF	9	$XTTF - 2 = 52$	$MTBF =_{15} 11,733$	$MTTR =_{6} 2,667$	

(a) scenario A: failures overview

n_{min}	3	$\ell^{\mathbb{C}}_{failure}\ (c^{(\mathbb{C},\ell)}_{max} - m^{(\mathbb{C},\ell)} < 0)$			8% (20 of 250)
n_{max}	9	$\sum n^{(\mathbb{C},\ell)} = 1992$	$\sum cr(e^{(\mathbb{C},\ell)}) = 1112(1236)$	$\triangle_{success}$	71% (5 of 7)
n_{init}	9	$\sum(n^{(\mathbb{C},\ell)} - cr(e^{(\mathbb{C},\ell)})) = 880(756)$		$\nabla_{failure}$	0% (0 of 0)
t_{max}	15,000	$w^{(\mathbb{C},\ell)}_{e} \in_{230} [0,000, 1,000]$	$\mu = 0,492$	$\sigma = 0,352$	$ci_{90\%}$: (4.54E-1, 5.31E-1)
$\ell^{\mathbb{C}}_{total}$	250	$n^{(\mathbb{C},\ell)} \in [5,9]$	$\mu = 7,968$	$\sigma = 1,165$	$ci_{90\%}$: (7.85, 8.09)
r_d	20	$\square_{fill}{}^{d} \in_{250} [0,20]$	$\mu = 8,520$	$\sigma = 6,005$	$ci_{90\%}$: (7.90, 9.14)
		$\square_{reset}{}^{d} \in_{16} [1,20]$	$\mu = 14,312$	$\sigma = 7,078$	$ci_{90\%}$: (11.40, 17.22)
r_u	1	$\square_{fill}{}^{u} \in_{250} [0,1]$	$\mu = 0,080$	$\sigma = 0,272$	$ci_{90\%}$: (5.17E-2, 1.08E-1)
		$\square_{reset}{}^{u} \in_{11} [1,1]$	$\mu = 1,000$	$\sigma = 0,000$	$ci_{90\%}$: (1.00, 1.00)
r_f	1	$\square_{corrupt}$	0% (0 of 250)		
TTF	13	$XTTF - 2 = 45$	$MTBF =_{19} 11,368$	$MTTR =_{9} 2,222$	

(b) scenario B: failures overview

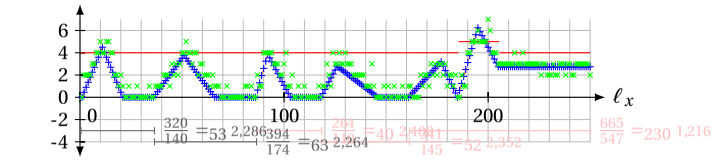
version	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
re-integration latency	n/a n/a	n/a n/a	n/a n/a	n/a n/a	n/a n/a	n/a n/a	n/a n/a	n/a n/a	n/a n/a
exclusion latency	n/a n/a	n/a n/a	n/a n/a	n/a n/a	n/a n/a	n/a n/a	n/a n/a	n/a n/a	n/a n/a
exclusion failures	n/a n/a	n/a n/a	n/a n/a	n/a n/a	n/a n/a	n/a n/a	n/a n/a	n/a n/a	n/a n/a
successive usage period	$\mu_1 = 250$ $\sigma = 0$	$\mu_1 = 250$ $\sigma = 0$	$\mu_1 = 250$ $\sigma = 0$	$\mu_1 = 250$ $\sigma = 0$	$\mu_1 = 250$ $\sigma = 0$	$\mu_1 = 250$ $\sigma = 0$	$\mu_1 = 250$ $\sigma = 0$	$\mu_1 = 250$ $\sigma = 0$	$\mu_1 = 250$ $\sigma = 0$
$\#rounds(\mathbb{C},v) - \#consent(\mathbb{C},v)$ $\rightsquigarrow \ell^{\mathbb{C}}_{total}$ $\rightsquigarrow \#rounds(\mathbb{C},v)$	61 24 % 24 %	70 28 % 28 %	59 24 % 24 %	52 21 % 21 %	60 24 % 24 %	68 27 % 27 %	68 27 % 27 %	54 22 % 22 %	64 26 % 26 %
$\#rounds(\mathbb{C},v)$ $\rightsquigarrow \ell^{\mathbb{C}}_{total}$	250 100 %	250 100 %	250 100 %	250 100 %	250 100 %	250 100 %	250 100%	250 100 %	250 100%
$\#consent(\mathbb{C},v)$ $\rightsquigarrow \#rounds(\mathbb{C},v)$	183 73 %	171 68 %	183 73 %	190 76 %	182 73 %	178 71 %	177 71 %	186 74 %	181 72%
normalised dissent	$\mu_{250} = 1,662$ $\sigma = 1,797$	$\mu_{230} = 1,762$ $\sigma = 1,640$	$\mu_{250} = 1,461$ $\sigma = 1,634$	$\mu_{30} = 1,305$ $\sigma = 1,318$	$\mu_{250} = 1,437$ $\sigma = 1,339$	$\mu_{30} = 1,940$ $\sigma = 2,285$	$\mu_{250} = 1,731$ $\sigma = 1,869$	$\mu_{30} = 1,376$ $\sigma = 1,316$	$\mu_{230} = 1,682$ $\sigma = 1,761$
end-to-end response time	$\mu_{250} = 9,227$ $\sigma = 2,098$	$\mu_{230} = 9,171$ $\sigma = 2,198$	$\mu_{250} = 9,435$ $\sigma = 2,182$	$\mu_{30} = 9,268$ $\sigma = 2,309$	$\mu_{250} = 9,022$ $\sigma = 1,829$	$\mu_{30} = 9,218$ $\sigma = 2,288$	$\mu_{250} = 9,510$ $\sigma = 2,361$	$\mu_{30} = 9,372$ $\sigma = 2,178$	$\mu_{230} = 9,505$ $\sigma = 2,195$
TTF $\rightsquigarrow$ inv	3 0,333	5 0,200	7 0,143	3 0,333	2 0,500	1 1,000	8 0,125	8 0,125	6 0,167
XTTF-2 $\rightsquigarrow$ inv	4 0,250	6 0,167	8 0,125	11 0,091	6 0,167	4 0,250	10 0,100	12 0,083	10 0,100
MTBF $\rightsquigarrow$ inv	3,436 0,291	3,000 0,333	3,939 0,254	4,714 0,212	3,339 0,299	3,316 0,302	3,528 0,283	4,333 0,231	3,765 0,266
MTTR $\rightsquigarrow$ inv	1,436 0,696	1,579 0,633	1,690 0,592	1,483 0,674	1,500 0,667	1,871 0,534	1,636 0,611	1,394 0,717	1,457 0,686

(c) scenario A: resource allocation & under-/overshooting

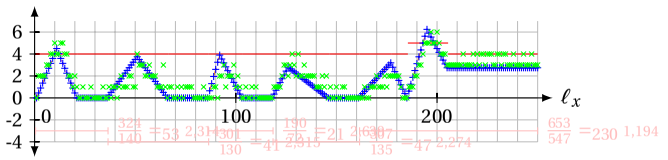
version	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
re-integration latency	$\mu_{23} = 1,522$ $\sigma = 0,790$	$\mu_{21} = 1,476$ $\sigma = 0,602$	$\mu_{22} = 1,273$ $\sigma = 0,456$	$\mu_{22} = 1,364$ $\sigma = 0,492$	$\mu_{19} = 1,368$ $\sigma = 0,831$	$\mu_{23} = 1,261$ $\sigma = 0,541$	$\mu_{22} = 1,545$ $\sigma = 1,057$	$\mu_{20} = 1,300$ $\sigma = 0,470$	$\mu_{16} = 1,188$ $\sigma = 0,403$
exclusion latency	$\mu_{12} = 14,000$ $\sigma = 14,032$	$\mu_{13} = 13,538$ $\sigma = 15,463$	$\mu_{10} = 15,900$ $\sigma = 16,829$	$\mu_{11} = 15,273$ $\sigma = 13,777$	$\mu_{13} = 12,538$ $\sigma = 13,408$	$\mu_{9} = 18,333$ $\sigma = 15,305$	$\mu_{10} = 17,300$ $\sigma = 16,787$	$\mu_{9} = 19,000$ $\sigma = 17,464$	$\mu_{8} = 23,000$ $\sigma = 15,525$
exclusion failures	$\mu_{12} = 3,917$ $\sigma = 4,274$	$\mu_{13} = 4,231$ $\sigma = 4,919$	$\mu_{10} = 4,000$ $\sigma = 6,096$	$\mu_{11} = 3,455$ $\sigma = 2,979$	$\mu_{13} = 4,000$ $\sigma = 4,743$	$\mu_{9} = 5,889$ $\sigma = 5,159$	$\mu_{10} = 4,100$ $\sigma = 5,131$	$\mu_{9} = 5,000$ $\sigma = 4,975$	$\mu_{8} = 5,750$ $\sigma = 3,919$
successive usage period	$\mu_{35} = 6,000$ $\sigma = 10,952$	$\mu_{31} = 7,000$ $\sigma = 12,245$	$\mu_{28} = 7,857$ $\sigma = 13,534$	$\mu_{31} = 7,267$ $\sigma = 11,020$	$\mu_{36} = 8,577$ $\sigma = 12,494$	$\mu_{29} = 7,586$ $\sigma = 11,703$	$\mu_{31} = 6,294$ $\sigma = 12,264$	$\mu_{26} = 8,500$ $\sigma = 13,846$	$\mu_{19} = 11,895$ $\sigma = 15,463$
$\#rounds(\mathbb{C},v) - \#consent(\mathbb{C},v)$ $\rightsquigarrow \ell^{\mathbb{C}}_{total}$ $\rightsquigarrow \#rounds(\mathbb{C},v)$	58 23 % 27 %	63 25 % 29 %	55 22 % 25 %	41 16% 19%	62 25 % 28 %	59 24 % 27 %	46 18 % 21 %	56 22 % 25 %	53 21 % 23 %
$\#rounds(\mathbb{C},v)$ $\rightsquigarrow \ell^{\mathbb{C}}_{total}$	215 86 %	219 88 %	222 89 %	220 88%	224 90 %	221 88 %	216 86 %	224 90 %	231 92 %
$\#consent(\mathbb{C},v)$ $\rightsquigarrow \#rounds(\mathbb{C},v)$	150 70 %	149 68 %	158 71 %	169 77%	155 69 %	155 70 %	164 76 %	163 73 %	167 72 %
normalised dissent	$\mu_{250} = 1,545$ $\sigma = 1,744$	$\mu_{250} = 1,304$ $\sigma = 1,625$	$\mu_{30} = 1,777$ $\sigma = 2,406$	$\mu_{230} = 1,102$ $\sigma = 1,301$	$\mu_{250} = 1,675$ $\sigma = 1,849$	$\mu_{30} = 1,614$ $\sigma = 1,823$	$\mu_{250} = 1,177$ $\sigma = 1,605$	$\mu_{230} = 1,471$ $\sigma = 1,644$	$\mu_{250} = 1,247$ $\sigma = 1,347$
end-to-end response time	$\mu_{215} = 9,519$ $\sigma = 2,353$	$\mu_{219} = 9,306$ $\sigma = 1,972$	$\mu_{22} = 9,283$ $\sigma = 2,301$	$\mu_{230} = 8,979$ $\sigma = 1,869$	$\mu_{224} = 9,454$ $\sigma = 2,401$	$\mu_{21} = 9,047$ $\sigma = 2,002$	$\mu_{216} = 9,401$ $\sigma = 2,051$	$\mu_{224} = 9,475$ $\sigma = 2,412$	$\mu_{231} = 9,286$ $\sigma = 2,169$
TTF $\rightsquigarrow$ inv	3 0,333	5 0,200	7 0,143	3 0,333	2 0,500	1 1,000	8 0,125	8 0,125	6 0,167
XTTF-2 $\rightsquigarrow$ inv	4 0,250	6 0,167	8 0,125	11 0,091	6 0,167	4 0,250	15 0,067	10 0,100	10 0,100
MTBF $\rightsquigarrow$ inv	4,083 0,245	3,357 0,298	4,261 0,235	5,432 0,184	3,765 0,266	3,731 0,268	4,854 0,206	4,333 0,231	4,283 0,234
MTTR $\rightsquigarrow$ inv	1,647 0,607	1,765 0,567	2,000 0,500	1,556 0,643	1,600 0,625	1,867 0,536	1,885 0,531	1,600 0,625	1,457 0,686

(d) scenario B: resource allocation & under-/overshooting

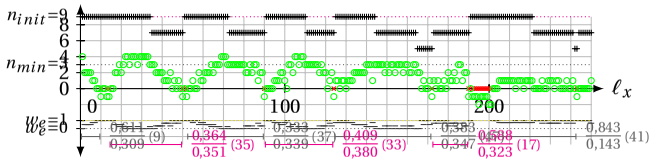
Table 7.7: Experiment 7.9: redundancy configuration/allocation and failure statistics (scenario A — *i.e.* traditional NVP — *vs.* scenario A).



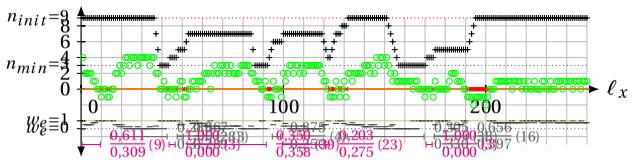
(a) scenario C: failures overview



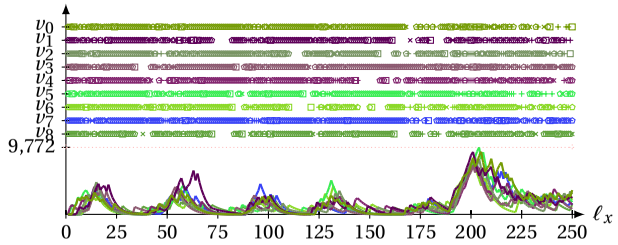
(b) scenario D: failures overview



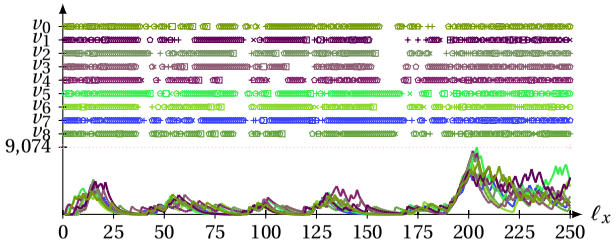
(c) scenario C: resource allocation & under-/overshooting



(d) scenario D: resource allocation & under-/overshooting



(e) scenario C: normalised dissent & observed disturbances



(f) scenario D: normalised dissent & observed disturbances

Figure 7.10: Experiment 7.9: comparison of various redundancy management strategies (scenarios C and D).

n_{min}	3	$\ell^{\mathcal{C}}_{failure}\ (c^{(\mathcal{C},\ell)}_{max} - m^{(\mathcal{C},\ell)} < 0)$			8% (20 of 250)
n_{max}	9	$\sum n^{(\mathcal{C},\ell)} = 1984$	$\sum cr(e^{(\mathcal{C},\ell)}) = 1116 (1262)$	$\triangle_{success}$	100% (6 of 6)
n_{init}	9	$\sum (n^{(\mathcal{C},\ell)} - cr(e^{(\mathcal{C},\ell)})) = 868 (722)$		$\nabla_{failure}$	0% (0 of 7)
t_{max}	15,000	$w_e^{(\mathcal{C},\ell)} \in_{230} [0,000, 1,000]$	$\mu = 0{,}510$	$\sigma = 0{,}373$	$ci_{90\%}$: (4.70E-1, 5.51E-1)
$\ell^{\mathcal{C}}_{total}$	250	$n^{(\mathcal{C},\ell)} \in [5, 9]$	$\mu = 7{,}936$	$\sigma = 1{,}149$	$ci_{90\%}$: (7.82, 8.06)
r_d	20	$\square_{fill}{}^{d} \in_{250} [0, 20]$	$\mu = 8{,}376$	$\sigma = 5{,}929$	$ci_{90\%}$: (7.76, 8.99)
		$\square_{reset}{}^{d} \in_{17} [1, 20]$	$\mu = 13{,}118$	$\sigma = 7{,}841$	$ci_{90\%}$: (9.99, 16.25)
r_u	1	$\square_{fill}{}^{u} \in_{250} [0, 1]$	$\mu = 0{,}080$	$\sigma = 0{,}272$	$ci_{90\%}$: (5.17E-2, 1.08E-1)
		$\square_{reset}{}^{u} \in_{10} [1, 1]$	$\mu = 1{,}000$	$\sigma = 0{,}000$	$ci_{90\%}$: (1.00, 1.00)
r_f	1	$\square_{corrupt}$	0% (0 of 250)		
TTF	9	$XTTF - 2 = 12$	$MTBF_{=19}$ 11,263	$MTTR_{=10}$ 2,000	

(a) scenario C: failures overview

n_{min}	3	$\ell^{\mathcal{C}}_{failure}\ (c^{(\mathcal{C},\ell)}_{max} - m^{(\mathcal{C},\ell)} < 0)$			12% (29 of 250)
n_{max}	9	$\sum n^{(\mathcal{C},\ell)} = 1775$	$\sum cr(e^{(\mathcal{C},\ell)}) = 1024 (1160)$	$\triangle_{success}$	53% (10 of 19)
n_{init}	9	$\sum (n^{(\mathcal{C},\ell)} - cr(e^{(\mathcal{C},\ell)})) = 751 (615)$		$\nabla_{failure}$	0% (0 of 14)
t_{max}	15,000	$w_e^{(\mathcal{C},\ell)} \in_{221} [0,000, 1,000]$	$\mu = 0{,}507$	$\sigma = 0{,}391$	$ci_{90\%}$: (4.64E-1, 5.50E-1)
$\ell^{\mathcal{C}}_{total}$	250	$n^{(\mathcal{C},\ell)} \in [3, 9]$	$\mu = 7{,}100$	$\sigma = 2{,}095$	$ci_{90\%}$: (6.88, 7.32)
r_d	20	$\square_{fill}{}^{d} \in_{250} [0, 20]$	$\mu = 11{,}232$	$\sigma = 7{,}884$	$ci_{90\%}$: (10.41, 12.05)
		$\square_{reset}{}^{d} \in_{13} [2, 20]$	$\mu = 9{,}538$	$\sigma = 8{,}212$	$ci_{90\%}$: (5.79, 13.28)
r_u	1	$\square_{fill}{}^{u} \in_{250} [0, 1]$	$\mu = 0{,}116$	$\sigma = 0{,}321$	$ci_{90\%}$: (8.26E-2, 1.49E-1)
		$\square_{reset}{}^{u} \in_{13} [1, 1]$	$\mu = 1{,}000$	$\sigma = 0{,}000$	$ci_{90\%}$: (1.00, 1.00)
r_f	1	$\square_{corrupt}$	0% (0 of 250)		
TTF	9	$XTTF - 2 = 12$	$MTBF_{=28}$ 5,821	$MTTR_{=13}$ 2,231	
c_{sm}	f	$c_{sm} = 0$	n/a	n/a	$ci_{90\%}$: (0.00, 0.00)
		$\ell_{suspect}\ (0 \le c^{(\mathcal{C},\ell)}_{max} - m^{(\mathcal{C},\ell)} < c_{sm})$			0% (0 of 250)

(b) scenario D: failures overview

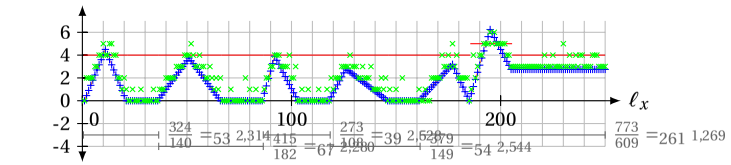
version	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
re-integration latency	$\mu_{12} = 2{,}000$ $\sigma = 1{,}044$	$\mu_{10} = 3{,}600$ $\sigma = 3{,}134$	$\mu_{10} = 3{,}000$ $\sigma = 2{,}261$	$\mu_6 = 2{,}300$ $\sigma = 2{,}345$	$\mu_8 = 3{,}500$ $\sigma = 2{,}878$	$\mu_{13} = 2{,}231$ $\sigma = 1{,}013$	$\mu_{18} = 2{,}833$ $\sigma = 2{,}431$	$\mu_6 = 1{,}667$ $\sigma = 0{,}816$	$\mu_{10} = 4{,}100$ $\sigma = 2{,}767$
exclusion latency	$\mu_{11} = 18{,}636$ $\sigma = 35{,}973$	$\mu_{10} = 17{,}800$ $\sigma = 21{,}658$	$\mu_9 = 22{,}222$ $\sigma = 23{,}231$	$\mu_6 = 33{,}333$ $\sigma = 36{,}247$	$\mu_6 = 34{,}167$ $\sigma = 39{,}957$	$\mu_{13} = 13{,}692$ $\sigma = 16{,}510$	$\mu_{15} = 10{,}733$ $\sigma = 12{,}848$	$\mu_6 = 34{,}500$ $\sigma = 63{,}406$	$\mu_9 = 18{,}556$ $\sigma = 15{,}923$
exclusion failures	$\mu_{11} = 4{,}818$ $\sigma = 6{,}226$	$\mu_{10} = 5{,}500$ $\sigma = 5{,}720$	$\mu_9 = 5{,}222$ $\sigma = 6{,}379$	$\mu_6 = 6{,}367$ $\sigma = 6{,}432$	$\mu_6 = 8{,}667$ $\sigma = 9{,}688$	$\mu_{13} = 3{,}923$ $\sigma = 4{,}873$	$\mu_{15} = 2{,}867$ $\sigma = 2{,}503$	$\mu_6 = 7{,}333$ $\sigma = 9{,}331$	$\mu_9 = 4{,}667$ $\sigma = 3{,}606$
successive usage period	$\mu_{24} = 9{,}375$ $\sigma = 26{,}441$	$\mu_{37} = 5{,}757$ $\sigma = 14{,}926$	$\mu_{11} = 7{,}065$ $\sigma = 16{,}555$	$\mu_{15} = 14{,}267$ $\sigma = 28{,}029$	$\mu_{18} = 7{,}821$ $\sigma = 22{,}465$	$\mu_{21} = 7{,}483$ $\sigma = 14{,}868$	$\mu_{51} = 3{,}882$ $\sigma = 9{,}046$	$\mu_{10} = 22{,}700$ $\sigma = 52{,}955$	$\mu_{41} = 4{,}732$ $\sigma = 11{,}360$
$\#rounds(\mathcal{C},v) - \#consent(\mathcal{C},v)$ $\hookrightarrow\ \ell^{\mathcal{C}}_{total}$ $\hookrightarrow\ \#rounds(\mathcal{C},v)$	61 24 % 27 %	67 27 % 31 %	54 22 % 25 %	47 19 % 20 %	61 24 % 27 %	58 23 % 26 %	52 21 % 26 %	55 22 % 23 %	51 20 % 24 %
$\#rounds(\mathcal{C},v)$ $\hookrightarrow\ \ell^{\mathcal{C}}_{total}$	226 90 %	213 85 %	219 88 %	235 94 %	222 89 %	221 88 %	199 80 %	240 96 %	209 84 %
$\#consent(\mathcal{C},v)$ $\hookrightarrow\ \#rounds(\mathcal{C},v)$	157 69 %	138 65 %	159 73 %	181 77 %	157 71 %	156 71 %	138 69 %	175 73 %	149 71 %
normalised dissent	$\mu_{250} = 1{,}753$ $\sigma = 1{,}980$	$\mu_{250} = 1{,}906$ $\sigma = 1{,}858$	$\mu_{250} = 1{,}388$ $\sigma = 1{,}640$	$\mu_{230} = 1{,}213$ $\sigma = 1{,}337$	$\mu_{250} = 1{,}486$ $\sigma = 1{,}446$	$\mu_{230} = 1{,}634$ $\sigma = 1{,}846$	$\mu_{250} = 1{,}428$ $\sigma = 1{,}395$	$\mu_{250} = 1{,}525$ $\sigma = 1{,}696$	$\mu_{230} = 1{,}305$ $\sigma = 1{,}405$
end-to-end response time	$\mu_{261} = 9{,}360$ $\sigma = 2{,}339$	$\mu_{213} = 9{,}400$ $\sigma = 2{,}285$	$\mu_{219} = 9{,}228$ $\sigma = 2{,}120$	$\mu_{235} = 9{,}212$ $\sigma = 2{,}199$	$\mu_{222} = 9{,}226$ $\sigma = 2{,}096$	$\mu_{221} = 9{,}193$ $\sigma = 2{,}043$	$\mu_{193} = 9{,}218$ $\sigma = 2{,}180$	$\mu_{240} = 9{,}448$ $\sigma = 2{,}179$	$\mu_{208} = 9{,}356$ $\sigma = 2{,}259$
TTF $\hookrightarrow$ inv	3 0,333	5 0,200	7 0,143	3 0,333	2 0,500	1 1,000	8 0,125	8 0,125	6 0,167
XTTF-2 $\hookrightarrow$ inv	4 0,250	6 0,167	8 0,125	11 0,091	6 0,167	4 0,250	10 0,100	10 0,100	10 0,100
MTBF $\hookrightarrow$ inv	3,692 0,271	3,500 0,286	4,174 0,240	5,000 0,200	3,765 0,266	3,824 0,262	4,690 0,213	4,217 0,237	4,786 0,209
MTTR $\hookrightarrow$ inv	1,919 0,521	1,788 0,559	2,192 0,456	1,643 0,609	2,086 0,479	2,667 0,375	2,250 0,444	1,636 0,611	2,107 0,475

(c) scenario C: resource allocation & under-/overshooting

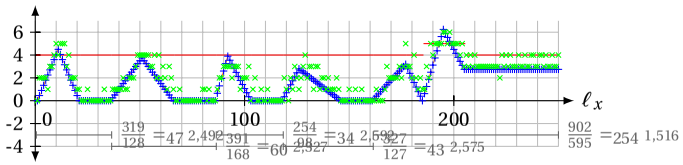
version	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
re-integration latency	$\mu_{17} = 2{,}941$ $\sigma = 2{,}106$	$\mu_{17} = 3{,}588$ $\sigma = 4{,}124$	$\mu_{12} = 4{,}083$ $\sigma = 4{,}420$	$\mu_{13} = 4{,}538$ $\sigma = 4{,}807$	$\mu_{12} = 5{,}333$ $\sigma = 4{,}599$	$\mu_{18} = 2{,}444$ $\sigma = 1{,}688$	$\mu_{13} = 4{,}308$ $\sigma = 5{,}186$	$\mu_{15} = 2{,}067$ $\sigma = 0{,}884$	$\mu_{17} = 3{,}588$ $\sigma = 3{,}743$
exclusion latency	$\mu_{14} = 7{,}500$ $\sigma = 11{,}889$	$\mu_{14} = 4{,}429$ $\sigma = 8{,}582$	$\mu_{12} = 8{,}500$ $\sigma = 12{,}139$	$\mu_9 = 8{,}556$ $\sigma = 12{,}914$	$\mu_9 = 9{,}889$ $\sigma = 14{,}031$	$\mu_{10} = 8{,}800$ $\sigma = 16{,}226$	$\mu_{10} = 8{,}100$ $\sigma = 11{,}140$	$\mu_{14} = 3{,}286$ $\sigma = 7{,}710$	$\mu_{10} = 8{,}100$ $\sigma = 11{,}426$
exclusion failures	$\mu_{14} = 1{,}857$ $\sigma = 1{,}610$	$\mu_{14} = 1{,}857$ $\sigma = 2{,}214$	$\mu_{12} = 1{,}750$ $\sigma = 1{,}485$	$\mu_9 = 1{,}889$ $\sigma = 1{,}537$	$\mu_9 = 2{,}667$ $\sigma = 2{,}872$	$\mu_{10} = 1{,}900$ $\sigma = 1{,}912$	$\mu_{10} = 1{,}900$ $\sigma = 1{,}287$	$\mu_{14} = 1{,}500$ $\sigma = 1{,}345$	$\mu_{10} = 2{,}100$ $\sigma = 1{,}524$
successive usage period	$\mu_{50} = 2{,}840$ $\sigma = 7{,}654$	$\mu_{61} = 2{,}148$ $\sigma = 6{,}675$	$\mu_{49} = 2{,}898$ $\sigma = 8{,}506$	$\mu_{31} = 2{,}271$ $\sigma = 7{,}119$	$\mu_{44} = 1{,}984$ $\sigma = 6{,}606$	$\mu_{41} = 3{,}386$ $\sigma = 8{,}966$	$\mu_{16} = 2{,}429$ $\sigma = 7{,}383$	$\mu_{31} = 5{,}161$ $\sigma = 10{,}674$	$\mu_{61} = 2{,}180$ $\sigma = 6{,}627$
$\#rounds(\mathcal{C},v) - \#consent(\mathcal{C},v)$ $\hookrightarrow\ \ell^{\mathcal{C}}_{total}$ $\hookrightarrow\ \#rounds(\mathcal{C},v)$	53 21 % 27 %	59 24 % 31 %	54 22 % 27 %	44 18 % 23 %	56 22 % 30 %	56 22 % 27 %	46 18 % 24 %	42 17 % 19 %	45 18 % 24 %
$\#rounds(\mathcal{C},v)$ $\hookrightarrow\ \ell^{\mathcal{C}}_{total}$	200 80 %	189 76 %	201 80 %	191 76 %	186 74 %	206 82 %	194 78 %	219 88 %	189 76 %
$\#consent(\mathcal{C},v)$ $\hookrightarrow\ \#rounds(\mathcal{C},v)$	138 69 %	123 65 %	138 69 %	135 71 %	123 66 %	142 69 %	140 72 %	166 76 %	134 71 %
normalised dissent	$\mu_{230} = 1{,}457$ $\sigma = 1{,}570$	$\mu_{250} = 1{,}632$ $\sigma = 1{,}732$	$\mu_{230} = 1{,}443$ $\sigma = 1{,}600$	$\mu_{230} = 1{,}202$ $\sigma = 1{,}124$	$\mu_{250} = 1{,}593$ $\sigma = 1{,}593$	$\mu_{230} = 1{,}663$ $\sigma = 1{,}901$	$\mu_{250} = 1{,}145$ $\sigma = 1{,}298$	$\mu_{230} = 1{,}129$ $\sigma = 1{,}259$	$\mu_{250} = 1{,}139$ $\sigma = 1{,}208$
end-to-end response time	$\mu_{200} = 9{,}237$ $\sigma = 2{,}199$	$\mu_{189} = 9{,}302$ $\sigma = 2{,}307$	$\mu_{201} = 9{,}220$ $\sigma = 2{,}165$	$\mu_{191} = 9{,}295$ $\sigma = 2{,}147$	$\mu_{186} = 9{,}222$ $\sigma = 2{,}181$	$\mu_{206} = 9{,}569$ $\sigma = 2{,}385$	$\mu_{194} = 9{,}266$ $\sigma = 2{,}148$	$\mu_{219} = 9{,}510$ $\sigma = 2{,}172$	$\mu_{189} = 9{,}310$ $\sigma = 2{,}085$
TTF $\hookrightarrow$ inv	3 0,333	5 0,200	7 0,143	3 0,333	2 0,500	1 1,000	8 0,125	8 0,125	6 0,167
XTTF-2 $\hookrightarrow$ inv	4 0,250	6 0,167	8 0,125	11 0,091	6 0,167	4 0,250	10 0,100	10 0,100	10 0,100
MTBF $\hookrightarrow$ inv	4,191 0,239	3,980 0,251	4,500 0,222	6,059 0,165	4,283 0,234	4,721 0,212	5,000 0,200	5,154 0,194	5,750 0,174
MTTR $\hookrightarrow$ inv	2,182 0,458	2,690 0,372	2,640 0,379	1,833 0,545	2,250 0,444	2,143 0,467	2,400 0,417	2,429 0,412	2,478 0,404

(d) scenario D: resource allocation & under-/overshooting

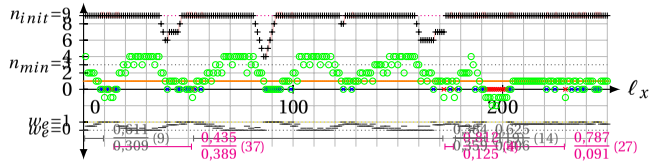
Table 7.8: Experiment 7.9: redundancy configuration/allocation and failure statistics (scenarios C and D).



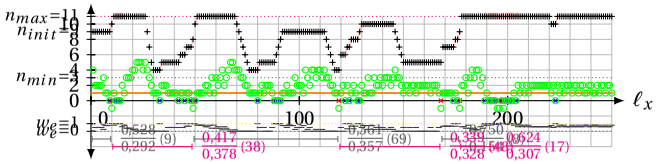
(a) scenario E: failures overview



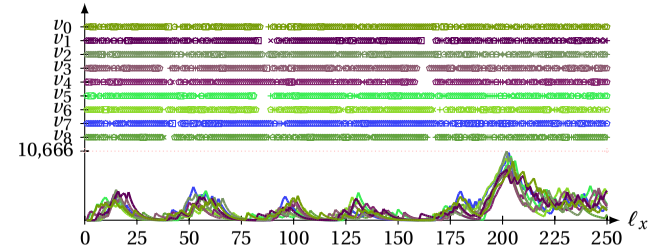
(b) scenario F: failures overview



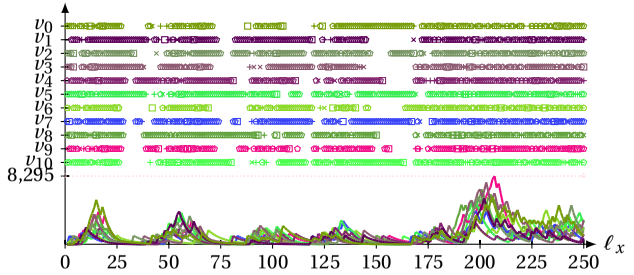
(c) scenario E: resource allocation & under-/overshooting



(d) scenario F: resource allocation & under-/overshooting



(e) scenario E: normalised dissent & observed disturbances



(f) scenario F: normalised dissent & observed disturbances

Figure 7.11: Experiment 7.9: comparison of various redundancy management strategies (scenarios E and F).

n_{min}	3	$\ell^{\mathbb{C}}_{failure}\ (c^{(\mathbb{C},\ell)}_{max} - m^{(\mathbb{C},\ell)} < 0)$			7% (18 of 250)
n_{max}	9	$\sum n^{(\mathbb{C},\ell)} = 2164$	$\sum cr(e^{(\mathbb{C},\ell)}) = 1188 (1334)$	$\triangle_{success}$	100% (11 of 11)
n_{init}	9	$\sum (n^{(\mathbb{C},\ell)} - cr(e^{(\mathbb{C},\ell)})) = 976 (830)$		$\nabla_{failure}$	0% (0 of 12)
t_{max}	15,000	$w^{(\mathbb{C},\ell)}_e \in_{232} [0,000, 1,000]$	$\mu = 0,514$	$\sigma = 0,364$	$ci_{90\%}$: (4.75E-1, 5.53E-1)
$\ell^{\mathbb{C}}_{total}$	250	$n^{(\mathbb{C},\ell)} \in [4,9]$	$\mu = 8,656$	$\sigma = 0,928$	$ci_{90\%}$: (8.56, 8.75)
r_d	20	$\square_{fill}{}^{d} \in_{250} [0,20]$	$\mu = 7,300$	$\sigma = 7,545$	$ci_{90\%}$: (6.52, 8.08)
		$\square_{reset}{}^{d} \in_{23} [1,20]$	$\mu = 6,565$	$\sigma = 7,031$	$ci_{90\%}$: (4.15, 8.98)
r_u	1	$\square_{fill}{}^{u} \in_{250} [0,1]$	$\mu = 0,240$	$\sigma = 0,428$	$ci_{90\%}$: (1.95E-1, 2.85E-1)
		$\square_{reset}{}^{u} \in_{23} [1,1]$	$\mu = 1,000$	$\sigma = 0,000$	$ci_{90\%}$: (1.00, 1.00)
r_f	1	$\square_{corrupt}$	0% (0 of 250)		
TTF	9	$XTTF - 2 = 12$	$MTBF =_7 11,941$	$MTTR =_7 2,571$	
c_{sm}	f	$c_{sm} = 1$	n/a	n/a	$ci_{90\%}$: (1.00, 1.00)
		$\ell_{suspect}\ (0 \leq c^{(\mathbb{C},\ell)}_{max} - m^{(\mathbb{C},\ell)} < c_{sm})$			17% (42 of 250)

(a) scenario E: failures overview

n_{min}	3	$\ell^{\mathbb{C}}_{failure}\ (c^{(\mathbb{C},\ell)}_{max} - m^{(\mathbb{C},\ell)} < 0)$			3% (8 of 250)
n_{max}	11	$\sum n^{(\mathbb{C},\ell)} = 2193$	$\sum cr(e^{(\mathbb{C},\ell)}) = 1116 (1284)$	$\triangle_{success}$	87% (20 of 23)
n_{init}	9	$\sum (n^{(\mathbb{C},\ell)} - cr(e^{(\mathbb{C},\ell)})) = 1077 (909)$		$\nabla_{failure}$	0% (0 of 22)
t_{max}	15,000	$w^{(\mathbb{C},\ell)}_e \in_{242} [0,000, 1,000]$	$\mu = 0,479$	$\sigma = 0,343$	$ci_{90\%}$: (4.43E-1, 5.16E-1)
$\ell^{\mathbb{C}}_{total}$	250	$n^{(\mathbb{C},\ell)} \in [4,11]$	$\mu = 8,772$	$\sigma = 2,469$	$ci_{90\%}$: (8.52, 9.03)
r_d	10	$\square_{fill}{}^{d} \in_{250} [0,10]$	$\mu = 6,000$	$\sigma = 4,074$	$ci_{90\%}$: (5.58, 6.42)
		$\square_{reset}{}^{d} \in_{19} [1,10]$	$\mu = 5,368$	$\sigma = 3,947$	$ci_{90\%}$: (3.88, 6.86)
r_u	1	$\square_{fill}{}^{u} \in_{250} [0,1]$	$\mu = 0,160$	$\sigma = 0,367$	$ci_{90\%}$: (1.22E-1, 1.98E-1)
		$\square_{reset}{}^{u} \in_{19} [1,1]$	$\mu = 1,000$	$\sigma = 0,000$	$ci_{90\%}$: (1.00, 1.00)
r_f	1	$\square_{corrupt}$	0% (0 of 250)		
TTF	9	$XTTF - 2 = 48$	$MTBF =_7 25,857$	$MTTR =_7 1,143$	
c_{sm}	f	$c_{sm} = 1$	n/a	n/a	$ci_{90\%}$: (1.00, 1.00)
		$\ell_{suspect}\ (0 \leq c^{(\mathbb{C},\ell)}_{max} - m^{(\mathbb{C},\ell)} < c_{sm})$			13% (32 of 250)

(b) scenario F: failures overview

version	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9
re-integration latency	$\mu_3 = 1,800$ $\sigma = 0,837$	$\mu_4 = 3,500$ $\sigma = 2,380$	$\mu_2 = 2,000$ $\sigma = 0,000$	$\mu_6 = 2,600$ $\sigma = 1,817$	$\mu_3 = 4,000$ $\sigma = 4,359$	$\mu_4 = 2,500$ $\sigma = 3,000$	$\mu_5 = 2,600$ $\sigma = 1,949$	$\mu_1 = 3,000$ -	$\mu_3 = 2,667$ $\sigma = 1,528$
exclusion latency	$\mu_5 = 31,000$ $\sigma = 31,686$	$\mu_4 = 37,250$ $\sigma = 41,484$	$\mu_2 = 77,500$ $\sigma = 0,707$	$\mu_6 = 28,600$ $\sigma = 13,465$	$\mu_3 = 50,000$ $\sigma = 56,630$	$\mu_4 = 36,750$ $\sigma = 41,732$	$\mu_5 = 29,400$ $\sigma = 16,772$	$\mu_1 = 34,000$ -	$\mu_3 = 49,000$ $\sigma = 22,113$
exclusion failures	$\mu_3 = 5,000$ $\sigma = 4,528$	$\mu_4 = 7,500$ $\sigma = 8,266$	$\mu_2 = 11,000$ $\sigma = 5,657$	$\mu_6 = 4,000$ $\sigma = 1,225$	$\mu_3 = 9,000$ $\sigma = 9,849$	$\mu_4 = 6,000$ $\sigma = 6,272$	$\mu_5 = 4,400$ $\sigma = 2,191$	$\mu_1 = 7,000$ -	$\mu_3 = 8,000$ $\sigma = 2,646$
successive usage period	$\mu_9 = 17,889$ $\sigma = 28,476$	$\mu_{14} = 11,286$ $\sigma = 27,767$	$\mu_4 = 41,750$ $\sigma = 48,224$	$\mu_{13} = 11,615$ $\sigma = 17,400$	$\mu_{12} = 12,917$ $\sigma = 33,735$	$\mu_{10} = 15,800$ $\sigma = 31,626$	$\mu_{13} = 12,231$ $\sigma = 19,036$	$\mu_8 = 14,000$ $\sigma = 24,249$	$\mu_8 = 19,875$ $\sigma = 29,338$
$\#rounds(\mathbb{C},v) - \#consent(\mathbb{C},v)$ $\hookrightarrow\ \ell^{\mathbb{C}}_{total}$ $\hookrightarrow\ \#rounds(\mathbb{C},v)$	65 26 % 27 %	67 27 % 28 %	59 24 % 24 %	55 22 % 23 %	61 24 % 26 %	62 25 % 26 %	60 24 % 25 %	59 24 % 24 %	54 22 % 22 %
$\#rounds(\mathbb{C},v)$ $\hookrightarrow\ \ell^{\mathbb{C}}_{total}$	241 96 %	236 94 %	246 98 %	237 95 %	238 95 %	240 96 %	237 95 %	247 99 %	242 97 %
$\#consent(\mathbb{C},v)$ $\hookrightarrow\ \#rounds(\mathbb{C},v)$	167 69 %	162 69 %	181 74 %	173 73 %	171 72 %	171 71 %	168 71 %	179 72 %	181 75 %
normalised dissent	$\mu_{250} = 1,748$ $\sigma = 1,789$	$\mu_{250} = 1,765$ $\sigma = 1,763$	$\mu_{230} = 1,594$ $\sigma = 1,955$	$\mu_{250} = 1,523$ $\sigma = 1,597$	$\mu_{250} = 1,458$ $\sigma = 1,526$	$\mu_{230} = 1,685$ $\sigma = 1,915$	$\mu_{230} = 1,633$ $\sigma = 1,853$	$\mu_{250} = 1,780$ $\sigma = 2,028$	$\mu_{230} = 1,301$ $\sigma = 1,497$
end-to-end response time	$\mu_{241} = 9,440$ $\sigma = 2,322$	$\mu_{216} = 9,255$ $\sigma = 2,248$	$\mu_{245} = 9,245$ $\sigma = 2,159$	$\mu_{217} = 9,378$ $\sigma = 2,134$	$\mu_{218} = 9,061$ $\sigma = 2,078$	$\mu_{240} = 9,093$ $\sigma = 2,040$	$\mu_{237} = 9,298$ $\sigma = 2,328$	$\mu_{247} = 9,632$ $\sigma = 2,317$	$\mu_{242} = 9,311$ $\sigma = 1,943$
TTF $\hookrightarrow$ inv	3 0,333	5 0,200	7 0,143	3 0,333	2 0,500	1 1,000	8 0,125	8 0,125	6 0,167
XTTF-2 $\hookrightarrow$ inv	4 0,250	6 0,167	8 0,125	11 0,091	6 0,167	4 0,250	10 0,100	10 0,100	10 0,100
MTBF $\hookrightarrow$ inv	3,519 0,284	3,136 0,319	3,745 0,267	4,854 0,206	3,585 0,279	3,473 0,288	3,898 0,257	4,000 0,250	3,860 0,259
MTTR $\hookrightarrow$ inv	1,595 0,627	1,605 0,623	1,767 0,566	1,483 0,674	1,541 0,649	1,933 0,517	1,839 0,544	1,441 0,694	1,758 0,569

(c) scenario E: resource allocation & under-/overshooting

version	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9	v_{10}
re-integration latency	$\mu_{11} = 5,616$ $\sigma = 5,900$	$\mu_{10} = 4,000$ $\sigma = 6,467$	$\mu_{10} = 4,200$ $\sigma = 3,084$	$\mu_{11} = 5,818$ $\sigma = 7,821$	$\mu_9 = 4,889$ $\sigma = 4,833$	$\mu_{11} = 2,818$ $\sigma = 1,991$	$\mu_{14} = 5,143$ $\sigma = 6,373$	$\mu_{12} = 3,300$ $\sigma = 2,023$	$\mu_9 = 5,625$ $\sigma = 5,553$	$\mu_{14} = 4,500$ $\sigma = 4,071$	$\mu_9 = 4,778$ $\sigma = 4,265$
exclusion latency	$\mu_9 = 16,111$ $\sigma = 17,829$	$\mu_9 = 11,333$ $\sigma = 13,276$	$\mu_9 = 11,500$ $\sigma = 11,339$	$\mu_9 = 9,667$ $\sigma = 12,001$	$\mu_9 = 12,375$ $\sigma = 12,959$	$\mu_9 = 9,778$ $\sigma = 14,202$	$\mu_9 = 4,000$ $\sigma = 5,701$	$\mu_9 = 9,125$ $\sigma = 11,205$	$\mu_5 = 17,400$ $\sigma = 17,069$	$\mu_{10} = 8,000$ $\sigma = 8,907$	$\mu_9 = 12,000$ $\sigma = 14,122$
exclusion failures	$\mu_9 = 4,222$ $\sigma = 5,974$	$\mu_9 = 1,778$ $\sigma = 1,641$	$\mu_9 = 2,625$ $\sigma = 1,996$	$\mu_9 = 3,667$ $\sigma = 4,770$	$\mu_9 = 2,375$ $\sigma = 1,598$	$\mu_9 = 2,111$ $\sigma = 2,088$	$\mu_9 = 1,333$ $\sigma = 0,707$	$\mu_9 = 1,750$ $\sigma = 1,035$	$\mu_5 = 3,200$ $\sigma = 2,168$	$\mu_{10} = 2,300$ $\sigma = 1,337$	$\mu_9 = 2,375$ $\sigma = 1,996$
successive usage period	$\mu_{62} = 2,597$ $\sigma = 9,112$	$\mu_{46} = 2,674$ $\sigma = 8,110$	$\mu_{42} = 3,190$ $\sigma = 7,875$	$\mu_{58} = 2,453$ $\sigma = 7,938$	$\mu_{44} = 3,023$ $\sigma = 9,179$	$\mu_{51} = 4,813$ $\sigma = 10,110$	$\mu_{72} = 1,264$ $\sigma = 3,989$	$\mu_{52} = 3,190$ $\sigma = 8,199$	$\mu_{45} = 2,844$ $\sigma = 10,009$	$\mu_{63} = 1,794$ $\sigma = 4,981$	$\mu_{43} = 2,907$ $\sigma = 8,527$
$\#rounds(\mathbb{C},v) - \#consent(\mathbb{C},v)$ $\hookrightarrow\ \ell^{\mathbb{C}}_{total}$ $\hookrightarrow\ \#round(\mathbb{C},v)$	55 22 % 29 %	41 16 % 20 %	47 19 % 23 %	48 19 % 26 %	43 17 % 21 %	45 18 % 21 %	47 19 % 26 %	40 16 % 19 %	48 19 % 23 %	53 21 % 28 %	50 20% 24%
$\#rounds(\mathbb{C},v)$ $\hookrightarrow\ \ell^{\mathbb{C}}_{total}$	188 75 %	202 81 %	208 83 %	186 74 %	206 82 %	219 88 %	178 71 %	208 83 %	205 82 %	186 74 %	207 83%
$\#consent(\mathbb{C},v)$ $\hookrightarrow\ \#round(\mathbb{C},v)$	130 69 %	160 79 %	160 77 %	134 72 %	161 78 %	171 78 %	126 71 %	165 79 %	153 75 %	130 70 %	154 74%
normalised dissent	$\mu_{250} = 1,301$ $\sigma = 1,344$	$\mu_{250} = 0,847$ $\sigma = 0,994$	$\mu_{250} = 1,008$ $\sigma = 1,129$	$\mu_{250} = 1,191$ $\sigma = 1,533$	$\mu_{250} = 0,759$ $\sigma = 0,637$	$\mu_{230} = 0,946$ $\sigma = 1,053$	$\mu_{250} = 1,009$ $\sigma = 0,873$	$\mu_{250} = 0,879$ $\sigma = 0,826$	$\mu_{250} = 1,088$ $\sigma = 1,357$	$\mu_{250} = 1,212$ $\sigma = 1,457$	$\mu_{250} = 1,035$ $\sigma = 1,141$
end-to-end response time	$\mu_{188} = 9,531$ $\sigma = 2,301$	$\mu_{202} = 9,313$ $\sigma = 2,026$	$\mu_{204} = 9,203$ $\sigma = 2,162$	$\mu_{186} = 9,238$ $\sigma = 2,212$	$\mu_{206} = 9,290$ $\sigma = 2,056$	$\mu_{219} = 9,099$ $\sigma = 2,035$	$\mu_{178} = 9,404$ $\sigma = 2,315$	$\mu_{208} = 9,417$ $\sigma = 2,305$	$\mu_{205} = 9,269$ $\sigma = 2,254$	$\mu_{186} = 9,369$ $\sigma = 2,359$	$\mu_{207} = 9,144$ $\sigma = 2,073$
TTF $\hookrightarrow$ inv	4 0,250	6 0,167	11 0,091	9 0,111	8 0,125	9 0,111	1 1,000	7 0,143	10 0,100	4 0,250	2 0,500
XTTF-2 $\hookrightarrow$ inv	5 0,200	7 0,143	12 0,083	10 0,100	9 0,111	13 0,077	6 0,167	18 0,056	11 0,091	6 0,167	10 0,100
MTBF $\hookrightarrow$ inv	4,455 0,224	6,147 0,163	4,923 0,203	5,103 0,196	5,886 0,170	5,103 0,196	5,861 0,171	6,344 0,158	5,128 0,195	4,829 0,207	4,591 0,218
MTTR $\hookrightarrow$ inv	1,812 0,552	1,957 0,511	1,690 0,592	3,100 0,316	1,708 0,585	2,148 0,466	1,438 0,696	1,889 0,529	1,533 0,652	1,606 0,623	1,866 0,509

(d) scenario F: resource allocation & under-/overshooting

Table 7.9: Experiment 7.9: redundancy configuration/allocation and failure statistics (scenarios E and F).

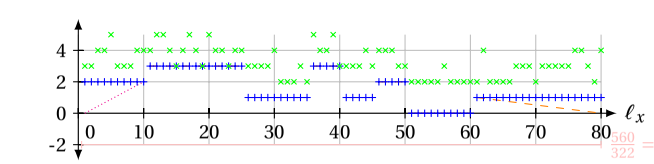
trustworthy.
However, theoretically, the most accurate measurements that would closely keep track of a version's behaviour would be to engage it in every voting round, so that its operational status could be deduced from the voting/partitioning procedure — *v.* Sect. 2.1. As it may be advantageous to temporarily exclude faulty versions from the redundancy configuration, it is important to conduct an up-front analysis to properly model the environment — including the fault model — and find an acceptable balance that will effectively boost the scheme's objectives.

– When all versions available in the system periodically or frequently fail, it might cause more harm than good to apply higher levels of redundancy. Due to the nature of the replica ranking, obviously less *reliable* versions would be selected. The optimisation problem would then reduce to a risk assessment problem, where the risk of failure should be minimised. One should therefore find a balance between normalised dissent measurement levels that indicate versions that have performed acceptably, and for which the risk of using them in subsequent voting rounds is judged low, and versions that are not likely to support the redundancy scheme's availability at all.
Imposing a safety margin c_{sm} will obviously lead to higher redundancy levels being used. Therefore, in addition to the $n^{(\mathfrak{C},\ell)}$ best versions that will be selected by the replica selection model, the redundancy configuration will — generally speaking — include an additional amount of "less trustworthy" versions. This in itself can obviously put at risk the proper functioning of the redundancy scheme. One can therefore expect to benefit from applying a safety margin if there are only few poorly performing versions present for use; if the vast majority is medium to highly trustworthy and are observed to perform well, there should be no problem.

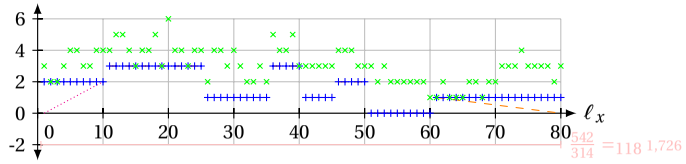
Experiment 7.10

In this experiment, we will evaluate the following dependability strategies, and compare their effectiveness against a classic NVP scheme (scenario A):
– scenarios B and C are examples are examples of Strategy B, Variant 1: in the former no safety margin is applied; in the latter, a margin $c_{sm} = 1$ is maintained;
– scenario D corresponds to Strategy B, Variant 2.

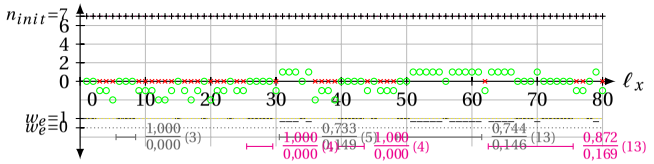
A global value t_{max} is set to 15 discrete time units. $k_1 = 0.85$, $k_2 = 0.75$, and $k_{max} = 0.95$; $r_d = 5$, and $r_u = r_f = 1$. Apart from scenario A (static redundancy configuration with a fixed redundancy $n^{(\mathfrak{C},\ell)} = 7$), the employed redundancy is initialised as $n_{init} = 5$ and can then vary between $[1,7]$. We assume a pool of $|V| = 7$ functionally-equivalent resources (versions).
Each scenario will be simulated when being subject to similar environmental conditions. Failures are injected using the trend-based mechanism; they will materialise as EVF and RVF content failures with a 20, respectively 80% probability, with response values being sampled from a uniformly distributed random variable — *v.* p. 51. In general, version service response times are sampled from an exponentially distributed random variable with $\lambda = 0.4$; network transmissions are assumed to take



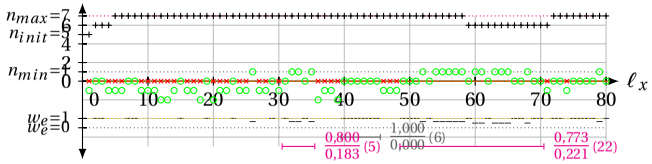
(a) scenario A: failures overview



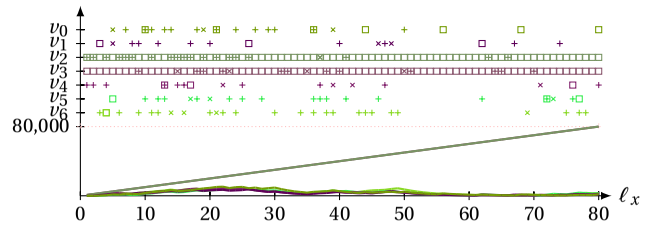
(b) scenario B: failures overview



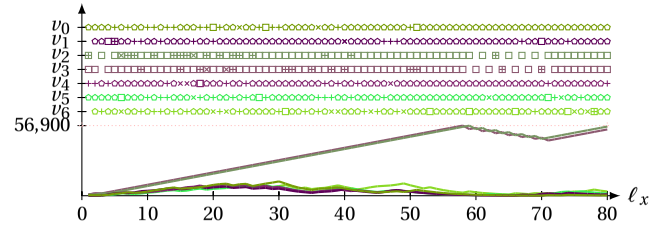
(c) scenario A: resource allocation & under-/overshooting



(d) scenario B: resource allocation & under-/overshooting



(e) scenario A: normalised dissent & observed disturbances



(f) scenario B: normalised dissent & observed disturbances

Figure 7.12: Experiment 7.10: a lower degree of redundancy, using an optimal replica selection, may result in better availability of the redundancy scheme (scenarios A and B).

n_{min}	7	$\ell^{\mathcal{C}}_{failure}(c^{(\mathcal{C},\ell)}_{max} - m^{(\mathcal{C},\ell)} < 0)$			38% (30 of 80)
n_{max}	7	$\sum n^{(\mathcal{C},\ell)} = 560$	$\sum cr(e^{(\mathcal{C},\ell)}) = 322(594)$	$\triangle_{success}$	0% (0 of 0)
n_{init}	7	$\sum(n^{(\mathcal{C},\ell)} - cr(e^{(\mathcal{C},\ell)})) = 238(-34)$		$\nabla_{failure}$	0% (0 of 0)
t_{max}	15,000	$w_e^{(\mathcal{C},\ell)} \in_{50} [0,667, 1,000]$	$\mu = 0,867$	$\sigma = 0,165$	$ci_{90\%}$: (8.28E-1, 9.05E-1)
$\ell^{\mathcal{C}}_{total}$	80	$n^{(\mathcal{C},\ell)} \in [7, 7]$	$\mu = 7,000$	$\sigma = 0,000$	$ci_{90\%}$: (7.00, 7.00)
TTF	2	$XTTF - 2 = 3$	$MTBF =_{29} 1,655$	$MTTR =_{11} 2,636$	

(a) scenario A: failures overview

n_{min}	1	$\ell^{\mathcal{C}}_{failure}(c^{(\mathcal{C},\ell)}_{max} - m^{(\mathcal{C},\ell)} < 0)$			38% (30 of 80)
n_{max}	7	$\sum n^{(\mathcal{C},\ell)} = 542$	$\sum cr(e^{(\mathcal{C},\ell)}) = 314(566)$	$\triangle_{success}$	67% (2 of 3)
n_{init}	5	$\sum(n^{(\mathcal{C},\ell)} - cr(e^{(\mathcal{C},\ell)})) = 228(-24)$		$\nabla_{failure}$	0% (0 of 1)
t_{max}	15,000	$w_e^{(\mathcal{C},\ell)} \in_{50} [0, 500, 1,000]$	$\mu = 0,867$	$\sigma = 0,193$	$ci_{90\%}$: (8.22E-1, 9.12E-1)
$\ell^{\mathcal{C}}_{total}$	80	$n^{(\mathcal{C},\ell)} \in [5, 7]$	$\mu = 6,775$	$\sigma = 0,449$	$ci_{90\%}$: (6.69, 6.86)
r_d	5	$\square_{fill}{}^{d} \in_{80} [0,5]$	$\mu = 2,112$	$\sigma = 2,129$	$ci_{90\%}$: (1.72, 2.50)
		$\square_{reset}{}^{d} \in_{11} [1,5]$	$\mu = 2,364$	$\sigma = 1,748$	$ci_{90\%}$: (1.50, 3.23)
r_u	1	$\square_{fill}{}^{u} \in_{80} [0,1]$	$\mu = 0,375$	$\sigma = 0,487$	$ci_{90\%}$: (2.85E-1, 4.65E-1)
		$\square_{reset}{}^{u} \in_{12} [1,1]$	$\mu = 1,000$	$\sigma = 0,000$	$ci_{90\%}$: (1.00, 1.00)
r_f	1	$\square_{corrupt}$	0% (0 of 80)		
TTF	0	$XTTF - 2 = 3$	$MTBF =_{29} 1,517$	$MTTR =_{12} 2,500$	
c_{sm}	f	$c_{sm} = 0$	n/a	n/a	$ci_{90\%}$: (0.00, 0.00)
		$\ell_{suspect}(0 \leq c^{(\mathcal{C},\ell)}_{max} - m^{(\mathcal{C},\ell)} < c_{sm})$			0% (0 of 80)

(b) scenario B: failures overview

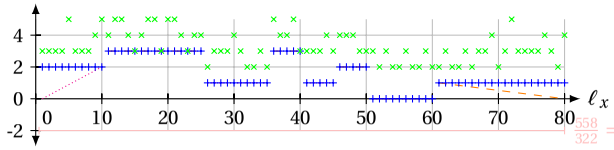
version	v_0	v_1	v_2	v_3	v_4	v_5	v_6
re-integration latency	n/a n/a	n/a n/a	n/a n/a	n/a n/a	n/a n/a	n/a n/a	n/a n/a
exclusion latency	n/a n/a	n/a n/a	n/a n/a	n/a n/a	n/a n/a	n/a n/a	n/a n/a
exclusion failures	n/a n/a	n/a n/a	n/a n/a	n/a n/a	n/a n/a	n/a n/a	n/a n/a
successive usage period	$\mu_1 = 80$ $\sigma = 0$	$\mu_1 = 80$ $\sigma = 0$	$\mu_1 = 80$ $\sigma = 0$	$\mu_1 = 80$ $\sigma = 0$	$\mu_1 = 80$ $\sigma = 0$	$\mu_1 = 80$ $\sigma = 0$	$\mu_1 = 80$ $\sigma = 0$
$\#rounds(\mathcal{C},v) - \#consent(\mathcal{C},v)$ $\hookrightarrow \ell^{\mathcal{C}}_{total}$ $\hookrightarrow \#rounds(\mathcal{C},v)$	20 25 % 25 %	15 19 % 19 %	80 100 % 100 %	80 100 % 100 %	16 20 % 20 %	20 25 % 25 %	26 33 % 33 %
$\#rounds(\mathcal{C},v)$ $\hookrightarrow \ell^{\mathcal{C}}_{total}$	80 100 %	80 100 %	80 100 %	80 100 %	80 100 %	80 100 %	80 100 %
$\#consent(\mathcal{C},v)$ $\hookrightarrow \#rounds(\mathcal{C},v)$	43 54 %	45 56 %	n/a n/a	n/a n/a	45 56 %	45 56 %	42 53 %
normalised dissent	$\mu_{80} = 3,850$ $\sigma = 3,025$	$\mu_{80} = 2,996$ $\sigma = 2,032$	$\mu_{80} = 40,500$ $\sigma = 23,238$	$\mu_{80} = 40,500$ $\sigma = 23,238$	$\mu_{80} = 3,215$ $\sigma = 2,417$	$\mu_{80} = 3,270$ $\sigma = 2,288$	$\mu_{80} = 4,053$ $\sigma = 2,676$
end-to-end response time	$\mu_{80} = 9,302$ $\sigma = 2,278$	$\mu_{80} = 9,250$ $\sigma = 2,185$	$\mu_{80} = 15,000$ $\sigma = 0,000$	$\mu_{80} = 15,000$ $\sigma = 0,000$	$\mu_{80} = 9,272$ $\sigma = 2,131$	$\mu_{80} = 9,392$ $\sigma = 2,214$	$\mu_{80} = 9,351$ $\sigma = 2,233$
TTF $\hookrightarrow$ inv	4 0,250	4 0,250	0 n/a	8 0,125	0 n/a	9 0,111	2 0,500
XTTF-2 $\hookrightarrow$ inv	6 0,167	7 0,143	1 1,000	14 0,071	1 1,000	11 0,091	5 0,200
MTBF $\hookrightarrow$ inv	2,000 0,500	5,273 0,190	2,421 0,413	3,067 0,326	5,077 0,197	2,882 0,347	2,125 0,471
MTTR $\hookrightarrow$ inv	1,333 0,750	1,333 0,750	1,538 0,650	1,455 0,688	1,182 0,846	1,385 0,722	1,471 0,680

(c) scenario A: resource allocation & under-/overshooting

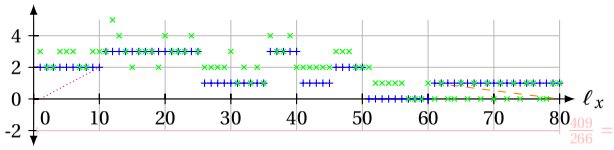
version	v_0	v_1	v_2	v_3	v_4	v_5	v_6
re-integration latency	n/a n/a	n/a n/a	$\mu_8 = 1,000$ $\sigma = 0,000$	$\mu_8 = 1,000$ $\sigma = 0,000$	n/a n/a	n/a n/a	n/a n/a
exclusion latency	n/a n/a	n/a n/a	$\mu_3 = 18,667$ $\sigma = 30,600$	$\mu_3 = 17,333$ $\sigma = 28,290$	n/a n/a	n/a n/a	n/a n/a
exclusion failures	n/a n/a	n/a n/a	$\mu_3 = 5,667$ $\sigma = 8,083$	$\mu_3 = 5,000$ $\sigma = 6,928$	n/a n/a	n/a n/a	n/a n/a
successive usage period			$\mu_8 = 7,750$ $\sigma = 19,092$	$\mu_8 = 7,875$ $\sigma = 19,045$			
$\#rounds(\mathcal{C},v) - \#consent(\mathcal{C},v)$ $\hookrightarrow \ell^{\mathcal{C}}_{total}$ $\hookrightarrow \#rounds(\mathcal{C},v)$	19 24 % 24 %	14 18 % 18 %	72 90 % 100 %	72 90 % 100 %	15 19 % 19 %	21 26 % 26 %	30 38 % 38 %
$\#rounds(\mathcal{C},v)$ $\hookrightarrow \ell^{\mathcal{C}}_{total}$	80 100 %	79 99 %	72 90 %	72 90 %	80 100 %	80 100 %	79 99 %
$\#consent(\mathcal{C},v)$ $\hookrightarrow \#rounds(\mathcal{C},v)$	45 56 %	46 58 %	n/a n/a	n/a n/a	46 57 %	44 55 %	36 46 %
normalised dissent	$\mu_{80} = 3,840$ $\sigma = 3,279$	$\mu_{80} = 3,087$ $\sigma = 2,180$	$\mu_{80} = 34,018$ $\sigma = 17,923$	$\mu_{80} = 34,344$ $\sigma = 17,310$	$\mu_{80} = 3,327$ $\sigma = 2,675$	$\mu_{80} = 3,275$ $\sigma = 2,235$	$\mu_{80} = 4,936$ $\sigma = 2,682$
end-to-end response time	$\mu_{80} = 9,498$ $\sigma = 2,155$	$\mu_{79} = 9,095$ $\sigma = 2,091$	$\mu_{72} = 15,000$ $\sigma = 0,000$	$\mu_{72} = 15,000$ $\sigma = 0,000$	$\mu_{80} = 9,179$ $\sigma = 1,949$	$\mu_{80} = 9,372$ $\sigma = 2,122$	$\mu_{79} = 9,371$ $\sigma = 2,515$
TTF $\hookrightarrow$ inv	4 0,250	4 0,250	0 n/a	8 0,125	0 n/a	9 0,111	2 0,500
XTTF-2 $\hookrightarrow$ inv	6 0,167	7 0,143	5 0,200	14 0,071	1 1,000	11 0,091	5 0,200
MTBF $\hookrightarrow$ inv	2,000 0,500	5,273 0,190	2,875 0,348	3,357 0,298	5,077 0,197	2,882 0,347	2,125 0,471
MTTR $\hookrightarrow$ inv	1,333 0,750	1,333 0,750	1,727 0,579	1,545 0,647	1,182 0,846	1,385 0,722	1,471 0,680

(d) scenario B: resource allocation & under-/overshooting

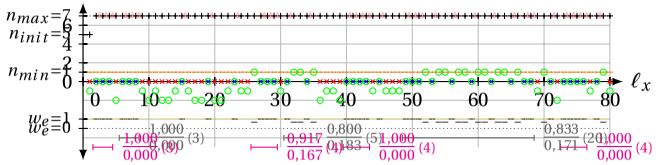
Table 7.10: Experiment 7.10: redundancy configuration/allocation and failure statistics (scenarios A and B).



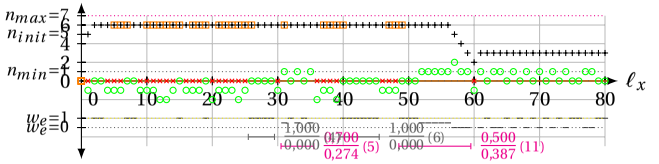
(a) scenario C: failures overview



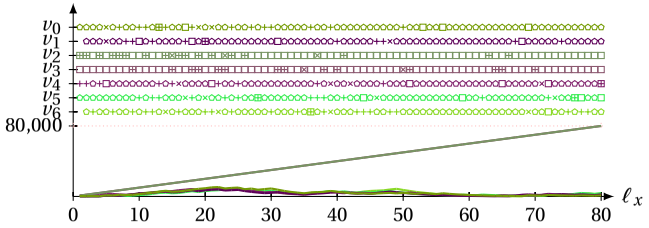
(b) scenario D: failures overview



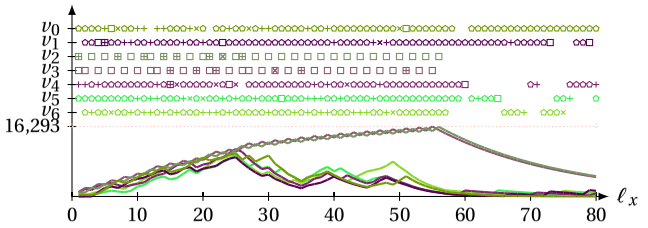
(c) scenario C: resource allocation & under-/overshooting



(d) scenario D: resource allocation & under-/overshooting



(e) scenario C: normalised dissent & observed disturbances



(f) scenario D: normalised dissent & observed disturbances

Figure 7.13: Experiment 7.10: a lower degree of redundancy, using an optimal replica selection, may result in better availability of the redundancy scheme (continued; scenarios C and D).

n_{min}	1	$\ell^{\mathcal{C}}_{failure}\,(c^{(\mathcal{C},\ell)}_{max} - m^{(\mathcal{C},\ell)} < 0)$			36% (29 of 80)
n_{max}	7	$\sum n^{(\mathcal{C},\ell)} = 558$	$\sum cr(e^{(\mathcal{C},\ell)}) = 322(606)$	$\triangle_{success}$	100% (1 of 1)
n_{init}	5	$\sum(n^{(\mathcal{C},\ell)} - cr(e^{(\mathcal{C},\ell)})) = 236(-48)$		$\triangledown_{failure}$	0% (0 of 0)
t_{max}	15,000	$w_e^{(\mathcal{C},\ell)} \in_{51} [0,667, 1,000]$	$\mu = 0,895$	$\sigma = 0,156$	$ci_{90\%}$: (8.59E-1, 9.31E-1)
$\ell^{\mathcal{C}}_{total}$	80	$n^{(\mathcal{C},\ell)} \in [5, 7]$	$\mu = 6,975$	$\sigma = 0,224$	$ci_{90\%}$: (6.93, 7.02)
r_d	5	$\square_{fill}{}^{d} \in_{80} [0, 2]$	$\mu = 0,262$	$\sigma = 0,568$	$ci_{90\%}$: (1.58E-1, 3.67E-1)
		$\square_{reset}{}^{d} \in_{11} [1, 2]$	$\mu = 1,455$	$\sigma = 0,522$	$ci_{90\%}$: (1.20, 1.71)
r_u	1	$\square_{fill}{}^{u} \in_{80} [0, 1]$	$\mu = 0,800$	$\sigma = 0,403$	$ci_{90\%}$: (7.26E-1, 8.74E-1)
		$\square_{reset}{}^{u} \in_{11} [1, 1]$	$\mu = 1,000$	$\sigma = 0,000$	$ci_{90\%}$: (1.00, 1.00)
r_f	1	$\square_{corrupt}$	0% (0 of 80)		
TTF	0	$XTTF - 2 = 4$	$MTBF =_{28} 1,821$	$MTTR =_{13} 2,154$	
c_{sm}	f	$c_{sm} = 1$	n/a	n/a	$ci_{90\%}$: (1.00, 1.00)
		$\ell_{suspect}\,(0 \le c^{(\mathcal{C},\ell)}_{max} - m^{(\mathcal{C},\ell)} < c_{sm})$			44% (35 of 80)

(a) scenario C: failures overview

n_{min}	1	$\ell^{\mathcal{C}}_{failure}\,(c^{(\mathcal{C},\ell)}_{max} - m^{(\mathcal{C},\ell)} < 0)$			35% (28 of 80)
n_{max}	7	$\sum n^{(\mathcal{C},\ell)} = 409$	$\sum cr(e^{(\mathcal{C},\ell)}) = 266(378)$	$\triangle_{success}$	100% (2 of 2)
n_{init}	5	$\sum(n^{(\mathcal{C},\ell)} - cr(e^{(\mathcal{C},\ell)})) = 143(31)$		$\triangledown_{failure}$	25% (1 of 4)
t_{max}	15,000	$w_e^{(\mathcal{C},\ell)} \in_{52} [0, 000, 1,000]$	$\mu = 0,673$	$\sigma = 0,430$	$ci_{90\%}$: (5.75E-1, 7.71E-1)
$\ell^{\mathcal{C}}_{total}$	80	$n^{(\mathcal{C},\ell)} \in [2, 6]$	$\mu = 5,112$	$\sigma = 1,369$	$ci_{90\%}$: (4.86, 5.36)
r_d	5	$\square_{fill}{}^{d} \in_{80} [0, 5]$	$\mu = 2,350$	$\sigma = 2,165$	$ci_{90\%}$: (1.95, 2.75)
		$\square_{reset}{}^{d} \in_{8} [1, 5]$	$\mu = 3,125$	$\sigma = 1,808$	$ci_{90\%}$: (2.07, 4.18)
r_u	1	$\square_{fill}{}^{u} \in_{80} [0, 1]$	$\mu = 0,350$	$\sigma = 0,480$	$ci_{90\%}$: (2.62E-1, 4.38E-1)
		$\square_{reset}{}^{u} \in_{9} [1, 1]$	$\mu = 1,000$	$\sigma = 0,000$	$ci_{90\%}$: (1.00, 1.00)
r_f	1	$\square_{corrupt}$	0% (0 of 80)		
TTF	0	$XTTF - 2 = 3$	$MTBF =_{27} 1,185$	$MTTR =_{9} 3,111$	
c_{sm}	f	$c_{sm} = 0$	n/a	n/a	$ci_{90\%}$: (0.00, 0.00)
		$\ell_{suspect}\,(0 \le c^{(\mathcal{C},\ell)}_{max} - m^{(\mathcal{C},\ell)} < c_{sm})$			0% (0 of 80)

(b) scenario D: failures overview

version	v_0	v_1	v_2	v_3	v_4	v_5	v_6
re-integration latency	n/a n/a	n/a n/a	n/a n/a	n/a n/a	n/a n/a	n/a n/a	n/a n/a
exclusion latency	n/a n/a	n/a n/a	n/a n/a	n/a n/a	n/a n/a	n/a n/a	n/a n/a
exclusion failures	n/a n/a	n/a n/a	n/a n/a	n/a n/a	n/a n/a	n/a n/a	n/a n/a
successive usage period							
$\#rounds(\mathcal{C},v) - \#consent(\mathcal{C},v)$ $\hookrightarrow \ell^{\mathcal{C}}_{total}$ $\hookrightarrow \#rounds(\mathcal{C},v)$	20 25 % 25 %	16 20 % 20 %	80 100 % 100 %	80 100 % 100 %	19 24 % 24 %	22 28 % 28 %	26 33 % 33 %
$\#rounds(\mathcal{C},v)$ $\hookrightarrow \ell^{\mathcal{C}}_{total}$	80 100 %	79 99 %	80 100 %	80 100 %	80 100 %	80 100 %	79 99 %
$\#consent(\mathcal{C},v)$ $\hookrightarrow \#rounds(\mathcal{C},v)$	44 55 %	45 57 %	n/a n/a	n/a n/a	44 55 %	44 55 %	43 54 %
normalised dissent	$\mu_{80} = 3,816$ $\sigma = 2,918$	$\mu_{80} = 3,106$ $\sigma = 2,101$	$\mu_{80} = 40,500$ $\sigma = 23,238$	$\mu_{80} = 40,500$ $\sigma = 23,238$	$\mu_{80} = 3,559$ $\sigma = 2,375$	$\mu_{80} = 3,420$ $\sigma = 2,206$	$\mu_{80} = 3,919$ $\sigma = 2,646$
end-to-end response time	$\mu_{80} = 9,205$ $\sigma = 2,250$	$\mu_{79} = 9,581$ $\sigma = 2,247$	$\mu_{80} = 15,000$ $\sigma = 0,000$	$\mu_{80} = 15,000$ $\sigma = 0,000$	$\mu_{80} = 9,409$ $\sigma = 2,155$	$\mu_{80} = 9,200$ $\sigma = 2,337$	$\mu_{79} = 9,371$ $\sigma = 2,077$
TTF $\hookrightarrow$ inv	4 0,250	4 0,250	0 n/a	8 0,125	0 n/a	9 0,111	2 0,500
XTTF-2 $\hookrightarrow$ inv	6 0,167	7 0,143	1 1,000	14 0,071	1 1,000	11 0,091	5 0,200
MTBF $\hookrightarrow$ inv	2,000 0,500	5,273 0,190	2,421 0,413	3,067 0,326	5,077 0,197	2,882 0,347	2,125 0,471
MTTR $\hookrightarrow$ inv	1,333 0,750	1,333 0,750	1,538 0,650	1,455 0,688	1,182 0,846	1,385 0,722	1,471 0,680

(c) scenario C: resource allocation & under-/overshooting

version	v_0	v_1	v_2	v_3	v_4	v_5	v_6
re-integration latency	$\mu_3 = 1,333$ $\sigma = 0,577$	$\mu_1 = 3,000$ -	$\mu_{26} = 1,000$ $\sigma = 0,000$	$\mu_{25} = 1,000$ $\sigma = 0,000$	$\mu_3 = 4,667$ $\sigma = 4,041$	$\mu_3 = 4,000$ $\sigma = 3,606$	$\mu_2 = 5,000$ $\sigma = 4,243$
exclusion latency	$\mu_3 = 16,667$ $\sigma = 15,885$	$\mu_1 = 69,000$ -	$\mu_8 = 1,125$ $\sigma = 0,354$	$\mu_6 = 1,000$ $\sigma = 0,000$	$\mu_3 = 16,667$ $\sigma = 13,577$	$\mu_3 = 18,333$ $\sigma = 27,465$	$\mu_3 = 19,000$ $\sigma = 31,177$
exclusion failures	$\mu_3 = 5,000$ $\sigma = 1,732$	$\mu_1 = 11,000$ -	$\mu_8 = 1,000$ $\sigma = 0,000$	$\mu_6 = 1,000$ $\sigma = 0,000$	$\mu_3 = 4,333$ $\sigma = 3,512$	$\mu_3 = 5,333$ $\sigma = 7,506$	$\mu_3 = 7,667$ $\sigma = 11,547$
successive usage period	$\mu_4 = 14,000$ $\sigma = 16,021$	$\mu_4 = 18,750$ $\sigma = 35,528$	$\mu_{50} = 0,600$ $\sigma = 0,606$	$\mu_{50} = 0,600$ $\sigma = 0,639$	$\mu_{14} = 4,357$ $\sigma = 10,810$	$\mu_{12} = 5,583$ $\sigma = 16,898$	$\mu_{15} = 4,267$ $\sigma = 14,380$
$\#rounds(\mathcal{C},v) - \#consent(\mathcal{C},v)$ $\hookrightarrow \ell^{\mathcal{C}}_{total}$ $\hookrightarrow \#rounds(\mathcal{C},v)$	17 21 % 22 %	15 19 % 20 %	30 38 % 100 %	30 38 % 100 %	16 20 % 24 %	18 23 % 26 %	23 29 % 36 %
$\#rounds(\mathcal{C},v)$ $\hookrightarrow \ell^{\mathcal{C}}_{total}$	76 95 %	75 94 %	30 38 %	30 38 %	66 83 %	68 85 %	64 80 %
$\#consent(\mathcal{C},v)$ $\hookrightarrow \#rounds(\mathcal{C},v)$	45 59 %	43 57 %	n/a n/a	n/a n/a	33 50 %	35 51 %	30 47 %
normalised dissent	$\mu_{80} = 3,564$ $\sigma = 3,141$	$\mu_{80} = 2,902$ $\sigma = 2,435$	$\mu_{80} = 9,962$ $\sigma = 4,247$	$\mu_{80} = 10,031$ $\sigma = 4,210$	$\mu_{80} = 3,491$ $\sigma = 2,806$	$\mu_{80} = 3,258$ $\sigma = 2,611$	$\mu_{80} = 3,635$ $\sigma = 2,677$
end-to-end response time	$\mu_{76} = 9,120$ $\sigma = 1,908$	$\mu_{75} = 9,691$ $\sigma = 2,307$	$\mu_{30} = 15,000$ $\sigma = 0,000$	$\mu_{30} = 15,000$ $\sigma = 0,000$	$\mu_{66} = 9,330$ $\sigma = 2,283$	$\mu_{68} = 8,977$ $\sigma = 2,026$	$\mu_{64} = 8,910$ $\sigma = 1,560$
TTF $\hookrightarrow$ inv	4 0,250	4 0,250	0 n/a	14 0,071	0 n/a	9 0,111	2 0,500
XTTF-2 $\hookrightarrow$ inv	6 0,167	7 0,143	6 0,167	18 0,056	1 1,000	11 0,091	5 0,200
MTBF $\hookrightarrow$ inv	2,214 0,452	5,200 0,192	2,571 0,389	6,200 0,161	5,077 0,197	3,400 0,294	2,273 0,440
MTTR $\hookrightarrow$ inv	1,600 0,625	1,375 0,727	2,500 0,400	2,000 0,500	1,636 0,611	1,583 0,632	1,600 0,625

(d) scenario D: resource allocation & under-/overshooting

Table 7.11: Experiment 7.10: redundancy configuration/allocation and failure statistics (continued; scenarios C and D).

a constant 3.5 time units (one-way). It is however assumed that a specific subset of versions — say, v_2 and v_3 — very often suffer from failure. Such situation could occur, for instance, if these would be deployed in a specific network, and the link to connect to that network is highly congested. Subsequent invocations of either version are therefore expected to suffer from LRF performance failures.

The following table gives a brief overview of the key findings at the end of simulation when each of these scenarios is subject to the same environment:

scenario	strategy	variant	c_{sm}	$\ell^{\mathcal{C}}_{failure}$	MTBF	MTTR	$\sum n^{(\mathcal{C},\ell)}$	saving
A	—	n/a	n/a	38%	1.655	2.636	560	
B	B	1	0	38%	1.517	2.500	542	-0.03%
C	B	1	1	36%	1.821	2.154	558	-0.00%
D	B	2	n/a	35%	1.185	3.111	409	-26.96%

The key messages to convey here are that:

– Maintaining a redundancy configuration using a lower degree of redundancy, but using an optimal subset of the available redundancy tends to yield better results in terms of dependability. Furthermore, compared to traditional NVP and even A-NVP, it is likely to result in a significantly reduction in resource expenditure. This statement is true especially when there is a marked difference in the performance of the available redundant resources (versions) in the system. The drawback of this approach is, however, that the overall load on the allocated versions will increase, and won't be offloaded to other, stand-by versions (which are observed to perform worse). This can be observed when comparing scenario D with scenarios A–C: given an operational life of the redundancy scheme of 80 voting rounds, we can record a spectacular reduction of up to 27% compared to traditional NVP when refraining from upscaling the redundancy if the stand-by versions are judged insufficiently trustworthy. In this particular experiment, one can even see that — because the use of unreliable versions is reduced/avoided — 3% less scheme failures are recorded.
– In this experiment, versions v_2 and v_3 perform poorly. They are intensively allocated and used in scenarios A–C (as can be seen in Fig. 7.12e, 7.12f, and 7.13e. When using versions that are untrustworthy, or that underperform with respect to the average performance recorded for the available system resources, the dependability of the redundancy scheme may suffer, both in terms of availability and reliability. Indeed, in scenario D, 3% less voting rounds are observed during which the redundancy scheme failed to adjudicate an outcome. This is because the use of the worst performing versions in the system is avoided, as can be observed from Fig. 7.13f (and by comparing Tables 7.10a, 7.10b and 7.11a with 7.11b.
– Strategy B, Variant 2 (scenario D) is successful in reducing the inclusion of underperforming resources in the redundancy configuration. Still, because normalised dissent measurements for idling versions gradually

decrease, previously penalised/excluded versions will eventually be re-introduced in the redundancy configuration. Whereas versions v_2 and v_3 are engaged in all 80 voting rounds in scenarios A and C, and in 72 of the 80 voting rounds in scenario B, a mere 30 voting rounds is noted for scenario D. Furthermore, because the normalised dissent measurements do not inflate as much, the algorithm is capable of safely downscaling when a lesser degree of disturbances is detected.

The above clearly corroborates the statement that the use of replicas of poor reliability can result in a system tolerant of faults but with poor reliability [3, Sect. 4.3.3]. A higher degree of redundancy therefore does not necessarily result in higher level of fault tolerance.

To conclude, a few side notes though:

- For visualisation and readability purposes, we have chosen to use a trend-based failure injection approach. However, such approach will generally inject the same number of disturbances for different redundancy levels. If we would know the exact details of the fault model, in general, less failures would be injected for lower redundancy levels. As a consequence, the probability of voting round failure would further diminish. Compare the number of injected disturbances in Fig. 7.12a, 7.12b, 7.13a and 7.13b (in particular for $\ell \geq 60$).
- The number of injected disturbances is denoted by ×. For many voting rounds, one can observe a number that exceeds the defined trend. This is because performance failures are not accounted for in the trend definition. In this particular experiment, only LRF performance failures materialise, since no crash failures are injected.
- The cumulative amount of allocated redundancy $\sum n^{(\mathcal{C},\ell)}$ recorded for scenario C is lower than what the algorithm would have considered as optimal redundancy levels. This can be observed from Fig. 7.13c, where the upper limit $|V| = 7$ is reached, even though the algorithm would instruct the system to scale the degree of redundancy up (conditions denoted as △). Had more resources been available, the cumulative degree of redundancy would be even higher. And — depending on the dependability of the additional resources — the scheme itself would exhibit a better or worse availability (number of voting round failures).
- We would like to remind the reader to the fact that the pentagonal symbols that indicate normal, fault-free behaviour, are only displayed in those scenarios in which a dynamic redundancy configuration is applied. Its purpose is mainly to visualise which versions are engaged in the configuration, and which remain idling.
- We would like to conclude this experiment by pointing out that the discerning reader may observe a different number of injected disturbances for a similar redundancy configuration, but in different scenarios. This may occur because of differences in sampling from random number generators, where the sequence of sampling may not fully be identical. As an example, additional LRF failures may emerge because of a random variate sampled from the exponentially distibuted

service time distribution, where the variate — added to the network transmission round-trip time — would exceed the t_{max} timeout. For instance voting round $\ell = 20$ has an identical redundancy configuration in scenarios A–C, yet the number of disturbances varies.

– In the above experiments, for readability purposes, requests were injected to hit the fault-tolerant composite one after another, one at a time. Athough this does not reflect reality, in which requests would hit with rather unpredictable inter-arrival rates and patterns, it does guarantee that there is no window corruption at the time the algorithm will adjust the redundancy configuration — *v.* p. 101. Because of this, decisions with respect to changes in the degree of redundancy to use, or which selection of versions to use, will be based on the latest, most complete and therefore most accurate data.
In contemporary distributed computing systems, the arrival process with which requests typically *arrive* at the processing facility is typically characterised by some type of distribution. Few aspects will determine the extent that contextual information is partially missing in order to determine a well-considered change in the redundancy configuration:

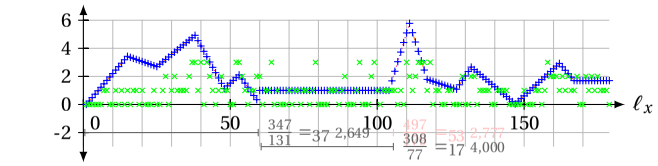
  + The end-to-end response times may vary, even for specific version — *cf.* Sect. 6.1.1.
  + With the exception of failures in the content domain, all other sorts of failures will be detected very late — after the t_{max} timeout has lapsed. Imagine this is the case for a voting round $(\mathcal{C}, \ell)$, and that no failures in the timing domain would occur throughout the next voting round $(\mathcal{C}, \ell + 1)$. Then, assuming the end-to-end response times do not substantially vary, it is quite likely that round $(\mathcal{C}, \ell + 1)$ will have completed before $(\mathcal{C}, \ell)$ has. Then, the measurements deduced for round $(\mathcal{C}, \ell)$ may not be available when initialising subsequent rounds $(\mathcal{C}, \ell + \mathrm{x})$, $x \geq 2$[8].
  + The inter-arrival rates at which requests hit the redundancy scheme: intense use of the scheme will result in more pending requests that are concurrently being processed[9], which will magnify the two phenomena listed here above.

Experiment 7.11

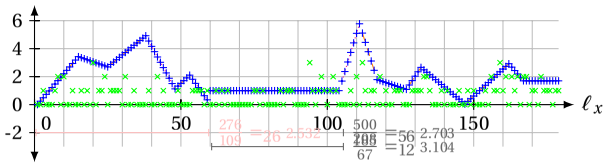
In this experiment, a variation of the trend-based failure injection used in Fig. 7.8 is applied, and the following scenarios are evaluated:
– In scenario A, voting rounds are scheduled one after another: there is no concurrent processing of subordinate requests pertaining to different voting

[8] Depending on the time that request hits (arrives at) the redundancy scheme, response times, *etc.* Note that larger t_{max} values may result in a worse degree of window corruption.

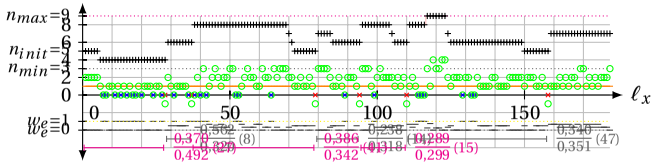
[9] Such behaviour would be observed especially when a queuing model $A/S/c/k/n/D$ would be applied with a limited processing capacity defined (*i.e.* $c \neq \infty$). Furthermore, when processing facilities — versions, that are — are subject to increased load, one may reasonably expect to see a subsequent increase of (inbound) waiting and/or request processing times. This has a direct impact on the recorded end-to-end response times, as per Fig. 6.1, p. 91.



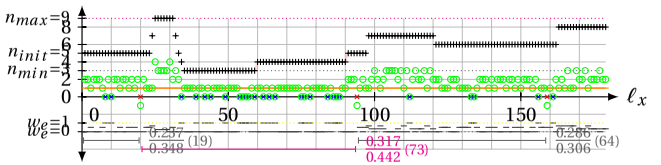
(a) scenario A: failures overview



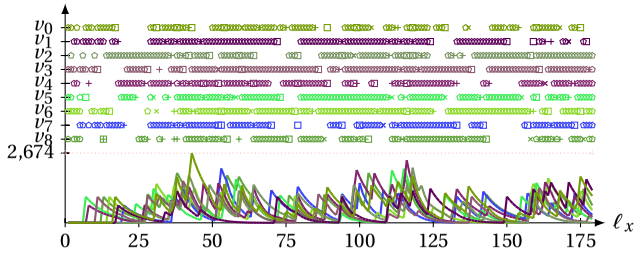
(b) scenario B: failures overview



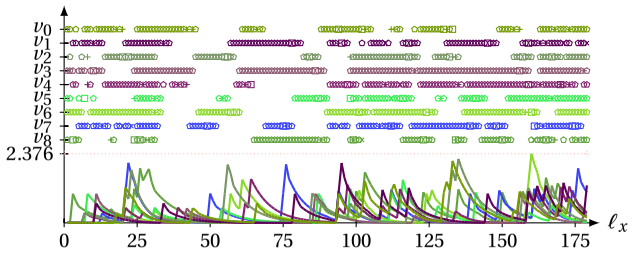
(c) scenario A: resource allocation & under-/overshooting



(d) scenario B: resource allocation & under-/overshooting



(e) scenario A: normalised dissent & observed disturbances



(f) scenario B: normalised dissent & observed disturbances

Figure 7.14: Experiment 7.11: the effectiveness of A-NVP strategies is determined by the system-environment fit (scenarios A and B).

n_{min}	3	$\ell^{\mathcal{C}}_{failure}\,(c^{(\mathcal{C},\ell)}_{max} - m^{(\mathcal{C},\ell)} < 0)$			3% (6 of 179)
n_{max}	9	$\sum n^{(\mathcal{C},\ell)} = 1152$	$\sum cr(e^{(\mathcal{C},\ell)}) = 387(513)$	$\triangle_{success}$	0% (0 of 0)
n_{init}	5	$\sum(n^{(\mathcal{C},\ell)} - cr(e^{(\mathcal{C},\ell)})) = 765(639)$		$\nabla_{failure}$	0% (0 of 0)
t_{max}	20,000	$w^{(\mathcal{C},\ell)}_e \in_{173} [0{,}000, 1{,}000]$	$\mu = 0{,}364$	$\sigma = 0{,}356$	$ci_{90\%}$: (3.20E-1, 4.09E-1)
$\ell^{\mathcal{C}}_{total}$	179	$n^{(\mathcal{C},\ell)} \in [4,9]$	$\mu = 6{,}436$	$\sigma = 1{,}430$	$ci_{90\%}$: (6.26, 6.61)
r_d	5	$\square_{fill}{}^{d} \in_{179} [0,5]$	$\mu = 3{,}045$	$\sigma = 2{,}025$	$ci_{90\%}$: (2.80, 3.29)
		$\square_{reset}{}^{d} \in_{23} [1,5]$	$\mu = 3{,}130$	$\sigma = 1{,}687$	$ci_{90\%}$: (2.55, 3.71)
r_u	3	$\square_{fill}{}^{u} \in_{179} [0,3]$	$\mu = 0{,}240$	$\sigma = 0{,}574$	$ci_{90\%}$: (1.70E-1, 3.11E-1)
		$\square_{reset}{}^{u} \in_{23} [1,3]$	$\mu = 1{,}391$	$\sigma = 0{,}656$	$ci_{90\%}$: (1.17, 1.62)
r_f	1	$\square_{corrupt}$	0% (0 of 179)		
TTF	27	$XTTF\text{-}2 = 36$	$MTBF =_5 25{,}000$	$MTTR =_6 1{,}000$	
c_{sm}	f	$c_{sm} = 1$	n/a	n/a	$ci_{90\%}$: (1.00, 1.00)
		$\ell_{suspect}\,(0 \leq c^{(\mathcal{C},\ell)}_{max} - m^{(\mathcal{C},\ell)} < c_{sm})$			15% (26 of 179)

(a) scenario A: failures overview

n_{min}	3	$\ell^{\mathcal{C}}_{failure}\,(c^{(\mathcal{C},\ell)}_{max} - m^{(\mathcal{C},\ell)} < 0)$			2% (3 of 179)
n_{max}	9	$\sum n^{(\mathcal{C},\ell)} = 984$	$\sum cr(e^{(\mathcal{C},\ell)}) = 361(405)$	$\triangle_{success}$	100% (7 of 7)
n_{init}	5	$\sum(n^{(\mathcal{C},\ell)} - cr(e^{(\mathcal{C},\ell)})) = 623(579)$		$\nabla_{failure}$	0% (0 of 14)
t_{max}	20.000	$w^{(\mathcal{C},\ell)}_e \in_{176} [0.000, 1.000]$	$\mu = 0.304$	$\sigma = 0.370$	$ci_{90\%}$: (2.58E-1, 3.50E-1)
$\ell^{\mathcal{C}}_{total}$	179	$n^{(\mathcal{C},\ell)} \in [3,9]$	$\mu = 5.497$	$\sigma = 1.667$	$ci_{90\%}$: (5.29, 5.70)
r_d	5	$glssymb: red-window-fill^{d} \in_{179} [0,5]$	$\mu = 3.112$	$\sigma = 2.002$	$ci_{90\%}$: (2.87, 3.36)
		$\square_{reset}{}^{d} \in_{22} [1,5]$	$\mu = 3.364$	$\sigma = 1.706$	$ci_{90\%}$: (2.77, 3.96)
r_u	3	$\square_{fill}{}^{u} \in_{179} [0,3]$	$\mu = 0.235$	$\sigma = 0.600$	$ci_{90\%}$: (1.61E-1, 3.08E-1)
		$\square_{reset}{}^{u} \in_{22} [1,3]$	$\mu = 1.273$	$\sigma = 0.631$	$ci_{90\%}$: (1.05, 1.49)
r_f	1	$\square_{corrupt}$	9% (17 of 179)		
TTF	19	$XTTF\text{-}2 = 93$	$MTBF =_2 68.500$	$MTTR =_3 1.000$	
c_{sm}	f	$c_{sm} = 1$	n/a	n/a	$ci_{90\%}$: (1.00, 1.00)
		$\ell_{suspect}\,(0 \leq c^{(\mathcal{C},\ell)}_{max} - m^{(\mathcal{C},\ell)} < c_{sm})$			15% (27 of 179)

(b) scenario B: failures overview

version	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
re-integration latency	$\mu_{18} = 3{,}389$ $\sigma = 2{,}747$	$\mu_{11} = 4{,}182$ $\sigma = 3{,}430$	$\mu_{14} = 3{,}214$ $\sigma = 2{,}940$	$\mu_{30} = 3{,}000$ $\sigma = 2{,}449$	$\mu_{12} = 4{,}417$ $\sigma = 3{,}147$	$\mu_{15} = 3{,}067$ $\sigma = 2{,}604$	$\mu_{13} = 2{,}923$ $\sigma = 3{,}068$	$\mu_{13} = 4{,}385$ $\sigma = 4{,}032$	$\mu_{15} = 4{,}533$ $\sigma = 3{,}777$
exclusion latency	$\mu_{10} = 2{,}400$ $\sigma = 3{,}406$	$\mu_6 = 12{,}333$ $\sigma = 17{,}739$	$\mu_7 = 1{,}571$ $\sigma = 0{,}976$	$\mu_4 = 15{,}500$ $\sigma = 17{,}234$	$\mu_9 = 5{,}778$ $\sigma = 7{,}886$	$\mu_{10} = 2{,}600$ $\sigma = 3{,}534$	$\mu_6 = 7{,}667$ $\sigma = 9{,}953$	$\mu_6 = 9{,}333$ $\sigma = 8{,}238$	$\mu_{12} = 5{,}000$ $\sigma = 8{,}334$
exclusion failures	$\mu_{10} = 1{,}200$ $\sigma = 0{,}422$	$\mu_6 = 1{,}833$ $\sigma = 1{,}329$	$\mu_7 = 1{,}000$ $\sigma = 0{,}000$	$\mu_4 = 1{,}250$ $\sigma = 0{,}500$	$\mu_9 = 1{,}556$ $\sigma = 1{,}014$	$\mu_{10} = 1{,}500$ $\sigma = 0{,}972$	$\mu_6 = 1{,}833$ $\sigma = 0{,}983$	$\mu_6 = 1{,}167$ $\sigma = 0{,}408$	$\mu_{12} = 1{,}583$ $\sigma = 0{,}793$
successive usage period	$\mu_{65} = 1{,}754$ $\sigma = 3{,}820$	$\mu_{48} = 2{,}688$ $\sigma = 9{,}226$	$\mu_{45} = 2{,}511$ $\sigma = 5{,}417$	$\mu_{30} = 4{,}433$ $\sigma = 11{,}355$	$\mu_{53} = 2{,}075$ $\sigma = 5{,}049$	$\mu_{46} = 2{,}761$ $\sigma = 6{,}385$	$\mu_{38} = 3{,}632$ $\sigma = 8{,}930$	$\mu_{57} = 2{,}000$ $\sigma = 5{,}335$	$\mu_{68} = 1{,}603$ $\sigma = 4{,}936$
$\#rounds(\mathcal{C},v) - \#consent(\mathcal{C},v)$	23	16	16	13	24	18	17	17	23
$\hookrightarrow \ell^{\mathcal{C}}_{total}$	13 %	9 %	9 %	7 %	13 %	10 %	9 %	9 %	13 %
$\hookrightarrow \#rounds(\mathcal{C},v)$	20 %	12 %	12 %	9 %	19 %	14 %	12 %	14 %	21 %
$\#rounds(\mathcal{C},v)$	114	129	133	149	124	133	141	118	111
$\hookrightarrow \ell^{\mathcal{C}}_{total}$	64 %	72 %	74 %	83 %	69 %	74 %	79 %	66 %	62 %
$\#consent(\mathcal{C},v)$	88	112	116	132	99	114	121	101	87
$\hookrightarrow \#rounds(\mathcal{C},v)$	77 %	87 %	87 %	89 %	80 %	86 %	86 %	86 %	78 %
normalised dissent	$\mu_{179} = 0{,}670$ $\sigma = 0{,}506$	$\mu_{179} = 0{,}403$ $\sigma = 0{,}424$	$\mu_{179} = 0{,}458$ $\sigma = 0{,}451$	$\mu_{179} = 0{,}364$ $\sigma = 0{,}355$	$\mu_{179} = 0{,}397$ $\sigma = 0{,}480$	$\mu_{179} = 0{,}472$ $\sigma = 0{,}440$	$\mu_{179} = 0{,}435$ $\sigma = 0{,}422$	$\mu_{179} = 0{,}504$ $\sigma = 0{,}413$	$\mu_{179} = 0{,}616$ $\sigma = 0{,}476$
end-to-end response time	$\mu_{114} = 9{,}648$ $\sigma = 5{,}124$	$\mu_{139} = 8{,}964$ $\sigma = 4{,}734$	$\mu_{131} = 9{,}072$ $\sigma = 5{,}065$	$\mu_{149} = 9{,}197$ $\sigma = 4{,}801$	$\mu_{124} = 9{,}152$ $\sigma = 5{,}204$	$\mu_{133} = 8{,}293$ $\sigma = 4{,}327$	$\mu_{141} = 8{,}308$ $\sigma = 4{,}580$	$\mu_{118} = 9{,}138$ $\sigma = 5{,}428$	$\mu_{111} = 9{,}393$ $\sigma = 5{,}211$
TTF	41	17	35	31	7	23	30	19	12
$\hookrightarrow$ inv	0,024	0,059	0,029	0,032	0,143	0,043	0,033	0,053	0,083
XTTF-2	59	33	39	51	26	29	36	36	27
$\hookrightarrow$ inv	0,017	0,030	0,026	0,020	0,038	0,034	0,028	0,028	0,037
MTBF	11,000	14,300	15,500	28,200	10,000	9,500	11,800	23,167	8,056
$\hookrightarrow$ inv	0,091	0,070	0,065	0,035	0,100	0,105	0,085	0,043	0,124
MTTR	3,273	2,900	2,000	1,667	3,923	2,923	3,125	2,429	3,250
$\hookrightarrow$ inv	0,306	0,345	0,500	0,600	0,255	0,342	0,320	0,412	0,308

(c) scenario A: resource allocation & under-/overshooting

version	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
re-integration latency	$\mu_{11} = 6.727$ $\sigma = 6.513$	$\mu_{30} = 6.600$ $\sigma = 5.582$	$\mu_{30} = 8.100$ $\sigma = 7.680$	$\mu_6 = 6.333$ $\sigma = 5.645$	$\mu_{10} = 6.000$ $\sigma = 8.300$	$\mu_{11} = 7.909$ $\sigma = 7.231$	$\mu_6 = 9.833$ $\sigma = 10.265$	$\mu_{10} = 5.900$ $\sigma = 5.384$	$\mu_{11} = 8.364$ $\sigma = 8.903$
exclusion latency	$\mu_7 = 3.143$ $\sigma = 0.690$	$\mu_6 = 4.667$ $\sigma = 3.204$	$\mu_3 = 9.667$ $\sigma = 12.503$	$\mu_2 = 29.500$ $\sigma = 23.335$	$\mu_8 = 8.000$ $\sigma = 10.043$	$\mu_5 = 5.800$ $\sigma = 3.899$	$\mu_3 = 10.000$ $\sigma = 10.440$	$\mu_6 = 5.667$ $\sigma = 4.082$	$\mu_6 = 3.833$ $\sigma = 2.229$
exclusion failures	$\mu_7 = 1.571$ $\sigma = 0.535$	$\mu_6 = 1.333$ $\sigma = 0.816$	$\mu_3 = 2.333$ $\sigma = 1.155$	$\mu_2 = 1.500$ $\sigma = 0.707$	$\mu_8 = 2.375$ $\sigma = 1.506$	$\mu_5 = 2.200$ $\sigma = 1.643$	$\mu_3 = 2.000$ $\sigma = 1.000$	$\mu_6 = 1.333$ $\sigma = 0.516$	$\mu_6 = 1.667$ $\sigma = 0.816$
successive usage period	$\mu_{74} = 1.189$ $\sigma = 3.817$	$\mu_{66} = 1.621$ $\sigma = 4.841$	$\mu_{81} = 1.099$ $\sigma = 3.923$	$\mu_{38} = 3.263$ $\sigma = 11.418$	$\mu_{60} = 1.917$ $\sigma = 6.171$	$\mu_{67} = 0.736$ $\sigma = 2.674$	$\mu_{89} = 1.847$ $\sigma = 5.974$	$\mu_{61} = 1.869$ $\sigma = 4.703$	$\mu_{92} = 0.880$ $\sigma = 3.393$
$\#rounds(\mathcal{C},v) - \#consent(\mathcal{C},v)$	17	13	13	5	14	11	9	15	16
$\hookrightarrow \ell^{\mathcal{C}}_{total}$	9 %	7 %	7 %	3 %	8 %	6 %	5 %	8 %	9 %
$\hookrightarrow \#rounds(\mathcal{C},v)$	16 %	12 %	13 %	4 %	12 %	12 %	8 %	13 %	18 %
$\#rounds(\mathcal{C},v)$	105	111	97	141	117	92	120	114	87
$\hookrightarrow \ell^{\mathcal{C}}_{total}$	59 %	62 %	54 %	79 %	65 %	51 %	67 %	64 %	49 %
$\#consent(\mathcal{C},v)$	89	98	83	134	102	80	109	99	71
$\hookrightarrow \#rounds(\mathcal{C},v)$	85 %	88 %	86 %	95 %	87 %	87 %	91 %	87 %	82 %
normalised dissent	$\mu_{179} = 0.354$ $\sigma = 0.401$	$\mu_{176} = 0.320$ $\sigma = 0.382$	$\mu_{179} = 0.411$ $\sigma = 0.466$	$\mu_{179} = 0.135$ $\sigma = 0.229$	$\mu_{178} = 0.295$ $\sigma = 0.332$	$\mu_{179} = 0.270$ $\sigma = 0.275$	$\mu_{179} = 0.271$ $\sigma = 0.402$	$\mu_{178} = 0.377$ $\sigma = 0.431$	$\mu_{179} = 0.434$ $\sigma = 0.429$
end-to-end response time	$\mu_{105} = 9.197$ $\sigma = 4.841$	$\mu_{111} = 8.857$ $\sigma = 4.586$	$\mu_{92} = 10.013$ $\sigma = 5.511$	$\mu_{141} = 9.113$ $\sigma = 4.808$	$\mu_{117} = 9.091$ $\sigma = 4.741$	$\mu_{92} = 8.780$ $\sigma = 5.322$	$\mu_{120} = 9.022$ $\sigma = 5.068$	$\mu_{114} = 8.986$ $\sigma = 5.163$	$\mu_{87} = 9.582$ $\sigma = 5.675$
TTF	17	9	7	31	7	23	30	12	15
$\hookrightarrow$ inv	0.059	0.111	0.143	0.032	0.143	0.043	0.033	0.083	0.067
XTTF-2	19	33	97	113	26	24	105	19	27
$\hookrightarrow$ inv	0.053	0.030	0.010	0.009	0.038	0.042	0.010	0.053	0.037
MTBF	12.000	20.125	20.167	47.667	8.167	10.308	24.600	20.714	15.000
$\hookrightarrow$ inv	0.083	0.050	0.050	0.021	0.122	0.097	0.041	0.048	0.067
MTTR	1.917	1.000	5.833	1.000	2.692	1.400	1.200	1.125	2.600
$\hookrightarrow$ inv	0.522	1.000	0.171	1.000	0.371	0.714	0.833	0.889	0.385

(d) scenario B: resource allocation & under-/overshooting

Table 7.12: Experiment 7.11: redundancy configuration/allocation and failure statistics (scenarios A and B).

rounds. A service time distribution S with request processing times are sampled from an exponentially distributed random variate with λ set to $1/2.5$. Likewise, incoming/outgoing network transmission RTT times are sampled from an exponentially distributed random variable with λ set to $1/3.5$. Similar distributions are applied in scenarios B and C. The A-NVP algorithm is set to apply a safety margin $c_{sm} = 1$. The reward model parameters are set as follows: $k_1 = 0.85$, $k_2 = 0.75$, and $k_{max} = 0.95$, and the redundancy dimensioning model is configured with $r_d = 5$, and $r_u = 3$ and $r_f = 1$. We assume a pool of $|V| = 9$ functionally-equivalent resources (versions): the initial degree of redundancy is set to $n_{init} = 5$, and is then allowed to subsequently vary between $n_{min} = 3$ and $n_{max} = 9$. RVF and EVF failures are injected with an 80%, resp. with 20% probability, with RVF failures sampled from a uniform distribution (refer to p. 51).

– Scenario B is identical to scenario A, except that voting rounds no longer hit the NVP scheme one after another. Instead, they will *arrive* at a constant, evenly paced rate. We will inject new voting rounds every 5.65 discrete time units (which should be sufficient to ensure there will be overlapping processing of multiple voting rounds, given the expected average RTT and service times).
– Scenario C is identical to scenario B, except that the inter-arrivals are sampled from a uniformly distributed arrival process A with $a = 2$ and $b = 9.3$, with an average $a+b/2 = 5.65$ comparable to previous scenario.
– Finally, in scenario D, we will apply the same configuration as in scenario C, but with slightly worse RTT and RPT (service) times: for the service time distribution, λ is set to $1/3.25$, and the exponential distribution used to sample RTT times will have $\lambda = 1/3.75$.

Given a similar fault model, the behaviour of the algorithm under these varying environmental conditions can be observed in Fig. 7.14 and 7.15 (overview of failure occurrences and resource allocation), and the tables below (statistics collected for scheme and individual versions). The following table gives a brief overview of the key findings when each of these scenarios is subjected to the same environment:

scenario	strategy	variant	c_{sm}	$\ell^{\mathcal{C}}_{failure}$	$\square_{corrupt}$	$\sum n^{(\mathcal{C},\ell)}$	inter-arrival time
A	B	1	1	3%	0%	1152	—
B	B	1	1	2%	9%	984	$\mu = 5.650$, $\sigma = 0$
C	B	1	1	2%	10%	851	$\mu = 5.669$, $\sigma = 2.103$
D	B	1	1	6%	15%	1147	$\mu = 5.669$, $\sigma = 2.103$

We have applied Strategy B, Variant 1 as defined in Sect. 7.3.1.2, and have set a safety margin c_{sm} to ease proactive upscaling. As we introduced more variability in the environmental behaviour (RTT and service response and inter-arrival times), we can observe an increase in the number of voting rounds for which the redundancy dimensioning model determined the redundancy level to be used based on a so-called *corrupted* window of cached contextual information $\square_{corrupt}$ — *v.* p. 101. As one would expect, the more pending requests (voting rounds), the higher the degree of window corruption. No direct correlation can be found between window corruption and the scheme's overall availability (using $\ell^{\mathcal{C}}_{failure}$ as an indicator for unavailability).

When comparing the end-to-end version response times in Tables 7.12c, 7.12d, 7.13c and 7.13d, scenarios A–C show similar average response times for all versions, within range $[8.30, 9.65]$ of discrete event time steps. The increase in average RTT and service response times in scenario D translates in another range

[9.30, 10.97]. Obviously, compared to scenario C, this results in a higher percentage of window corruption because of the additional variability in the overall RTT, which can also be observed by an additional amount of LRF failures in scenario D. It is therefore not surprising to see that the scheme is somewhat more unavailable: 6% compared to a mere 2%. But when comparing Tables 7.12d, 7.13c and 7.13d, one can deduce from the measurements of $\#rounds(\mathcal{C}, v) - \#consent(\mathcal{C}, v)$ that with 170[a], scenario D had to endure far more dissent — potentially due to additional disturbances — than scenarios B and C (where a mere 113, resp. 99 were recorded). We have kept the t_{max} timeout stable across the different scenarios. There is however a direct correlation between the degree of window corruption and t_{max}: the higher its value, the longer it will take to collect ballots. However, the value should be chosen in line with the response times recorded for the underlying resources, otherwise the voting algorithm may not be able to adjudicate an outcome as too few versions returned their response in due time.

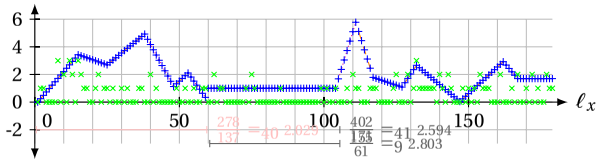
[a] Obtained by adding the measurements for all involved versions.

– **Whilsicality of the environment**: as mentioned before, the system-environment fit may change. At times, the environment may behave quite differently than what the designer initially foresaw. Although an A-NVP composite will eventually detect a change in behaviour — be it in terms of availability, response times or load of/on the underlying versions — a transient state is likely during which the composite is just operating without any accurate or precise perception or awareness of its surrounding environment — *cf.* experiment 7.8.

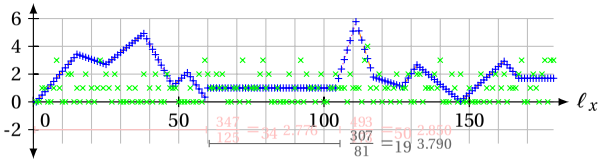
7.5 Conclusion

The various experiments conducted as part of this chapter seem to confirm that the suggested A-NVP algorithm can be effectively applied to identify situations necessitating an adjustment of the redundancy configuration. We have shown that dynamic redundancy configurations can achieve a substantial improvement in dependability, compared to traditional, static redundancy strategies. Given the availability of spare system resources, our redundancy dimensioning model is indubidably capable of scaling up the employed degree of redundancy, either in response to a failure of the scheme, or as a precautionary measure if the effectiveness of the employed redundancy is observed to deteriorate. Furthermore, tuning the adopted degree of redundancy to the actually observed disturbances allows unnecessary resource expenditure to be reduced, therefore enhancing cost effectiveness.
Our experimentation has revealed that:

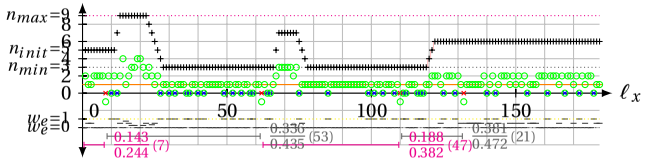
– Our A-NVP algorithm will only be successful if it is configured in line with the behavioural characteristics of the environment in which it will be put into operation — in particular the applicable fault model. Even if these characteristics would be unknown, the algorithm and the provided simulation framework will support the designer in assessing how the redundancy scheme would perform under various conditions, which would avoid wrong assumptions being made and therefore help to realise a better system-environment fit.



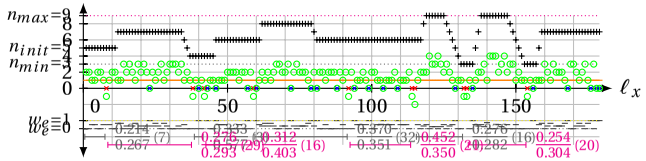
(a) scenario C: failures overview



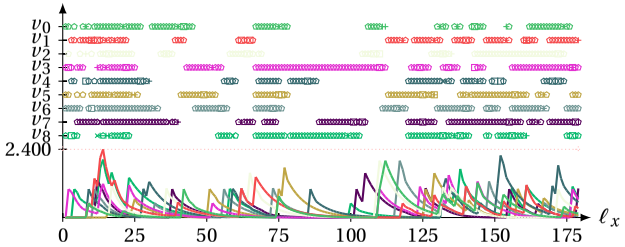
(b) scenario D: failures overview



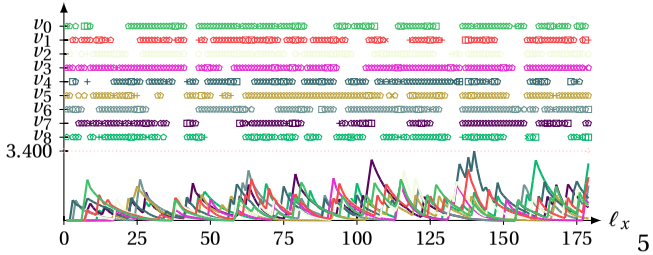
(c) scenario C: resource allocation & under-/overshooting



(d) scenario D: resource allocation & under-/overshooting



(e) scenario C: normalised dissent & observed disturbances



(f) scenario D: normalised dissent & observed disturbances

Figure 7.15: Experiment 7.11: the effectiveness of A-NVP strategies is determined by the system-environment fit (continued; scenarios C and D).

n_{min}	3	$\ell^{\mathbb{C}}_{failure}\ (c^{(\mathbb{C},\ell)}_{max} - m^{(\mathbb{C},\ell)} < 0)$			2% (4 of 179)
n_{max}	9	$\sum n^{(\mathbb{C},\ell)} = 851$	$\sum cr(e^{(\mathbb{C},\ell)}) = 353(377)$	$\triangle_{success}$	100% (13 of 13)
n_{init}	5	$\sum(n^{(\mathbb{C},\ell)} - cr(e^{(\mathbb{C},\ell)})) = 498(474)$		$\nabla_{failure}$	0% (0 of 11)
t_{max}	20.000	$w_v^{(\mathbb{C},\ell)} \in_{175} [0.000, 1.000]$	$\mu = 0.298$	$\sigma = 0.404$	$ci_{90\%}$: (2.48E-1, 3.48E-1)
$\ell^{\mathbb{C}}_{total}$	179	$n^{(\mathbb{C},\ell)} \in [3, 9]$	$\mu = 4.754$	$\sigma = 1.807$	$ci_{90\%}$: (4.53, 4.98)
r_d	5	$glssymb: red-window-fill^d \in_{179} [0, 5]$	$\mu = 2.447$	$\sigma = 1.952$	$ci_{90\%}$: (2.21, 2.69)
		$\square_{reset^d} \in_{33} [1, 5]$	$\mu = 2.970$	$\sigma = 1.686$	$ci_{90\%}$: (2.49, 3.45)
r_u	3	$\square_{fill^u} \in_{179} [0, 3]$	$\mu = 0.279$	$\sigma = 0.590$	$ci_{90\%}$: (2.07E-1, 3.52E-1)
		$\square_{reset^u} \in_{33} [1, 3]$	$\mu = 1.182$	$\sigma = 0.528$	$ci_{90\%}$: (1.03, 1.33)
r_f	1	$\square_{corrupt}$	10% (18 of 179)		
TTF	7	$XTTF_{-2} = 61$	$MTBF_{=3} 40.333$	$MTTR_{=4} 1.000$	
c_{sm}	f	$c_{sm} = 1$	n/a	n/a	$ci_{90\%}$: (1.00, 1.00)
		$\ell_{suspect}\ (0 \leq c^{(\mathbb{C},\ell)}_{max} - m^{(\mathbb{C},\ell)} < c_{sm})$			20% (36 of 179)

(a) scenario C: failures overview

n_{min}	3	$\ell^{\mathbb{C}}_{failure}\ (c^{(\mathbb{C},\ell)}_{max} - m^{(\mathbb{C},\ell)} < 0)$			6% (10 of 179)
n_{max}	9	$\sum n^{(\mathbb{C},\ell)} = 1147$	$\sum cr(e^{(\mathbb{C},\ell)}) = 379(519)$	$\triangle_{success}$	100% (20 of 20)
n_{init}	5	$\sum(n^{(\mathbb{C},\ell)} - cr(e^{(\mathbb{C},\ell)})) = 768(628)$		$\nabla_{failure}$	10% (2 of 20)
t_{max}	20.000	$w_v^{(\mathbb{C},\ell)} \in_{169} [0.000, 1.000]$	$\mu = 0.338$	$\sigma = 0.341$	$ci_{90\%}$: (2.95E-1, 3.81E-1)
$\ell^{\mathbb{C}}_{total}$	179	$n^{(\mathbb{C},\ell)} \in [3, 9]$	$\mu = 6.408$	$\sigma = 1.509$	$ci_{90\%}$: (6.22, 6.59)
r_d	5	$glssymb: red-window-fill^d \in_{179} [0, 5]$	$\mu = 3.022$	$\sigma = 2.008$	$ci_{90\%}$: (2.78, 3.27)
		$\square_{reset^d} \in_{26} [1, 5]$	$\mu = 3.231$	$\sigma = 1.861$	$ci_{90\%}$: (2.63, 3.83)
r_u	3	$\square_{fill^u} \in_{179} [0, 2]$	$\mu = 0.201$	$\sigma = 0.467$	$ci_{90\%}$: (1.44E-1, 2.58E-1)
		$\square_{reset^u} \in_{25} [1, 2]$	$\mu = 1.160$	$\sigma = 0.374$	$ci_{90\%}$: (1.04, 1.28)
r_f	1	$\square_{corrupt}$	15% (26 of 179)		
TTF	7	$XTTF_{-2} = 37$	$MTBF_{=9} 15.222$	$MTTR_{=8} 1.250$	
c_{sm}	f	$c_{sm} = 1$	n/a	n/a	$ci_{90\%}$: (1.00, 1.00)
		$\ell_{suspect}\ (0 \leq c^{(\mathbb{C},\ell)}_{max} - m^{(\mathbb{C},\ell)} < c_{sm})$			12% (21 of 179)

(b) scenario D: failures overview

version	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
re-integration latency	$\mu_{11} = 8.182$ $\sigma = 9.141$	$\mu_9 = 11.000$ $\sigma = 14.474$	$\mu_{10} = 9.400$ $\sigma = 10.895$	$\mu_8 = 7.375$ $\sigma = 5.041$	$\mu_{11} = 8.182$ $\sigma = 10.496$	$\mu_9 = 9.444$ $\sigma = 10.394$	$\mu_9 = 9.111$ $\sigma = 10.043$	$\mu_7 = 8.429$ $\sigma = 7.390$	$\mu_9 = 8.889$ $\sigma = 9.466$
exclusion latency	$\mu_9 = 4.778$ $\sigma = 3.032$	$\mu_5 = 5.200$ $\sigma = 4.658$	$\mu_5 = 2.200$ $\sigma = 1.304$	$\mu_3 = 12.667$ $\sigma = 8.083$	$\mu_6 = 2.833$ $\sigma = 1.169$	$\mu_5 = 11.000$ $\sigma = 7.314$	$\mu_5 = 5.667$ $\sigma = 4.619$	$\mu_4 = 12.000$ $\sigma = 13.928$	$\mu_5 = 8.000$ $\sigma = 5.831$
exclusion failures	$\mu_9 = 1.778$ $\sigma = 0.972$	$\mu_5 = 1.000$ $\sigma = 1.304$	$\mu_5 = 1.600$ $\sigma = 0.894$	$\mu_3 = 1.667$ $\sigma = 1.155$	$\mu_6 = 1.667$ $\sigma = 0.816$	$\mu_5 = 1.800$ $\sigma = 0.837$	$\mu_5 = 1.667$ $\sigma = 1.155$	$\mu_4 = 2.500$ $\sigma = 3.000$	$\mu_5 = 2.200$ $\sigma = 1.304$
successive usage period	$\mu_{93} = 0.925$ $\sigma = 3.163$	$\mu_{99} = 0.677$ $\sigma = 2.482$	$\mu_{99} = 0.798$ $\sigma = 3.687$	$\mu_{59} = 1.627$ $\sigma = 6.491$	$\mu_{95} = 0.863$ $\sigma = 2.995$	$\mu_{85} = 1.071$ $\sigma = 4.070$	$\mu_{82} = 1.122$ $\sigma = 3.700$	$\mu_{59} = 1.780$ $\sigma = 6.242$	$\mu_{80} = 1.175$ $\sigma = 4.281$
$\#rounds(\mathbb{C}, v) - \#consent(\mathbb{C}, v)$	12	12	9	10	12	11	9	10	14
$\hookrightarrow \ell^{\mathbb{C}}_{total}$	7 %	7 %	5 %	6 %	7 %	6 %	5 %	6 %	8 %
$\hookrightarrow \#rounds(\mathbb{C}, v)$	14 %	15 %	11 %	8 %	15 %	12 %	9 %	9 %	14 %
$\#rounds(\mathbb{C}, v)$	86	78	79	120	82	94	97	116	99
$\hookrightarrow \ell^{\mathbb{C}}_{total}$	48 %	44 %	44 %	67 %	46 %	53 %	54 %	65 %	55 %
$\#consent(\mathbb{C}, v)$	73	64	69	109	70	83	87	106	85
$\hookrightarrow \#rounds(\mathbb{C}, v)$	85 %	82 %	87 %	91 %	85 %	88 %	90 %	91 %	86 %
normalised dissent	$\mu_{179} = 0.267$ $\sigma = 0.351$	$\mu_{179} = 0.324$ $\sigma = 0.401$	$\mu_{179} = 0.307$ $\sigma = 0.384$	$\mu_{179} = 0.273$ $\sigma = 0.331$	$\mu_{179} = 0.344$ $\sigma = 0.421$	$\mu_{179} = 0.263$ $\sigma = 0.338$	$\mu_{179} = 0.267$ $\sigma = 0.337$	$\mu_{179} = 0.249$ $\sigma = 0.324$	$\mu_{179} = 0.367$ $\sigma = 0.423$
end-to-end response time	$\mu_{86} = 8.949$ $\sigma = 4.597$	$\mu_{78} = 9.054$ $\sigma = 5.275$	$\mu_{79} = 9.441$ $\sigma = 5.559$	$\mu_{120} = 9.171$ $\sigma = 5.128$	$\mu_{82} = 8.780$ $\sigma = 5.646$	$\mu_{84} = 9.880$ $\sigma = 5.469$	$\mu_{87} = 8.527$ $\sigma = 4.986$	$\mu_{116} = 9.335$ $\sigma = 5.145$	$\mu_{99} = 9.486$ $\sigma = 5.079$
TTF	7	9	7	11	7	16	30	8	11
$\hookrightarrow$ inv	0.143	0.111	0.143	0.091	0.143	0.062	0.033	0.125	0.091
XTTF-2	17	14	35	14	10	23	116	11	12
$\hookrightarrow$ inv	0.059	0.071	0.029	0.071	0.100	0.043	0.009	0.091	0.083
MTBF	10.067	15.900	17.143	32.200	17.333	17.750	33.000	16.333	13.909
$\hookrightarrow$ inv	0.099	0.063	0.058	0.031	0.058	0.056	0.030	0.061	0.072
MTTR	2.200	1.111	5.500	1.000	2.000	2.125	1.250	3.333	2.500
$\hookrightarrow$ inv	0.455	0.900	0.182	1.000	0.500	0.471	0.800	0.300	0.400

(c) scenario C: resource allocation & under-/overshooting

version	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
re-integration latency	$\mu_{11} = 4.727$ $\sigma = 3.927$	$\mu_{13} = 4.231$ $\sigma = 3.295$	$\mu_{14} = 3.214$ $\sigma = 2.293$	$\mu_8 = 3.125$ $\sigma = 2.800$	$\mu_{12} = 4.500$ $\sigma = 2.541$	$\mu_{12} = 3.667$ $\sigma = 2.229$	$\mu_8 = 5.875$ $\sigma = 5.083$	$\mu_9 = 7.222$ $\sigma = 4.919$	$\mu_{14} = 4.571$ $\sigma = 3.877$
exclusion latency	$\mu_6 = 14.167$ $\sigma = 13.197$	$\mu_8 = 5.000$ $\sigma = 4.408$	$\mu_7 = 5.857$ $\sigma = 6.122$	$\mu_4 = 13.000$ $\sigma = 13.441$	$\mu_8 = 6.500$ $\sigma = 8.401$	$\mu_6 = 5.500$ $\sigma = 3.619$	$\mu_4 = 7.250$ $\sigma = 10.532$	$\mu_4 = 12.750$ $\sigma = 9.500$	$\mu_6 = 8.167$ $\sigma = 7.935$
exclusion failures	$\mu_6 = 1.833$ $\sigma = 1.169$	$\mu_8 = 1.375$ $\sigma = 0.744$	$\mu_7 = 1.714$ $\sigma = 0.756$	$\mu_4 = 1.750$ $\sigma = 0.957$	$\mu_8 = 1.875$ $\sigma = 2.100$	$\mu_6 = 1.667$ $\sigma = 0.816$	$\mu_4 = 1.500$ $\sigma = 1.000$	$\mu_4 = 2.000$ $\sigma = 1.414$	$\mu_6 = 2.333$ $\sigma = 1.862$
successive usage period	$\mu_{31} = 2.377$ $\sigma = 6.520$	$\mu_{35} = 2.164$ $\sigma = 4.849$	$\mu_{45} = 2.933$ $\sigma = 6.376$	$\mu_{25} = 5.280$ $\sigma = 10.306$	$\mu_{57} = 2.105$ $\sigma = 5.112$	$\mu_{41} = 2.568$ $\sigma = 7.875$	$\mu_{47} = 2.468$ $\sigma = 6.734$	$\mu_{51} = 1.662$ $\sigma = 5.304$	$\mu_{64} = 1.688$ $\sigma = 4.170$
$\#rounds(\mathbb{C}, v) - \#consent(\mathbb{C}, v)$	19	23	18	10	24	14	17	20	25
$\hookrightarrow \ell^{\mathbb{C}}_{total}$	11 %	13 %	10 %	6 %	13 %	8 %	9 %	11 %	14 %
$\hookrightarrow \#rounds(\mathbb{C}, v)$	15 %	19 %	14 %	6 %	20 %	10 %	13 %	18 %	22 %
$\#rounds(\mathbb{C}, v)$	126	122	133	154	120	135	132	110	115
$\hookrightarrow \ell^{\mathbb{C}}_{total}$	70 %	68 %	74 %	86 %	67 %	75 %	74 %	61 %	64 %
$\#consent(\mathbb{C}, v)$	105	97	112	140	93	119	114	88	89
$\hookrightarrow \#rounds(\mathbb{C}, v)$	83 %	80 %	84 %	91 %	78 %	88 %	86 %	80 %	77 %
normalised dissent	$\mu_{179} = 0.576$ $\sigma = 0.486$	$\mu_{179} = 0.635$ $\sigma = 0.478$	$\mu_{179} = 0.516$ $\sigma = 0.461$	$\mu_{179} = 0.341$ $\sigma = 0.414$	$\mu_{179} = 0.758$ $\sigma = 0.611$	$\mu_{179} = 0.360$ $\sigma = 0.385$	$\mu_{179} = 0.489$ $\sigma = 0.516$	$\mu_{179} = 0.638$ $\sigma = 0.559$	$\mu_{179} = 0.662$ $\sigma = 0.543$
end-to-end response time	$\mu_{136} = 11.051$ $\sigma = 5.133$	$\mu_{122} = 10.851$ $\sigma = 5.529$	$\mu_{131} = 10.090$ $\sigma = 4.980$	$\mu_{154} = 9.797$ $\sigma = 4.901$	$\mu_{120} = 10.777$ $\sigma = 5.288$	$\mu_{135} = 10.735$ $\sigma = 5.073$	$\mu_{132} = 10.644$ $\sigma = 5.433$	$\mu_{110} = 10.274$ $\sigma = 5.784$	$\mu_{115} = 10.737$ $\sigma = 5.464$
TTF	21	14	7	14	7	16	62	8	11
$\hookrightarrow$ inv	0.048	0.071	0.143	0.071	0.143	0.062	0.016	0.125	0.091
XTTF-2	45	33	35	16	35	23	83	12	12
$\hookrightarrow$ inv	0.022	0.030	0.029	0.062	0.029	0.043	0.012	0.083	0.083
MTBF	13.600	13.909	10.545	22.286	10.786	13.000	16.333	21.286	10.714
$\hookrightarrow$ inv	0.074	0.072	0.095	0.045	0.093	0.077	0.061	0.047	0.093
MTTR	1.000	2.500	2.091	2.125	2.273	2.200	1.429	1.000	4.091
$\hookrightarrow$ inv	1.000	0.400	0.478	0.471	0.440	0.455	0.700	1.000	0.244

(d) scenario D: resource allocation & under-/overshooting

Table 7.13: Experiment 7.11: redundancy configuration/allocation and failure statistics (continued; scenarios C and D).

– Despite the benefit of purposeful redundancy upscaling, we have also observed that upscaling is not always beneficial, and that the significance of extracting, analysing and using correct contextual information when dynamically selecting eligible versions at runtime is of equal, if not higher, importance.
One could also enhance the solution to upscale the level of redundancy only when the potential gains significantly exceed the potential risk of doing so. For instance, one could consider a (dynamic) upper limit for normalised dissent measurements, and use this to assess whether to upscale/downscale/maintain the current redundancy level.

– Collecting runtime statistics throughout the scheme's operational life is essential, as is illustrated by the normalised dissent. This metric can be used to assess a specific version's reliability over time, in the context of a specific redundancy scheme. Analysis of the evolution of recorded measurements can shed clarity on the failure model for individual versions.

– The perceptional abilities of the algorithm could be improved by adding probing mechanisms to gain more accurate insight into the the nature of disturbances, *e.g.* crash failures. Although such enhancement would improve the scheme's overall dependability, it would come at the expense of additional resource consumption.

– Finally, when determining a suitable level of redundancy to be applied, to ensure full reliability, the designer will likely need to apply a safety margin c_{sm} so as to realise (i) a more efficient proactive upscaling, and (ii) a less aggressive downscaling. In doing so, it is expected that the reduction in resource consumption will be of less significance — though, in general, still better compared to using a static redundancy scheme.

CHAPTER 8

How WS-* Specifications can Ease the Development of FCUs

"Web service technology allows the integration of applications across different organisations and facilitates interoperability among distributed heterogeneous applications and components, independent of the development platform, middleware, operating system and hardware type" [131]. The use of stateless web services and the confinement of any communication to take place via explicitly defined service interfaces appear to suggest that web services are an adequate technology for implementing FCUs. In this chapter, a prototypical service-oriented implementation of the proposed adaptive fault-tolerant strategy is presented, demonstrating that WS- specifications can be leveraged not only to gather and disseminate contextual information, but to sustain adaptive redundancy management as well, broadening the applicability of NVP schemata by increased interoperability. We argue that the WSDM and WSRF specifications in particular can aid in isolating effective implementations of autonomic capabilities from the underlying managed resources, and in achieving proper separation of concerns. With message-oriented middleware solutions increasingly being used to underpin the operations of web services-based service-oriented architectures, we deemed it useful to conclude this chapter by elaborating on the queuing model that such solutions typically administer when handling service requests. The content of this chapter has been disseminated to the public through publication [83]. Related research question(s): RQ-4.*

8.1 On the Role of Message-oriented Middleware in Contemporary Distributed Computing Systems

There is a growing move to transform legacy distributed systems into service-oriented architectures (SoA), mainly driven by the prospects of interoperability, agility and legacy leverage. The widespread adherence to the **service-oriented computing paradigm** can be justified as it comprises the best practices in distributed computing

of, roughly estimated, the past twenty years, and by the numerous standardisation initiatives backed by major industry consortia. It emerged as an architectural pattern in response to increasingly complex challenges in the domain of **enterprise application integration** (EAI), aiming to overcome the technological disparities that are commonly observed between the various heterogeneous legacy systems that can be found in an enterprise's ICT landscape.

Service-oriented architectures are a continuation of traditional client-server architectures. In this type of architecture, there is at least one **service provider** — in control of and responsible for managing a server — that is capable of delivering a specific service. One or more **service requestors** — acting as clients — use (consume) this service and communicate with the service provider by exchanging messages, honouring the terms described in the service interface definition. They are distributed computing systems, where a network infrastructure is responsible for interconnecting the various entities like service providers and requestors, and as such supports the exchange of message data in a robust and performant manner.

Among the available technological solutions to SoA, XML-based web services, which have become the predominant implementation technology for encapsulating and deploying software components, are now being used in a diversity of application domains, ranging from enterprise software to embedded systems. Such type of web services in particular offer a high degree of interoperability, which mainly stems from the use of the SOAP messaging protocol to envelop messages to be exchanged, and which is well-suited for carrying additional attributes to ease message transmission, routing and processing — of which many are predefined attributes of various specifications in the WS-* protocol stack. Furthermore, such type of web services are easily approachable after introspection of the interface definition, which usually takes the form of a WSDL document. This document contains a listing of entry points (the so-called **endpoints**) through which the service is accessible, and includes a rigorous description of syntactically valid inbound request messages, and the expected structure of outbound response messages.

As demand grew for transforming enterprise applications into web services-based service-oriented architectures, so grew trust in WS-* specifications. And as these specifications gained in popularity, we have seen a gradual shift from custom-built proprietary integration implementation technology to reusable toolsets like software libraries and code generators, and finally to **message-oriented middleware (MoM)**. A MoM solution is a set of software libraries that collectively support the sending and receiving of messages that result from requesting the service and functionalities exposed by a distributed computing system. MoM solutions are available both as commercial and open-source offerings; either type usually incurs a license fee. They come with an embedded application server, and software runtime libraries to support various specifications in support of common tasks and duties like message processing, transaction management, orchestration, service discovery and federation, security and access control, *etc.* Examples include specifications that are part of the core Java™ platform, and additional libraries in support of the WS-* protocol stack. The JAX-WS and JAX-RS specifications are an essential part of the Java™ platform, and help to expose business logic as web services resources. There are also numerous open-source runtime libraries available, many of which emerged from the Apache Web Services project. For sure, Apache Axis2 and Glassfish Metro

are the most wide-spread. Many of these libraries are included in commercial integration solutions.

As can be seen from Fig. 8.1, the application-to-application (A2A) interaction in SoA solutions is often accomplished through the use of middleware technology. Emphasis is placed on the SOAP messaging layer that is governed by a WSDL layer, in which the required message exchange patterns are defined. MoM solutions usually result in an improved separation of concerns and maintainability, since most of the technical duties it should fulfill no longer require the developer to write (all of the) implementation logic. The integration code left in the business logic is minimal, and is usually well-readable and maintainable when using annotations. Apart from the key responsibilities listed here above, MoM solutions introduce an abstraction layer on top of the actual network technology used to exchange messages — be it an intranet, the Internet, or simply an interprocess communication facility. The underlying communication channels are thus completely transparent to the actual business logic, requiring only little or no integration logic at all.

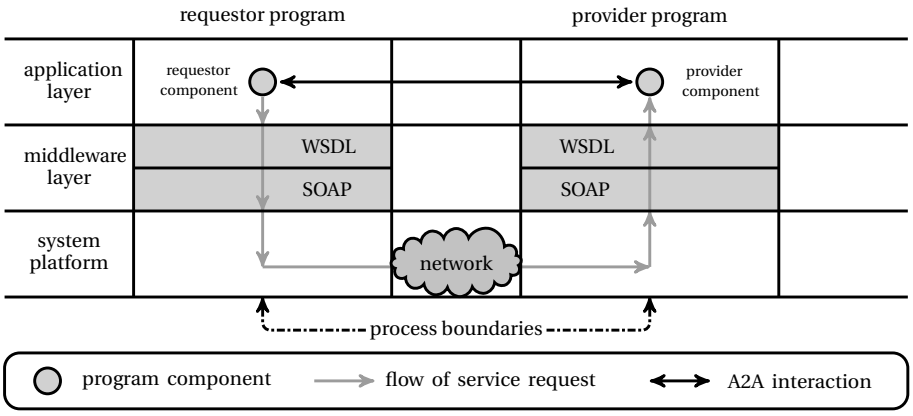


Figure 8.1: Reproduction of [51, Fig. 9.1]: application-to-application communication in which the middleware layer is responsible for managing the network connectivity and handling the messaging complexity, while adhering to WS-* specifications.

8.2 Overview of Key Web Service Specifications and Standards

There exist numerous **web services (WS-*) specifications** that emerged from intense and unwavering standardisation initiatives coordinated by the W3C® and OASIS® organisations, and backed by major industry consortia. Due to the complementary approach in the way most of these specifications have been defined, the resulting **web service protocol stack** has proved to be extremely useful in defining, locating, and implementing XML-based web services, as well as making services interact with each other. Most of these specifications define languages and vocabularies compliant to the **eXtensible Markup Language (XML)**, and aim to solve specific tasks and duties that are commonly expected to be fulfilled by message-oriented middleware in an interoperable, platform-, technology- and vendor-neutral way.

As can be seen in Fig. 8.2, the WS-* stack covers transport, messaging and eventing, description, discovery, management, orchestration, and security protocols (the latter are not shown though) [22, Chapt. 6–7, 17]. This figure shows a layered representation of the specifications relevant to our A-NVP implementation. Although they are not explicitly shown because they are not directly related to the research reported here, other WS-* specifications exist that are intended to contribute to the overall dependability of XML-based web services, mainly in the areas of reliable messaging, transactional support and end-to-end-security [22].

Due to space restrictions, only relevant specifications will be addressed, and apart from high-level introductory explanations that provide the reader with information about the key concerns addressed by and the main concepts of the proposed solution, specific features cannot and will not be provided. The reader may wish to consult `https://www.oasis-open.org/standards` for in-depth information about the listed specifications, as well as an introduction to those not covered in this section. Detailed information on W3C®-driven specifications, particularly XML-related standards and first-generation WS-* standards such as WSDL, WS-Addressing and SOAP, may be retrieved from `https://www.w3.org/TR/`.

The use of WS-* specifications is, in itself, no guarantee for achieving ICT solutions of higher quality with improved dependability characteristics. Some specifications like UDDI have often been perceived as overly complex and have never seriously gained ground. The decision on which subset of the available specifications is to be used should be made without prejudice and requires adequate forethought, withholding those that cannot be purposefully used to support the desired non-functional requirements of the solution that is to be developed and/or re-engineered. The same applies to the assessment of competing specifications, or that show a significant overlap — such as the WS-Eventing and WS-Notification specifications, for instance.

8.2.1 The Foundations: Connectivity & Message Exchange Patterns

In this section, an overview is given on the foundational technologies and specifications that are indispensable to effectively implement distributed computing systems. From a **connectivity perspective**, "the [...] significant advantage that XML web services have over previous efforts is that they work with standard Web protocols — XML, HTTP and TCP/IP" [132].

At the very lowest layer, there is a need for a robust and reliable network communication infrastructure, to realise the intended connectivity. Packetised network traffic resulting from the exchange of messages is usually transmitted over TCP/IP-enabled networks. The **Internet Protocol Suite** comprises a set of networking protocols upon which the Internet and most computer networks rely. Maintained by the IETF®, the relevant RFCs describe protocols for data packetisation, addressing, transmission, and routing. Together, they provide a robust solution for end-to-end networked connectivity.

As it is, web services are network-accessible software components that encapsulate and expose the underlying business logic in a structured, managed and standardised way. They are usually placed in production by deploying them to an application server, which will expose them to the network through a managed set of **endpoint references (EPRs)** through which clients can communicate and consume the exposed

services by the exchange of messages. An EPR is an XML fragment that encapsulates the information necessary for identifying a web service endpoint, such that messages can be routed to the intended target web service. The structure of these XML fragments should comply with the syntax and semantics outlined in the **WS-Addressing** specification. WS-Addressing was set out by the W3C® consortium as a set of "transport-neutral mechanisms that allow web services [and clients] to communicate addressing information" and conversational attributes [133]. It has become an essential part of the WS-* protocol stack, defining a series of XML vocabularies for identifying and communicating references to concrete web service endpoints, enriching the expressiveness of regular **uniform resource identifiers** — compact sequences of characters adhering to the syntactical guidelines set out in RFC 3986 that are used to identify, name, and address network resources — by including additional reference parameters [54, Sect. 18.2].

Emerging from a joint standardisation effort coordinated by IETF® and W3C®, the HTTP protocol has become the foundation for (textual) data exchange over the Internet [28, Sect. 7.3.4]. It is commonly used in XML-based SoA solutions to exchange SOAP messages amongst web services by POSTing these messages as HTTP request payloads. Over the years, this approach has proved most effective in achieving the intended connectivity required to access remote web services whose endpoints are often protected by the use of corporate **firewalls**.

From a **messaging perspective**, the predominant specifications in the WS-* stack are, without doubt, SOAP and WSDL. Both specifications have been well received and have been widely and successfully adopted.

The **Simple Object Access Protocol (SOAP)** specification introduces a lightweight procotol that is intended to support the exchange of structured information between XML-based web services. It was designed to be platform- and technology-independent, bridging the technological disparities that can be observed when integrating various middleware, implementation and transport technologies. Even though it was designed with a particular focus on the exchange of messages in XML-based SoA, its application is by no means limited to such context. Valid SOAP messages should be structured as *envelopes*, in which the message *body* carrying the XML payload is clearly separated from the optional SOAP *headers* which carry specific information in support of the WS-* feature set. Among SOAP header blocks, WS-Addressing elements are commonly found. Valid messages are usually exchanged over HTTP, although other transmission protocols are available [17] [54, Chapt. 11].

"The SOAP messaging protocol provides only basic communication and does not describe what pattern of message exchanges are required to be followed by specific service requestors and providers" [51, Chapt. 9]. To address this deficiency, a web service is typically exposed through a well-defined open XML interface described in a **Web Services Description Language (WSDL)** document that formally describes the syntax of standardised and application-specific messages in XML Schema Definition (XSD) format [18]. The WSDL specification has been highly acclaimed and widely used, and, being an XML-based interface description language, it outlines the details for structuring and describing the functionality offered by web services, *port types* — *i.e.* the set of exposed operations — and the permissible message (payload) data types and message exchange and interaction patterns [22, Chapt. 7–9].

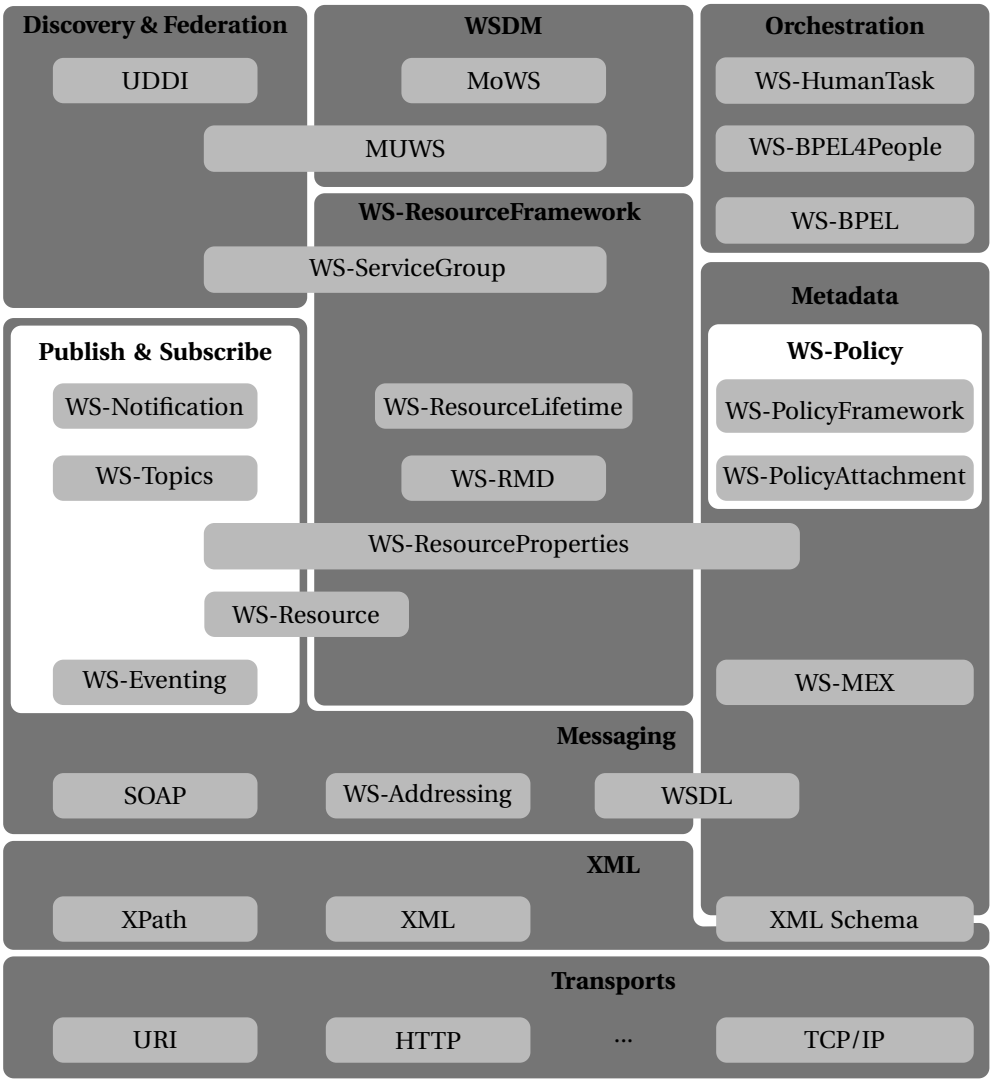


Figure 8.2: Layered overview of WS-* specifications illustrating WSRF and WSDM and their interdependencies relative to other industry standards.

The WSDL specification has been designed to allow for the inclusion of additional extensibility elements that can be used to optionally augment the expressiveness and semantics of the interface description. In the case of stateful WS-Resources, references to WSRF-RP documents and associated WS-RMD descriptors are usually added — specifications that will be covered in Sect. 8.2.4.

8.2.2 Service Introspection & Metadata Retrieval

Considering the expressiveness of WS-* specifications like WSDL, and — in case of stateful WS-Resources — WSRF-RP and WS-RMD, the interface of a web service

component can be described in great detail. Publishing an exhaustive description including all relevant **metadata** in a machine-interpretable way is useful, in that it allows potential service requestors to retrieve this metadata, and to introspect (analyse) it, even in an automated way. The **Web Services Metadata Exchange (WS-MEX)** specification was standardised under the coordination of the W3C® and proposes a standardised interface and a set of operations that can be used for retrieving all or part of the metadata associated with a specific web service, using only a single endpoint (reference) [134, 135] [22, Sect. 7.5]. Using specific dialect identifiers, the requesting party itself can decide on whether to retrieve the WSDL in its entirety, or in part, *e.g.* the embedded WSRF-RP document, or the referenced WS-RMD document [136, 137].

8.2.3 Publish-and-Subscribe Eventing Models

Whereas WSDL mainly describes message exchange patterns — most of which will be initiated by the service requestor — it comes with limited support for asynchronous messaging patterns. Fortunately, several WS-* specifications are available that can be used to implement service-oriented, event-driven architectures. Event-driven architectures rely on a **publish-and-subscribe model** for the asynchronous exchange of specific data fragments called *events* [138, Sect. 10.6]. In an attempt to apply the concepts of publish-and-subscribe models to web services-based SoA solutions, two standardisation initiatives emerged: **WS-Eventing** by the W3C®, and **WS-Notification** by OASIS®. Both WS-* specifications show considerable overlap, in that they define a specific set of operations to facilitate asynchronous message exchange between event *sources* and *sinks*, as well as a generic event message format — *cf.* Sect. 1.1 [139] [24]. It is fair to say that the WS-Notification family of specifications is by far superior to WS-Eventing. Apart from the core WS-BaseNotification specification, it also includes the WS-Topics specification that describes various filtering options with which the *subscriber* can express its particular interests, *e.g.* by means of an XPath query that is validated against the notification message payload.

8.2.4 Exposing and Managing Stateful Resources

The recent advances in autonomic computing have resulted in numerous standardisation initiatives under the auspices of the OASIS®, which have resulted in a comprehensive set of WS-* specifications, most notably the WSRF and WSDM families of specifications.

Serving as the core foundational specification, **WS-Resource** outlines how a WS-Resource results from the composition of a (stateful) resource and a web service through which the resource can be accessed, controlled and/or monitored. Several specifications in the WSRF and WSDM family may be required to support such features though. Note how such type of web services is implicitly assumed to be stateful, whereas statelessness is otherwise considered as one of the key charactistics of web services [140, Chapt. 11].

At the highest level, one can find the **Web Services Distributed Management (WSDM)** family of specifications. These define a standardised, extensible model for exposing a web services-based manageability layer through which an underlying

stateful resource — *e.g.* an application or a device wrapped within a WS-Resource-compliant entity — can be managed and controlled. This will result in improved controllability and interoperability, even across enterprise and organisational boundaries [109]. A WSDM-enabled WS-Resource is essentially an aggregation of several **manageability capabilities** that are collectively exposed through a single, cohesive WSDL interface. A manageability capability defines a set of resource properties, operations, events, metadata and other semantics supporting a particular management aspect of a WS-Resource service. Apart from a set of predefined foundational manageability capabilities, WSDM was designed for extensibility, allowing the development of domain-specific capabilities comprising customised manageability logic or that extend any of the foundational capabilities as appropriate. Such approach clearly accommodates the principle of separation of concerns, not in the least because application-specific logic is separated from the application-agnostic predefined manageability capabilities.

It is comprised of the **Management Using Web Services (MUWS)** and **Management of Web Services (MoWS)** specifications. In addition to the definition of a set of core foundational manageability capabilities, the MUWS specification outlines how to encapsulate and expose customised manageability and/or business logic as additional domain- and/or application-specific capabilities. The MoWS specification takes this one step further by zooming in on the specificities of wrapping and managing web services as such as WS-Resource-compliant entities using MUWS [109, MoWS]. In order to expose information regarding the operational status of the underlying managed web service, it defines several service- and operation-level metrics and status models, as well as a request processing state model, which was derived from the **Web Service Management: Service Life Cycle** specification.

Many of the features introduced by WSDM heavily rely on the **Web Services Resource Framework (WSRF)** family of OASIS®-authored specifications that aim to "define a generic and open framework for modelling and accessing stateful resources using web services". Its two key specifications are, without doubt, the **Web Services Resource Framework: Resource Properties (WSRF-RP)** and **Web Services Resource Framework: Service Group (WSRF-SG)**.

WSRF-RP defines an extensible mechanism for exposing additional, stateful information by means of a set of typed values — called resource properties — in the WSDL interface of a WS-Resource. "The declaration of the WS-Resource's properties represents a projection of or a view on the WS-Resource's state. This projection is defined in terms of a resource properties document. This [...] document serves to define a basis for access to the resource properties through web service interfaces", as the specification introduces robust protocols for querying, reading and manipulating these metadata properties in a standardised format [136].

Resource properties and resource property documents can be formally described using the syntax set out in the **WS-ResourceMetadataDescriptor (WS-RMD)** specification. This concise, albeit expressive language allows to enrich a WS-Resource's core WSDL interface definition with additional metadata in order to describe the semantics of individual resource properties in detail. In doing so, the designer can enforce and indicate any value restrictions and access control constraints that are applicable, supporting, *e.g.* mutability and modifiability [137].

Web Services Resource Framework: Service Group, a related specification, comes with interfaces and operations for managing service groups, *i.e.* "[potentially] heterogeneous by-reference collections of web services". It can be used as a lightweight alternative for UDDI, for it can serve as a lightweight service registry solution, capable of structuring entire federations of web services. It supports representing a logical group of functionally-equivalent services that expose a common WSDL interface as a WS-Resource. It relies on WSRF-RP for group membership management, and on the companion **WS-ResourceLifetime** specification for lifecycle management of the underlying referenced resources.

8.3 A-NVP WS-* SoA Prototype

In this section, we present a prototypical service-oriented implementation of the adaptive fault-tolerant strategy as proposed in Chapt. 5. The framework was conceived leveraging a set of ratified WS-* specifications, mainly capitalising on the features offered by the WSRF, WSDM and WS-Notification families of OASIS®-published standards. The framework was developed using the latest release of the Apache MUSE project to date, supplemented by our own implementation of the MoWS specification[1]. As illustrated in Fig. 5.2, we have implemented an NVP redundancy scheme as a WSDM-enabled WS-Resource-compliant web service aggregating several manageability capabilities. Effectively realising a proper fault containment unit (FCU), where the underlying redundant resources (versions) are *members* of WSRF-SG service group, the core of its implementation consists of three manageability capabilities. The following subsections will cover each in greater detail.

8.3.1 Enhanced WS-ServiceGroup Capability

The composite A-NVP web service leverages the WSRF-SG specification and the notion of membership content rules defined therein to manage federations of functionally-equivalent web services. The entries of the group represent locally or remotely hosted member web services, and **membership content rules** can be used to express constraints on the member services. Such rules can impose limitations on the WSDL port types that services in the service group must implement, as well as the resource properties the member services are expected to expose. The rationale behind the mandatory use of membership content rules is that web services implementing a common WSDL port type and exposing the same set of resource properties can be considered as functionally-equivalent. If needed, adapter interfaces can be used to apply the necessary transformation logic, and expose the underlying service through a common interface, while shielding the specific implementation details and technological differences, and wrapping the originally published WSDL interface, in case this is not fully compliant [141, Sect. 15.1].

We have extended the foundational WSRF-SG capability in order to support advanced replica management. This includes facilities to compensate for the occasional emerging and disappearing of web services in the system. A freshly

[1] For more information, refer to `https://attic.apache.org/projects/muse.html` and `http://52north.org/communities/sensorweb/amused/`.

discovered service may be added to the group as the result of an incoming MUWS advertisement notification, provided the reported service complies to the membership content rules. Upon addition of a replica member web service, its metadata will be validated, and the service group will automatically issue a WS-Notification subscription request so as to be notified of changes in any additional mandatory resource properties that were declared in the membership content rules set on the service group — *cf.* Sect. 8.3.3. Conversely, the receipt of a WS-ResourceLifetime destruction event will trigger the removal of the member from the service group.

The A-NVP composite has been explicitly designed as a generic WSDM-enabled utility WS-Resource so as to support a diversity of applications, without the need to generate application-specific proxy classes at design time. When assembling the deployment artefact, the user is expected to supply the WSDL interface definitions containing the port type descriptions for admissible service group members. During the initialisation of the composite WS-Resource, the provided WSDL definitions will be inspected, and for each non-standardised, request-response operation declared within, a new message handler will be registered. Furthermore, the system will automatically initialise the membership content rules, given the port types that were found whilst scanning the user-supplied interface definitions. Note that the WSDL interface advertised for the A-NVP composite itself is predefined and exposes a single port type combining only the standardised operations defined for the WSRF-SG and WS-Notification Consumer capabilities.

Figure 5.2 shows how message handlers enable the A-NVP composite to accept application-specific SOAP request messages and hand these over to the A-NVP manageability capability for execution. It can be seen from the message handlers that port type A exposes 3 operations and port type B exposes 2. All versions implementing port type A are assumed to be unreachable. When detected, the service group disables the corresponding message handlers. Should there remain no active member services in the group for a particular port type, the respective handlers will be disabled, such that they will dismiss any incoming SOAP request by reporting a WS-Addressing `ActionNotSupported` fault message.

8.3.2 Domain-Agnostic A-NVP Capability

Context information for any of the member web services within the federation is managed at operation level — *cf.* (A29). Specifically, for each operation for which a dynamic message handler was registered, the context manager provides adequate data structures for storing the values $D(\mathcal{C}, v)$, $L(\mathcal{C}, v)$, $T(\mathcal{C}, v)$ and the respectively corresponding maxima $\delta_D^{\mathcal{C}}$, $\delta_L^{\mathcal{C}}$ and $\delta_T^{\mathcal{C}}$ as defined in Sect. 5.3, as well as the counters $\#rounds(\mathcal{C}, v)$ and $\#consent(\mathcal{C}, v)$ that were introduced in Sect. 4.3. Furthermore, application-specific configuration parameters can be specified for individual operations, thereby overriding the system defaults. One may do so by editing a deployment descriptor, in which a service operation can be uniquely identified by the service port type name and the WS-Addressing action URI.

The capability provides a single operation to accept NVP service requests. Upon invocation of the A-NVP composite, the system first determines the set of eligible functionally-equivalent member services in the service group, *i.e.* V. In order to do so, the payload of the incoming SOAP request as well as its WS-Addressing message headers are inspected so as to establish which of the registered port types exposes

the targeted service operation. After acquiring all registered member services that implement the given port type, the capability proceeds by applying the algorithm introduced in Chapt. 5 so as to determine an adequate selection of versions $V^{(\mathcal{C},\ell)}$. Such selection is carried out referring to the context information pertaining to the targeted operation, as stored in the context manager. The SOAP request is then simultaneously forwarded to each of the selected versions. As soon as an absolute majority $m^{(\mathcal{C},\ell)}$ of the selected $n^{(\mathcal{C},\ell)}$ versions have returned their response, the voting scheme will determine and return the outcome of the current voting round ℓ, without awaiting the remaining replicas to return — *cf.* (A18). At the same time, the $n^{(\mathcal{C},\ell)} - m^{(\mathcal{C},\ell)}$ pending results will be collected after the response was sent to the client such that the $dtof^{(\mathcal{C},\ell)}$ and normalised dissent $D(\mathcal{C}, v)$ can be computed and subsequently reported to the context manager as soon as the voting round ℓ has completed and all $n^{(\mathcal{C},\ell)}$ ballots have been acquired and processed.

It is noteworthy to point out that the voting procedure will assign any two versions to the same equivalence class of the partition $\wp^{(\mathcal{C},\ell)} \setminus P_F^{(\mathcal{C},\ell)}$ if the XML fragments enclosed within the body of their SOAP response messages are found to be syntactically equivalent, given the XSD schema definitions included in the WSDL interface. Special attention is paid to SOAP faults, however, which are typically used to convey error condition information when an exceptional situation occurs. In particular, one needs to clearly distinguish between application-specific and application-agnostic fault messages. Whereas the former type of fault messages are expected to carry domain-specific fault data and are processed like ordinary SOAP response messages, application-agnostic fault messages will directly be classified in $P_F^{(\mathcal{C},\ell)}$. Examples of this second category of messages include, *e.g.*, standardised fault messages from various WS-* specifications, or SOAP faults reported for versions that were detected to be affected by performance or omission failures (*cf.* Sect. 5.1 and 5.3).

8.3.3 Externally Supplied Context Information

As pointed out in Sect. 5.3, the vast majority of the metrics and counters stored in the context manager is updated using information that was collected within the A-NVP composite itself, upon completion of a voting round. An exception to this approach though, is the number of pending requests $L(\mathcal{C}, v)$, which needs to be supplied externally as it is conceivable that a member replica may concurrently be used by services other than the A-NVP composite. Specifically, we require any member WS-Resource to expose the metrics defined by the MoWS operation metrics manageability capability. As such, the resource property `OperationMetrics` is supposed to be included in the membership content rules of the A-NVP composite.

Upon addition of a new member service, the enhanced service group capability will consequently issue a WS-Notification subscription request in order to be notified for changes in the values of this resource property. Any valid value for the `OperationMetrics` resource property is defined to hold three direct XML child elements, *i.c.* `NumberOfRequests`, `NumberOfFailedRequests` and `NumberOfSuccessfulRequests`. Considering the non-negative integer values of these metrics, the context manager can easily determine the number of pending requests as `NumberOfRequests` − (`NumberOfFailedRequests` + `NumberOfSuccessfulRequests`). The estimation of the load on any of the registered member services is always a

rough approximation, due to potential latency in the issuance and processing of the WS-Notification notification messages — *cf.* (D04).

8.4 Additional Contributions & Related Work

This section will cover related research activities that are not directly related or in line with the main research track reported throughout this dissertation. Nonetheless, these activities are worth mentioning, as they further substantiate the key statement raised in this chapter that WS-* specifications can aid in achieving proper separation of concerns while implementing reliable and autonomic distributed computing systems.

8.4.1 Reflective and Refractive Variables

Reflective and refractive variables were originally announced in [53] as an application-layer construct that can be used to implement feedback loops — *cf.* Sect. 1.2. Being an abstraction to perform concealed tasks, they are "volatile variables whose identifier links them with an external device", a sensor, or an actuator. Reflective variables are periodically and "asynchronously updated by dedicated service threads that interface [the corresponding] external devices" and/or sensors; the referenced values will therefore accurately *reflect* the values that were actually measured by those devices — within some reasonable delay. Note that reflective variables may also expose analysed and processed measurements. Furthermore, a modification of the value referenced by a refractive variable will be caught and interpreted as a request to trigger actuation and realise a change in the configuration of the corresponding external device or sensor.

Such model allows to clearly express potentially complex operations in the application layer in a structured way, while encapsulating and shielding the complexity of underlying application logic, communication protocols and hardware-specific details from the end-user. This translates not only in a strong separation of design concerns, but in enhanced maintainability as well [53]. Because of the additional layer of abstraction, the technique is well-suited to underpin the monitoring and execution activities on which feedback loops typically rely — *cf.* Sect. 1.2.

Taking this a step further up to the level of distributed computing systems, the author argues that reflective and refractive variables can be effectively implemented as WSRF-RP resource properties exposed using WSDM-enabled WS-Resources. In doing so, each variable — resource property — is formally described using a WS-RMD definition: refractive variables have read-write modifiability; reflective variables read-only. Both types are mutuable. Here, it is common to have one or more dedicated manageability capabilities that hold all logic for updating reflective variables and exposing them as resource properties. The same applies for reflective variables, and the specific logic that is responsible for realising state change and reconfiguration. Furthermore, all details for interacting with the underlying managed resource(s), are shielded within. Interested parties can be automatically notified whenever the values of a reflective variable — resource property — has changed by means of WS-Notification.

Additional contributions can be found in the design of a middleware solution in support of a distributed telemonitoring solution, developed within the scope of

the **Little Sister research project** reported in [142, Sect. IV][2]. The resulting solution includes the entire backend software infrastructure, including

(i) a modular C++ library designed to support the additional integration of the software routines and functionality supplied by the other academic partners participating to the project;
(ii) a comprehensive set of software routines for managing and using image sensors;
(iii) an integration layer built as a federation of WSDM-enabled web services, which is used to control the C++ backend components by means of a Java™ Native Interface (JNI) bridge, and to integrate the system in the applications developed and commercialised by the participating industrial partners; and
(iv) tools for runtime monitoring of sensor value and configuration changes.

In addition to these activities, the author extended the Apache MUSE project with support for the MoWS specification, thereby adding support for standardised, application-agnostic service-level and operation-level metrics and request processing state models, and for modelling and exposing application-specific status models.

8.4.2 Implementing Fault-tolerant Orchestration Logic as Workflows

With the advent of XML-based SoA, the OASIS® **Web Services Business Process Execution Language (WS-BPEL)** specification swiftly became a widely accepted standard for modelling business processes. As its name implies, WS-BPEL is a standardised executable XML-based language in which long-running business process activities can be described by orchestrating data flows, in which information is retrieved from and communicated to web service endpoints [143]. The specification is largely centered around the use of web services for process decomposition and assembly, aiming to maximally leverage key WS-* standards like SOAP and WSDL, as can be observed from Fig. 8.2 [22, Sect. 16.1].

A WS-BPEL process definition is a **self-contained, centrally-managed coordination routine** that formally describes the various interactions with and data flows between web services, controlling the sequences in which message exchange patterns occur, as well as intermediate data manipulation and transformation operations. Orchestration tools supporting the WS-BPEL language generally come with robust state persistence and lifecycle management features. This is needed not only because there may be activities that involve human interaction, but also because specific processes may need to be suspended, resumed or terminated. The WS-BPEL4People and WS-HumanTask specifications are recommended to be used for implementing activities that require human intervention [144–146]. As can be observed from Fig. 8.2, these specifications have been devised as modular extensions building on top of the core WS-BPEL specification, in an attempt to cover the complete spectrum of human-to-process interaction — interaction patterns that the core WS-BPEL specification did not originally foresee.

Whereas business processes are usually long-running, the underlying web services are assumed to expose software routines implementing short-lived, stateless business

[2] The objective of this project was to deliver a low-cost solution for ambient-assisted living, including ample features so as to sufficiently protect and assist the elderly. Funded by iMinds, the project was a joint research initiative of the universities of Antwerp, Ghent and Brussels, in collaboration with industrial partners Niko Projects, JFOceans and Seris Belgium, and supported by the Christelijke Mutualiteiten.

logic, although WS-Resources services are typically stateful, due to the underlying managed resources. Such **long-running transaction model** calls for adequate support for failure recovery in order to compensate parts of long-running business processes. Accordingly, WS-BPEL comes with a specific set of **synactical language constructs** so as to delineate scopes and attach compensation handlers that hold application-specific forward error recovery routines [147, Chapt. 4] — *v.* Sect. 1.3.2.

Despite the proliferation of WS-* specifications, and the availability of redimentary syntactical constructs for failure recovery in languages like WS-BPEL, XML-based SoA does not, in itself, contribute to the construction of dependable web services. The author has pointed out in [68, Sect. 3.4.3] though how these constructs can be used to implement redundancy schemata like NVP and RB, and how a clear separation of concerns can be achieved by isolating the actual business logic encapsulated within the underlying web services — versions, that are — from the dedicated fault-tolerant orchestration logic. For more information about this contribution, we refer to App. C.

8.5 Request Processing is Managed by Message-oriented Middleware

8.5.1 Effective Capacity Planning Calls for Realistic Queuing Models

"Servers must offer cost-effective and highly-available services in the elongated period" [148]. Adequate forethought should be spent on capacity planning, aiming to sustain service dependability by the reconciliation of the various objectives and expectations expressed by various stakeholders. From a functional perspective, there is a clear need for a scalable deployment infrastructure in order to be able to handle the expected load and safeguard the accessibility of the published services — *cf.* Sect. 1.1. From an operational and economical point of view though, "efficient policies that avoid over-provisioning are clearly desirable" [30].

Effective capacity planning and analysis require a sufficient degree of insight into a software system's structure and its inner working, a realistic view on the environment in which it will be deployed and will be operating, as well as estimations based on models that faithfully represent the rate at which incoming requests are actually received. This includes, but is by no means limited to, the type and bandwidth of the network communication infrastructure and the number of computing resources. Simulation is a vital technique for planning and analysing capacity. Once a simulation model is built using the knowledge obtained from the aforementioned details, this model can then be used to determine the extent to which the modelled system is capable of handling the anticipated load. This investigation usually involves the use of **queuing models**, which allow to zoom in on specific properties of the system itself and its environment, and to analyse how to maximise the system's **utilisation** — minimising idling time and avoiding resource over-provisioning — and **throughput** (processing capacity). The use of the Kendall notation to classify queuing models and to denote their properties was already covered in Sect. 6.1.1 on p. 91.

Although differentiated QoS levels may be desirable in commercial enterprise (cloud) environments, only few platforms actually enforce a policy- and tier-based service model. Rather than applying such priority-based (PNPN) scheduling discipline, servers are usually found to apply a simple **first come, first served (FCFS)**

policy, where service requests will be handled — serviced — in the order they arrived, and where they will be processed entirely without any intermediate interruption[3] [30]. Whenever a sufficient share of the processing capacity is released, the facility will start processing the request at the front of the waiting queue (if available). This type of scheduling discipline is usually chosen because it ensures fairness amongst the service requests that, upon arrival, were added to the waiting queue, and because it can be implemented with little complexity [118].

Throughout this dissertation, the processing resource is assumed to be a remotely deployed, network-accessible software component, *e.g.* a web service, which is referred to as version or replica **(A41)**. Assuming that there is sufficient servicing capacity available to handle requests in the long run, the model is said to be *stable* when subject to normal load[4]. Since requests "do not arrive at a constant, evenly paced rate, nor are they all served in an equal amount of time", a waiting queue will be "continually increasing and decreasing in length (and [will] sometimes [be] empty)" [149, Chapt. 16]. Intuitively, one could say that the queue is likely to grow when the facility is subject to a burst of requests, whereas it will shorten during periods of low demand or inactivity.

8.5.2 Application Servers and Servlet-driven Request Processing

Most message-oriented middleware solutions rely on an embedded application server for deploying business logic and exposing it to external parties. Such application servers are designed to manage the influx of requests in a controlled and systematic way, and to ensure each will be serviced in due time. This clearly indicates the presence of an underlying queuing model. A suitable model should represent the characteristics of conventional service endpoint implementation technologies. Not only do these technologies define ways to create entry points through which the service can be accessed; they also cover transport-specific details and suggest — at times implicitly — processing and scheduling policies.

Without doubt, the Java™ Servlet Specification has significantly contributed to the success of HTTP-based web service endpoints [101]. Most message-oriented middleware solutions are shipped with libraries to support servlet-based request handing and processing. Although most request processing is handled directly by the servlet runtime, the specification does outline a generic API that should be

[3] This is different from the **processor sharing (PS)** scheduling discipline, where each request will receive an equal share of the available processing capacity by applying techniques similar to time slicing, often realised in time-shared computer systems by means of round-robin scheduling algorithms. With PS, servicing of the request will immediately start upon arrival; there is no need to wait. The services time distribution will take the potential overhead that will surface due to context switching into acount, including the performance penalties resulting from lock acquisition and relinquishment delays of shared resources.

[4] Most queuing models have a limited processing and/or queuing capacity, which is usually the result from capacity planning prior to placing the modelled system in production. Given that the estimations used to approximate the pace at which load is arriving at the facility is realistic, the capacity of the queue should be sufficient to overcome transient bursts of incoming service requests. In such scenario, the queue length will allow to overcome the variations in arrival rate and the model will exhibit stable behaviour. However, the model cannot be expected to reliably serve unrestrained load, as one would expect to observe during denial-of-service (DoS) attacks. In such scenario, the queue would grow to become infinite over time, and, given the limited processing capacity, the system would suffer from (partial) interruptions in accessibility and therefore availability — *cf.* Sect. 1.1.

implemented to reach out to the actual business logic and to trigger the actual functional request processing. In reality, most message-oriented middleware solutions are shipped with a custom implementation of the servlet API. This eases the integration of custom business logic, requiring the developer only to register custom implementation classes by modifying configuration files, and is useful to trigger additional message processing and validation logic in due course — *e.g.* libraries in support of WS-* specifications. The servlet model has shown to be a robust and effective model for web services-based SoA, since web service requests are issued by POSTing SOAP messages as part of an HTTP message body.

Throughout this thesis, it is assumed that a queuing model underpinning a contemporary message-oriented middleware solution constitutes a **multiple-channel, single-phase process (A42)**. Waiting line structures are often categorised using the number of channels and phases that can be observed in the queuing process, where the number of *channels* is indicative of the available processing capacity to serve multiple requests in parallel — referred to as c in the Kendall notation [149, Chapt. 16]. There is only a single *phase* in handling requests, as any potential form of composition or orchestration of the underlying business logic is fully masked and unknown to the client (service requestor). As the primary scope of this thesis is the application of redundancy schemata in contemporary SoA architectures, we assume each underlying resource — web services acting as versions — corresponds to a software component containing a self-contained, short-lived atomic unit of business logic **(A43)**.

Many different application servers have been developed with the purpose of deploying and hosting (servlet-based) web service endpoints. It is common for these server programs to maintain a pool of standby threads: all share the duty of processing incoming requests. For the sake of simplicity, let us assume that a single pool of $c \geq 1$ threads is maintained (for each version), each of which can be used to serve a single request at a time. Each request runs independently and in complete isolation of any other. Requests are admitted for processing using an FCFS queuing discipline. Once a request has been completely processed, the allocated thread will be released, after which it can then be reallocated for serving another request. These system properties clearly imply that the processing capacity of each version is managed by a multiple-channel, single-phase queuing system. The worker threads correspond to identical server instances — *channels*, that are — that service requests in parallel. Once a request has been processed by a worker thread, it no longer requires further service (single-phase processing).

Most servers allow to easily adjust the size of the worker thread pool c and the waiting buffer k by means of a simple change in configuration, although the capacity of the waiting buffer is very often assumed to be infinite and constrained only by the available memory of the host. Not all requests will immediately go into service; some may temporarily be stalled and remain in the buffer, awaiting the availability of a worker thread in the pool. The maximum amount of worker threads that can be managed by an application server is constrained by the available memory, or by the operating system of the host, although recent advances in computer hardware and in distributed clustering technologies have allowed application servers to scale and measure up to virtually any demand. Hence, for the sake of convenience, we assume $k = c = \infty$. The properties of the queuing model employed by most

contemporary application servers can thus be summarised as a $G/G/c$ model, conform assumption (A07).

Due to the multi-threaded nature of servlet request processing, context switching will inevitably take place when multiple requests are being handled in parallel. Although it may appear as though there is hardly any waiting before newly arrived requests are treated, this does not imply a PS scheduling discipline. The primary characteristic of the model is that requests are admitted using an FCFS policy: requests are treated independently, and although they may be processed in parallel, no specific time slicing is applied.

CHAPTER 9

Conclusions & Future Research

Adopting classic redundancy-based fault-tolerant design patterns, such as NVP, in highly dynamic distributed computing systems does not necessarily result in the anticipated improvement in dependability. This primarily stems from the statically predefined redundancy configurations hardwired within such dependability strategies, *i.e.* a fixed degree of redundancy and, accordingly, an immutable selection of functionally-equivalent software components, which may negatively impact the schemes' overall effectiveness, and this from a dependability, a timeliness as well as a resource expenditure perspective.

Furthermore, we see a growing tendency to transform legacy enterprise applications into service-oriented and microservices-based architectures — architectural patterns which uphold principles such as contract-first, loosely-coupled (dynamic) composition and late binding. For business-critical applications, such transformation requires effective techniques to realise dependable fault-tolerant SoA "in terms of autonomic searching, discovering and selecting" candidate services in a robust and reliable way [87].

9.1 Key Contributions of this Research

In this thesis, a novel dependability strategy has been introduced encompassing advanced redundancy management, aiming to autonomously tune its internal redundancy configuration in function of the observed disturbances. Designed to sustain high availability and reliability, this adaptive fault-tolerant strategy may dynamically alter the amount of redundancy and the selection of functionally-equivalent resources employed within the redundancy scheme.

The publications through which the research reported throughout this dissertation were initially announced, have been occasionally cited in the literature, which show the relevance of our contributions in the domain of **fault-tolerant engineering of SoA solutions** [87, 88, 91–93]. We will conclude by listing the various contributions of this research, and by referring to the research questions that were initially raised in the introductory chapter (refer to pp. 31–34).

– The main contribution of the research reported in this dissertation can be found in the formal description of a **parameterised algorithm** that is **responsible for the adjustment of the redundancy configuration used in NVP redundancy schemata** in view of the environmental context in which the scheme is operating (**Chapt. 5**). From a dependability perspective, its design objective is to maximally sustain availability, whereas from a resource expenditure perspective it should focus on the exclusion of replicas that do not significantly contribute to the application's objectives and goals. The algorithm addresses the **research questions RQ-2 and RQ-5**.
  In [36], the authors propose a classification to categorise fault-tolerant solutions by analysing resilient behaviour as the result of four properties: perception, awareness, planning and dynamicity.

  ✖ Although our A-NVP algorithm is categorised as a resilient software system lacking perceptional abilities to *directly* detect environmental change,

  ✔ it is believed to possess the property of awareness.

  ✔ Unlike traditional NVP redundancy schemata, which are said to be only purposefully aware because they merely rely on fault masking, A-NVP is believed to attain a higher level of resiliency through "parametric and structural adaptation [of the underlying redundancy configuration]".

  ✔ Such reconfiguration is triggered by changes in metric measurements, which are used to *indirectly* monitor the scheme's environmental state. It is therefore classified as a system that is capable of purposefully planning.

  ✔ Its dynamicity is judged medium, as versions are dynamically (un)selected at runtime, unlike traditional NVP in which the redundancy configuration is static and determined at design time.

  Moreover, the resilient behaviour of A-NVP is categorised as predictive [150]. It is "identified as a **second-order predictive mechanism**, in that its behaviour is a response computed by correlating two perception dimensions: the overall *dtof* as well as the version's *trustworthiness* [— normalised dissent, that is —] with respect to the majority of votes" [36].

– In [93], the authors compared sixteen redundancy-based techniques for software fault tolerance — including our A-NVP model — by scoring each technique's ability to address various essential parameters for optimal behaviour, in terms of (i) adjudication, (ii) design diversity, and (iii) adaptiveness[1]. The authors acknowledge that **"A-NVP [...] added a significant value to conventional NVP with an added feature of configuration of quality and quantity of [versions] to participate [in a redundancy scheme]"**. Overall, our algorithm is ranked as the second best technique that maximally addresses all identified essential parameters. We believe this to be a meritorious place, **acknowledging the value of our contribution to the domain of software fault tolerance**, especially since A-NVP was confirmed to have "the highest score in [terms of] adaptive[ness]-related parameters".

[1] Here, "adaptiveness implies awareness about operating environment and mutating the technique to suit that environment", corresponding to the concept of system-environment fit used throughout this dissertation. Refer to [93, Table II] for specific properties of redundancy schemata that support the properties of awareness and planning.

– Another contribution is the **evaluation of the effectiveness of the proposed algorithm** in **Chapt. 7**, including a preliminary assessment of some promising policies for redundancy management, thereby addressing the **research questions RQ-3 and RQ-5**. We have shown that it is possible to realise an increase in dependability, or economise on redundancy expenditure without jeopardising the overall effectiveness of NVP redundancy schemata when applying dynamic redundancy configurations. The effectiveness of doing so depends on the applied policies for redundancy management, whose parameters should be configured to ensure there is a good system-environment fit. Suboptimal configuration that is not fully in line with the application objectives or with the environment's properties, may obviously result in a deterioration of the scheme's effectiveness.

– Another key contribution is to be found in the formal definition of a **mathematical structure that is capable of efficiently capturing how a specific version has affected the reliability of the fault-tolerant redundancy scheme throughout its operational life span.** This addresses **research question RQ-1**. This can be found in **Chapt. 3–4** and **App. B**. In addressing **research question RQ-3**, we have shown how the dependability of individual (software) components can be approximated by aggregating runtime information and statistics, and thus can be used to indirectly perceive environmental change, failures in particular. Normalised dissent measurements are used for steering the adjustment of the scheme's underlying redundancy configuration, and can be used to boost environmental awareness — provided that inbound requests hit the fault-tolerant composite at a constant, evenly paced and sufficiently high arrival rate.

– Discrete event simulations have been profitably used in many research disciplines — including dependability engineering — to conduct experiments to analyse the behaviour and performance of new algorithms and techniques, especially in the early phases of their development. They have become an indispensible tool for researchers to manage complexity, to try and find ways to solve issues and improve performance, by zooming in on specific phenomena and controlling environmental conditions that may otherwise complicate in-depth analysis. In response to **research questions RQ-2 and RQ-5**, we have developed a **discrete event simulation framework** that can be used to conduct rigorous performance analyses of policies for (autonomic) redundancy management within the scope of various types of redundancy schemata. As described in **Chapt. 2 and 6**, the framework comes with:

  (i) a wide range of artefacts and templates that support the designer in properly modelling the environment to a level of sufficient detail, and to define, implement, and examine the effectiveness of various strategies and policies for static and/or dynamic redundancy management;

  (ii) tools to exert control on the environment, its behaviour and properties whilst conducting large-scale experiments;

  (iii) an implementation of various predefined metrics than can **provide insight on the system-environment fit or mismatch**, which can optionally be extended at will to support other, custom-defined metrics.

– Furthermore, in **Chapt. 8**, we have shown how **autonomous redundancy management can be implemented as a well-separated concern** within the context of service-oriented architectures, thus providing an affirmative answer to **research question RQ-4**. Having scrutinised the self-managing capabilities that autonomic computing systems are expected to implement, and how these directly map to properties of resilient software, our contribution can be found in the design of a flexible implementation library to support the **development of fault containment units**. We have shown that it is beneficial to implement FCUs as managed WS-Resources, and have described our WS-*-compliant implementation of A-NVP. Building on top of the WSDM and WSRF specifications helps to isolate the implementation of autonomic capabilities from the underlying managed resources, and allows to expose a single, stable and consolidated service interface, where the common functional part is clearly separated from the standardised operations for steering the autonomous management (proper separation of concerns). Moreover, the WS-ServiceGroups managed by FCUs ensure the frictionless addition and/or removal of functionally-equivalent services, hence supporting key principles such as autonomic searching, discovering and selection of candidate services (versions).
NVP schemata are modular, component-based architectures *in se* — *cf.* Sect. 1.1. Implementing NVP as WS-*-compliant FCUs effectively realises n-version software execution environments (NVX), thereby meeting all of the requirements listed in [151]:

  ✔ **Independence of the program modules from the programming language** and **encapsulation**: the common functional behaviour implemented by each individual version (WS-Resource) is exposed through a formal service interface definition (WSDL). Technical implementation details are effectively encapsulated, and remain hidden.

  ✔ **Dynamic connection of the program modules**: the use of WS-ServiceGroups allows to replace individual WS-Resources — program modules or versions — at runtime.

  ✔ **Inter-module access (protection)**: when deploying web services on application servers, the application server will ensure their execution is constrained to a specific, well-isolated deployment and runtime context. Furthermore, the service is consumed uniquely through the service interface and the explicitly defined/exposed endpoints. Adding the distributed nature of service-oriented architectures to that, WS-Resource-compliant versions typically run in different processes or on different network hosts than the process where the NVP composite itself is running. Hence, the components in the architecture are not expected to interfere with one another.

  Our algorithm for autonomously adjusting the redundancy configuration has been observed by the authors of [91], who have analysed its properties and have acknowledged its ability to autonomously adjust the redundancy configuration when needed.
  Apart from a WSDM-driven implementation, we have also pointed out that some of the linguistic constructs available in WS-BPEL can be used to implement redundancy schemata like NVP and RB, and that such an approach can help

to achieve a clear separation of concerns by isolating the actual business logic encapsulated within the underlying web services — versions — from the dedicated fault-tolerant orchestration logic (refer to **App. C** for more information).

The key message conveyed in this section is that A-NVP is capable of realising a reduction in resource consumption/expenditure, while safeguarding the availability of the redundancy scheme as a whole. It does so by applying a dynamic redundancy configuration that is adjusted so as to exclude poorly performing versions, and by allocating a more suitable share of the available redundancy. This statement was corroborated by means of extensive discrete event simulation, and confirmed in [91] and [93]. In traditional NVP, however, the maximum number of tolerable disturbances that a given redundancy scheme can tolerate at a time is determined upfront, based on estimations, assumptions and analyses of the environment, and the available versions in the system. This applies especially to safety-critical systems, in particular in industrial settings, such as the control systems of a particle accelerator connected to a fusion target (*e.g.* the MYRRHA project[2]). Or one might consider a traditional nuclear power plant, in which safety relies on the availability of cooling systems. In that case, the exception is rare with respect to the rule, although of course its malfunction — unfortunately exemplified by the 2011 Fukushima incident — may lead to environmental disaster, human injury and/or significant monetary penalties. If worst-case dimensioning results in undershooting, or when wrong assumptions about the environment and the threats it poses to the system in consideration, there is a concrete possibility of catastrophic failure — *v.* Sect. 3.1 and [152, Fig. 3]. In such a case, proper use of the A-NVP scheme may result in excellent operational costs, where it can be used to optimise energy consumption, with no impact on safety[3].

9.2 Reliability Engineering: Over Four Decades of Research

Ever since its introduction back in 1977, the NVP methodology has inspired many researchers to investigate how it can be used to realise truly dependable software systems [67, Chapt. 2]. It has become a well-known architectural blueprint in reliability engineering, and has been successfully applied in mission-critical control systems found in nuclear power and chemical processing plants, aviation and aerospace, the military and waste treatment facilities. Over the past fourty years, many have studied how NVP can be successfully applied, and have assessed its

[2] For more information on the Multi-purpose hYbrid Research Reactor for High-tech Applications (MYRRHA) project, refer to `https://myrrha.be/myrrha-project/`.

[3] If we continue the example of a power plant and the availability of a cooling facility, one may argue that the cost resulting from redundant cooling installations could decrease, either because these installations should be less subject to wear, or simply because there will be less need for fuel to power all of them at the same time. So indeed, if one can guarantee that upon failure or suboptimal use of the "master" installation(s), one can swap these for backup equipment that is in stand-by mode within a reasonable and guaranteed delay, the safety and availability of the plant as a whole would not suffer. Of course, this depends on the risk appetite of the organisation, and the feasibility of a timely failover. Either way, at the very least, A-NVP would be able to observe subsets of resources — versions, in fact — that are underperforming, compared to the overall functioning of the replicas. It goes without saying that for HVAC systems, one would probably need to redefine the *dtof* metric to reflect more meaningful data, *e.g.* a delta between the temperatures of water recorded both at the ingress and the egress from the installation.

potential to significantly increase the dependability of software systems [93]. Since then, an overwhelming number of publications that have appeared in the literature, in which the technique has been applied to various emerging concepts and technologies. Research continues to date, where topics like fault prevention and/or detection, finding the optimal redundancy configuration at an improved cost-benefit ratio, and measuring design diversity are predominant [153–157].

Fault prevention/detection: developing dependable software. While it is true that introducing redundancy in software comes at a significant higher investment cost, no qualitative software can ever be delivered without some form of redundancy. Software engineering involves many steps and tasks that are to be carried out by humans. Humans that — regardless of their education, training and proficiency level — occasionally err, thereby introducing faults. In [158], the authors argue that no matter what modelling frameworks, development tools and verification processes are used or introduced during a project, the only option to further reduce the number of residual faults is to **introduce redundancy throughout the various stages of the software development lifecycle** (SDLC). That may include reviewing done at the level of requirements analysis, architecture and design, peer reviews during programming, and additional quality assurance by a testing team. However, all of these only *reveal* faults; they do not show their absence. For mission-critical systems, the only way to further reduce the likelihood of errors occurring is to introduce redundant software components, where multiple versions of a component with the same functional behaviour are independently developed by different teams.

Our research takes a different approach though, although we adhere strongly to the idea of introducing redundancy throughout the SDLC to ensure that high-quality software be delivered. Throughout this dissertation however, it is assumed that n functionally-equivalent versions have already been developed and have been deployed — in either production (regular operations) or pre-production (testing) environments. The presence of residual design faults — whose activations would materialise as disturbances (failures) — is likely to be detected by the majority voting algorithm, and would be accounted for in normalised dissent measurements. Further in-depth analysis of system logs may indicate the nature of these faults, and possibly their whereabouts. A similar approach is taken in [153].

Measuring design diversity. "Classic engineering approaches rely on different forms of redundancy explicitly added at design time, and suitably exploited at runtime" [154]. Examples include hardware, information and time redudancy, as well as software redundancy [2]. NVP rests on the assumption that "coincidental faults in independently developed components are very unlikely" — an assuption that has oftentimes been disputed. In this research track, researchers are trying to address and refute this critique. When using truly diversely designed versions in a redundancy scheme, the scheme would indeed realise its dependability objectives (*i.c.* increased reliability and fault-tolerant behaviour). But unless adequate precautions are taken to ensure that their design and implementation is truly diverse, applying NVP may lead to expectations not being met and, worst-case, result in catastrophic failure. A particularly accurate, albeit informal definition of redundancy was suggested in [155]: "two [code] fragments are redundant when they are functio-

nally equivalent [— they produce indistinguishable functional results from an external viewpoint —] and at the same time their executions are different" [154].

In [159], the authors propose a dissimilarity measure that can be used to analyse different code fragments and reveal the extent to which these fragments' executions differ algorithmicly. As such, the measure can be used to determine deep code differences from shallow code differences — the latter which would put at risk the effectiveness when both fragments would be part of versions used in the same redundancy scheme.

Furthermore, software redundancy can sometimes be achieved by carefully using third-party libraries and/or runtime environments, simply by using different software routines whose underlying implementation is different, yet produce identical functional results — this principle of intrinsic software redundancy was put forth in [154].

Finding the optimal redundancy configuration at an improved cost-benefit ratio. Developing truly dependable software solutions by means of software redundancy and fault tolerance introduces a significant additional cost that can easily add up to levels that cannot be justified given revenue forecasts, time to market, or the risk and the amount of financial losses against which the software should protect[4]. Recent research on NVP is largely centered around the following two problems:

(i) Given a set of versions, what is the best combination that will realise the application's objectives? Apart from dependability attributes, other attributes like timeliness and resource expenditure are also taken in consideration.

(ii) How can reliable software systems be produced at affordable costs?

A multi-attribute decision making model is used in [160] to determine which alternative redunancy configuration performs best, with a ranking obtained based on a combination of attributes, including, *e.g.* implementation and computing resource cost, memory usage and reliability (MTTF). The approach described in the cited publication seems quite similar to the redundancy dimensioning and replica selection models that were introduced in Chapt. 5, which clearly indicates that the research problem covered throughout this dissertation is actively being studied and is relevant in the domain of reliability engineering.
However, on second glance, the cited research differs from ours, in that:

– The ranking of alternative redundancy configurations is determined once, whereas our algorithm allows to re-evaluate throughout the system's operational life. In other words: ours has the advantage that it can periodically re-evaluate the cost-benefit ratio and adjust the redundancy configuration. However, our colleagues can cover the entire possible space of redundancy configurations, whereas in our solution, we cannot afford to solve such computationally intensive calculation without introducing significant latencies at runtime.

[4]For true mission-critical systems where the protection of the environment, the society or the operation itself is paramount, the additional cost would obviously be justifiable. Nonetheless, any approach that could result in a cost reduction would be welcomed, as long as it can be proved that system's dependability will not suffer from its application.

– From our understanding, the attribute values are assessed upfront, and are calculated once for one particular redundancy configuration (at the level of the NVP composite). In our A-NVP solution, such information is collected at runtime, at the end of each voting round, for each individual version, then consolidated so that it can be used for adjusting the redundancy configuration in subsequent voting rounds (feedback loop). We believe our approach is better suited to capture unforeseen exogenous factors, in which case another redundancy configuration may prove more effective.

– Both approaches foresee a means to compensate on resource expenditure, although the approach is quite different: in [160], this is a trade-off that is used in computing an acceptable, properly performing redundancy configuration, whereas in our model, the availability of the system is of primary order, and economising on resource allocation is of secondary order. Only after a prolonged period of redundancy overshooting, the system will assume it is safe to reduce the employed degree of redundancy.

9.3 Recommendations for Future Research and Practical Applications

9.3.1 Combining Different Techniques into an Enhanced Hybrid Solution

It was already pointed out that A-NVP was ranked as second best technique for software fault tolerance in [93]. In their final conclusion, the authors argue that the ideal solution would be a hybrid solution that would include "A-NVP for adaptiveness, **two-pass adjudication** (TPA) for diversity, and **acceptance voting** (AV) for adjudication". Although such solution remains out of scope, we will briefly cover its feasibility.

"The AV pattern is a hybrid pattern, which represents an extension of the NVP approach by incorporating [in] this approach [... the concept of ...] acceptance test [... commonly] used in the RB approach" [161]. Each ballot is presented to an acceptance test to determine if the output is reasonable; only accepted results will be used by the voting algorithm to adjudicate the final outcome of the scheme. In its basic form, the AV technique seems to assume the application of PV, but other voting algorithms like MV can easily be used as well. "The tests need not be as vigorous as those used in RB because of the presence of a voting [algorithm]. They are to serve as coarse filters so that clearly erroneous results are not presented" and therefore not taken into account by the voting algorithm [71, pp. 162–172].

As the name implies, two-pass adjudication includes two voting round *passes*: the first pass is fed with the original inputs; if that fails to adjudicate an outcome, a second pass is initiated, which is fed by re-expressed parameters [71, pp. 218–231]. If an outcome can be adjudicated at the end of the first pass, TPA is identical to normal NVP. Otherwise, a second pass is initiated and the original inputs are run through data re-expression algorithms (DRAs) to be normalised. A second voting round (with the same redundancy configuration applied) is then fed the re-expressed data as input, hoping this will allow an outcome to be determined nonetheless. A DRA is an algorithm that is used to transform the original input data sent upon invocation of a redundancy scheme. Rather than replicating the inputs, they can be preprocessed so that normalised and/or slightly rounded or rectified values are used for the subsequent version invocation requests. It is a form of data redundancy that

can be used for tolerating software faults, primarily in software implementations that are fed with noisy or imprecise data, or that include lots or arithmetic operations on floating-point numbers[5] [71, p. 21].

Although there is no impediment to extending our A-NVP algorithm and simulation toolbox to include the AV and TPA techniques, a few remarks are in place though:

– To include support for AV, it would suffice to add an additional state `validate ballot` in Fig. 2.2 immediately after the state `request handled`, just before the corresponding ballot will be processed by the voting algorithm. Or, put differently, this additional state should occur immediately before state `(c)` in Fig. 2.1. Ballots that do not qualify will simply be classified in $P_F^{(\mathcal{C},\ell)}$. Note however that acceptance tests are by nature very implementation-specific. Furthermore, not all functionality and implementations will allow to define effective acceptance tests to check the correctness of the results they deliver (ballot validation).

– A TPA-based redundancy scheme is essentially a *process* with one mandatory voting round (pass 1), optionally followed with an additional voting round (pass 2) in case the first pass was not able to successfully adjudicate an outcome. The inclusion of DRA is as simple as inserting an additional state `re-express input` between the states `request initialised` and `request sent` in Fig. 2.2. This would involve only a small additional layer on top of the existing simulation models, which in itself is quite easy to implement. However, there remain quite some open questions. In the second pass, should the same redundancy configuration be used as in the first pass? Should we just see both passes as two stand-alone voting rounds, or do we need redefine the normalised dissent metric to issue a single reward/penalty (at the end of whatever pass results in a valid outcome for the scheme? If we consider a TPA invocation as a single voting round, what measures should be redefined? And to what extent will we need to re-engineer the computation of measurements to consider these aggregated values (execution time, resource consumption *etc.*)? For sure, we would need to adjust the notation to identify individual version invocations into $\langle \mathcal{C}, \ell, p, i \rangle$, with $p \in [1,2]$ indicating a specific pass.

– Both techniques will introduce additional latency and runtime overhead; although negligible for AV, it will be considerable for TPA in case a second pass is required. Hence, any application consuming the service exposed and protected by a redundancy scheme should not be very latency-sensitive. For TPA specifically, the functionality implemented by the different versions should be idempotent (which is in line with our implicit assumption of stateless service implementations).

In Sect. 1.3.2.1, a reference was made to the hybrid approach published in [72] which combines NVP and RB. Although the approach bears great resemblance with our approach, in that runtime QoS measurements are used to determine, at

[5] It was in this context, where situations in which multiple (in)correct ballots would impede the adjudication of an outcome, that TPA was originally designed.

runtime, an optimal redundancy scheme and strategy[6], the computational footprint is significantly higher. Nonetheless, it would be useful to study to what extent part of the principles behind it could be incorporated in our A-NVP. Clearly, A-NVP is a **dynamic parallel strategy**; however, it does not foresee in a **dynamic sequential strategy** (which is a combination of a basic retry scheme and RB). It might also be useful to pursue the same idea of varying the voting algorithm used[7], and to vary it in time based on, *e.g.*, *dtof* and even RTT times.

9.3.2 Enhancing A-NVP's Proactiveness by Integrating Techniques to Anticipate and Prevent Failures from Occurring

In its current form, A-NVP is purely reactive, in the sense that it will only act once contextual knowledge was deduced at the end of individual voting rounds, and once a specific trend or risk was identified after correlation. Indeed, the algorithm will identify how many disturbances have been masked by the redundancy scheme, and will assess how effective the applied redundancy configuration was (and how trustworthy the underlying versions are). Although our experimentation shows that this approach is able to adjust the redundancy configuration, and to apply a redundancy level close to, though above, the contextual redundancy, additional techniques may **add proactiveness** to the approach and actually **prevent failures from occurring (or recurring)** and affecting the availability and effectiveness of the redundancy scheme at all[8].

Such capabilities could be added to the algorithm itself, or one could foresee an additional component as part of the A-NVP architecture. The former approach has several drawbacks:

(i) The additional complexity is likely to result in additional overhead and latency in determining the redundancy configuration, or at the end of the voting procedure itself. Furthermore, the additional implementation logic should not be intrusive, and should be isolated from the core A-NVP logic[9]. Furthermore, such logic may be very application-specific, whereas A-NVP was designed to be completely application-agnostic.

(ii) Such additional detection and prevention capabilities may be able to more rapidly detect trends or anomalies in the system. If embedded in the A-NVP

[6] The NVP-RB hybrid approach considers the time required before a response is obtained from versions, and estimations of the individual failure rates. It does not consider load distribution. Furthermore, it relies on prediction of RTT, a calculation that involves recursion. In case NVP is chosen as the most applicable redundancy strategy for a subsequent voting round, the model will predict the RTT for $n!/u!\cdot(n-u)!$ combinations, with $u < n$ and u odd, and n any $n^{(\mathcal{C},\ell)}$. Perhaps this procedure will not be triggered for every newly initiated voting round, but it should be repeated periodically and frequently to ensure the redundancy strategy is still effective.

[7] The approach taken in [72] is a combination of AV and PV, in which MV is applied to the u properly-returned responses. This could be added by introducing a threshold-base PV-like mechanism. However, an impact analysis remains to ensure the maximum accuracy of *dtof* measurements, and, by consequence, normalised dissent values.

[8] One may consider two levels of granularity here: one could focus on avoiding failure of the scheme itself, or one could focus on the (un)availability of individual versions.

[9] To achieve proper separation of concerns, it would be better to isolate different concerns into different architectural building blocks. As mentioned in Sect. 8.3, one could realise such separation by encapsulating each functional capability in a dedicated manageability capability.

algorithm itself, this would mean they can only be triggered at the start or at the end of each voting round, which may not leverage to the full extent the functionality such capabilities would bring. Housing these capabilities in dedicated components could support increasing the sampling rate to as little as few hundreds of milliseconds.

(iii) Additional diagnostic or detection capabilities will usually use a different type of knowledge base data, and are likely to monitor the system and its components on a periodic basis, trying to deduce meaningful conditions that might benefit from immediate or proactive intervention or system reconfiguration. The so-called diagnostic rounds are not likely to coincide with voting rounds, which depend on specific conditions like arrival rates, service times *etc.* [5, 77].

Because of this, it is recommendable to add such additional logic to the system by encapsulating it within one or more dedicated software component(s). However, what additional capabilities may be useful to include? And how can they improve the overall dependability and timeliness of the scheme and the underlying redundant resources? A few potential candidates are listed below:

– Versions can be invoked periodically, a technique often referred to as **probing**. This is typically combined by some kind of soundness check to check if the response is valid[10] (very much comparable to the concept of an acceptance test in RB). The technique will result in additional background load, which may affect the selection of versions by the replica selection algorithm — *v.* Sect 2.7, 5.1 and 5.3. Periodic polling by dedicated software components may help to assess the responsiveness of individual versions, as well as their availability. It could be used to more proactively detect potential crash failures, *e.g.* when observing an increased recurrence rate for intermittent failures. For example, after observing x successive failed trials, one could mark the version as untrustworthy, and temporarily exclude it from the redundancy configuration used for subsequent voting rounds. Probing should continue in the background, to evaluate if the failure behaviour is intermittent or eventually resulting in crash mode, in which case the version should be permanently excluded/disabled. This technique is very similar to the work of [114, Fig. 1], where faulty versions are isolated and re-integrated upon recovery.

– The flexibility and velocity of deployment in virtualised (cloud) environments allows to take techniques like **chaos engineering**[11] and **recovery-oriented com-**

[10] Various options are available here: in microservices-based or service-oriented architectures, a simple check of the HTTP response might be sufficient to indicate failure (response codes in the 20x range may indicate success, whereas the 50x range may indicate failure). In specific scenarios, some logical validation of the actual response value or payload may be more suitable.

[11] "Chaos Engineering is the discipline of experimenting on a [distributed computing] system in order to build confidence in the system's [ability] to withstand turbulent conditions [and unexpected conditions] in production" — definition as per `https://principlesofchaos.org`. The basic idea is to formulate a hypothesis about the system's steady state, and — in a way similar to reflective variables — to define some measurable measures to reflect wellbeing or aberrant behaviour (potentially after aggregation and/or analysis) — *v.* Sect. 8.4.1. This can reveal anomalies at various layers, ranging from disruptions of the service at application layer, malfunctioning hardware, unusual workload patterns *etc.*

puting to another level and to rapidly and **proactively rejuvenate** whenever a drop in performance or trustworthiness is observed (by deploying a new *replica* and undeploying the previous instance) — a technique used in [63, 162]. This highly relies on the assumption that software components (deployments) are reliable in the early phases of their operational life, and that there is some kind of **software aging** over time — a phenomenon confirmed in [45, Sect. 11.8].

– In [163], the authors point out that such **fidelity drifting** may occur at random times, but that in some cases, it is hard-bound and guaranteed to occur after a certain time in operational mode [31]. As an example, they refer to the incident with the Patriot missile-defense system during the Persian Gulf War, which failed to reliably track and intercept an incoming hostile missile, thereby causing unneeded death and severe injury to several dozens of people. Therefore, the detection of gradual behavioural driftings, if any such technique can be devised, is indicative of a drop in trustworthiness, for which rejuvenation might help as a compensatory control.

– Another approach would be to rely on **machine learning (ML) and other artificial intelligence (AI) techniques** that can help to understand and **anticipate trends**, *e.g.* patterns of disturbances that are about to occur, or periods of system-environment (mis)match. In the assumption that such techniques would be effective and reliable, they could be used to classify the next x time slots (*e.g.* voting rounds), and see if a chosen redundancy configuration is expected to properly mask the disturbances of the environment. Such analysis would best run in isolation of the core A-NVP logic, to avoid additional latencies during the selection of the redundancy configuration and the voting procedure[12]. However, the predicted outcomes could be used while determining the redundancy configuration — mainly from a quantitative, but also from a qualitative point of view — *v.* Chapt. 5. For instance, if a prediction indicates a risk of undershooting — failure, that is — one could contemplate more aggressive upscaling to prevent service disruption and/or unavailability. Reversely, one could use the prediction as an additional confirmation that it is safe to scale down resource consumption and, in doing so, prevent unneeded resource and power consumption.

9.3.3 Reducing A-NVP Computational Overhead

Although A-NVP was designed in such a way to avoid significant impact on end-to-end response times, one could wonder if the actions of determining the redundancy configuration and of voting may contribute to a non-negligible latency, thus in case assumptions (A37) and (A38) would be violated. One could record the actual latencies for voting procedure (including the extraction of context information and the updating of the windows of context information), as well as for the computation of the redundancy configuration — *v.* Fig. 5.1, p. 82. In addition, A-NVP could be extended with an upper bound on the overall response time for individual voting

[12] Of course, there might be a need to share contextual data between A-NVP and these additional ML/AI modules: the actual data will likely be harvested during the voting procedure (or by additional modules with dedicated diagnostic and/or detection capabilities). How that data is shared remains undefined and should be evaluated based on the characteristics of the system: it is likely shared memory will be preferred in case of embedded systems.

rounds, unlike the upper bound t_{max} set on the end-to-end response times for subordinate version invocations. By doing so, hard time limits could be enforced for voting round processing, which would include the latencies resulting from the algorithm itself and the voting procedure.

Although very unlikely, if one would notice that the algorithm would cause non-negligible overhead (and therefore occasionally translate in performance failures at composite level), one could consider several optimisations:

– One could accept the algorithm in its current form, yet reduce the frequency with which the redundancy configuration is computed. By default, it will be determined at the start of every subsequent voting round. Yet, one may consider to only **aperiodically update the configuration**. Examples could be once every x voting rounds. Obviously, such approach will increase the risk of undershooting, as rapid changes in the amount of failure occurrences may not be considered in due time to realise a swift and decisive reconfiguration of the allocation of redundant resources.

– One may also **update the redundancy configuration on a periodic basis**, *e.g.* every x seconds. That would require moving the logic for the redundancy dimensioning and replica selection models into some sort of scheduled task — *v.* Sect. 5.2 and 5.3. The same drawback as before would apply, in that in some cases — especially in cases of elevated load — the approach might not be able to upscale or select a more appropriate replica selection in time. The knowledge base, also known as the window of contextual information, will be shared between the core (A-)NVP logic (encompassing the logic for the voting algorithm, including extraction of context data and populating records in the knowledge base) and the entity containing the logic for determining the redundancy configuration, and scheduling that activity (in fact the consumer of that context data).

– For this second option, one could simply share the same data structure, or one could also isolate the knowledge base in a separate component. During voting, once context data has been deduced, it can be pushed to this additional dedicated component, typically by means of a robust message queue solution. Note that the adjudication procedure (voting algorithm) by its very nature is an integral part at the end of any voting round, and this latency must be absorbed in any time constraint defined at the level of the redundancy scheme.

Note that some overhead, most particularly queuing delays, will stem from the architectural choices made during the design of the system. Throughout this thesis, the focus was set to distributed applications, which is reflected in assumption (A07). Because of this, and the assumption of using modern MoM platforms in the software platform layer, little or no waiting times should be expected. However, in resource-constrained environments, or in case of embedded systems, similar workloads would probably result in more requests being stalled in the waiting buffer (unless the workload pattern would be different, *e.g.* periodical, evenly-paced arrivals of requests).

As a final note, in specific cases, one could potentially **vary the voting algorithm** used at runtime. Such approach was foreseen in [72], where AV and MV voting algorithms are used, depending on the predicted response times. Although this may

help to reduce latencies and overhead, it will usually increase the risk of inaccurate or erroneous outcomes being adjudicated. The impact of different voting algorithms is illustrated in Fig. B.1, 226.

9.3.4 Safeguarding A-NVP's Availability by Tolerating Failures in the Underlying Platform Layers

In Sect. 1.3.2.2, we elaborated on synergistic approaches in which multiple error mitigation and resilience techniques are applied at different abstraction layers to defend against (functional) errors occurring and to prevent them from propagating to higher-level layers. Layers that are commonly observed in the design of digital systems include the hardware platform layer, the underlying devices or electronic circuitry, the platform software stack (including operating system, runtime engines and/or interpreters, and middleware solutions), and the actual application layer [42, Fig. 1]. Of particular concern here is to ensure that the core A-NVP functionality — residing in the upper application layer — will remain reliable, and that its availability and integrity is not affected by failures affecting the lower abstraction layers. Specific error mitigation techniques can be applied at different layers to prevent failure propagation, allowing to abstract away the details of lower layers and assuming these layers will perform as expected.

Apart from that, in a context of embedded systems, it is conceivable that specific failure classes affecting, *e.g.* the hardware platform layer or dedicated circuitry, would require a very **fast reaction time** to mask such failure occurrences (and thus contain them and prevent them from propagating to higher abstraction layers). Unfortunately, A-NVP will only act during the initialisation of a new voting round, by selecting an adequate redundancy configuration and excluding suboptimal or unreliable versions. With (A-)NVP relying on fault masking by means of an adjudication (voting) algorithm, such redundancy schemata will not be able to swiftly and decisively act to recover from the described types of failures classes. In the end, it will be the workload characteristics that will determine when the redundancy configuration may be next adjusted. So for the described failure classes, additional **specific hardware support** would be needed to **detect and contain the disturbance and compensate its** detrimental **effects**.

Another approach could be to include auxiliary dedicated entities for **fault monitoring** to detect disturbances as soon as they occur, so *during* voting round execution, and not at the end when the voting procedure is triggered. This could even result in a more accurate view on how well versions behave, and by how many disturbances they are struck[13]. Such additional diagnostic capabilities could then be used to more accurately **detect/predict faults**, the rate with which they (re)occur, and potentially also the specific conditions under which they occur. They would allow to exclude faulty versions more rapidly, without the need to await the completion of the voting round during which those disturbances were triggered. In doing so, we eliminate the dependency on the workload pattern (determining arrival and duration of new voting rounds).

[13] Note that so far, because a disturbance is deduced from dissentient or missing ballots, the algorithm will be aware only of a single failure, more specifically the failure class of highest failure severity — *v.* Fig. 2.3, p. 45. For instance, if a version is affected by an RVF followed by an EVF disturbance, only the latter will be detected during voting.

9.3.5 Including Context Parameters and their Causal Relationship Towards Failure

The notion of context has been used so far to refer to attributes reflecting how well a redundancy scheme is functioning, with respect to the environment in which it is operating. This includes various measures to assess the system-environment (mis)match — including, *e.g.* normalised dissent, nett redundancy *etc.* — as well as a handful of operational measures that reflect load- and timing-related performance characteristics — *v.* Fig. 5.1.

However, it might be useful to consider context parameters for **specific operating conditions** that could in their own right affect the functional and/or parametric reliability of the system and the underlying components. Examples could be found in temperature and humidity levels. Specific ranges of values could be identified that could potentially significantly increase the **risk of malfunction**. Obviously this would be applicable to embedded systems that would include dedicated circuitry. At a higher level, an example could be rain- or snowfall in the context of autonomous vehicles, rendering specific sensors or algorithms — versions — less accurate or even unreliable. In such situations, the use of alternative sensor types may be considered, and it may even be useful to use different sensor (types) in parallel to **filter out inaccuracies or noise**, albeit at the expense of increased power consumption [164]. If the causal relationship between specific context parameters and potential failure occurrence could be discovered during the design phase, the overall dependability of the system would benefit from keeping track of these parameters by recording measurements and using them to adjust the system whenever needed or recommendable.

So how then should the A-NVP framework be enriched to enhance its teleological behaviour? Because of the nature of these additional context parameters, measurements will likely be **sampled periodically**. Because of this periodic sampling, it is recommendable to include an additional component whose sole responsibility would be to keep track of and measure these environmental context properties[14].

Changes in this perceived environmental state — the context in which the redundancy scheme is operating, that is — can be used to trigger **reconfiguration** of the underlying redundancy configuration. It can also be used to trigger specific reconfiguration of individual components, if it is possible to adjust their configuration at runtime (*e.g.* to temporarily accept a lower degree of accuracy, but to prolong service availability and avoid downtime as a whole)[15]. This could be done as part of the replica selection procedure (and if the redundancy level would be affected, the redundancy dimensioning model as well). Or, one may act directly from the dedicated module. This last option seems to be the preferred way, since the core A-NVP algorithm is completely application-agnostic, and this type of actuation rules/logic are typically very specific to the application/system itself.

[14] Note that A-NVP relied on *apperceptional* abilities to interpret the context properties listed throughout Chapt. 5. For context parameters like temperature and humidity, the operating conditions could be *directly* sampled, which adds true *perceptional* abilities.

[15] In the classification of resilience techniques proposed in [42], this approach would correspond to the concepts of internal functionality reuse and operating conditions control (in case of platform software, respectively hardware). The basic idea is that by adjusting some parameters of the platform, a **more resilient execution** can be achieved.

Note that in some cases, it might be useful to **vary the voting algorithm** throughout time. For example, to filter out outlier ballots (and thus noise), a technique called weighted voting was used in [164]. Or it could be useful to temporarily activate some normalisation and validation logic to **preprocess the acquired ballots** before feeding them into the voting algorithm — very similar to the use of DRAs applied to input values (request message payloads, that is).

9.3.6 Potential Next Steps that Remain Unexplored

Although we believe to have evaluated the proposed A-NVP algorithm well, and ran a sufficient amount of discrete event simulations to corroborate our statements, it may be worth pursuing this research and zoom in on a number of open research questions that remain:

- The research on autonomous redundancy management in NVP-based redundancy schemata should be continued: **new policies** should be devised and their effectiveness should be analysed. We believe we have paved the way to ease future development and experimentation by implementing an extensible and modular discrete event simulation framework.

- Further experimentation on the first set of policies presented throughout Chapt. 7 would be useful, particularly to assess their effectiveness under **various failure occurrence and load patterns**. Another potential topic could be to investigate how a dynamic safety margin may help to improve the scheme's resilience when submitted to a deployment environment that exhibits graver whimsicality. Detailed fault models reflecting the internal structure of a version's programming logic and composition of software components can further help to zoom in on real-world failure occurrence patterns.

- As pointed out before, it remains extremely challenging to properly model (design) faults, both in terms of manifestative behaviour as well as occurrence patterns. Consensus cannot be found in the literature, nor a common agreement or **empirical evidence on failure (occurrence) rates**. All of this is aggravated by an ever increasing complexity of (software) systems. Even though one can find promising work to characterise failure behaviour in [123, 127], few (recent) publications can be found to approximate the amount of residual design faults in software implementations.
  The lack of clear guidance here have forced us to formulate a set of assumptions and to conduct discrete event simulation to verify the effectiveness of A-NVP. One might be interested to further investigate two things here:

  (i) First, in order to further substantiate the findings reported in Chapt. 7 and corroborate our claims about the effectiveness of A-NVP, one could consider **emulation** by means of the implementation described in Chapt. 8. One could programmatically inject software disturbances (*i.e.*, a hybrid emulation/simulation model), or simply purposefully introduce design faults in the source code for the functional behaviour of the application. Either way, if available, one could use representative **failure occurrence data** and emulate the failure behaviour one would expect to see in a specific

target deployment environment. One could deploy this implementation in a cloud environment, similar to the approach taken in [165], to obtain realistic end-to-end response times and network connectivity issues (*e.g.* congestion or varying RTT times). Such approach could be used as an **experimental validation of our simulation results**, and to confirm that the distributions used throughout our experimentation match reality (or at the very least to discover more representative distribution parameters). This type of emulation will also be able to validate the assumptions made with respect to the processing capacity and queuing model of the underlying MoM — *v.* Sect. 8.5. Emulation does come with specific drawbacks compared to discrete event simulation: no strict control can be exerted on specific environmental parameters — something that was already pointed out in Sect. 1.4.

The approach described above may also be used to collect real-world data that can be used to feed into our discrete event simulation framework — a technique commonly referred to as **profiling**. The framework could be extended to inject disturbances based on the failure occurrences detected in a production environment, and to reflect the actual end-to-end response times. And although collecting experimental data that is gathered during emulation will be more representative of the target deployment environment, one would have to be willing to take the risk and put the system in production, which in itself incurs risk. One could consider to profile the behaviour of the system in pre-production environments, but oftentimes, the workload would be completely different from what would be observed in production. Apart from that, the acquired data could be used to calibrate the simulation model, and to determine the root causes that lead to suboptimal performance during specific intervals.

(ii) Second, the discrete event simulation framework described in Chapt. 6 was based on the Stochastic Simulation in Java™ (SSJ) library developed at the Université de Montréal. That library was chosen because it provided a robust foundation on top of which specific (and at times complex) models can be developed. Other, more widely accepted **simulation tooling** that does not seem to offer the same flexibility or freedom of doing so, which would impose tool-specific boundaries and limitations.

Reliability and performability assessments of NVP redundancy schemata have been successfully performed by simulation based on generalised stochastic **Petri nets** (GSPN), which support to simulate timed transitions that fire after a random delay sampled from exponentially distributed random variables associated to each specific transition [166–169]. Even though it may be technically feasible to define the models defined in Chapt. 2 as GSPNs, the assumption of using exponential distributions would be limitative and would contradict research stating that realistic service and response times are more precisely characterised by other distributions [30, 45, 165]. An extension can be found in **stochastic activity networks** (SAN) that allow to define alternative distributions and that offer a more practical high-level language for modelling system behaviour[16] [170]. But

[16] SANs have been used to evaluate the effectiveness of the α-count approach proposed in [77], a technique that inspired the normalised dissent measure defined in Chapt. 4.

even then, it would be extremely challenging to try and define all aspects of our discrete event simulation models into a combination of SANs, and this would become even more challenging if we would include other failure classes (*e.g.* hardware or network failures). Either way, it should be technically possible to translate A-NVP into a SAN model.

– We have demonstrated that an A-NVP redundancy scheme can take a snapshot of the environmental conditions and act to ensure and/or regain a normal operational state, thereby regaining or further sustaining the scheme's dependability. There is a whole spectrum of environmental conditions, and only one scenario was explored where design faults materialise. As per the previous remark, fault models can be explored that are highly likely, but also very unlikely (when analysing the behaviour in exceptional situations).
  Moreover, one could also study the system behaviour from a **cybersecurity** point of view. As an attacker would exploit system vulnerabilities (which include more than just design faults), (s)he would be able to force the system to become unavailable. In this case, one would also have to try and identify the environmental conditions and the evolution thereof that are representative of advanced attacks.

– Throughout this thesis, the emphasis has been placed on the effect design faults have on the dependability of redundancy schemata. The **impact of disturbances that affect the underlying network and hardware infrastructure** can easily be analysed by implementing the relevant models using our simulation framework. Likewise, if, for any reason at all, the reader would disagree with any of the assumptions listed, the default artefacts can easily be adjusted to analyse the effect of such change.
  It would also be worthwhile to study the potential impact of hardware faults, *e.g.* electrical failures, on the functioning and availability of the software running on top, and to zoom in into these types of complex fault models. Such analysis would become even more important in the context of embedded and real-time systems, where the complexity of the hardware technology cannot easily be abstracted or decoupled from the firmware/software running on top of it[17]. One would also want to analyse how specific workload patterns may lead to a gradual build-up of parametric failures that may exacerbate the reliability of the underlying electrical circuits, and how that may (in)directly translate into disturbances in the hardware and software layers [56, 171]. Any new insights in these complementary research domains might suggest additional potential techniques for mitigating the detrimental effects on system reliability, whereby specific protections could be introduced at different layers.
  As a lot of research has already been done on **analytical models** to approximate hardware fault models [3]. A valid research question that most definitely is worthwhile to investigate is if accuracy would benefit if these models were to be **combined with runtime methods** like our approach (**for calibration** and profiling).

– It will be considerably harder to **apply autonomous adjustment of the applied redundancy configuration in RB-based redundancy schemata**, for the norma-

[17] For software applications, the hardware would be mostly abstracted away by the operating system, development tools (like compilers/interpreters) and/or supporting software libraries.

lised dissent measurements will be updated less frequently. This is because RB will invoke the underlying versions in sequence, rather than in parallel, which obviously leads to a significant reduction in version invocations. We expect it will be much more difficult to devise solutions that are meaningfully effective to realise similar objectives. However, researchers can build upon our framework to model software components, to implement the fault-tolerant logic underpinning specific RB-based redundancy schemata, and to implement novel policies and algorithms and analyse their effectiveness and performance.

– The approach to *perceive* a change of the environment — including disturbances affecting specific versions — indirectly through changes in measurements for the normalised dissent may be enhanced to improve the actual *awareness* of the redundancy scheme, and various techniques for failure detection may be considered [36, 98, 99, 172].

– Throughout this thesis, we have considered a relaxed form of majority voting, mainly to reduce the average response times recorded for NVP invocations, and to lessen the amount of required computing power [7]. For those interested in the matter, it may be useful to investigate the effects of **applying an alternative voting/adjudication mechanism**, both in terms of dependability and timeliness, and — indirectly — the impact this change may have on resource expenditure.

– The **use of machine learning (ML) and other techniques for artificial intelligence (AI)** to optimise the redundancy configuration used by NVP redundancy schemata has not been explored to date, nor was it part of this research. If we would be able to rely on ML/AI techniques to reliably anticipate how the environment will evolve in the near future, our algorithm could proactively and aggressively upscale when needed, or prepare a parsimonious, yet safe allocation of system resources. Or, we might be able to use ML/AI to self-assess how effective the selected redundancy configurations were in the context of specific properties observed as the current system-environment fit, and use that to anticipate what (change in) configuration might be beneficial. In fact, it is the only property that is missing in order for A-NVP to qualify as a **computationally antifragile** system [90, 150, 163]:

  ✔ it is "able to exercise teleological behaviour [— meaning that a feedback loop is involved —] that evolve the system and its identity in such a way as to *systematically* improve the fit with [the] environment" in which it was set to run [90];

  ✔ the model aims to keep an accurate view on and is aware of the current system-environment fit, based on *dtof* and normalised dissent measurements;

  ✔ in cases where the environmental behaviour is stable, the algorithm may gradually adjust the applied redundancy configuration so as to improve the overall system-environment fit;

  ✖ although capable to assess how effective a specific redundancy configuration was at a given point in time, the algorithm does not apply any real form of (machine) learning — although it can steer the redundancy allocation process towards intervals of elastic (maintain specific redundancy level),

entelechial (strive towards a certain level of fault tolerance) or antifragile (steer the system towards an optimal system-environment fit) planning behaviour;

✔ contextual data is cached and metrics are computed and tracked to ensure the observed behaviour can be interpreted and used in subsequent voting rounds to optimally sustain the scheme's non-functional requirements (availability in particular) — *v.* Sect. 5.1.

However, if it were technically possible to meaningfully apply ML techniques to further enhance re-configuration planning, one may expect to see a non-negligible impact on the scheme's overhead, resulting from markedly higher CPU and memory consumption.

– Nonetheless, "ML systems can [...] benefit from [...] a multi-version approach", with research being reported in which multiple ML models are combined within an NVP scheme, mainly to improve system reliability, but especially to improve prediction accuracy and to overcome perturbations that might lead to corruption of input data [157]. The main applications can be found in the research on autonomic vehicles, and include use cases such as steering angle prediction and stop sign recognition [157, 164].
In [164], a technique referred to as weighted voting scheme is introduced. As in regular NVP, its purpose is to improve the overall robustness by tolerating and masking occasional failures. It includes an innovative mechanism to (i) detect and exclude outlier ballots from the voting procedure, and (ii) to adjudicate outcomes with a higher accuracy. The outcome itself is computed based on the acquired ballots, in such a way that the differences between ballots observed in prior voting rounds is taken into account.

APPENDIX A

Auxiliary Lemmas

Theorem A.1 For any voting round $(\mathcal{C},\ell)$ during which the employed redundancy configuration $V^{(\mathcal{C},\ell)}$ proved to be cr-dependable, an additional amount of $\lfloor (n^{(\mathcal{C},\ell)} - cr(e^{(\mathcal{C},\ell)}))/2 \rfloor = dtof^{(\mathcal{C},\ell)} - 1$ disturbances could have been tolerated.

Considering the definition of the function $cr : \mathbb{N}_0 \mapsto \mathbb{N}^+$ introduced in Sect. 3.1 as a means to quantify the contextual redundancy, and the notion of cr-dependability introduced in Sect. 3.1.3, several presuppositions can be stated to hold:

(a) For a redundancy configuration $V^{(\mathcal{C},\ell)}$ to be cr-dependable, implies that, throughout the course of the voting round $(\mathcal{C},\ell)$, it was subject to a number of disturbances $e^{(\mathcal{C},\ell)} \leqslant n^{(\mathcal{C},\ell)} - m^{(\mathcal{C},\ell)}$, such that the exact result is obtained from the existence of a unique consensus block $P^{(\mathcal{C},\ell)} = b_0$ constituted by a majority of consentient replicas — *cf.* (A30) Sect. 3.1.

(b) Recall that $d^{(\mathcal{C},\ell)}$ was defined as a measure so as to intuitively represent how many of the versions in $V^{(\mathcal{C},\ell)}$ returned either a syntactically invalid response, or a response that differs from the majority, if any such majority could be identified at the end of round $(\mathcal{C},\ell)$ — *cf.* Sect. 3.1. The premiss above excludes the adjudication of an incorrect result, as would have been the case had $e^{(\mathcal{C},\ell)} \geqslant |V_{rvf}^{(\mathcal{C},\ell)}| \geqslant m^{(\mathcal{C},\ell)}$. Given these conditionalities, one can see that $d^{(\mathcal{C},\ell)} = e^{(\mathcal{C},\ell)}$, for $d^{(\mathcal{C},\ell)}$ was defined as $n^{(\mathcal{C},\ell)} - c_{max}^{(\mathcal{C},\ell)}$, and $c_{max}^{(\mathcal{C},\ell)} = |P^{(\mathcal{C},\ell)}| \geqslant m^{(\mathcal{C},\ell)}$.

(c) Finally, a redundancy configuration $V^{(\mathcal{C},\ell)}$ is cr-dependable if and only if $n^{(\mathcal{C},\ell)} \geqslant cr(e^{(\mathcal{C},\ell)})$, which in itself implies the adjudication procedure outlined in (a), as well as $dtof^{(\mathcal{C},\ell)} \geqslant 1$, in line with Eq. 3.2 — *cf.* Sect. 3.1.

In regard to the premisses mentioned hereabove, one can now simply determine the corresponding values for the measures $\lfloor (n^{(\mathcal{C},\ell)} - cr(e^{(\mathcal{C},\ell)}))/2 \rfloor$ and $dtof^{(\mathcal{C},\ell)} - 1$[1], for integral values $n^{(\mathcal{C},\ell)} \geqslant c_{max}^{(\mathcal{C},\ell)} \geqslant m^{(\mathcal{C},\ell)}$ — *cf.* (b):

[1] Note that this second measure can be easily computed as $c_{max}^{(\mathcal{C},\ell)} - m^{(\mathcal{C},\ell)}$, for Eq. 3.3 is applicable in view of (b) and (c).

$c_{max}^{(\mathbb{C},\ell)}$		$d^{(\mathbb{C},\ell)}$		$cr(e^{(\mathbb{C},\ell)})$		$\lfloor (n^{(\mathbb{C},\ell)} - cr(e^{(\mathbb{C},\ell)}))/2 \rfloor$
7	→	0	→	1	→	3
6	→	1	→	3	→	2
5	→	2	→	5	→	1
4	→	3	→	7	→	0
7	→	4	→	3		
6	→	3	→	2		
5	→	2	→	1		
4	→	1	→	0		
$c_{max}^{(\mathbb{C},\ell)}$		$dtof^{(\mathbb{C},\ell)}$		$dtof^{(\mathbb{C},\ell)} - 1$		

Table A.1: Illustrating agreement between both measures for an odd degree of redundancy $n^{(\mathbb{C},\ell)} = 7$ ($c_{max}^{(\mathbb{C},\ell)} \geqslant m^{(\mathbb{C},\ell)} = 4$).

Comparing the quantities obtained from both measures confirms the correlation between either of these two measures, for eligible values $c_{max}^{(\mathbb{C},\ell)}$. We will now elaborate on this correlation and provide formal proofs in case of an odd or even amount of redundancy $n^{(\mathbb{C},\ell)}$.

Proof (for $\mathbf{n}^{(\mathbb{C},\ell)}$ odd):
Let us start with the observation that the notions of $e^{(\mathbb{C},\ell)}$ and $d^{(\mathbb{C},\ell)}$ coincide, as put forth in premiss (b). Multiplication by –1 then allows the sign to be flipped in both sides of the initial equation, after which $m^{(\mathbb{C},\ell)} - 1$ is added to both sides. Applying some rewriting in the right-hand side, we obtain $dtof^{(\mathbb{C},\ell)} - 1$ in (A.1.2):

$$
\begin{aligned}
e^{(\mathbb{C},\ell)} &= d^{(\mathbb{C},\ell)} \\
-e^{(\mathbb{C},\ell)} &= -d^{(\mathbb{C},\ell)} \\
\underline{(m^{(\mathbb{C},\ell)} - 1)} - e^{(\mathbb{C},\ell)} &= \underline{(m^{(\mathbb{C},\ell)} - 1)} - d^{(\mathbb{C},\ell)} && \text{(A.1.1)} \\
&= (m^{(\mathbb{C},\ell)} - d^{(\mathbb{C},\ell)}) - 1 \\
&= dtof^{(\mathbb{C},\ell)} - 1 && \text{(A.1.2)}
\end{aligned}
$$

Next, the underlined minuend of the subtraction in the left-hand side of (A.1.1) is rewritten by substituting $m^{(\mathbb{C},\ell)}$ by its representative value, as defined in Eq. 3.1 in Sect. 3.1. Immediately after, the right-hand side of (A.2.1) is rearranged by using the proposition that states that $\lceil x \rceil + n = \lceil x + n \rceil$, for any integer $n \in \mathbb{Z}$ and $x \in \mathbb{R}$, as stated in [173, Eq. 3.6a]. Considering that $n^{(\mathbb{C},\ell)} - 1$ is an even number, $(n^{(\mathbb{C},\ell)}-1)/2$ is known to be a positive integer number $x \in \mathbb{Z}$, for which goes that $\lceil x \rceil = \lfloor x \rfloor = x$ [173, p. 68].

$$
\begin{aligned}
\underset{\cdots\cdots}{m}^{(\mathbb{C},\ell)} - 1 &= \left\lceil \frac{n^{(\mathbb{C},\ell)} + 1}{2} \right\rceil - 1 \quad \text{by Eq. 3.1, Sect. 3.1} \\
&= \left\lceil \frac{n^{(\mathbb{C},\ell)} + 1}{2} - 1 \right\rceil \quad \text{by [173, Eq. 3.6]} && \text{(A.2.1)} \\
&= \left\lceil \frac{n^{(\mathbb{C},\ell)} - 1}{2} \right\rceil \\
&= \left\lfloor \frac{n^{(\mathbb{C},\ell)} - 1}{2} \right\rfloor \quad \text{by [173, p. 68]} && \text{(A.2.2)}
\end{aligned}
$$

We can now carry on with the left-hand side of (A.1.1) and proceed by the substitution of the underlined term for its alternative representation as per (A.2.2). By analogy with (A.2.1), the right-hand side is now rearranged such that the outer term is moved inside the floor expression. To do so, we make use of the proposition that states that $\lfloor x \rfloor + n = \lfloor x + n \rfloor$, for any integer $n \in \mathbb{Z}$ and $x \in \mathbb{R}$ [173, Eq. 3.6b]. Some further rewriting yields the expression shown in (A.3).

$$\begin{aligned}(m^{(\mathcal{C},\ell)} - 1) - e^{(\mathcal{C},\ell)} &= \left\lfloor \frac{n^{(\mathcal{C},\ell)} - 1}{2} \right\rfloor - e^{(\mathcal{C},\ell)} \\ &= \left\lfloor \frac{n^{(\mathcal{C},\ell)} - 1}{2} - e^{(\mathcal{C},\ell)} \right\rfloor \\ &= \left\lfloor \frac{n^{(\mathcal{C},\ell)}}{2} - \frac{2e^{(\mathcal{C},\ell)}}{2} - \frac{1}{2} \right\rfloor \\ &= \left\lfloor \frac{n^{(\mathcal{C},\ell)} - (2e^{(\mathcal{C},\ell)} + 1)}{2} \right\rfloor \end{aligned} \tag{A.3}$$

Finally, synthesis of (A.1.2) and (A.3), combined with the definition of the function $cr(e^{(\mathcal{C},\ell)}) = 2e^{(\mathcal{C},\ell)} + 1$ as defined in Sect. 3.1, provide adequate grounds to conclude this proof:

$$\Rightarrow dtof^{(\mathcal{C},\ell)} - 1 = \left\lfloor \frac{n^{(\mathcal{C},\ell)} - cr(e^{(\mathcal{C},\ell)})}{2} \right\rfloor \qquad \square$$

Similar to Table A.1, one can empirically show that, for any even $n^{(\mathcal{C},\ell)}$ and for eligible values $c_{max}^{(\mathcal{C},\ell)} \geqslant m^{(\mathcal{C},\ell)}$, both $\lfloor (n^{(\mathcal{C},\ell)} - cr(e^{(\mathcal{C},\ell)}))/2 \rfloor$ and $dtof^{(\mathcal{C},\ell)} - 1$ will take the same value whenever the premisses (a)–(c) hold:

$c_{max}^{(\mathcal{C},\ell)}$		$d^{(\mathcal{C},\ell)}$		$cr(e^{(\mathcal{C},\ell)})$		$\lfloor (n^{(\mathcal{C},\ell)} - cr(e^{(\mathcal{C},\ell)}))/2 \rfloor$
6	→	0	→	1	→	2
5	→	1	→	3	→	1
4	→	2	→	5	→	0
6	→	3	→	2		
5	→	2	→	1		
4	→	1	→	0		
$c_{max}^{(\mathcal{C},\ell)}$		$dtof^{(\mathcal{C},\ell)}$		$dtof^{(\mathcal{C},\ell)} - 1$		

Table A.2: Illustrating agreement between both measures for even degree of redundancy $n^{(\mathcal{C},\ell)} = 6$ ($c_{max}^{(\mathcal{C},\ell)} \geqslant m^{(\mathcal{C},\ell)} = 4$).

Proof (for $\mathbf{n}^{(\mathcal{C},\ell)}$ even):
The argumentation in this proof exhibits close resemblance to its counterpart for an odd degree of redundancy. As before, we start with the equality $e^{(\mathcal{C},\ell)} = d^{(\mathcal{C},\ell)}$, and seek to reduce both left- and right-hand sides into a desired form, *i.e.* one of the measures under investigation. The measure $dtof^{(\mathcal{C},\ell)} - 1$ in (A.4.2) is obtained by

some simple rewriting of the intermediate equation in (A.4.1) that in itself originates from flipping the sign and then adding $m^{(\mathcal{C},\ell)} - 2$ in both sides of the initial equation:

$$\begin{aligned} e^{(\mathcal{C},\ell)} &= d^{(\mathcal{C},\ell)} \\ -e^{(\mathcal{C},\ell)} &= -d^{(\mathcal{C},\ell)} \\ \underline{(m^{(\mathcal{C},\ell)} - 2)} - e^{(\mathcal{C},\ell)} &= \underline{(m^{(\mathcal{C},\ell)} - 2)} - d^{(\mathcal{C},\ell)} && \text{(A.4.1)} \\ &= (m^{(\mathcal{C},\ell)} - d^{(\mathcal{C},\ell)} - 1) - 1 \\ &= dtof^{(\mathcal{C},\ell)} - 1 && \text{(A.4.2)} \end{aligned}$$

Again, $m^{(\mathcal{C},\ell)}$ is substituted by its representative value in the left-hand side of (A.4.1), yielding (A.5.1) after splitting the fractional argument of the floor function. As it is known that $n^{(\mathcal{C},\ell)}$ is an even number in $\mathbb{N}^+$, $x = {}^{n^{(\mathcal{C},\ell)}}/_2 \in \mathbb{N}^+$. Per definition, $\lceil x + r \rceil = x + 1$, if and only if $x \in \mathbb{Z}$ and $r \in \mathbb{R}_{]0,1]}$. Some additional rewriting consequentially results in (A.5.2), which in itself is another integer $x \in \mathbb{N}_0$, for which goes that $x = \lfloor x \rfloor$, such that (A.5.3) is eventually attained [173, p. 68].

$$\begin{aligned} \underset{\cdots\cdots}{m}^{(\mathcal{C},\ell)} - 2 &= \left\lceil \frac{n^{(\mathcal{C},\ell)} + 1}{2} \right\rceil - 2 \qquad \text{by Eq. 3.1, Sect. 3.1} \\ &= \left\lceil \frac{n^{(\mathcal{C},\ell)}}{2} + \frac{1}{2} \right\rceil - 2 && \text{(A.5.1)} \\ &= \left(\frac{n^{(\mathcal{C},\ell)}}{2} + 1 \right) - 2 = \frac{n^{(\mathcal{C},\ell)}}{2} - 1 && \text{(A.5.2)} \\ &= \left\lfloor \frac{n^{(\mathcal{C},\ell)}}{2} - 1 \right\rfloor = \left\lfloor \frac{n^{(\mathcal{C},\ell)} - 2}{2} \right\rfloor && \text{(A.5.3)} \end{aligned}$$

Rearranging the left-hand side of (A.4.1) by splitting the fraction ${}^{(n^{(\mathcal{C},\ell)}-2)}/_2$, moving the outer term $-e^{(\mathcal{C},\ell)}$ inside the floor expression, and some simple rewriting results in (A.6.1) [173, Eq. 3.6a].

$$\begin{aligned} (m^{(\mathcal{C},\ell)} - 2) - e^{(\mathcal{C},\ell)} &= \left\lfloor \frac{n^{(\mathcal{C},\ell)} + 2}{2} \right\rfloor - e^{(\mathcal{C},\ell)} \\ &= \left\lfloor \frac{n^{(\mathcal{C},\ell)}}{2} - \frac{2e^{(\mathcal{C},\ell)}}{2} - \frac{2}{2} \right\rfloor \\ &= \left\lfloor \frac{n^{(\mathcal{C},\ell)} - (2e^{(\mathcal{C},\ell)} + 1)}{2} - \frac{1}{2} \right\rfloor && \text{(A.6.1)} \\ &= \left\lfloor \frac{n^{(\mathcal{C},\ell)} - (2e^{(\mathcal{C},\ell)} + 1)}{2} \right\rfloor && \text{(A.6.2)} \end{aligned}$$

The reduction of (A.6.1) into (A.6.2) is justified by reason of the closure properties of integer addition, subtraction and multiplication. Given $n^{(\mathcal{C},\ell)}, e^{(\mathcal{C},\ell)} \in \mathbb{N}$, then $2e^{(\mathcal{C},\ell)} + 1 \in \mathbb{N}^+$, and $n^{(\mathcal{C},\ell)} - (2e^{(\mathcal{C},\ell)} + 1) \in \mathbb{Z}^+$ — *cf.* premiss (c). Since $n^{(\mathcal{C},\ell)}$ is even and $cr(e^{(\mathcal{C},\ell)})$ is odd, the numerator $x = n^{(\mathcal{C},\ell)} - (2e^{(\mathcal{C},\ell)} + 1)$ in (A.6.1) is odd, hence there exists a number $i \in \mathbb{N}_0$ such that $x = 2i + 1$. Then ${}^x/_2 = i + {}^1/_2$, and by definition $\lfloor i + {}^1/_2 \rfloor = i = \lfloor i \rfloor$ [173, p. 68]. Working backwards, the correlation between (A.6.1) and (A.6.2) can now be motivated as follows:

$$\lfloor i \rfloor = \left\lfloor i + \frac{1}{2} - \frac{1}{2} \right\rfloor = \left\lfloor \frac{2i+1}{2} - \frac{1}{2} \right\rfloor = \left\lfloor \frac{x}{2} - \frac{1}{2} \right\rfloor .$$

Finally, the proof can be concluded by replacing the subtract $2e^{(\mathbb{C},\ell)}+1$ in the numerator of (A.6.2) with $cr(e^{(\mathbb{C},\ell)})$, and combining the result with (A.4.2):

$$\Rightarrow dtof^{(\mathbb{C},\ell)} - 1 = \left\lfloor \frac{n^{(\mathbb{C},\ell)} - cr(e^{(\mathbb{C},\ell)})}{2} \right\rfloor \qquad \square$$

A.1 Epilogue

The second proof clearly illustrates that a redundancy configuration encompassing an even amount of redundancy $n^{(\mathbb{C},\ell)} = 2e^{(\mathbb{C},\ell)}+2$ can tolerate no more failures than its counterpart that involves an odd number of versions $cr(e^{(\mathbb{C},\ell)})$. The attentive reader may already have considered the overall ineffectiveness of the spare version in addition to the required $cr(e^{(\mathbb{C},\ell)})$ versions, and may wonder if the use of even levels of redundancy may be useful at all — *cf.* Fig. 3.1 in Sect. 3.1. Indeed, previous work in the domain of reliability engineering on n-version programming schemata seems to support this conjecture, in that it has mostly been presented with the implicit assumption of an odd degree of redundancy [3, Sect. 3.4.1, 3.7.3].

Maintaining an odd degree of redundancy throughout the operational lifetime of an NVP composite by consistently adjusting the employed level op redundancy by (multiples of) 2, may however result in a downscaling procedure which proves to be too aggressive — *cf.* Strategy A, p. 151. Such a redundancy management scheme may bring about discontinuities in the piecewise constant redundancy function n_c, with oscillations that can drive the employed redundancy level straight into an undershooting region.

Furthermore, a tendency can be observed in contemporary distributed system design to discriminate fault messages on the basis of syntactical validity, a matter addressed in Chapt. 8 in the domain of service-oriented architectures **(A44)**. Specific types of fault messages may be anticipated to carry a meaningful payload, and may be recognised as syntactically valid response messages that may contribute to the intended behaviour of the system. When such a message is received in response to some invocation, the invocation under consideration should not be considered to have been affected by a disturbance of higher severity than an RVF failure (unlike EVF failures whose syntactically invalid payload does signal non-persistent disturbances of higher severity) — *cf.* (A19) and Sect. 2.6.5. For instance, an implementation of a square root function without support for complex numbers should issue a fault message when it is invoked with a negative radicand rather than returning an arbitrary value. Supposing that, within the scope of some voting round, this version would be used in conjunction with another version that does offer support for complex numbers, then one or more RVF failures will emerge.

Regardless of the voting algorithm employed, the scheme will always attempt to identify a sufficiently large degree of consent amongst versions $v_i \in V^{(\mathbb{C},\ell)}$ based on the equivalence of syntactically valid response values acquired from the corresponding invocations $\langle \mathbb{C}, \ell, i \rangle$ — *cf.* Sect. 3.1 and 2.6.1. No notice is taken of the remaining versions v_i whose invocation $\langle \mathbb{C}, \ell, i \rangle$ reported an invalid response value and that were classified into $P_F^{(\mathbb{C},\ell)}$ accordingly. As disturbances originate unexpectedly and their manifestative behaviour cannot generally be anticipated, there is no certainty that an odd amount of the $n^{(\mathbb{C},\ell)}$ engaged versions are classified in $\wp^{(\mathbb{C},\ell)} \setminus P_F^{(\mathbb{C},\ell)}$. And even if there were, ambiguity cannot be excluded.

The redundancy dimensioning and replica selection models were introduced in Sect. 5 as loosely coupled algorithms that operate autonomously on the basis of contextual information that has been harvested throughout the system's operational life span. The decision to maintain an even degree of redundancy by the former model (especially in the course of a downscaling procedure) should be considered as an intermediate, transitionary redundancy level during which the latter can attempt to substitute poorly performing replicas by alternative, idling replicas, before instructing the use of a lesser (odd) redundancy level.

APPENDIX B

Alternative Voting Algorithms

An NVP redundancy scheme relies on a decision algorithm in an attempt to overcome any disparities in the results acquired for each of the subordinate invocations $\langle \mathcal{C}, \ell, i \rangle$, and to adjudicate a satisfactory result to be returned for the voting round $(\mathcal{C}, \ell)$ nonetheless. Many different types of decision algorithms have been developed, which are usually implemented as generic voters. And although a procedure inspired by formalised majority voting was used throughout this chapter for the adjudication of a result from at least $m^{(\mathcal{C},\ell)}$ consentient versions amongst the $n^{(\mathcal{C},\ell)}$ participating versions, many other voting mechanisms have been devised: examples include, amongst others, active, plurality and unanimity voting [7] [5, Chapt. 4]. Related research question(s): RQ-1.

B.1 Formalising Unanimity Voting

Even though the adaptive fault-tolerant strategy introduced in Chapt. 5 was designed with (formalised) majority voting in mind, its applicability is not inherently limited to this specific type of voting mechanism, and support for alternative voting algorithms may require only few modifications. In particular, some formulae may require change so as to preserve the semantics of the measures and concepts introduced throughout this chapter.

In what follows we will outline which changes are required in order to support unanimity voting (UV). The term unanimity emphasises the requirement for all participating versions $v_i \in V^{(\mathcal{C},\ell)}$ to be in mutual agreement before a response can be adjudicated, *i.e.* $m^{(\mathcal{C},\ell)} = n^{(\mathcal{C},\ell)}$ — *cf.* Sect. 3.1. Note that the likelihood of such agreement materialising may increase when an "inexact notion of equality between version outputs" is used, such as the equivalence relations $\mathscr{R}_d$ set forth in Sect. 2.6.1.1 [7]. Given this reformulation of $m^{(\mathcal{C},\ell)}$, it follows that an NVP/UV scheme is not tolerant of any failures at all, as it is resilient to withstand disturbances affecting at most $n^{(\mathcal{C},\ell)} - m^{(\mathcal{C},\ell)} = 0$ of the $n^{(\mathcal{C},\ell)}$ versions participating in round $(\mathcal{C}, \ell)$ — *cf.* Sect. 3.1.1. Even though such type of decision algorithm may be perceived to have limited applicability and to squander valuable system resources while at the

same time not offering a significant improvement in availability, there do exist some applications for which it is preferable to return all but an incorrect result.

The *dtof* measure was originally introduced in Sect. 3.1.1 as a measure to quantify the proximity of hazardous situations in which the scheme would have failed to adjudicate a result, had the current redundancy configuration been exposed to additional disturbances. Whenever a result can effectively be determined for some voting round $(\mathcal{C}, \ell)$, the current redundancy configuration $V^{(\mathcal{C},\ell)}$ should have an associated value $dtof^{(\mathcal{C},\ell)} \geqslant 1$. However, as a redundancy scheme based on unanimity voting requires unanimous consensus, the total amount $n^{(\mathcal{C},\ell)}$ of redundancy used is consistently exhausted; a single (additional) disturbance would have caused a failure of the scheme to determine a result. Considering that the scheme cannot tolerate any failures in spite of a strictly positive *dtof* value, one can see that the nett redundancy $dtof^{(\mathcal{C},\ell)} - 1$ yields 0 — *cf.* Sect. 3.1.2. Hence, Eq. (3.2b) and (3.2c) are to be replaced by Eq. (B.1b), in which a critically low value $dtof^{(\mathcal{C},\ell)} = 1$ is attributed to the redundancy configuration $V^{(\mathcal{C},\ell)}$. The slightest difference between version outputs will, however, result in the (temporary) unavailability of the scheme, and the inadequate redundancy configuration will be judged according to Eq. (B.1a), which is identical to Eq. (3.2a). $Ran(dtof)$ will therefore narrow to $[0,1]$, which is captured accurately by the following equation:

$$dtof^{(\mathcal{C},\ell)} = \begin{cases} 0 & c_{max}^{(\mathcal{C},\ell)} < n^{(\mathcal{C},\ell)} \quad \text{(B.1a)} \\ 1 & P_F^{(\mathcal{C},\ell)} = \emptyset \wedge |\wp^{(\mathcal{C},\ell)} \setminus P_F^{(\mathcal{C},\ell)}| = 1 \wedge c_{max}^{(\mathcal{C},\ell)} = n^{(\mathcal{C},\ell)} \quad \text{(B.1b)} \end{cases}$$

The normalised dissent measure $D(\mathcal{C}, v)$ was gradually introduced throughout Chapt. 4 as a means to approximate the reliability of an individual version v by iteratively issuing penalties and rewards reflecting the impact on its operational context, *i.e.* a redundancy scheme $\mathcal{C}$ within which it operates. We will now provide a rundown of any required changes for and the implications of supporting unanimity voting.

The pivotal iterative updating procedure formalised by Eq. (4.2) in Chapt. 4 does not require change at all, and initialisation and penalisation and reward mechanisms will account for any deduced contextual information as before. A side-effect of using unanimity voting is, however, that premiss $dtof^{(\mathcal{C},\ell)} > 0$ implies unanimous consent, *i.e.* $P^{(\mathcal{C},\ell)} = V^{(\mathcal{C},\ell)}$, which excludes the possibility of the existence of other equivalence classes. For that reason, Eq. (4.2b) can and will never occur.

Another aspect to bear in mind is that, even if this premiss holds and a result can be adjudicated, the robustness of the redundancy configuration is consistently jeopardised. Observe how the denominator in Eq. (4.3b) will evaluate to 0, for it was already argued that a scheme based on unanimous voting is not resilient of any failures. For that reason, the definition of the robustness measure provided in Eq. (4.3) in Sect. 4.1 will be replaced by Eq. B.2, in which the repeated exhaustion of the available redundancy is revealed:

$$w_e^{(\mathcal{C},\ell)} = 1 \quad \text{(B.2)}$$

Careful examination of the premisses shows that Eq. (4.5a) of the penalisation mechanism is used by Eq. (4.2b) to determine the penalty that needs to be inflicted on some dissentient version. However, both equations have the premiss $dtof^{(\mathcal{C},\ell)} > 0$

in common, which, in case of unanimity voting, implies unanimous consensus and thus $P^{(\mathcal{C},\ell)} = V^{(\mathcal{C},\ell)}$, such that a reward should be awarded to each of the participating versions by means of Eq. (4.7b) rather than inflicting a penalty. Versions returning a syntactically invalid response message are penalised as before according to Eq. (4.5c). Apart from this, penalties can only originate from Eq. (4.5b), initiated by Eq. (4.2c). Recall how $c^{(\mathcal{C},\ell)}(v)$ was introduced in Sect. 4.1 as the cardinality of the consensus block in which version v was classified throughout round $(\mathcal{C},\ell)$. As there is no mutual agreement between the engaged versions in $V^{(\mathcal{C},\ell)}$, $dtof^{(\mathcal{C},\ell)} = 0$, therefore eligible values for $c^{(\mathcal{C},\ell)}(v)$ and $c_{max}^{(\mathcal{C},\ell)}$ lie in $\mathbb{N}_{[1,n^{(\mathcal{C},\ell)}[}$ and thus $Ran(p^{(\mathcal{C},\ell)}(v)) = \mathbb{R}_{]0,1]}$. The severity of the penalty inflicted on a version is inversely proportionate to the relative degree of consent within the equivalence class in which it was classified; refer to Sect. 4.2 for more information.

Finally, we consider the reward model that was introduced in Sect. 4.3. Recalling that Eq. (4.7b) is used to obtain a reward factor in view of Eq. (4.2e), which requires a strictly positive *dtof* value, one can observe that all engaged versions are attributed the maximum reward value k_1. This prudent downscaling of the accumulated penalties is motivated by the increased vulnerability and poor robustness of an NVP/UV scheme, which was captured by Eq. (B.2). On the whole, the downward evolution of normalised dissent values may be expected to be significantly more gradual than it would be the case for a scheme based on majority voting that would operate in a similar environment, with comparable redundancy configurations exposed to identical disturbances. This can be further supported when reasoning about Eq. (4.7a): let us consider a redundancy scheme $\mathcal{C}$ using a static redundancy configuration. Then, given an identical failure occurrence model, values for $\#consent(\mathcal{C}, v)$ may reasonably be expected to be lower if the scheme is using unanimity voting rather than majority voting. As unanimous consensus is required for the NVP/UV scheme to be able to adjudicate a result, as opposed to a mere qualified majority for an equivalent NVP/MV scheme, the former may fail to determine a result, whereas the latter may have succeeded — note that the inverse is not true. The use of unanimity voting will therefore result in a more rapid increase of the normalised dissent values for participating versions, which is reflected in larger values of $w_i^{(\mathcal{C},\ell)}(v)$, and in turn may delay re-integration of idling versions — *cf.* Eq. (4.8b) and Eq. (4.7a).

B.2 Ramifications of Voting Mechanism

When contemplating the use of an alternative voting mechanism, one should be aware of the repercussions of doing so. Depending on the type of voting mechanism used, the perceived responsiveness of an NVP composite could be sped up by returning an adequate result for a voting round $(\mathcal{C},\ell)$ as soon as one can be uniquely ascertained, possibly ahead of the completion of the partitioning procedure and before a result has been acquired for each of the subordinate invocations $\langle\mathcal{C},\ell,i\rangle$. In spite of pre-emptively returning a result for an ongoing voting round $(\mathcal{C},\ell)$, execution of the remaining, pending invocations $\langle\mathcal{C},\ell,i\rangle$ will proceed as before — *cf.* the voting round state transition model in Fig. 2.1. The conditionalities are specific to each type of voting mechanism, which can only inspect the (partially) constructed partition immediately after each successive partition update, *i.e.* whenever a result for some

version $v_i \in V^{(\mathcal{C},\ell)}$ has been acquired and the version has been classified into some equivalence class in $\wp^{(\mathcal{C},\ell)}$ accordingly. Recall that a generic result value is used for invocations that failed to return a response before the t_{max} timeout has lapsed since their initialisation, which is indicated by the hatched area in Fig. 2.1 and B.1. It is at this stage of the lifecycle of a voting round $(\mathcal{C},\ell)$, *i.e.* the transition into state `(c)`, that a result has been secured for all of the subordinate invocations $\langle \mathcal{C},\ell,i\rangle$ of the involved versions $v_i \in V^{(\mathcal{C},\ell)}$ — either by having received a response message, or because of a performance failure having occurred. It also marks the completion of the partitioning procedure, during which each of the engaged $n^{(\mathcal{C},\ell)}$ versions were classified. Contextual information is then extracted by means of the measures listed in Sect. 3.1 and 3.4 and fed into the models defined in Chapt. 4 and Sect. 5.2 and 5.3.

We will now elaborate on the impact of a chosen voting mechanism on the response time of NVP redundancy schemata. Throughout our discussion, we will refer to Fig. B.1 in which a single voting round is depicted involving $n^{(\mathcal{C},\ell)} = 5$ versions. The length of the curved pink lines is indicative of the relative response time observed for an invocation $\langle \mathcal{C},\ell,i\rangle$. Should it extend beyond the dashed line inside the area hatched in orange, a predefined result message will be used to signal a performance failure for the corresponding version v_i. This scenario is exemplified in Fig. B.1 by version v_4, which failed to return a response within the imposed t_{max} timeout because of a late response failure (omission failures would be caught in a similar way) — *cf.* Sect. 2.6.4. Recall that it is assumed that, within the scope of a given voting round $(\mathcal{C},\ell)$, versions $v_i \in V^{(\mathcal{C},\ell)}$ are classified in the order in which their responses are acquired from the corresponding invocations $\langle \mathcal{C},\ell,i\rangle$ (increasing response times) — *cf.* (A18), Sect. 17. The order in which versions affected by performance failures are classified remains undefined, though this procedure takes place precisely when the t_{max} timeout has lapsed. In discussing the response time of a scheme and the adjudication of a result, one needs to discriminate between a response (value), often denoted by the term outcome, or a failure message to be returned.

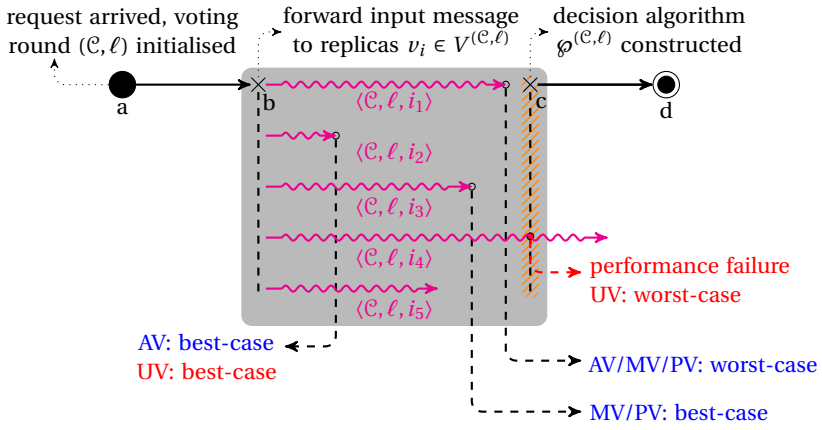


Figure B.1: Comparison of scheme response times for active, plurality, majority and unanimity voting. Blue cases relate to adjudication of outcomes; red ones to failures.

An outcome can only be adjudicated for an invocation $(\mathcal{C}, \ell)$ of a scheme using unanimity voting when a syntactically valid response has been timely secured for all of the subordinate invocations $\langle \mathcal{C}, \ell, i \rangle$ of the involved versions $v_i \in V^{(\mathcal{C},\ell)}$ and all versions returned equivalent responses under an equivalence relation $\mathscr{R}_d$. In other words: unanimity voting does not allow for a pre-emptive return of an outcome; a failure message could be considered to be issued pre-emptively when the timeout has expired. If all versions return an equivalent response before the timeout, the response time for an outcome equals the maximum response time of the $n^{(\mathcal{C},\ell)}$ subordinate invocations. Failures can be issued as soon as a new result has been acquired for a participating version that exhibits anomalous behaviour. Content failures mark a violation of the requirement of unanimity; RVF failures would typically result in the creation of additional consensus block(s), whereas EVF failures will cause the affected versions to be classified in $P_F^{(\mathcal{C},\ell)}$ accordingly. The worst-case response time for identifying a failure is the occurrence of performance failures after all received responses (if any) reported equivalent response values. The best-case scenario would be if, from the first result retrieved from an invocation $\langle \mathcal{C}, \ell, i \rangle$, the version v_i is perceived to have been affected by an EVF failure.

Another, less commonly used voting mechanism is active voting (AV), in which the first acquired syntactically valid response from an invocation $\langle \mathcal{C}, \ell, i \rangle$ will immediately be returned as the outcome for round $(\mathcal{C}, \ell)$, *i.e.* version v_i was classified in some class in $\wp^{(\mathcal{C},\ell)} \setminus P_F^{(\mathcal{C},\ell)}$. If no eligible response can be acquired before the timeout, a failure message will be issued to signal that an outcome could not be adjudicated. Although a substantial reduction in response time can be realised by this approach, it is not suitable for applications designed with stringent correctness concerns, as it cannot overcome discrepancies that may occur due to RVF failures.

Finally, we move on to the majority voting and the closely related plurality voting (PV) mechanisms. They differ in that the latter may adjudicate a result from a consensus block that does not form a qualified majority, as opposed to a threshold $m^{(\mathcal{C},\ell)}$ required by the former — *cf.* Sect. 3.1 [7]. A unique response can be adjudicated as soon as there exists a consensus block $P^{(\mathcal{C},\ell)}$ different from $P_F^{(\mathcal{C},\ell)}$ of cardinality greater than or equal to $m^{(\mathcal{C},\ell)}$. In that case, the outcome can be returned for either of the two voting mechanisms. The best-case response time for determining an outcome of the scheme corresponds to a scenario in which the first $m^{(\mathcal{C},\ell)}$ responses from invocations $\langle \mathcal{C}, \ell, i \rangle$ were acquired before the t_{max} timeout, and were found to be equivalent under the applicable relation $\mathscr{R}_d$. If no such qualified majority has been identified before the timeout expires, a failure message will be returned for an NVP/MV scheme. An NVP/PV scheme may, however, still return an outcome based on the (unique) consensus block of largest cardinality.

APPENDIX C

Optimisation of WS-BPEL Workflows through Business Process Re-engineering Patterns

For the sake of completeness, this appendix reports on research activities that were conducted at an early stage, and that were discontinued due to a lack of mature, open-source software implementations and libraries of the investigated tools and specifications. Although the reported research activities are by no means directly related to or in line with the main research track reported throughout this dissertation, there is a small overlap with the contributions described in Sect. 8.4. With the courtesy and permission of the publisher IGI Global, this appendix includes publication [86] in full, which is an improved and extended version of the research that was previously published as [68] and [85]. It has been included to further substantiate the claim that some of the available linguistic constructs in WS-BPEL can be used to implement redundancy schemata like NVP and RB, and that such an approach can help to achieve a clear separation of concerns by isolating the actual business logic encapsulated within the underlying web services — versions — from the dedicated fault-tolerant orchestration logic. Related research question(s): RQ-4.

This chapter appeared in *Technological Innovations in Adaptive and Dependable Systems: Advancing Models and Concepts* edited by Vincenzo De Florio. Copyright 2012, IGI Global. Posted by permission of the publisher.

With the advent of XML-based SoA, WS-BPEL swiftly became a widely accepted standard for modelling business processes. Even though SoA is said to embrace the principle of business agility, WS-BPEL business processes are still manually crafted into their final executable version. While SoA has proven to be a giant leap forward in building flexible IT systems, this static WS-BPEL workflow model should be enhanced to better sustain continual process evolution. In this seminal paper,

we point out the potential for adding business intelligence with respect to business processes re-engineering patterns to the system to allow for automatic business processes optimisation. Furthermore, we point out how these re-engineering patterns may be implemented leveraging techniques that were already applied successfully in computer science. Several practical examples illustrate the benefit of such adaptive process models. Our preliminary findings indicate that techniques such as the resequencing and parallelisation of instructions, further optimised by introspection, as well as techniques for achieving software fault tolerance, are particularly valuable for optimising business processes. Finally, we elaborate on the design of people-oriented business processes using common human-centric re-engineering patterns.

C.1 Introduction

A cutthroat competition is currently raging between enterprises in which companies are compelled to constantly evolve in order to realise a competitive advantage. This goal of attaining market leadership is pursued by iteratively altering business processes[1] and strategies aimed at improving operational efficiency [174]. Business processes are thus continuously refined, mainly to resolve recurrent issues and as such rectify process performance. This concept is commonly referred to as **business process re-engineering (BPR)**[2].

Large enterprises have extensively deployed information technology (IT) systems, and have recently started to automate their business processes. Regrettably enough, most of these volatile business processes are enlaced into rigid IT systems and this imposes limitations with respect to the speed with which changes are possible. In the beginning of this decade, this issue led to the concept of service-oriented architectures (SoA) in which IT is flexibly structured to better alleviate the re-engineering of processes by splitting up so-called business logic into a number of software components that are exposed as services [22]. With service (operations) as an implementation for individual process activities, a business process can be automated by appropriately orchestrating and coordinating a set of services. Actually this service-oriented computing paradigm has adopted the best practices in distributed computing of — roughly estimated — the past twenty years, and commercially backed by major industry concerns, SoA continues to gain adherence [175].

As one possible SoA implementation technology, web services have managed to become the de facto standard for enterprise software in which various distributed, heterogeneous software systems are integrated in support of corporate e-business and e-commerce activities [22]. A web service is typically exposed through a well-defined open XML interface described in the Web Services Description Language (WSDL) document that formally describes the syntax of application-specific messages in XSD Schema format [18] [54]. Clients communicate with a web service through an endpoint reference that represents the address and context path where the

[1] The notion of business process is defined as an orchestration of several process activities carried out by computer systems or people within an enterprise with the objective of supplying a product or service to the customer.

[2] Because of the vague definitions found in most text books, the BPR acronym is commonly used interchangeably for business process re-engineering as well as business process redesign. The former has an evolutionary character, while the latter is revolutionary. For more information, we refer to [174].

service is deployed [133]. The Web Services Business Process Execution Language (WS-BPEL) XML language is one of the standards that resulted from intensive standardisation initiatives by industrial consortia, and shortly became a widely accepted standard for workflow modelling [143]. The benefit of the central WS-BPEL orchestration component is that the process definition is no longer interwoven inside the implementation code of the business logic. Because of this separation, SoA is said to alleviate the transformation and restructuring of business processes using highly reusable services that can easily be re-orchestrated into WS-BPEL workflows [22].

The service-oriented paradigm turned out to be a giant leap forward in the construction of flexible IT systems indeed. XML-based SoA with WS-BPEL further added to business agility, allowing for the quick development of new business processes leveraging service-wrapped legacy IT assets (*i.e.* business process redesign). But in spite of the popularity of WS-BPEL and its clear separation of process and business logic, there remain some shortcomings [85, 176]. One of these issues is that a WS-BPEL process definition is extremely static: it is designed manually using some software tools and is then loaded into the WS-BPEL engine. Since service orchestration and business processes are at the core of SoA, it is imperative to continuously optimise WS-BPEL process definitions to achieve an increase in system performance, besides having economic implications in realising a competitive advantage required by the actual continual process evolution.

Although the BPR methodology originated in the early Nineties, until recently, businesses were still generally managed using an approach based on experience and intuition. As BPR is gaining adherence, we are on the verge of unifying the IT-driven service-oriented paradigm and the BPR managerial methodology: automatically applying prevailing BPR principles to WS-BPEL process definitions can help in the further optimisation of these process models, thereby help sustain process evolution.

This article starts with an introduction on how BPR patterns can be applied to WS-BPEL process definitions using established techniques from computer science (Sect. C.2). Next, in Sect. C.3, we illustrate the applicability of BPR patterns to WS-BPEL workflows, and show how this can result in performance improvements, such as a reduction in execution time. Sect. C.3.1 to C.3.3 will then elaborate on the resequencing and parallelisation of process activities, further optimised by introspection, after which the relationship is examined between techniques for achieving software fault tolerance and critical process activities. Lastly, Sect. C.3 will highlight the key role we envisage for human-centric re-engineering patterns in the design of people-oriented business processes.

C.2 Business Process Re-engineering and WS-BPEL

Numerous BPR principles (best practices, design patterns, heuristics) have been proposed in the literature, yet there has not been any thorough inquiry into combining IT and BPR so far [177]. In order to support a higher level of process agility, we propose to design an intelligent system able to optimise the WS-BPEL processes in accordance with these conceptual BPR principles. An overview of some potentially useful patterns for WS-BPEL process improvement is shown in Table C.1.

The WS-BPEL XML language defines a set of primitives with which business processes can be modeled: basic activities (receive, assign, invoke, reply, *etc.*) can

be set in order using control and data flow supported by structured activities (*e.g.* sequence, loop, pick) [22, 143]. These rudimentary structural activities turn out to be limited to the common control and decision structures available in most imperative programming languages.

It is no surprise, then, that we can spot similarities between the techniques for program optimisation in computer science, operating on atomic units of instructions and the BPR patterns for business processes, acting on coarser units of instruction blocks with a variable size: service operations. Consequently, it is plausible to try and automate economic BPR patterns leveraging existing techniques from computer science.

BPR directives	basic techniques	human-centric	Sect.
resequencing	data and control flow analysis [178]	no	C.3.1
parallelisation	Tomasulo, scoreboarding [179]	no	C.3.1
exception	control flow and flow variable	no	C.3.2
knock-out (minimise process cost)	speculation [179]	no	C.3.3
reliability	transactional support [180]	no	C.3.4.2
dependability	redoing, design diversity [67, Chapt. 1, 2]	no	C.3.4.3
order assignment & distribution	chain of execution [145]	yes	C.3.5.2
flexible assignment	nomination [145, 146]	yes	C.3.4.3
specialist-generalist	nomination [145, 146]	yes	C.3.4.3
split responsibilities	4-eyes principle [145]	yes	C.3.5.4
case manager	delegation, escalation, process administrator role [145]	yes	C.3.5.5

Table C.1: some business process re-engineering patterns

One possibility to combine both disciplines is to add BPR intelligence into workflow design tools that will preprocess and transform the process model prior to its execution. Alternatively, attributing business intelligence to SoA could allow for the dynamic application of BPR principles to the original static WS-BPEL process definitions, aiming at the **optimisation of the process at runtime** depending on the system's current state and resource availability. This runtime information may be used to adjust either the overall process model or individual process instances. Obviously, this second approach is more powerful than the former which is operating exclusively at design-time, as it enables the system to tune a process taking into account the system's running internal state as well as environmental conditions.

C.3 Business Process Re-engineering Patterns

In this section, some examples will illustrate that applying BPR patterns to WS-BPEL processes can have a beneficial influence on the overall performance of the process model. We suggest some techniques from computer science with the potential to implement these patterns. For the purpose of clarity, the examples are presented in Business Process Model and Notation (BPMN), a common graphical representation of the actual XML WS-BPEL definition. This does not limit our contribution, as WS-BPEL can easily be mapped to BPMN and vice versa [181]. For detailed information about the WS-BPEL 2.0 specification, please refer to [143].

C.3.1 Resequencing and Parallelisation BPR Patterns

The execution of a WS-BPEL process is essentially sequential, though the WS-BPEL specification also contains syntactical facilities for executing activities in parallel [143]. As a primary BPR pattern, the execution order of process activities, *i.e.* service invocations, can be optimised by considering data flow dependencies so as to execute mutually independent activities in parallel [177]. The underlying idea of simultaneously executing activities and advancing activity initialisation is that some time can be gained by avoiding performance-degrading stalls caused by dependencies. Throughout this paper, we assume an SoA-based environment aggregating a set of distributed IT system allowing for optimisation by parallel execution, but the amount of parallelism that can actually be achieved is also limited by the number of resource (web services or employees) replicas and their processing capacity in the system.

Data dependencies can arise between different activities and relate to variables defined in the WS-BPEL process definition. For instance, a service invocation activity has a read-dependency on whatever variable is used to hold the input message for the service to be invoked, and also a write dependency to the variable that will ultimately store the service's reply. Assignment statements normally construct and write to a variable after reading values from one or more other variables. Structural activities may also read certain variables during the evaluation of control flow variables.

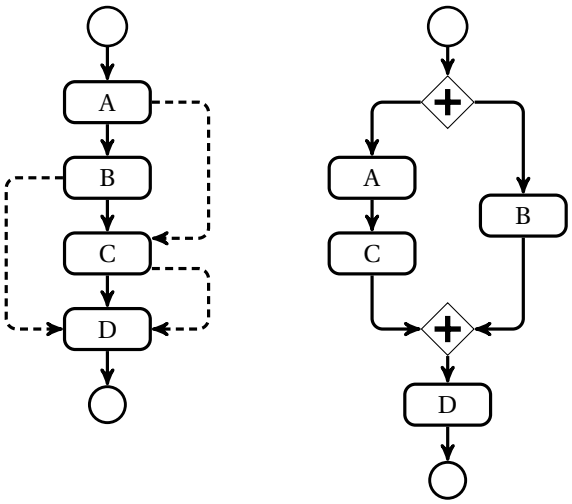


Figure C.1: On the left the original WS-BPEL process definition; on the right the optimised process. The rhombic symbols stand for parallel execution, *i.e.* AND-split/join.

Techniques for the dynamic scheduling of instructions, such as the Tomasulo approach and scoreboarding, have been successfully used in numerous domains of computer science and allow for an optimised, out-of-order execution of sequential streams of program instructions, which could be used as the basis for individual process instances [179]. These techniques could be extended and applied to WS-BPEL activities to avoid pointless waiting as the result of data dependencies. However,

these approaches are limited to basic blocks of non-branching sequences of instructions, *cf.* one sequential scope in WS-BPEL, and can benefit from techniques such as speculation to work around control flow statements and as such artificially increasing the number of instructions in these basic blocks (*cf.* Sect. C.3.3). The Tomasulo approach exploits the knowledge on dependencies unraveled at runtime; thus it clearly outperforms all strategies that statically analyse the data and control flow of the WS-BPEL process; nevertheless, it is considerably easier to analyse and restructure the overall WS-BPEL process model at design-time [178].

The process model in Fig. C.1 merely represents successive service invocation activities. It is furthermore assumed that an unaltered output message from a particular service invocation is stored in a WS-BPEL variable, which is used as input for invoking another service. Note that the dashed arrows representing these data dependencies are not part of the official BPMN notation and have been added for improved readability. The start event corresponds to the reception of a message that triggers execution of a new WS-BPEL process instance and the end event represents the process replying to its requester. The solid arrows in the diagrams indicate sequential flow. Supposing the respective execution times for activities A, B, C and D (invocation of an operation on a service) are 9, 4, 12 and 6 seconds, the execution time in the original process would simply be 31 seconds, whereas the optimised version would result in an execution time of 27 seconds. Obviously, the time required for the new scope containing the parallel flows to complete is determined by the branch that takes most time to complete ($A - C$).

C.3.2 Exception BPR Pattern

In re-engineering business processes, it is common to isolate the exceptional part from the normal process flow. Techniques like speculation are already applied in compiler optimisations to improve control flow, branches in particular [179]. This can be accomplished by conditionally executing the branch with highest probability, and compensating in case of misprediction. Moreover, the amount of parallelism that one can exploit is also limited by control dependencies. Speculation is a technique that can be used to overcome the penalty of control dependencies in some cases by shifting highly probable activities to eliminate control dependencies so as to match the parallelism offered by the execution environment. To achieve speculation techniques in WS-BPEL process definitions, scopes of activities may be shifted, provided that an estimation on the probability of each branch is available. This information can either be a constant value chosen at design time, or alternatively be gathered at runtime, during the execution of the program, through a monitoring component (as it is the case *e.g.* in feedback loops and autonomic computing systems) [50].

Consider for instance the example in Fig. C.2. Imagine the left branch in the original process model has a 30% chance of being executed. For the service invocations A, B, C and X, we suppose execution times being 15 seconds each. Then the execution of the original process model would take about 45 seconds for sequentially executing activities X, A and B, or 30 seconds for the other execution path comprising activities X and C. The time required to evaluate control flow structures, *e.g.* branch variables, is considered negligible.

Applying speculation will restructure the initial model to always execute *C* (because of the 70% chance the containing branch is chosen). The branching condition remains identical as long as the result of invoking service *C* is not written to a variable that is used for evaluating the branching condition. In case the process instance proceeds accordingly, the execution time equals 30 seconds. However, speculation may deteriorate and delay process execution in case the execution does not fit: because of the undoing[3], the best-case execution time of the speculated WS-BPEL process is now 60 seconds.

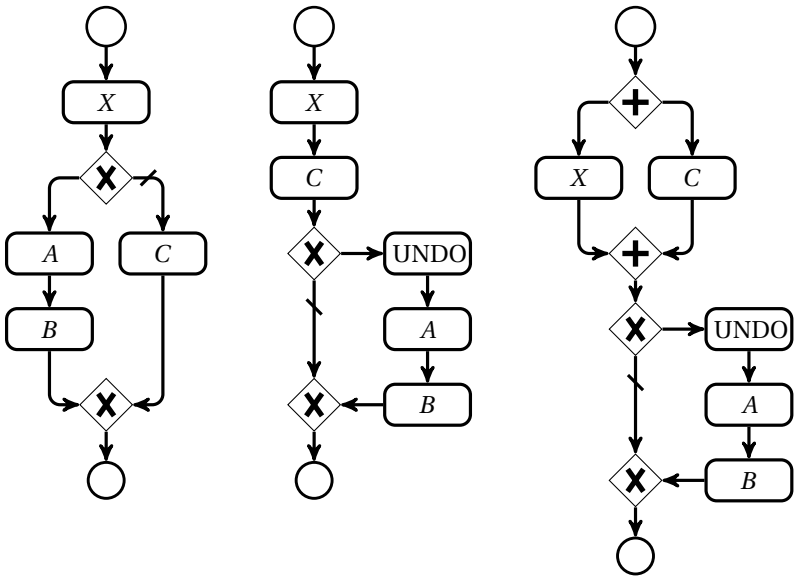


Figure C.2: From left to right: the original process, which is then restructured using speculation (phase 1), and finally the version optimised using parallelisation (phase 2). The rhombic symbols in the original process denote conditional branches.

Consequently, speculation in itself does not necessarily result in an enhanced process model, but combined with parallelisation, a significant gain in execution time can be harvested. In the absence of data dependencies between activities *X* and *C*, the intermediate model that was transformed using speculation can now be restructured to execute *C* in parallel to *X*. In case of the normal process flow (the branch comprising *C* in the original model), there is a considerable improvement: 15 seconds instead of 30 seconds (a speedup by factor 2). In the worst case, should the time required to execute *C* not exceed the time lapse for the execution of the other parallel flow of activities, the performance degradation is given by the overhead for undoing, and the alternative flow will perform no worse than in the original model.

In conclusion, the exception re-engineering pattern should only be applied if there are no data dependencies between the activities in the branch that is most likely to be chosen (the normal flow) and the activities before the branching condition. Furthermore, we suggest this pattern to be applied only if the difference

[3] As the service-oriented computing paradigm promotes the development of stateless web services, the overhead of the undo activity may be considered negligible.

in probabilities of the alternative branches exceeds a minimal threshold. In that case, the overall performance might benefit from speculation, and the negative impact of non-fitting WS-BPEL instances might be subdued.

C.3.3 Knock-out BPR Pattern

Business processes generally contain a number of knock-outs, conditional checks that may cause the complete process instance to cease, skipping all subsequent process activities. Upon occurrence, the WS-BPEL process instance should be abandoned, possibly compensating in order to reverse the service invocations that were required for evaluating the knockout conditions. The knock-out BPR pattern is a special version of the resequencing pattern aiming to manipulate the process yielding on average the least costly execution by arranging knock-outs in decreasing order of effort and increasing order of termination probability [177]. The rationale behind this pattern is that knock-outs should be inserted in the process flow as early as possible to avoid the allocation of resources during the execution of other process activities for process instances that halt. We illustrate this principle in Fig. C.3:

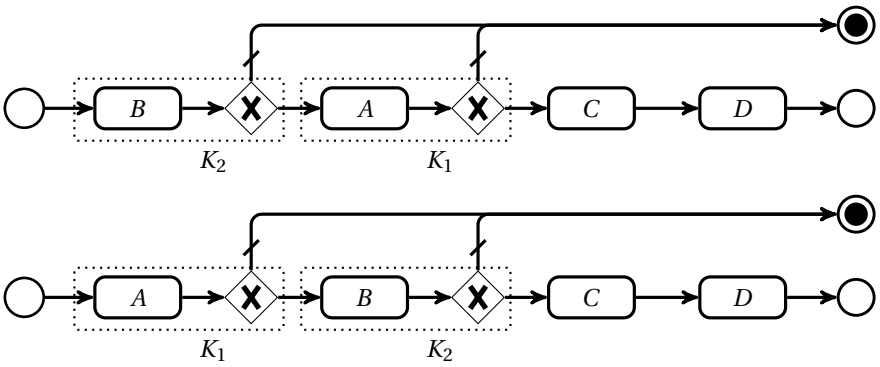


Figure C.3: An example to illustrate the knock-out BPR pattern.

Suppose knock-out condition K_1 has a 40% probability of evaluating negatively and it takes 2 seconds to invoke service A and compute this branching condition. Likewise, for knock-out K_2, these values respectively equal 65% and 4 seconds including the invocation of service B. Then the ratio $0.40/2$ for K_1 is higher then $0.65/4$ for K_2. Hence, assuming the absence of data dependencies, application of the knock-out pattern should restructure the arrangement of knock-outs in the upper diagram shown in Fig. C.3 into the lower process model, *i.e.* K_1 should be checked before K_2.

As time goes by and more process instances have been executed, for both the speculation and knock-out re-engineering patterns, the estimated probabilities of the branching conditions might change, which in turn may trigger new changes to the process model at runtime, resulting in adaptive business processes.

C.3.4 Dependability Aspects

During its execution, failures may occur that impede a WS-BPEL process instance from proceeding correctly or even worse, simply terminate it. Moreover, a process

may contain a number of business-critical scopes that require a higher degree of reliability or some form of transactional support. In this section, we present a survey on how processes can be designed for increased reliability using proven techniques for application-level fault tolerance.

C.3.4.1 WS-BPEL and Fault Tolerance

Fault-tolerant mechanisms can only be expressed in a syntactically adequate linguistic structure [11]. In spite of numerous WS-* specifications related to reliable messaging and security, little emphasis has been placed upon the fault tolerance aspect of SoA [176, 180]. In this section, we first introduce the syntactical constructs for error recovery as well as their semantics in the WS-BPEL specification. We will then point out that these features prove to be inadequate for complex scenarios aiming at increasing dependability by means of transactions and fault tolerance. A WS-BPEL process definition consists of a number of scopes that typically represent a particular piece of functionality, most likely a complex activity. Similar to high-level programming languages, scopes can be nested within the overall process definition as the outer scope. Three types of handlers can be defined upon a scope [143]:

– The purpose of WS-BPEL fault handlers is similar to that of catch blocks and exceptions in the Java™ programming language: to undo the partial and unsuccessful work of a scope and performing forward error recovery with the aim of re-attaining a state where the execution of the WS-BPEL instance can resume. A web service may explicitly throw a SOAP fault message when it is invoked from a WS-BPEL process. WS-BPEL also defines a number of standard faults that will be thrown by the WS-BPEL runtime as a consequence of erroneous conditions during process execution, for instance, a join failure. Furthermore, the process designer may include application-specific knockouts that throw WS-BPEL faults when deviations from normal behavior are detected (self-checking pattern) [5]. Faults are identified by an XML qualified name. Two types of fault handlers can be attached to a scope for intercepting faults thrown during the execution of the activities contained inside the scope: a fault handler can either handle one specific type of fault, or a "catch-all" fault handler can handle all faults for which no specific handler was defined. When a fault is raised, all remaining activities in the current scope are terminated, and an appropriate handler that is capable of handling the fault will be selected and activated. If a fault cannot be treated by the handlers pertaining to the current scope, it is recursively forwarded to the enclosing scope. If the fault is not caught by any fault handler, the process instance will exit, triggering the default termination handler. Note that fault handlers are only enabled when the execution of a scope is in progress.

– Compensation handlers represent the application-specific undo process for rolling back the effects of scoped activities that were already executed (forward error recovery). If compensation is triggered, for instance from within fault handlers, all nested scope will have their compensation handlers activated recursively.

– Lastly, termination handlers can be used kick in a series of activities when a WS-BPEL process exits unexpectedly. The default compensation handler will trigger compensation.

Apart from its compensation and fault handing, the WS-BPEL syntax is limited to describe the functional part of workflows. Moreover, these standard WS-BPEL recovery mechanisms prove to be inadequate to define sophisticated recovery patterns/procedures, for example rollback or the execution of alternative web services [85, 180]. Finally, a hidden assumption in WS-BPEL is that designers have complete knowledge of the fault and system model of the partner services, so that they are able to define a process flow that contains all the required strategies for recovering from faulty situations [176]. We believe this assumption not to be a realistic one.

C.3.4.2 Transactional Support

The WS-AtomicTransaction specification, part of the WS-Transaction family of specifications, enables the coordination of distributed transactions using the two phase-commit protocol with ACID-compliant transaction features (atomicity, consistency, isolation, durability) [22]. Even though backward error recovery techniques such as this commit and rollback approach are generally not suitable to apply to long-running WS-BPEL workflows, there are situations where consistency should be guaranteed during short-lived subprocesses (service invocations in a particular scope). WS-BPEL's compensation mechanism, which allows undoing the effects of completed activities, cannot cope with atomic transactions, as the coordination model of WS-BPEL is local to the process definition. As there is no external coordination, a partner service in the transaction may not be notified by the process, which may leave the system in an inconsistent state. The WS-BPEL enhancements published in [180] enable the use of atomic transactions and business activities in the context of WS-BPEL processes by using aspects to inject WS-BPEL code to use the external coordination mechanism defined in WS-AtomicTransaction.

C.3.4.3 Redoing and Design Diversity

This section will demonstrate the feasibility of using WS-BPEL to apply proven techniques for application-level fault tolerance.

Redoing and Recovery Blocks Recovery blocks is a technique that addresses residual software design faults. It is similar to the hardware fault tolerance approach known as "stand-by sparing". The approach works as follows: on entry to a recovery blocks, the current state of the system is checkpointed. A primary alternate is executed. When it ends, an acceptance test checks whether the primary alternate successfully accomplished its objectives. If not, a backward recovery step brings the system state back to its original value and a secondary alternate takes over the task of the primary alternate. When the secondary alternate ends, the acceptance test is executed again. The strategy goes on until either an alternate fulfills its tasks or all alternates are executed without success. In such a case, an error routine is executed.

The effectiveness of recovery blocks rests to a great extent on the acceptance test. A failure of the acceptance test is a failure of the whole recovery blocks strategy. For this reason, the acceptance test must be simple, must not introduce huge run-time overheads, and it must not retain data locally. Recovery blocks have been successfully adopted throughout 30 years in many different application fields. It

has been successfully validated by a number of statistical experiments and through mathematical modelling [67, Chapt. 1].

Retrying is not directly supported by WS-BPEL, nor is WS-BPEL capable of checkpointing the initial system state. Assuming stateless web services as alternates, consider a scope with a structured sequence activity. First it will invoke an operation on service *A*. Then, a conditional branch will evaluate the acceptance test on the service response message. If the acceptance test is successful, the process execution proceeds. However, in case the acceptance test fails, a fault is thrown. This fault is caught by a fault handler attached that in turn triggers compensation. The compensation handler contains another scope with precisely the same activities, only this time an alternate service *B* will be invoked. By nesting scopes inside compensation handlers recursively, WS-BPEL allows to redo a web service invocation [182].

***n*-version programming** *n*-version programming (NVP) systems are built from generic architectures based on redundancy and consensus. NVP is defined by its author as "the independent generation of $n > 1$ functionally-equivalent programs from the same initial specification" [6]. These *n* programs, called versions, are developed for being executed in parallel. This system constitutes a fault-tolerant software unit that depends on a generic decision algorithm to determine a consensus or majority result from the individual outputs of two or more versions of the unit.

Such a strategy has been developed under the fundamental conjecture that independent designs translate into random component failures. Such a result would guarantee that correlated failures do not translate into immediate exhaustion of the available redundancy, as it would happen, *e.g.*, by using *n* copies of the same version. Replicating software would also mean replicating any dormant software fault in the source version.

NVP is different from recovery blocks in that the latter is a sequential strategy, whereas NVP allows concurrent execution. Moreover, recovery blocks require the user to provide a fault-free, application-specific acceptance test, while NVP adopts a generic consensus or majority voting algorithm that can be provided by the execution environment. Finally, recovery blocks allow different correct outputs from the alternates, while the general-purpose character of the consensus algorithm of NVP calls for a single correct output. The two models collapse when the acceptance test of recovery blocks is done as in NVP, *i.e.*, when the acceptance test is a consensus on the basis of the outputs of the different alternates.

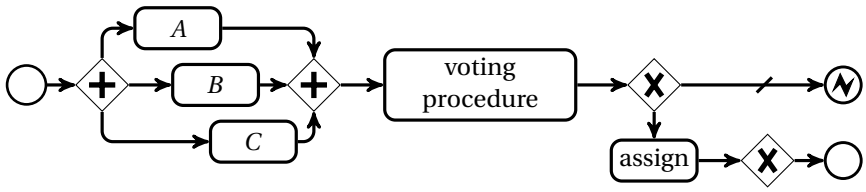


Figure C.4: *n*-version programming will execute multiple service implementations and perform majority voting to verify the result

Fig. C.4 shows how NVP may be implemented in WS-BPEL. Assuming stateless services *A*, *B* and *C* have the same WSDL interface, we concurrently execute the

same method on the three available services ($n = 3$). After joining, the voting procedure will compare the values returned. Finally, if the outcome of the voting procedure is negative, a WS-BPEL fault will be thrown to signal failure, or the value for which there is consensus will be stored in a variable, after which the process will continue.

C.3.5 Human-centric BPR in People-oriented Business Processes

Human interactions frequently occur in business processes for the manual execution of tasks, *e.g.* an approval [145]. Expenses resulting from employment of people are still a major cost factor in enterprises. Therefore, an efficient allocation of the staff is imperative, and IT can also help to achieve this goal.

This brings us to another shortcoming of WS-BPEL, which is rightly blamed of being too automation-centric since it lacks the recognition of employees in process workflows [143,145]. The WS-BPEL4People and WS-HumanTask specification drafts, recently submitted to OASIS® for ratification, allow for hybrid SoA in which human actors occur next to customary IT systems exposed as web services [146]. Consequently, we claim people-centric WS-BPEL4People processes should be designed baring human-centric BPR patterns in mind, so that the system can automatically determine which employees should take care of a process activity, depending on the current availability of these human resources. To our knowledge, this idea has not been previously investigated. Table C.1 shows a few of these patterns and points out relevant procedures defined in the WS-BPEL4People specifications. Most of these patterns deal with the issue of assigning the task to the best suitable person available.

C.3.5.1 Introduction to BPEL4People and WS-HumanTask Specifications

The term WS-BPEL4People actually covers two specifications that have been devised in a modular approach as an attempt to cover the complete spectrum of human-to-process interaction. WS-HumanTask proposes an industry standard for defining and managing human-based activities in a WS-BPEL4People process:

– It stipulates the syntax and semantics for defining human tasks in XML format, where a task is considered as an indivisible unit of work performed by a human process actor. Similar to a subprocess, the execution of a task is closely related to the context of the parent process. The interoperable WS-HumanTask coordination protocol has been conceived to attain a tight coupling with synchronisation of state between the task and the process, where state changes can be propagated in either direction.

– New tasks are usually offered to a task inbox, the central point of interaction for human actors. WS-HumanTask defines a comprehensive client API interface for the implementation of task boxes that can be used for manipulating tasks and controlling their life cycle in accordance to the WS-HumanTask coordination protocol. A task inbox is capable of rendering the user interface that is associated with a particular type of task so that all relevant information is displayed and the employee can successfully complete the work.

On a higher level, the WS-BPEL4People extension defines a number of features layered on top of WS-BPEL to seamlessly integrate WS-HumanTask tasks into WS-BPEL process definitions. Three concepts are at the core of WS-BPEL4People:

– People activities were introduced as a new type of basic activity, to allow the integration of user interactions within WS-BPEL processes. WS-HumanTask tasks can be declared either inside the activity declaration, or may be remotely deployed as a web service that supports the WS-HumanTask coordination protocol (though the service is not implemented by a piece of software, but by a task box such that a user will eventually perform the work manually). A people activity declares the inputs and outputs required to invoke the task, just like its equivalent invoke activity used for calling a web service.

– Similar to partner links which are used to bind a web service to a WS-BPEL process, people links bind a group of people to a business process. People links are generally associated with the generic human roles defined in WS-BPEL4People and WS-HumanTask and represent a group of people who are associated with the execution of a particular people activity.

– In order to determine the actual group of individuals involved in dealing with a particular activity, the action of people resolution has to be performed by assigning people queries to people links. An example of a people query may be an XPath expression to be evaluated against a people directory, a database describing an organisational model to represent the employees of some company or department (WS-BPEL4People only describes the entity and the XSD schema type the query should return; the actual implementation is not covered). People resolution is actually a two-phase procedure: first the people query is evaluated to determine the set of people that have the potential owner generic human role. The task infrastructure will subsequently offer the task to all potential owners who are eligible to claim that activity. Eventually a single potential owner that claimed the activity will become the actual owner and will be responsible for completing the activity.

One may wonder whether these specifications cover most commonly used constellations of human-to-process interaction. Extensive studies with regard to this issue were already published in [144] and [183]. Having compared these specifications against the universal transaction pattern as the core of the communication theory formalised in the Design and Engineering Methodology for Organisations (DEMO), we concluded that the specifications support a vast majority of the human-process interaction scenarios [26].

C.3.5.2 Order Assignment BPR Pattern

The order assignment pattern prefers the same employee to work on several successive process activities for a particular process instance. This is directly supported by the WS-BPEL4People concept of chain of execution, where the actual owner that took care of the previous activity is selected as the sole potential owner for the task at hand (see Lst. C.1, activities *A* and *B*). In addition, an escalation action should be defined to offer the task to the regular set of potential owners in case the

default scenario would fail (*i.e.* the owner of the previous activity does not claim the task before the expiration deadline). Assigning several consecutive process activities to one person should result in a reduction of execution time as this person has got acquainted with the case. The side effect, however, is that the employee's workload will slightly increase compared to his or her colleagues.

Listing C.1: Assuming a sample process comprising human tasks *A* and *B* (in that order), this excerpt from the WS-BPEL4People process definition illustrates the language constructs for chaining the execution of *B* to *A*.

```
<bpel:extensionActivity>
    <b4p:peopleActivity name="B">
        <htd:task name="B">
            <htd:peopleAssignments>
                <htd:potentialOwners>
                    <htd:from>b4p:getActualOwner("A")</htd:from>
                </htd:potentialOwners>
            </htd:peopleAssignments>
        </htd:task>
    </b4p:peopleActivity>
</bpel:extensionActivity>
```

C.3.5.3 Flexible Assignment and Specialist-Generalist BPR Pattern

Next, according to the flexible assignment BPR pattern, and supplemented by the specialist-generalist pattern, one should distinguish between highly specialised human resources and generalist employees that can be assigned to execute a diversity of tasks. The availability of generalists adds more flexibility to the business process and can lead to a better utilisation of resources. Unfortunately, the generic human roles defined in the specifications are insufficient, and the people query facility and the organisational people directory that is searched by this query, both proposed in the above mentioned specifications, remain undefined. We should find a way to annotate people in this directory describing their skills, capabilities and permissions so that the system can reason about the degree of an individual's specialisation. We envision an important role for techniques such as semantic processing and semantic matching in particular. The mutual assistance community, as it was proposed in [184], aiming to provide elderly people with the services they require in a timely and cost-effective way, introduces a system where human resources are registered with a semantic description according to an OWL-S ontology model [185]. Further research is required on semantic WS-HumanTask annotation before this type of service-wrapped registry can be used to determine the potential owners of a task.

C.3.5.4 Split Responsibilities BPR Pattern

Assigning different tasks within a process to people from different functional units should be avoided (split responsibilities pattern). Again, enhancing the expressiveness of people queries and the structure of the people directory could allow the system to optimise the dispatching of human tasks to the appropriate available human resources at runtime. Related to this pattern is the concept of segregation of duties, also referred to as the 4-eye principle in which mutually independent individuals each perform an instance of the same task for the purpose of combating

fraud and avoiding disastrous mistakes [146]. An example is shown in Lst. C.2, where activity *C* may not be executed by whomever performed task *A*.

Listing C.2: Assuming a sample process comprising human tasks *A* and *C* (in that order), this excerpt from the WS-BPEL4People process definition illustrates the language constructs for separation of duties: activity *C* cannot be handled by whomever completed activity *A*.

```
<bpel:extensionActivity>
    <b4p:peopleActivity name="C">
        <htd:task name="C">
            <htd:peopleAssignments>
                <htd:excludedOwners>
                    <htd:from>b4p:getActualOwner("A")</htd:from>
                </htd:excludedOwners>
            </htd:peopleAssignments>
        </htd:task>
    </b4p:peopleActivity>
</bpel:extensionActivity>
```

C.3.5.5 Case Manager BPR Pattern

The case manager BPR pattern originally introduced an additional process actor — the case manager — that is responsible for a business process. However, as the emphasis is on the management of the process rather than actually participating in its execution, the case manager is not necessarily the only resource that will work on process tasks. Providing a single point of contact from a client perspective, detour patterns, such as delegation and escalation which are directly supported in WS-BPEL4People, can result in delegating process activities to other people [144, 145]. Apart from this single point of contact, the case manager is also the person accountable for correcting mistakes. Fortunately, the business administrator and process stakeholder generic human roles, defined in the WS-BPEL4People specification, can be used to represent the case manager respectively when managing the entire process or merely a single process case [145].

In a situation where the execution of a process has gone astray, chances are that it will jam and require manual intervention of the case manager in order not to aggravate the situation. Therefore, one-way WS-HumanTask notifications can be used to notify the case manager of noteworthy events, or application-specific administration tasks for forward or backward recovery may be embedded inside WS-BPEL fault or termination handlers.

As a final consideration, human-computer interaction faults were rarely considered in fault tolerance designs as, in the past, they were considered external to the system boundaries. The WS-BPEL4People specifications finally allow extending these boundaries by seamlessly integrating human tasks and service-wrapped software components into hybrid WS-BPEL workflows that can consequently realise a higher degree of dependability. Also, because of WS-HumanTask, such interaction faults can be detected and dealt with by adding additional checks, possibly with the intervention of an external case manager.

Combining human-computer interaction by means of the WS-BPEL4People and WS-HumanTask specifications with BPR patterns for automatically and intelligently dispatching workload to human system resources is an exciting research challenge

with the potential for a substantial performance improvement in process execution, which may lead to increased productivity and competitiveness. At the same time this also endorses the significance of WS-BPEL4People in SoA, and we plead for a speedy ratification of the specification drafts.

C.4 Conclusion

We started this paper by briefly introducing BPR as a relatively new managerial methodology and SoA as a way to sustain the volatility of business processes resulting from the fierce competition in the market. It was pointed out that the static nature of WS-BPEL process definitions imposes limitations to quickly and easily re-engineer business processes in the quest for operational efficiency.

SoA strongly embraces the principle of business agility. Hence, incorporating BPR system intelligence into the WS-BPEL engine allows for the dynamic application of BPR principles to the original static WS-BPEL process definitions, with the goal of optimising the process at runtime with respect to the system's current state and runtime environment.

We propose an innovatory approach in which BPR principles are explicitly applied to WS-BPEL processes by means of established techniques and practices from computer science so that the process semantics are preserved and whereby at the same time the process execution is being optimised. It is expected that this will allow for a reduction in execution time, *e.g.* as the result of parallelisation. This conjecture has been corroborated by several small examples. We then addressed the issue of process reliability using a bottom-up approach starting from proven techniques for software fault tolerance. Furthermore, the WS-BPEL4People standard enables to design people-oriented business processes such that human-centric BPR patterns are applied at runtime to intelligently dispatch human tasks to suitable human process actors depending on availability.

Furthermore, BPR-aware SoA have the potential to turn static WS-BPEL process definitions into adaptive workflows that match the current environmental and systemic conditions so as to make a more efficient use of these system resources thus achieving higher performance. The WS-BPEL specification need not be modified: this ensures a smooth transition in adopting these ideas. Complex BPR patterns can be implemented using runtime system information and possibly service metadata and annotations. We are still in the early phase of elaborating on the ideas presented in this paper. As a proof of concept, we intend to develop a prototype illustrating the feasibility of the exemplified BPR patterns. Research on the introduced human-centric BPR patterns will depend on the ratification process of the WS-BPEL4People specification draft, and the availability of compliant implementations.

We conclude that BPR-aware SoA environments, automatically applying re-engineering patterns to WS-BPEL processes, result in adaptive business processes, which is a crucial requisite for achieving an enhanced form of business agility and as such better sustaining process evolution.

APPENDIX D

Nederlandstalige Samenvatting

Van bedrijfs- en missiekritische gedistribueerde applicaties wordt in toenemende mate verwacht dat ze uiterst betrouwbare kenmerken vertonen, met name op gebied van beschikbaarheid en tijdigheid. Voor dit soort toepassingen zal een volledige stopzetting of een subnormale prestatie van de dienst die ze geacht worden te leveren, evenals laattijdige of ongeldige resultaten, met grote waarschijnlijk leiden tot aanzienlijke financiële verliezen, milieurampen of menselijk letsel. Softwarecomponenten die deel uitmaken van gedistribueerde computersystemen kunnen echter lijden onder de beperkingen en uitdagingen die inherent zijn aan zulke omgevingen, zoals variabele responstijden of tijdelijke onbeschikbaarheid en onbereikbaarheid.

Het toepassen van klassieke redundantie-gebaseerde fouttolerante ontwerppatronen, zoals NVP, in zeer dynamische gedistribueerde computersystemen leidt niet noodzakelijkerwijs tot de verwachte verbetering van de betrouwbaarheid. Dit komt voornamelijk voort uit statische en vooraf gedefinieerde redundantieconfiguraties die binnen dergelijke betrouwbaarheidsstrategieën toegepast worden, d.w.z. een vast redundantieniveau en een vaste selectie van functioneel equivalente software componenten, wat op zich een negatieve invloed kan hebben op de algehele effectiviteit van het systeem, ten minste vanuit de volgende twee invalshoeken.
Ten eerste kan een statische, context-agnostische redundantieconfiguratie op termijn leiden tot een snellere uitputting van de beschikbare redundantie. Daardoor is het mogelijk dat eventuele verstoringen van de operationele status (context) van de onderliggende componenten niet naar behoren kunnen worden gecompenseerd.
Ten tweede bepaalt de hoeveelheid redundantie, in combinatie met het stemalgoritme, hoeveel simultaan falende versies een NVP-schema kan tolereren. Een vooraf bepaald niveau van redundantie is echter niet kosteneffectief: wanneer het werkelijke aantal storingen lager zou zijn dat wat tijdens de ontwerpfase vooropgesteld werd, dan zou een kleinere hoeveelheid redundantie (tijdelijk) kunnen volstaan, waardoor de computationele kost gereduceerd kan worden, en mogelijk tevens de kost van het daaraan geassocieerde energieverbruik.

In dit proefschrift wordt een nieuwe betrouwbaarheidsstrategie geïntroduceerd die geavanceerd redundantiebeheer toevoegt aan NVP, met als doel de interne

redundantieconfiguratie autonoom af te stemmen op de waargenomen verstoringen. Deze adaptieve fouttolerante strategie is ontworpen om een hoge beschikbaarheid en betrouwbaarheid te handhaven en kan redundantieniveau en de selectie van functioneel equivalente componenten die door het redundantieschema worden gebruikt, dynamisch aanpassen. Daarbij berust het algoritme op een aantal metrieken om de doeltreffendheid van de redundantieconfiguratie en de gebruikte onderliggende componenten te evalueren, op vlak van betrouwbaarheid en tijdigheid. Simulatietechnieken werden aangewend om de kenmerken van het algoritme, het systeem en de omgeving waarin het zal functioneren te modelleren. Hierdoor kon de doeltreffendheid van het algoritme geanalyseerd worden, en werd aangetoond hoe het algoritme de tekortkomingen aanpakt die typisch verbonden zijn aan het gebruik van de conventionele NVP-techniek.

List of Acronyms

$A/S/c/k/n/D$ A commonly used notation to theoretically classify queuing models, and to denote their analytical properties [115, Chapt. 1]. More specifically, the notation allows to characterise the input or **arrival process** A, the **service time distribution** S, the number of available **server instances** c, the **buffer size** k, the size of the **calling population** n, as well as the **scheduling discipline** D. Throughout this thesis though, it is assumed that $k = n = \infty$, while considering an FCFS-based scheduling discipline D, allowing the use of the shortened notation $A/S/c$ — *cf.* assumption (A07). A queuing model can be seen as some processing facility where requests arrive at random times, where they receive service, after which they depart. The (inter-)arrival times are characterised by the arrival process A, whereas the service (processing) times are characterised by the service time distribution S. The system encompasses a waiting queue buffer, in which a maximum of k requests may be temporarily kept on hold before they can be served. At any time, at most c requests can be simultaneously serviced by the processing facility. When the processing facility becomes available, the scheduling discipline D is responsible for deciding which deferred request is to be served next. The cardinality of the set of all types of requests permissible in the system is denoted by n — *cf.* (A41), p. 193.

AV The technique of **acceptance voting** refers to a hybrid approach in which a traditional NVP scheme is extended to include the concept of acceptance tests that is commonly found in RBs. Here, the acceptance test will be used to validate the correctness of individual ballots prior to feeding them to the voting algorithm [71, pp. 162–172]. See also: RB, NVP.

AV **Active voting**: a decision algorithm for use in NVP redundancy schemata in which, for a given invocation of the NVP composite, all n versions are queried simultaneously, and the first acquired syntactically valid response will be returned. Although it does succeed in masking EVF failures, this voting algorithm may fail to mask RVF failures — *v.* Sect. B.2.

BPMN — In contrast to the W3C® and OASIS® specifications mentioned throughout this dissertation, which are all XML-driven specifications, the **Business Process Model and Notation** specification is a standardised graphical notation to model business processes. It was proposed by the OMG® [95].

CORBA® — The **Common Object Request Broker Architecture** is a set of (protocol) specifications that emerged from intense standardisation initiatives coordinated by the OMG®, and has been designed as an "open, vendor-independent architecture and infrastructure" for bridging the technological disparities when integrating software components during the implementation of distributed computing applications.

DES — **Discrete event simulations** have proven to be extremely useful in analysing the behaviour and properties of complex systems. They can be seen as computer programs written in such a way that they mimic the behaviour of the system under investigation [73, pp. 380–382] [74]. In modelling the system, it is formalised from 2 distinct perspectives. Firstly, a set of variables and/or class objects is identified that can represent and reflect the state the system is currently in. Secondly, the system's operational and the environment's behaviour is analysed, resulting in a set of dedicated simulation entities (events). Each of these events have a specific life cycle, during which they can affect the system, its state and/or its environment. It is the aggregate result of the state changes brought about by these identified events that approximates the actual system's behaviour.

DRA — A **data re-expression algorithm** is an algorithm that is used to transform the original input data sent upon invocation of a redundancy scheme. Rather than replicating the inputs, they can be preprocessed so that normalised and/or slightly rounded or rectified values are used for the subsequent version invocation requests. It is a form of data redundancy that can be used for tolerating software faults, primarily in software implementations that are fed with noisy or imprecise data, or that include lots of arithmetic operations on floating-point numbers [71, p. 21].

dtof — The **distance-to-failure** metric was designed to assess the effectiveness of a given redundancy configuration used throughout the life span of a single completed voting round $(\mathcal{C}, \ell)$ from a dependability perspective. It can serve as an indication of the proximity of potentially hazardous situations that may necessitate the adjustment of the currently employed redundancy configuration so as to ensure the sustainment of the availability of the composite's service — *v.* Sect. 3.1.1 and Chapt. 4.

EDA — **Event-driven architectures** rely on **publish-and-subscribe models** for the asynchronous exchange of specific data fragments called *events* [138, Sect. 10.6]. In such type of models, there is an interested party — the event sink — that issues a subscription request, *i.e.* it states its interest in a specific type of events. The other party — commonly referred to as the event source, or the publisher — may accept the request, thereby pledging to send out event messages asynchronously [22, Sect. 7.7]. That is to say, events are published without delay, avoiding the need for the client to periodically poll. See also: WS-Notification and invocations in the context of the traditional client-server distributed model.

EPR — An **endpoint reference** is an XML fragment that encapsulates the information necessary for identifying a web service endpoint, such that SOAP messages can be addressed and routed to the corresponding target web service. A URI is commonly used to identify the destination, supplemented by additional reference parameters [54, Sect. 18.2].

EVF — **Erroneous value failures** may emerge from the activation of a latent software design fault, causing the normal flow of execution to be interrupted abruptly by means of an exception — fault message, that is — being thrown — *v.* Sect. 2.6.2.

FCFS — **First come, first served**: a type of scheduling discipline D that, despite its simplicity, is commonly used to ensure the principle of *fairness* in scheduling requests that are awaiting processing by the service facility. See also: the Kendall notation $A/S/c/k/n/D$.

FCU — A **fault containment unit** is a well-structured (software) solution that encapsulates a specific fault-tolerant scheme, and that is designed so as to "confine the effects of a fault [originating from the use of underlying resources]" to a limited locality, and "prevent the effects of that fault from propagating throughout [the] system" in which it is used [3]. Examples include NVP and RB schemata.

$G/G/c$ — A specific, though abstract type of queuing model in which a general, unspecified distribution G is used to emphasise independent arrival and service times. See also: the Kendall notation $A/S/c/k/n/D$.

HTTP — Emerging from a joint standardisation effort coordinated by IETF® and W3C®, the HTTP protocol has become the foundation for (textual) data exchange over the Internet [28, Sect. 7.3.4]. It is commonly used in XML-based SoA solutions to exchange SOAP messages amongst web services. Related: TCP/IP.

IETF® — The **Internet Engineering Task Force** is an open standardisation organisation whose primary objective is to propose, define, and promote standards related to Internet architecture. Standardisation initiatives are channelled through and published as *Request for Comments* (RFC): these memoranda are authored by the voluntarily participating members and describe the proposed innovations, methods and behaviours. The principal achievement emerging from the numerous standardisation initiatives is, without doubt, the Internet Protocol Suite (TCP/IP).

LAN — **Local area network**: a network offering connectivity between computing devices located into the same geographical entity.

LRF — When invoking a processing facility, the invocation is said to suffer from a **late response failure** whenever the system fails to deliver a response within the specified time constraints. This type of failure is usually caught and signaled by means of a performance failure — *v.* Sect. 2.6.4.

MAPE-K — The MAPE-K control loop structure is an abstract view on the essential capabilities that are needed to implement autonomic computing. It was originally published in [49] in an attempt to apply the concept of feedback loops to distributed computing environments. Such control loops are able to **monitor** the environment in which they operate, **analyse** the perceived environmental behaviour and characteristics, in order to **plan** the adjustment of their configuration, that will eventually be applied (**executed**). These four capabilities rely on the presence of contextual **knowledge**.

MoM — A **message-oriented middleware** solution is a set of software libraries that collectively support the sending and receiving of messages that result from requesting the service and functionalities exposed by a distributed computing system. MoM solutions are available both as commercial and open-source offerings; either type usually incurs a license fee. They come with an embedded application server, and software runtime libraries to support various specifications in support of common tasks and duties like message processing, transaction management, orchestration, service discovery and federation, security and access control, *etc.* Examples include specifications that are part of the WS-* stack, or the Java™ platform.

MoWS — Part of the WSDM specification, the **Management of Web Services** specification outlines how web services themselves can be used and managed as WS-Resource-compliant entities using MUWS. It also outlines a list of service-level and operation-level metrics and status models, as well as a request processing state model, which was derived from the WSLC specification.

MTBF — The **mean time between failures** is the average time span between any two successive points in time when the service the system is aiming to provide, suddenly becomes unavailable (again). Each such period is immediately proceeded by a period during which the system behaved correctly and the availability of the service is seeks to provide is sustained in full — *v.* Sect. 6.1.3. See also: MTTF, MTBF, and MTBFO.

MTBFO — A useful measure for approximating the failure rate of a specific software entity, the **mean time between failure occurrences** is defined as the average time between any two consecutive failure occurrences, both of which result in disturbances affecting the availability of the software entity under consideration — *cf.* MTBF.

MTTF — **Mean time to failure**: the average duration of all periods throughout the system's operational life during which its availability is sustained in full and without interruption, *i.e.* it behaves in line with its (non-)functional specifications — *v.* Sect. 6.1.3. See also: MTTR and MTBF.

MTTR — **Mean time to repair**: the average duration of all periods throughout the system's operational life, each of which is characterised by a prolongment of unavailability, as the system continuously fails to meet the objectives defined in its (non-)functional specifications, resulting from the occurrence of disturbances — *v.* Sect. 6.1.3. See also: MTTF and MTBF.

MUWS — Part of the WSDM specification, the **Management Using Web Services** specification defines a set of foundational *manageability capabilities*, each of which defines a set of operations, events, WSRF-RP-compliant metadata and other semantics supporting a particular management aspect of a WS-Resource service. Apart from a set of predefined foundational manageability capabilities, the specification also outlines how domain-specific capabilities can be designed, which comprise customised manageability logic, and which may extend any of the foundational capabilities as appropriate.

MV — **Majority voting**: a decision algorithm commonly used in NVP redundancy schemata in which, for a given invocation of the NVP composite, a qualified (absolute) majority is sought amongst the n acquired outputs in order to determine the result [7] — *cf.* Eq. 3.1, p. 67. See also: PV.

NVP — In reliability engineering, n**-version programming** has been successfully used as a design diversity pattern for achieving software fault tolerance [5, Chapt. 4] [4, pp. 60–66] [2, Sect. 7.3]. An n-version module constitutes a fault-tolerant software unit — a client-transparent replication layer in which all $n > 1$ functionally-equivalent programs, called versions, receive a copy of the user input and are orchestrated to independently perform their computations in parallel. It relies on a decision algorithm to determine a result from the individual outputs of the versions employed within the unit — *v.* Sect. 2.1.

OASIS® — The **Organization for the Advancement of Structured Information Standards**: a global consortium whose primary objective is to "drive the development, convergence and adoption of open standards for the global information society", especially for e-business and web service standards.

OMG® — The **Object Management Group**: an "international, open membership, non-profit technology standards consortium" whose objective is to develop "enterprise integration standards for a wide range of technologies". The consortium primarily focusses on modelling system properties and behaviour, resulting in popular standards like UML™ and BPMN.

PNPN — **Priority-based** queue: a type of scheduling discipline D that is commonly used to enforce various SLA levels in scheduling different types of requests that are awaiting processing by the service facility. See also: the Kendall notation $A/S/c/k/n/D$.

PV — **Plurality voting**: a decision algorithm occasionally used in NVP redundancy schemata in which, for a given invocation of the NVP composite, a non-qualified majority is sought amongst the n acquired outputs in order to determine the result [7] — *cf.* Sect. B.2 and Eq. 3.1, p. 67. See also: MV.

QoE — **Quality of experience**: see QoS.

QoS — **Quality of service**: an indication of the overall performance of a networked telephony or computing solution, as perceived by the end-user. Hence the synonymous use of the term quality of experience (QoE).

RB — In reliability engineering, **recovery blocks** has been successfully used as a design diversity pattern for achieving software fault tolerance [4, pp. 60–66] [2, Sect. 7.3]. The iterative behaviour of recovery blocks schemata stands is sharp contrast to the parallel constellation applied in NVP schemata, in that each of the n functionally-equivalent versions is probed in sequence. More specifically, each iteration involves the invocation of a single version, and encompasses 3 phases: (i) *checkpointing*, *i.e.* saving the system's current state information, (ii) invoking the next version and waiting for a response to be secured, and (iii) subject the acquired result to an *acceptance test*. In case the test evaluates positively, the result will be returned, and no further actions will be taken. Otherwise, the system will be rolled back to its previously checkpointed state, and another iteration will be initiated, given that there remain untested versions.

RPT — **Request processing time**: the duration during which the request is being serviced by the processing facility. In discrete event simulations, RPT values are commonly generated by drawing random variates from the service time distribution S, and are therefore commonly referred to as request service times.

RTT — Throughout this dissertation, the **round-trip time** is considered to be the time required for transferring the request and response messages to/from a $G/G/c$ processing facility throughout the invocation of an operation exhibiting a request-response message invocation pattern. The RTT does *not* include the RPT at the processing facility, *nor* potential waiting times.

RVF — Despite being classified as failures, the occurrence of **response value failures** will usually not make the system appear to fail — *cf.* Fig. 2.3 on p. 45. Rather, the content of the response returned via the service interface is syntactically correct, though diverges from implementing the service's functional specification — *v.* Sect.2.6.1.

SDLC — The term **software development lifecycle** refers to all activities required to create and deliver software, ranging from requirements analysis, architecture and design, implementation (development), quality assurance (testing), system operations (deployment, monitoring, *etc.*). It also includes activities like project and change management and the like.

SLA — A **service level agreement** is a formal definition of particular qualitative aspects of the service as it is expected to be delivered by some software entity. It is "agreed upon by the respective owners of a service [provider] and its requestors" [22, 140].

SoA — **Service-oriented architecture**: a popular, multi-disciplinary paradigm used in the design of enterprise architectures [22, 186]. Throughout this thesis, SoA is used to denote contemporary distributed systems with a particular focus on software components wrapped as XML-based web services.

SOAP — The **Simple Object Access Protocol** is a lightweight procotol intended for exchanging structured information between XML-based web services. Valid SOAP messages should be structured as *envelopes*, in which the message *body* carrying the payload is clearly separated from the optional SOAP *headers* which carry specific information in support of the WS-* feature set. Among SOAP header blocks, WS-Addressing elements are commonly found. Valid messages are usually exchanged over HTTP, although other transmission protocols are available [17] [54, Chapt. 11].

SSJ — **Stochastic Simulation in Java™**: a Java™-based software library developed at the Université de Montréal which provides numerous facilities for programming discrete event simulations [81]. Is has been used for conducting performance analyses of the algorithms that emerged from the research activities reported in this dissertation — *v.* Chapt. 6 and 7.

TCP/IP — The **Internet Protocol Suite** comprises a set of networking protocols upon which the Internet and most computer networks rely. Maintained by the IETF®, the relevant RFCs describe protocols for data packetisation, addressing, transmission, and routing. Together, they provide a robust solution for end-to-end networked connectivity. The **Internet Protocol (IP)** defines how packets ought to be structured from an addressing perspective, such that source and destination information is carried alongside the encapsulated payload [28, Sect. 5.6]. The **Transmission Control Protocol (TCP)** is responsible for "regulating the flow of internetwork [IP datagram] packets", and comes with a robust error detection scheme for retransmitting unacknowledged packets, as well as facilities for transmitting chunks of data and keeping packets properly ordered [28, Sect. 6.5]. Packetised network traffic resulting from HTTP sessions is usually transmitted over TCP/IP-enabled networks.

TMR — A term used to denote a redundancy scheme in which three redundant resources (versions) are used to mask failures. The concept can be applied to hardware as well as software redundancy patterns — an example of the latter being RB and NVP schemata. As the emphasis is placed on NVP, the term is used to indicate an NVP redundancy scheme with a fixed degree of redundancy $n = 3$ — regardless of whether a static or dynamic redundancy configuration (replica selection) applies. Similarly, for other redundancy levels, the term n-modular redundancy, or NMR, is used.

TPA — As the name implies, the technique of **two-pass adjudication** includes two voting round *passes*: the first pass is fed with the original inputs; if that fails to adjudicate an outcome, a second pass is initiated, which is fed by re-expressed parameters [71, pp. 218–231]. See also: DRA, NVP.

UDDI — Despite the attempt to position the **Universal Description Discovery and Integration** specification as a core web service standard, it has often been perceived as overly complex and has never seriously gained ground. It was proposed by OASIS® as a platform-independent, XML-based registry in which enterprises can publish web services, and which can be interrogated by regular SOAP messages so as to locate suitable web services [187].

UML™ — The **Unified Modelling Language** is a popular "general-purpose visual modelling language [proposed by the OMG®] that is used to specify, visualize, construct, and document the artifacts of a software system" [188].

URI — With syntactical guidelines set out in RFC 3986, **uniform resource identifiers** enable to effectively identify, name, and address network (internet) resources by means of a compact sequence of characters. They are used to identify web service endpoints by means of endpoint references [189].

UV — **Unanimity (or consensus) voting**: a decision algorithm rarely used in NVP redundancy schemata. The term unanimity emphasises the requirement for all n versions to be in mutual agreement before a response can be adjudicated. Despite its limited applicability and inability to mask failure occurrences, it does lend itself to applications for which it is preferable to return all but an incorrect result — *v.* Sect. B.1.

W3C® — The **World Wide Web Consortium**: an international organisation whose primary objective is the definition of web standards.

WAN — **Wide area network**: a network offering connectivity between computing devices situated across multiple local area networks.

WS-* A set of **web services specifications** that emerged from numerous W3C® - and OASIS®-driven standardisation initiatives. Due to the complementary approach in the way most of these specifications have been defined, the resulting *web service protocol stack* has proved to be extremely useful in defining, locating, and implementing XML-based web services, as well as making services interact with each other. As can be seen in Fig. 8.2 on p. 184, the WS-* stack covers transport, messaging and eventing, description, discovery, management, and security protocols (the latter are not shown though) [22, Chapt. 6–7, 17].

WS-Addressing The **Web Services Addressing** specification was set out by the W3C® as a set of "transport-neutral mechanisms that allow web services to communicate addressing information" and conversational attributes. It has become an essential part of the web services specifications, defining a series of XML vocabularies for identifying and communicating references to concrete web service endpoints, enriching the expressiveness of regular endpoint references. With WS-Addressing information being carried as SOAP header blocks, the specification effectively contributes to supporting message transmission and delivery over message-oriented middleware solutions, thereby offering a valuable solution to potential issues frequently caused by processing nodes such as gateways and firewalls. Furthermore, it outlines the fundamentals for establishing conversational contexts and message correlation [133] [54, Chapt. 18–19] [22, Sect. 7.1].

WS-BPEL As its name implies, the OASIS® **Web Services Business Process Execution Language** is a standardised executable XML-based language in which business process activities can be described by orchestrating data flows, in which information is retrieved from and communicated to web service endpoints [143]. See also: WS-BPEL4People and WS-HumanTask.

WS-BPEL4People The **WS-BPEL4People** specification emerged as an OASIS® standardisation initiative in an attempt to cover the complete spectrum of **human-to-process interaction** so as to fully support the modelling of hybrid, **people-centric business processes**. In this type of processes, human actors may participate in specific process activities, next to the customary IT systems exposed as web services. Its purpose was primarily to address the shortcomings of WS-BPEL, which had previously been regularly criticised for being too automation-centric, since it lacked the recognition of employees in process workflows. The WS-BPEL4People extension defines a number of features layered on top of WS-BPEL to seamlessly integrate WS-HumanTask tasks into WS-BPEL process definitions. For more information, please refer to App. C. See also: WS-BPEL and WS-HumanTask.

WS-HumanTask — Ratified by OASIS®, **WS-HumanTask** proposes an industry standard for defining and managing human-based activities in people-centric business processes. On a higher level, the WS-BPEL4People extension defines a number of features layered on top of WS-BPEL to seamlessly integrate WS-HumanTask tasks into WS-BPEL process definitions. The main contributions of this extension are (i) an interoperable coordination protocol that was conceived to attain a tight coupling with the synchronisation of state between the **task** itself — an individual unit of work to be processed by a human actor — and the process instance which it is part of, and (ii) the concept of a **task box** as a central point of interaction through which human actors may manipulate tasks and control their life cycle. For more information, please refer to App. C. See also: WS-BPEL and WS-BPEL4People.

WS-MEX — A W3C® standardisation initiative to define an interface and operations for retrieving all or part of the metadata associated with a specific web service endpoint, *e.g.* WSDL interface definitions, or WSRF-RP documents [134] [22, Sect. 7.5]. See also: WS-RMD, WSRF.

WS-Notification — **Web Services Notification**: a set of OASIS® WS-* specifications that form the foundation for building event-driven architectures (EDAs) in SoA. More specifically, the WS-BaseNotification specification covers event message formats and a set of predefined operations which apply the concepts of publish-and-subscribe models to web services [139] [24]. Supplementing the WS-Notification specification, the WS-Topics specification by OASIS® describes various filtering options with which the subscriber can express its particular interests, *e.g.* by means of an XPath query that is validated against the notification message payload. A competing, albeit inferior standard, is WS-Eventing, which resulted from a W3C® initiative.

WS-Policy — Authored by the W3C®, the **Web Services Policy** family of WS-* specifications encompasses a number of complementary specifications designed for associating non-functional constraints and requirements to web service endpoints, primarily in the areas of security and quality of service. Examples might be required security tokens, supported encryption algorithms or coordination protocols. An extensible mechanism is put forth (i) for advertising such attributes by means of *policy assertions*, and (ii) for the validation and enforcement of compliance to such attributes by service requestors [190] [22, Sect. 7.4] [54, Chapt. 16–17].

WS-Resource — Part of the OASIS®-driven WSRF standardisation initiative, the foundational **Web Services Resource Framework: Resource** specification outlines how a WS-Resource results from the composition of a (stateful) resource and a web service through which the resource can be accessed, controlled and/or monitored. Several specifications from the WSRF and WSDM family may be required to support such features though.

WS-RMD — An OASIS® specification for defining metadata for resource properties, which allows to enforce value restrictions and access control, supporting, *e.g.* mutability and modifiability [137]. See also: WSRF-RP and WS-MEX.

WSDL — Without doubt the predominant W3C® specification in the WS-* stack, the **Web Services Description Language** specification has been widely and successfully used as an XML-based interface description language [18]. It outlines the details for structuring and describing the functionality offered by web services, *port types*, *i.e.* the set of exposed operations, and the permissible message (payload) data types and interaction patterns [22, Chapt. 7–9].

WSDM — **Web Services Distributed Management**: a family of OASIS®-driven WS-* specifications conceived to expose a web services-based manageability layer for applications and/or stateful resources, resulting in improved controllability and interoperability, even across enterprise and organisational boundaries [109]. It is comprised of the MUWS and MoWS specifications. The underlying resource is wrapped inside a WS-Resource-compliant entity, through which the manageability layer can be accessed by means of a single, coherent WSDL interface. See also: WSRF.

WSLC — **Web Service Management: Service Life Cycle**: a W3C®-driven initiative to standardise the life cycles of web services, and request processing [94]. Formally expressed by means of state transition diagrams, they lay the foundation for the service and request processing state models defined in the MoWS specification.

WSRF — The **Web Services Resource Framework** is a family of specifications authored by OASIS® that aim to "define a generic and open framework for modelling and accessing stateful resources using web services". It builds on two foundational specifications: WS-Resource and WS-BaseFaults, the latter intent on outlining an extensible mechanism for defining rich SOAP faults. The other members of the family build on top of these:
- WS-ResourceLifetime: defines an interface and operations to manage the lifecycle of potentially short-lived WS-Resource resources;
- See also: WSRF-RP and WSRF-SG.

Related: WS-RMD. The manageability features targeted by WSDM heavily rely on WSRF.

WSRF-RP — **Web Services Resource Framework: Resource Properties**: an OASIS®-driven WS-* standardisation initiative which outlines an extensible mechanism for exposing additional, stateful information by means of a set of typed values — called *resource properties* — in the WSDL interface of a WS-Resource, and protocols for querying, reading and manipulating these metadata properties in a standardised format [136].

WSRF-SG — Part of the OASIS®-driven WSRF standardisation initiative, the **Web Services Resource Framework: Service Group** specification comes with interfaces and operations for managing service groups, *i.e.* "[potentially] heterogeneous by-reference collections of web services". It can be used as a lightweight alternative for UDDI, for it can serve as a lightweight service registry solution, capable of structuring entire federations of web services.

XML — The widely used **eXtensible Markup Language** specification was set out by the W3C® as an open standard for encoding documents, and arbitrary data held within, in a self-descriptive format that is both machine- and human-readable. Tagging is used to annotate the data held within documents, such that metadata is syntactically distinguisable from basic text values. WS-* specifications heavily rely on XML, primarily for defining the structure of standardised message formats that are exchanged when interacting with web services.

XPath — Authored by the W3C®, the **XML Path Language** specification describes a compact syntax for selecting nodes and values within an XML document. It operates on the "logical [tree] structure of an XML document" and proposes a set of operators for navigating the underlying DOM tree, as well as a basic set of operators for performing simple computations on (node) values [191]. Related to: WS-Topics.

XSD — **XML Schema Definition**: a W3C® extension of the XML specification, which can be used to formally describe the intended structure and content of XML documents. XSD schemas, which are themselves structured as XML documents, are commonly used for validation purposes: they allow to verify that XML fragments are syntactically correct and therefore interpretable in specific application contexts.

List of Terms

ballot	A ballot is the response value or failure message returned from a single subordinate invocation of one specific version used within an NVP redundancy scheme. Ideally, whilst processing a request, the scheme will be able to secure n ballots, one for each underlying version. These will then be analysed by the adjudication mechanism so as to overcome eventual discrepancies and determine a single satisfactory *outcome* to be returned.
business process	From a service-oriented perspective, a business process can be seen as a well-structured workflow, and the corresponding data flow, defined on a set of process activities, each of which is to be carried out by computer entities, computer systems and/or people within an enterprise, with the final objective of supplying a product or service to the customer [68, 86]. With web service (operations) encapsulating and exposing accessible implementations of individual process activities, a business process can be automated by appropriately orchestrating and coordinating a set of services by means of a WS-BPEL process definition.
disturbance	The event of a single request being struck by some type of failure, resulting in the perturbation and, consequently, the (temporary) unavailability of the service that the software entity that is processing the request is expected to provide.
error	Resulting from the activation of a fault, an error makes the affected entity transition into an inconsistent state, inhibiting the entity to pursue its normal operations as specified. The (in)direct effect this may have on the associated service is called a failure.
failure	A failure or malfunction is any situation, resulting from the activation of an error, under which the delivered service by the affected entity is behaving in an unexpected way, and deviates from the expected service as it was anticipated and specified.
fault	An unintentional defect, a design or implementation flaw that was overseen during the engineering and testing of a software entity. When activated, it will manifest itself in an error.

invocation	In the traditional client-server (distributed) model, when issuing a request for service, there are two software entities involved: a client entity, and a server entity. The invocation of the server entity is initiated by the client entity by issuing a **request message** destined for the server entity. As soon as the request message has been handed over to the server entity, the request will be served and processed, after which a **response message** will be sent back to the client entity. The lifecycle of a single invocation is modeled in the lower part of Fig. 2.1. When invoking an entity, a single operation — a handle to a specific software processing routine — is called that is defined in the entity's interface. As the subject matter of this dissertation is on redundancy management within NVP-based systems, only operations are considered that are accessible through a *request-response message exchange pattern* [22]. In XML-based SoAs, these operations would be defined in the WSDL interface definition, as would be the XSD definition of the permissible payload that will be transmitted inside the body of the SOAP request and response messages. The terms invocation and **request** are used interchangeably.
operational life	Once a software entity is deployed, the operational life is the total time in which the entity is running and processing requests, starting from the moment the entity is launched, *i.e.* it is instructed to start its operations, until the system ceases its operations, either because it is explicitly instructed to do so, or because of a crash failure.
pending request	A request is said to be pending as long as it it is being serviced, or planned to be serviced by the software entity being queried. As a software entity is essentially a processing facility, whose properties can be characterised by some queuing model, it corresponds to that part of the invocation's lifecycle during which it has entered, but not yet left the model — *v.* Fig. 2.1. In other words, the invocation has been received and accepted for treatment by the processing facility.
redundancy scheme	A commonly used architectural style for designing fault containment units, in which "some form of redundancy — time, information, [...] hardware and/or software redundancy — [is exploited]" for masking potential disturbances [11, Sect. 2.2]. Redundancy schemata rely on dedicated circuitry and/or logic in support of the execution, and for adjudication purposes in particular. When applied to software, the n underlying redundant resources correspond to functionally-equivalent software entities that supply the same service and functionality, though each has been designed and implemented by independent development teams. Examples are NVP and RB schemata.

software entity	Considering the compositional nature of many large-scale distributed applications and the role of *software components* as building blocks, the abstract notion of software entity is used for generalisation purposes. At the lowest level, it can be used to refer to *single-component software entities*, as well as *composite software entities* that encapsulate logic for properly orchestrating other software entities encompassed or used within. A component of the former type shall be referred to as ***version***. Redundancy schemata such as, for instance, NVP serve well to exemplify this latter type of software entities. At the highest level of composition, the term is used to refer to a (distributed) application in its entirety, that is to say the software system itself.
software system	At the highest compositional level, this type of software entities refers to software applications composed of multiple interconnected software entities, possibly deployed on top of a dedicated network infrastructure, in which lower-level software entities are arranged so as to provide an automated end-to-end software solution in support of specific functionality and/or a specific set of business processes. The term will be occasionally used to refer to a large-scale distributed computing solution in its entirety, and is used as synonymous with *enterprise application*.
version	A single-component software entity. The term is particularly used in the context of redundancy-based fault-tolerant schemata; software entities in which multiple versions are utilised for masking potential failure occurrences. It is in that context that each version is expected to encapsulate an "independent[ly developed] functionally-equivalent program[..., all sharing] the same initial specification" [6]. See also: RB, NVP.

List of Assumptions

In order to improve readability, all of the assumptions, hypotheses and design choices introduced throughout this dissertation have been summarised and listed here below. The abstract notion of **software entity** is used for generalisation purposes: it can be used to refer to single-component software entities, which shall be referred to as **version**, as well as composite entities that encapsulate logic for properly orchestrating other software entities encompassed or used within. Redundancy schemata such as, for instance, NVP serve well to exemplify this latter type of software entities.

(A01), p. 41 Request processing will commence as soon as the scheduling discipline removes the request from the waiting queue buffer and hands it over to the processing resource — version — *cf.* (A07). From that point on, until it has been completely treated, the request is said to be pending, and will receive service from the processing resource handling it, which will apply the business logic held within.

(A02), p. 45 Throughout this dissertation, the notion of disturbance is used to denote the (temporary) unavailability of the service that the affected version is expected to provide. It is the result of the activation and manifestation of one or more latent design faults in the underlying business logic, and is triggered when handling pending requests. Although not explicitly considered, other types of defects also qualify, including hardware and network connectivity issues.

(A03), p. 45 The processing of a request is assumed to yield a result compliant to the functional specification of the software component (version) that is responsible for its servicing, provided that this component was not affected by any type of disturbance during the request processing time. See also: (D01) and (D02).

(A04), p. 45 Contemporary distributed computing systems are assumed to exhibit the properties of a timed asynchronous distributed system model. The behaviour and the potential anomalies that may cause disturbances are characterised by means of four properties on p. 45.

(A05), p. 46 When a software component is struck by a crash failure, it will persistently exhibit omission failures for a prolonged period of time. Affected components usually never recover without administrative intervention by the system operator. See also: software rejuvenation (p. 96).

(A06), p. 46 Since the scope of this thesis is largely situated in the area of application-level fault tolerance, the emphasis is primarily placed on software design faults. The network datagram service, the middleware deployment environment and the NVP composite itself — including the embedded decision algorithm — are assumed to behave properly without any disturbances appearing.

(A07), p. 46 Throughout this thesis, it is assumed requests are admitted for processing by a specific version based on a $G/G/c$ queuing model that relies on an FCFS scheduling discipline. Recent advances in computer hardware and in distributed clustering technologies have allowed distributed computer systems to scale and measure up to virtually any demand, supporting the simplification that $k = c = \infty$.

(A08), p. 47 The activation of a dormant design fault pertaining to a specific software entity will directly result in the emergence of a disturbance, and thus immediately manifest into a failure without any delay, resulting the perturbation of the entity's availability.

(A09), p. 47 The manifestation of a non-persistent disturbance originating from the activation of a software design fault can only affect the availability of the corresponding software entity within the scope of execution resulting from a single invocation of the entity. Given that the disruptive effects of such disturbance be confined to a specific scope, other invocations of the same software entity, which may be serviced simultaneously, will remain unaffected. This does not apply to persistent disturbances — crash failures, that is. See also: (A21).

(A10), p. 47 The potential disruptive effects caused by non-persistent disturbances will dissipate as soon as the scope of execution from which they emerged due to the activation of a design fault, has completed, and a result is handed over to the middleware environment for transmission to the requesting party. See also: (A21).

(A11), p. 48 Software design faults can only manifest as content or crash failures — *cf.* Fig. 2.3 on p. 45.

(A12), p. 48 Content failures can emerge only when a latent software design fault is activated along the execution path of the business logic held within a software component. This can occur only when the component is actually processing requests.

(A13), p. 48 As an essential part of any voting round, a decision algorithm is responsible for securing all ballots and adjudicating a single outcome. To do so, it will compare the ballot values acquired, and create a partition to categorise these into equivalence classes.

(A14), p. 49 When classifying and comparing response values, a distance function can be used to determine whether any two values are equivalent. This function determines the equivalence classes that will be generated as elements of the partition — *cf.* (A14).

(A15), p. 50 Response values can be sampled as random variates drawn from a normally distributed random variable. In doing so, outlier response values are less likely to occur than minor deviations from the exact result. The exact result is the expected result that should be generated when a request was processed in full without triggering disturbances in the underlying software component during the request request processing time. See also: (A17) and (D02).

(A16), p. 50 Related to (A13), (A14) and (A15): each equivalence class will accommodate the same amount of variability in the response values obtained for the versions classified within, in comparison to the exact value.

(A17), p. 51 Response values can be sampled as random variates drawn from a uniformly distributed random variable. Compared to (A15), this is likely to cause more divergence of the generated response values, which typically reduces the ability to adjudicate a consensus (majority) due to a significantly smaller expected cardinality of the equivalence classes in the generated partition — *cf.* Fig. 2.5. See also: (D02).

(A18), p. 52 For performance reasons, we assume that response values are classified in the order they are acquired. Although this does affect the construction of the partition that is used by the decision algorithm, this does allow to return an outcome before all ballots have been secured. See also: (A13).

(A19), p. 54 The exceptional behaviour caused by the occurrence of an EVF failure is transient and will disappear immediately such that it will only affect the relevant pending request(s) that were being serviced at the time the failure occurred. See also: (A20).

(A20), p. 54 EVF failures can only affect pending requests that were being serviced by the affected software component. See also: (A12) and (A19).

(A21), p. 55 When a crash failure is activated during the servicing of a specific invocation of a software entity, the entity will become permanently unresponsive. The entity will henceforth exhibit omission failures for any pending request it was servicing at the time of failure occurrence, as well as any subsequent request it was offered for processing after the failure had originally occurred. See also: (A08) and (A10).

(A22), p. 55 Crash failures can only arise as the result of the activation of a latent software design fault along the current execution path followed while the affected software component was busy processing one or more pending requests.

(A23), p. 55 The activation of a crash failure is assumed to affect only the availability of the affected software component, and does not affect the behaviour of the deployment environment. Consequently, since the software component is assumed to be managed within a queue-based processing facility, inbound requests will be accepted and buffered in the waiting queue without ever receiving service — *cf.* (A07).

(A24), p. 55 For requests susceptible to crash failures, the associated omission failure will manifest as performance failures. See also: (A25).

(A25), p. 56 A performance failure manifests only when a response for a request could not be acquired in due time.

(A26), p. 56 A performance failure does not necessarily emerge due to the activation of a design fault; it can materialise from the interplay of several endogenous and exogenous conditions like the computational capacity, load pattern, *etc.*

(A27), p. 60 During the course of its lifetime, a request may be affected by multiple types of disturbances.

(A28), p. 60 During the RPT, a request can be affected by multiple occurrences of a particular type of failure class. Within the scope of a single request, these occurrences are assumed to be idempotent and need not necessarily all be treated by the simulation model — *cf.* (D01) and (D02). This does not apply to performance failures though: at most one such failure can occur for a given request.

(A29), p. 63 Intuitively, the business logic is a set of software routines, each of which is exposed as an *operation* in the interface that is published for a specific software component. Throughout this dissertation, unless explicitly stated otherwise, it was assumed that a software component exposes only a single operation. This does not restrict the applicability of our contribution and merely serves as a simplification to reduce complexity. Generally speaking, different units of context information should be maintained for each specific software routine. Since each software routine is expected to implement different functional specifications, the proposed algorithm will treat each as completely isolated units. See also: Sect. 2.7 and 5.4 and (A36).

(A30), p. 67 Even if a majority can be found by the voting algorithm, this outcome does not necessarily match the exact value — *cf.* (A18), (A03) and (A13). See also: App. B.

(A31), p. 68 Continuing on (A30): if the decision algorithm manages to find a majority, this will be assumed to match the exact value (allowing for some reasonable deviation). The inability to establish a majority, however, will be regarded as a failure of the redundancy scheme.

(A32), p. 82 Within the scope of an NVP redundancy scheme applying a majority voting-based decision algorithm, a given voting round is observed to have completed successfully if a sufficiently large degree of consent could be found between the acquired ballots such that a qualified majority could be found. Here, each ballot is the contribution of a specific version selected by the redundancy configuration applied for that voting round — *cf.* (A31).

(A33), p. 82 Applying more aggressive redundancy management policies may result in an incautious downscaling of the redundancy, which in itself might lead to failure of the redundancy scheme in subsequent voting rounds. Furthermore, temporarily refraining from increasing the employed degree of redundancy after observing a potentially hazardous situation might prolong the scheme's unavailability. See also: (A34).

(A34), p. 83 When applying defensive redundancy management policies, the degree of redundancy is proactively adjusted upwards whenever potantially hazardous situations are detected. This would come at the price of a more aggressive upscaling strategy though, in which system resources would be allocated rather lavishly — *cf.* (A33). Moreover, such type of policies typically translate in a delayed downscaling of the employed degree of redundancy, at the expense of postponing the relinquishment of excess redundancy.

(A35), p. 84 It is assumed the environment in which the system is operating behaves unpredictably, mainly due to exogenous conditions (primarily request arrival patterns, since request processing may trigger the activation of design faults) — *cf.* (A12), (A20) and (A22). As a result, the number of disturbances materialising at any point in time may vary considerably.

(A36), p. 93 It is assumed that the service exposed by a given software entity is remotely accessible as a set of software routines exhibiting a request-response message invocation pattern. A one-way message invocation pattern is not suitable given our current focus on NVP redundancy schemata, in which a comparison algorithm relies on the assumption that a version is expected to produce a result [22]. When exposing the service as a true XML-based web service, a formal service contract will be advertised by means of a WSDL document, in which each exposed software routine corresponds to an *operation* — *v.* Chapt. 8. See also: (A29).

(A37), p. 94 Within the scope of execution of a single invocation, once the actual processing is complete, it is assumed there is no further delay before the response is actually sent back to the requesting party over the network — *cf.* Fig. 6.1 on p. 91. The duties imposed on the message-oriented middleware to take the computed response value, serialising it and embedding it in a syntactically correct response message and have this sent to the intended recipient are assumed to occur instantaneously, without introducing any further delay. We argue that middleware solutions are optimised to achieve extreme levels of performance in dealing with these messaging-related tasks, and that careful capacity planning for outbound network traffic is achievable by proper analysis of the processing facility's queuing model — *v.* Sect. 6.1.1.

(A38), p. 94 The client-specific overhead resulting from preparatory and concluding activities before and after requesting service, are assumed to be negligible, especially when compared to the overall end-to-end response time.

(A39), p. 96 When a software entity is placed into operation, it initially performs correctly, fully in line with its (non-)functional specifications. The assumption is valid for mature software solutions that were submitted to rigorous testing routines in order to ensure the removal of critical software defects. Software is commonly assumed to age throughout its operational life span, and, by doing so, to become increasingly vulnerable to disturbances.

(A40), p. 97 A mission-critical system is not expected to be placed in production until it is found to be sufficiently mature, and has been submitted to rigorous testing routines to detect and remove as many defects as possible. Given that such systems often rely on redundancy schemata, faulty underlying components are deliberately kept running in the assumption that disturbances may be of transient or intermittent nature, and that faulty components can consequently recover and resume normal behaviour. It is assumed that the vast majority of defects that may translate in crash failures can be eliminated by ample and adequate testing.

(A41), p. 193 Throughout this dissertation, the processing resource managed within a queuing model is assumed to be a remotely deployed, network-accessible software component, *e.g.* a web service, which is referred to as version or replica.

(A42), p. 194 The queuing model underpinning a contemporary message-oriented middleware solution constitutes a multiple-channel, single-phase process waiting line structure. Multiple channels are available, since multiple requests can be independently processed in parallel — *cf.* Sect. 8.5.2. Request handling involves a single phase, as any potential form of compositionality of the underlying business logic is fully masked and unknown to the client (service requestor). See also: (A07).

(A43), p. 194 Each of the underlying versions in a redundancy scheme corresponds to a software component containing a self-contained, short-lived atomic unit of business logic. The redundancy scheme itself is a composition of versions.

(D01), p. 45 When simulating request handling, a fault manifestation model is used to indicate the type(s) of disturbance(s) the request was affected by during its processing. Although a request can be affected by multiple types of disturbances, the total order defined in Sect. 2.4 will ensure only the correct (most severe) type will be reported. See also: (A27) and (A28).

(D02), p. 49 In simulating the processing of individual version invocations, no specific request, response and fault messages/values are used. Instead, the simulation model will apply the fault manifestation model outlined in Sect. 2.4 to indicate the appropriate type of disturbance each of the requests was affected by (if applicable). Within NVP constellations, this ensures sufficient information is available when securing ballots to support voting and adjudication of an outcome. See also: (A15) and (A17).

(D03), p. 62 While undergoing service, any pending request is susceptible to crash failures affecting the software component — version — that is currently processing it.

(D04), p. 64 Load statistics are maintained for each version (operation) by the underlying deployment environment (middleware) — *cf.* (A29) and (A36). These are relayed to the NVP composite by means of an asynchronous publish-and-subscribe model. The potential timing overhead of such type of messaging model is accounted for by the counter update discrete event model defined in Sect. 2.7. See also: Sect. 1.1 and 8.2.3.

(D05), p. 79 The redundancy configuration to be used throughout a voting round is determined upon the arrival of the request at the NVP composite, and will take place prior to forwarding the incoming request message to each of the selected versions.

Bibliography

[1] William Wordsworth. *Ode: Intimations of Immortality from Recollections of Early Childhood*, volume 2 of *Poems, in Two Volumes*. Longman, Hurst, Rees and Orme, 1807.

[2] Elena Dubrova. *Fault Tolerant Design*. Springer-Verlag Berlin Heidelberg, 2013.

[3] Barry W. Johnson. *Design and Analysis of Fault Tolerant Digital Systems*. Addison-Wesley Series in Electrical and Computer Engineering. Addison-Wesley Publishing Company, 1989.

[4] Vincenzo De Florio. *Application-Layer Fault-Tolerance Protocols*. IGI Global, 2009.

[5] Hassan B. Diab and Albert Y. Zomaya, editors. *Dependable Computing Systems: Paradigms, Performance Issues, and Applications*. Wiley Series on Parallel and Distributed Computing. John Wiley & Sons, Inc., 2005.

[6] Algirdas A. Avižienis. The n-Version Approach to Fault-Tolerant Software. *IEEE Transactions on Software Engineering*, SE-11(12):1491–1501, December 1985.

[7] Paul R. Lorczak, Alper K. Caglaya, and Dave E. Eckhardt. A Theoretical Investigation of Generalized Voters for Redundant Systems. In *IEEE Digest of Papers on the* 19^{th} *International Symposium on Fault-Tolerant Computing*, pages 444–451, Chicago, IL, USA, 1989. IEEE Computer Society Press.

[8] Behrooz Parhami. Voting algorithms. *IEEE Transactions on Reliability*, 43(4):617–629, December 1994.

[9] Jim Gray and Daniel P. Siewiorek. High-Availability Computer Systems. *Computer*, 24(9):39–48, September 1991.

[10] Dependable Embedded Systems: Software Fault Tolerance. Retrieved 4 March 2017, `http://www.ece.cmu.edu/~koopman/des_s99/sw_fault_tolerance/`.

[11] Vincenzo De Florio. *A Fault-Tolerant Linguistic Structure for Distributed Applications*. PhD thesis, Katholieke Universiteit Leuven, October 2000.

[12] Jean-Claude Laprie. Dependable Computing and Fault Tolerance: Concepts and Terminology. In *Proceedings of the 25th International Symposium on Fault-Tolerant Computing, ' Highlights from Twenty-Five Years'*, volume III, pages 2–10, Pasadena, CA, USA, 1995. IEEE Computer Society Press.

[13] Mazeiar Salehie and Ladan Tahvildari. Autonomic Computing: Emerging Trends and Open Problems. *SIGSOFT Software Engineering Notes*, 30(4):1–7, 2005.

[14] Ian Sommerville. *Software Engineering*. Addison Wesley, Pearson Education, Inc., 8^{th} edition, 2007.

[15] Jan-Willem Hubbers, Art Ligthart, and Linda Terlouw. Ten Ways to Identify Services. *SOA Magazine*, XIII, December 2007.

[16] John W. Satzinger, Robert B. Jackson, and Stephen D. Burd. *Systems Analysis & Design in a Changing World*. Course Technology Press, 4^{th} edition, 2006.

[17] World Wide Web Consortium (W3C). Simple Object Access Protocol (SOAP) specification, May 2000. `http://www.w3.org/TR/2000/NOTE-SOAP-20000508/`.

[18] World Wide Web Consortium (W3C). Web Service Description Language (WSDL) specification, March 2001. `http://www.w3.org/TR/wsdl`.

[19] Alexander. D. Brown. OSGi Demystified: 5.1 — Declarative Services: A Tutorial, 2019. Retrieved 1 May 2020.

[20] Object Management Group (OMG). CORBA Component Model (CCM) specification, April 2006. `http://www.omg.org/spec/CCM/4.0/`.

[21] Organization for the Advancement of Structured Information Standards (OASIS). Reference Model for Service-oriented Architecture, October 2006. `https://www.oasis-open.org/committees/soa-rm/`.

[22] Thomas Erl. *Service-Oriented Architecture: Concepts, Technology, and Design*. The Prentice Hall Service-Oriented Computing Series from Thomas Erl. Prentice Hall PTR, 2005.

[23] World Wide Web Consortium (W3C). OpenAPI Specification (OAS), May 2017. `https://swagger.io/specification/`.

[24] S. Graham P. Niblett. Events and Service-oriented Arcitecture: the OASIS Web Services Notification Specifications. *IBM Systems Journal*, 44(4):869–886, 2005.

[25] Thomas Erl, Andre Tost, Satadru Roy, and Philip Thomas. *SOA with Java: Realizing Service-Orientation with Java Technologies*. The Prentice Hall Service Technology Series from Thomas Erl. Prentice Hall PTR, 2014.

[26] Jan L.G. Dietz. *Enterprise Ontology - Theory and Methodology*. Springer-Verlag Berlin Heidelberg, 2006.

[27] Ian Sommerville, Guy Dewsbury, Karen Clarke, and Mark Rouncefield. Dependability and Trust in Organisational and Domestic Computer Systems. In *Trust in Technology: A Socio-Technical Perspective*, volume 36 of *Computer Supported Cooperative Work*, chapter 8, pages 169–193. Springer-Verlag Berlin Heidelberg, 2006.

[28] Andrew Stuart Tanenbaum and David J. Wetherall. *Computer Networks*. Prentice Hall PTR, 5^{th} edition, 2010.

[29] Palo Alto Networks. What is a Service Level Agreement? Retrieved 18 March 2008, `https://www.paloaltonetworks.com/resources/learning-center/what-is-a-service-level-agreement-sla.html`.

[30] Michele Mazzucco, Isi Mitrani, Jennie Palmer, Mike Fisher, and Paul McKee. Web Service Hosting and Revenue Maximization. In *Proceedings of the* 5^{th} *IEEE European Conference on Web Services*, pages 45–54, Halle, Germany, 2007. IEEE Computer Society Press.

[31] Jorge Cardoso, John Miller, Amit Sheth, and Jonathan Arnold. Modeling Quality of Service for Workflows and Web Service Processes. Technical Report TR-02-002, LSDIS Lab, Computer Science Department, University of Georgia, December 2002.

[32] Anbazhagan Mani and Arun Nagarajan. Understanding Quality of Service for Web Services: Improving the Performance of your Web Services, January 2002. `http://www.ibm.com/developerworks/library/ws-quality/`.

[33] William Craig Carter. A Time for Reflection. In *Proceedings of the* 12^{th} *International Symposium on Fault-Tolerant Computing*, page 41, Santa Monica, CA, USA, 1982. IEEE Computer Society Press.

[34] James McGovern, Sameer Tyagi, Michael Stevens, and Sunil Matthew. *Java Web Services Architecture*. Morgan Kaufmann Publishers, 2003.

[35] John C. Knight, Elisabeth A. Strunk, and Kevin J. Sullivan. Towards a Rigorous Definition of Information System Survivability. In *Proceedings of the DARPA Information Survivability Conference and Exposition*, pages 78–89, Washington, DC, USA, 2003. IEEE Computer Society Press.

[36] Vincenzo De Florio. On the Constituent Attributes of Software and Organizational Resilience. *Interdisciplinary Science Reviews*, 38(2):122–148, June 2013.

[37] Nick Rozanski and Eóin Woods. *Software Systems Architecture: Working With Stakeholders Using Viewpoints*. Addison-Wesley Publishing Company, 1^{st} edition, 2005.

[38] IBM Software Group. An Architectural Blueprint for Autonomic Computing. White paper, June 2005.

[39] Bertrand Meyer. Dependable Software. In *Dependable Systems: Software, Computing, Networks*, volume 4028 of *Lecture Notes in Computer Science*, pages 1–33. Springer-Verlag Berlin Heidelberg, 2006.

[40] Behrooz Parhami. *Dependable Computing: A Multilevel Approach*. November 2015. http://www.ece.ucsb.edu/~parhami/text_dep_comp.htm.

[41] Kumar Saha Goutam. Fault tolerant computing issues. *International Journal of Applied Research on Information Technology and Computing*, 6:197, 1 2015.

[42] Georgia Psychou, Dimitrios Rodopoulos, Mohamed Sabry, Tobias Gemmeke, David Atienza, Tobias Noll, and Francky Catthoor. Classification of Resilience Techniques Against Functional Errors at Higher Abstraction Layers of Digital Systems. *ACM Computing Surveys*, 50:1–38, 10 2017.

[43] *Securing The API Stronghold: The Ultimate Guide to API Security*. Nordic APIs, 2015.

[44] Dave Patterson and Armando Fox. Recovery-oriented Computing: Overview, 2004. http://roc.cs.berkeley.edu/roc_overview.html.

[45] Daniel A. Menascé and Virgilio A. F. Almeida. *Capacity Planning for Web Services: Metrics, Models, and Methods*. Prentice Hall PTR, 2002.

[46] J.O. Kephart and D.M. Chess. The Vision of Autonomic Computing. *Computer*, 36(1):41–50, January 2003.

[47] Roy Sterritt. Autonomic computing. *Innovations in Systems and Software Engineering*, 1(1):79–88, April 2005.

[48] S. Schmid, M. Sifalakis, and D. Hutchison. Towards Autonomic Networks. In *Autonomic Networking*, volume 4195 of *Lecture Notes in Computer Science*, pages 1–11. Springer-Verlag Berlin Heidelberg, 2006.

[49] IBM Corporation. Autonomic Problem Determination: A First Step toward Self-healing Computing Systems. White paper, IBM, October 2003.

[50] A. G. Ganek and T. A. Corbi. The Dawning of the Autonomic Computing Era. *IBM Systems Journal*, 42(1):5–18, February 2003.

[51] Manish Parashar and Salim Hariri. *Autonomic Computing: Concepts, Infrastructure, and Applications*. Taylor & Francis, Inc., 2007.

[52] Elisabeth Hildt. Artificial Intelligence: Does Consciousness Matter? *Frontiers in Psychology*, 10, July 2019.

[53] V. De Florio and C. Blondia. Reflective and Refractive Variables: A Model for Effective and Maintainable Adaptive-and-Dependable Software. In *Proceedings of the* 33^{rd} *Euromicro Conference on Software Engineering and Advanced Applications*, Luebeck, Germany, 2007. IEEE Computer Society Press.

[54] Thomas Erl, Anish Karmarkar, Priscilla Walmsley, Hugo Haas, Umit Yalcinalp, Canyang Kevin Liu, David Orchard, Andre Tost, and James Pasley. *Web Service Contract Design and Versioning for SOA*. The Prentice Hall Service-Oriented Computing Series from Thomas Erl. Prentice Hall PTR, 2009.

[55] Vincenzo De Florio. On the Behavioral Interpretation of System-Environment Fit and Auto-Resilience. In *Proceedings of the IEEE 2014 Conference on Norbert Wiener in the* 21^{st} *Century*, Boston MA, USA, 2014. IEEE Computer Society Press.

[56] Halil Kükner, Pieter Weckx, Sébastien Morrison, Jacopo Franco, Maria Toledano-Luque, Moonju Cho, Praveen Raghavan, Ben Kaczer, Doyoung Jang, Kenichi Miyaguchi, Marie Bardon, Francky Catthoor, Liesbet van der perre, Rudy Lauwereins, and Guido Groeseneken. Comparison of NBTI Aging on Adder Architectures and Ring Oscillators in the Downscaling Technology Nodes. *Microprocessors and Microsystems*, 39:1039–1051, June 2015.

[57] Shantanu Dutt, Federico Rota, Franco Trovo, and Fran Hanchek. *Fault Tolerance in Computer Systems—From Circuits to Algorithms*, chapter 8, pages 427–457. Academic Press, December 2005.

[58] Jeannette M. Wing. A Specifier's Introduction to Formal Methods. *IEEE Computer*, 23(9):8–24, September 1990.

[59] Information Technology — Z Formal Specification Notation — Syntax, Typesystem and Semantics, July 2002.

[60] Rami Melhem. Advanced Topics in Distributed and Real-time CS3530: Introduction to Fault Tolerant Systems, January 2002. `https://people.cs.pitt.edu/~melhem/courses/3530/L4.pdf`.

[61] Ajit Kumar Verma, Srividya Ajit, and Manoj Kumar. Dependability of Networked Computer-based Systems. Springer Series in Reliability Engineering, chapter 1, pages 1–13. Springer-Verlag Berlin Heidelberg, 2011.

[62] Jean-Claude Laprie. *Dependability: Basic Concepts and Terminology*, volume 0932 of *Dependable Computing and Fault-Tolerant Systems*. Springer-Verlag Berlin Heidelberg, 1992.

[63] Aaron B. Brown and David A. Patterson. Rewind, Repair, Replay: Three R's to Dependability. In *Proceedings of the* 10^{th} *workshop on ACM SIGOPS European workshop*, pages 70–77, Saint-Emilion, France, 2002. Association for Computing Machinery, Inc.

[64] D. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, and N. Treuhaft. Recovery-oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies. Technical Report UCB//CSD-02-1175, University of California, Berkeley, March 2002.

[65] Kirstie Hawkey. A Comparison of Software Production vs that of Other Engineered Products (CSCI 3130), Summer 2011. `https://web.cs.dal.ca/~hawkey/3130/SEBackground2.pdf`.

[66] Barry W. Johnson. *The Electrical Engineering Handbook*, chapter 93, pages 2346–2357. CRC Press LLC, 2000.

[67] Michael R. Lyu, editor. *Software Fault Tolerance*. Trends in Software. John Wiley & Sons, Inc., 1995.

[68] Jonas Buys, Vincenzo De Florio, and Chris Blondia. Optimization of WS-BPEL Workflows through Business Process Re-engineering Patterns. *International Journal of Adaptive, Resilient and Autonomic Systems*, 1(3):25–41, July–September 2010.

[69] Patrick Rogers. *Software Fault Tolerance, Reflection and the Ada Programming Language*. PhD thesis, University of York: Department of Computer Science, 2003.

[70] David Kalinsky. Design Patterns for High Availability. *Embedded Systems Programming*, 15(8), August 2002.

[71] Laura L. Pullum. *Software Fault Tolerance: Techniques and Implementation*. Artech House, 1 2001.

[72] Zibin Zheng and Michael R. Lyu. An Adaptive QoS-Aware Fault Tolerance Strategy for Web Services. *Empirical Software Engineering*, 15(4):323–345, 2010.

[73] Bruno Richard Preiss. *Data Structures and Algorithms with Object-Oriented Design Patterns in Java*. John Wiley & Sons, Inc., 1999.

[74] Jerry Banks, John S. Carson II, Barry L. Nelson, and David M. Nicol. *Discrete-Event System Simulation*. Prentice-Hall International Series in Industrial and Systems Enineering. Prentice Hall PTR, 4^{th} edition, 2004.

[75] Shirish S. Sane, Neeta Deshpande, and Anuradha A. Puntambekar. *Data Structures*. Technical Publications, 2^{nd} edition, 2006.

[76] George Fishman. *Discrete-Event Simulation: Modeling, Programming, and Analysis*. Springer Series in Operations Research and Financial Engineering. Springer-Verlag Berlin Heidelberg, 2001.

[77] Andrea Bondavalli, Silvano Chiaradonna, Felicita Di Giandomenico, and Fabrizio Grandoni. Threshold-based Mechanisms to Discriminate Transient from Intermittent Faults. *IEEE Transactions on Computers*, 49(3):230–245, March 2000.

[78] Vincenzo De Florio. Software Assumptions Failure Tolerance: Role, Strategies, and Visions. In *Architecting Dependable Systems*, volume 7, pages 249–272. Springer-Verlag Berlin Heidelberg, 2010.

[79] Apache Software Foundation. Apache MUSE. Retrieved 10 June 2014, `http://52north.org/communities/sensorweb/amused/`.

[80] P. Martin, W. Powley, K. Wilson, W. Tian, T. Xu, and J. Zebedee. The WSDM of Autonomic Computing: Experiences in Implementing Autonomic Web Services. In *Proceedings of the 2007 International Workshop on Software Engineering for Adaptive and Self-Managing Systems*, Minneapolis, MN, USA, 2007. IEEE Computer Society Press.

[81] Pierre L'Ecuyer. SSJ: Stochastic Simulation in Java™. Retrieved 2 December 2017, `http://simul.iro.umontreal.ca/ssj/indexe.html`.

[82] Swapna S. Gokhale and Robert E. Mullen. Application of the Lognormal Distribution to Software Reliability Engineering. In *Handbook of Performability Engineering*, number XLVIII, chapter 73, pages 1209–1225. Springer-Verlag Berlin Heidelberg, 2008.

[83] Jonas Buys, Vincenzo De Florio, and Chris Blondia. Towards Context-Aware Adaptive Fault Tolerance in SOA Applications. In *Proceedings of the* 5^{th} *ACM International Conference on Distributed Event-Based Systems*, pages 63–74, New York, NY, USA, 2011. Association for Computing Machinery, Inc.

[84] Jonas Buys, Vincenzo De Florio, and Chris Blondia. Towards Parsimonious Resource Allocation in Context-Aware *n*-Version Programming. In *Proceedings of the* 7^{th} *IET System Safety Conference*, IET Conference Publications, pages 137–144, Edinburgh, Scotland, United Kingdom, 2012. The Institute of Engineering and Technology.

[85] Jonas Buys, Vincenzo De Florio, and Chris Blondia. Applying Business Process Re-engineering Patterns to Optimize WS-BPEL Workflows. In Ozgur Akan, Paolo Bellavista, Jiannong Cao, Falko Dressler, Domenico Ferrari, Mario Gerla, Hisashi Kobayashi, Sergio Palazzo, Sartaj Sahni, Xuemin (Sherman) Shen, Mircea Stan, Jia Xiaohua, Albert Zomaya, Geoffrey Coulson, Mihaela Ulieru, Peter Palensky, and Rene Doursat, editors, *IT Revolutions*, volume 11 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 151–160. Springer Berlin Heidelberg, Venice, Italy, 2008.

[86] Jonas Buys, Vincenzo De Florio, and Chris Blondia. Optimization of WS-BPEL Workflows through Business Process Re-engineering Patterns. In Vincenzo De Florio, editor, *Technological Innovations in Adaptive and Dependable Systems: Advancing Models and Concepts*, chapter 20, pages 345–361. IGI Global, 2012.

[87] Amanda S. Nascimento, Cecília M.F. Rubira, Rachel Burrows, and Fernando Castor. A Systematic Review of Design Diversity-based Solutions a for Fault-tolerant SOAs. In *Proceedings of the* 17^{th} *International Conference on Evaluation and Assessment in Software Engineering*, pages 107–118, Porto de Galinhas, Brazil. Association for Computing Machinery, Inc.

[88] Luiz Alexandre Hiane da Silva Maciel and Celso Massaki Hirata. Fault-tolerant Timestamp-based Two-phase Commit Protocol for RESTful Services. *Journal of Software: Practice and Experience*, 43(12):1459–1488, September 2013.

[89] Roeland Dillen, Jonas Buys, Vincenzo De Florio, and Chris Blondia. WSDM-enabled Autonomic Augmentation of Classical Multi-Version Software Fault Tolerance Mechanisms. In F. Ortmeier and P. Daniel, editors, *Proceedings of the SAFECOMP 2012 Workshops*, volume 7613 of *Lecture Notes in Computer Science*, pages 294–306, Berlin, Germany, 2012. Springer-Verlag. Presented at the 1^{st} Workshop on DEpendable and SEcure Computing for Large-scale Complex Critical Infrastructures (DESEC4LCCI2012).

[90] Vincenzo De Florio. On resilient behaviors in computational systems and environments. *Journal of Reliable Intelligent Environments*, 1(1):33–46, 2015.

[91] Amanda S. Nascimento, Cecília M.F. Rubira, Rachel Burrows, Fernando Castor, and Patrick H.S. Brito. Designing Fault-tolerant SOA-based on Design Diversity. *Journal of Software Engineering Research and Development*, 2(13), December 2014.

[92] Andrea Höller, Tobias Rauter, Johannes Iber, and Christian Kreiner. Towards Dynamic Software Diversity for Resilient Redundant Embedded Systems. In *Proceedings of the* 7^{th} *International Workshop on Software Engineering for Resilient Systems*, volume 9274, pages 16–30, Germany, 2015. Springer-Verlag Berlin Heidelberg.

[93] M. Rizwan, A. Nadeem, and M. B. Khan. An Evaluation of Software Fault Tolerance Techniques for Optimality. In *Proceedings of the* 11^{th} *International Conference on Emerging Technologies*, pages 1–6, Peshawar, Pakistan, 2015.

[94] World Wide Web Consortium (W3C). Web Service Management: Service Life Cycle (WSLC) specification, February 2004. `http://www.w3.org/TR/wslc`.

[95] Object Management Group (OMG). Business Process Model and Notation (BPMN) specification, January 2011. `http://www.omg.org/spec/BPMN/2.0/PDF/`.

[96] Martin Aigner. *Combinatorial Theory*. Classics in Mathematics. Springer-Verlag Berlin Heidelberg, 1997.

[97] Algirdas A. Avižienis, Jean-Claude Laprie, and Brian Randell. Dependability and its Threats — a Taxonomy. In *IFIP Congress Topical Sessions*, pages 91–120, Toulouse, France, 2004. Kluwer Academic Publishers.

[98] Cristian Fetzer. Perfect Failure Detection in Timed Asynchronous Systems. *IEEE Transactions on Computers*, 52(2):99–112, February 2003.

[99] Felix C. Gärtner. A Gentle Introduction to Failure Detectors and Related Problems. Technical Report TUD-BS-2001-01, Darmstadt University of Technology, Department of Computer Science, April 2001.

[100] Jim Gray. Why do Computers Stop and What can be done about it? Technical Report TR-85.7, Tandem, June 1985.

[101] Java™ Community Process. Java™ Servlet 2.4 specification (JSR 154), November 2003. `http://jcp.org/en/jsr/detail?id=154`.

[102] Flaviu Cristian. Understanding Fault-Tolerant Distributed Systems. *Communications of the ACM*, 34(2):56–78, February 1991.

[103] Nancy R. Tague. *The Quality Toolbox*. ASQ Quality Press, Milwaukee, WI, USA, 2^{nd} edition, 2004.

[104] Sheldon Ross. *A First Course in Probability*. Prentice Hall PTR, 5^{th} edition, 1997.

[105] Flaviu Cristian and Cristian Fetzer. The Timed Asynchronous Distributed System Model. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):642–657, June 1999.

[106] Vincenzo De Florio. Robust-and-Evolvable Resilient Software Systems — Open Problems and Lessons Learned. In *Proceedings of the ESEC/FSE 2011 Workshop on Assurances for Self-Adaptive Systems*, pages 10–17, Szeged, Hungary, 2011. Association for Computing Machinery, Inc.

[107] Paul G. Hoel. *Introduction to Mathematical Statistics*. Wiley Series in Probability and Mathematical Statistics. John Wiley & Sons, Inc., 6^{th} edition, 1984.

[108] Joshua Bloch. *Effective Java™*. The Java Series. Addison-Wesley Publishing Company, 2^{nd} edition, 2008.

[109] Organization for the Advancement of Structured Information Standards (OASIS). Web Services Distributed Management (WSDM) specifications, August 2006. `http://www.oasis-open.org/committees/wsdm`.

[110] Vincenzo De Florio, Geert Deconinck, and Rudy Lauwereins. Software Tool Combining Fault Masking with User-Defined Recovery Strategies. *IEE Proceedings — Software*, 145(6):203–211, December 1998.

[111] Gruia-Catalin Roman, Christine Julien, and Jamie Payton. A Formal Treatment of Context-Awareness. In *Fundamental Approaches to Software Engineering*, volume 2984 of *Lecture Notes in Computer Science*, pages 12–36. Springer-Verlag Berlin Heidelberg, 2004.

[112] Michael R. Lyu, editor. *Handbook of Software Reliability Engineering*. McGraw-Hill, Inc., 1996.

[113] Joachim Götze and Jochen Müller. Iterative Service Orchestration Based on Dependability Attributes. In *Proceedings of the* 34^{th} *Euromicro Conference on Software Engineering and Advanced Applications*, pages 353–360, Parma, Italy, September 2008. IEEE Computer Society Press.

[114] Marco Serafini, Andrea Bondavalli, and Neeraj Suri. Online Diagnosis and Recovery: On the Choice and Impact of Tuning Parameters. *IEEE Transactions on Dependable and Secure Computing*, 4(4):295–312, October–December 2007.

[115] Jewgeni H. Dshalalow, editor. *Advances in Queueing: Theory, Methods and Open Problems*. CRC Press, Inc., Boca Raton, FL, USA, 1995.

[116] Swapna S. Gokhale and Robert E. Mullen. From Test Count to Code Coverage Using the Lognormal Failure Rate. In *Proceedings of the* 15^{th} *International Symposium on Software Reliability Engineering*, pages 295–305, Saint-Malo, Bretagne, France, 2004. IEEE Computer Society Press.

[117] Thomas Erl. *Service-Oriented Architecture: A Field Guide to Integrating XML and Web Services*. The Prentice Hall Service-Oriented Computing Series from Thomas Erl. Prentice Hall PTR, 2004.

[118] Samir Youcef, Muhammad Usman Bhatti, Lynda Mokdad, and Valerie Monfort. Simulation-based Response-time Analysis of Composite Web Services. In *Proceedings of the* 10^{th} *IEEE International Multitopic Conference*, Islamabad, Pakistan, 2007. IEEE Computer Society Press.

[119] J.-P. Ebert and A. Willig. A Gilbert-Elliot Bit Error Model and the Efficient Use in Packet Level Simulation. Technical Report TKN-99-002, Telecommunication Networks Group, Technische Universität Berlin, March 1999.

[120] John Aitchison and J. Alan C. Brown. *The Lognormal Distribution*, volume 5 of *Monographs*. Jarrold & Sons Ltd, Norwich, United Kingdom, 1969.

[121] R. Mullen. The Lognormal Distribution of Software Failure Rates: Origin and Evidence. In *Proceedings of the* 9^{th} *International Symposium on Software Reliability Engineering*, pages 124–134, Paderborn, Germany, 1998. IEEE Computer Society Press.

[122] R. Mullen. The Lognormal Distribution of Software Failure Rates: Application to Software Reliability Growth Modeling. In *Proceedings of the* 9^{th} *International Symposium on Software Reliability Engineering*, pages 134–142, Paderborn, Germany, 1998. IEEE Computer Society Press.

[123] Peter G. Bishop and Robin E. Bloomfield. Using a Log-normal Failure Rate Distribution for Worst Case Bound Reliability Prediction. In *Proceedings of the* 14^{th} *International Symposium on Software Reliability Engineering*, pages 237–245, Denver, CO, USA, 2003. IEEE Computer Society Press.

[124] Robert E. Mullen and Swapna S. Gokhale. Software Defect Rediscoveries: A Discrete Lognormal Model. In *Proceedings of the* 16^{th} *IEEE International Symposium on Software Reliability Engineering*, pages 203–212, Chicago, IL, USA, 2005. IEEE Computer Society Press.

[125] Robert E. Mullen and Swapna S. Gokhale. A Discrete Lognormal Model of Defect Occurrence Counts with Applicability to Network Security Defects. In Dale S. Caffall, James B. Michael, and Jeffrey M. Voas, editors, *Proceedings of the* 5^{th} *Workshop on Software Assessment*, pages 1–33, Monterey, CA, USA, 2005. IEEE Computer Society Press.

[126] Edward N. Adams. Optimizing Preventive Service of Software Products. *IBM Journal of Research and Development*, 28(1):2–14, January 1984.

[127] Daniel P. Siewiorek and Robert S. Swarz. Faults and their Manifestations. In *Reliable Computer Systems: Design and Evaluation*, chapter 2, pages 22–78. A K Peters/CRC Press, 3^{rd} edition, October 1998.

[128] D. N. Prabhakar Murthy, Min Xie, and Renyan Jiang. *Type V Weibull Models*, chapter 13, pages 238–246. Wiley Series in Probability and Statistics. John Wiley & Sons, Inc., 2003.

[129] Hongyu Zhang. On the Distribution of Software Faults. *IEEE Transactions on Software Engineering*, 34(2):301–302, March–April 2008.

[130] Mei-Chen Hsueh, Timothy K. Tsai, and Ravishankar K. Iyer. Fault Injection Techniques and Tools. *IEEE Computer*, 30(4):75–82, April 1997.

[131] Neeraj Saxena and Anita Goel. Using Web Services for Distributed Computing. In *Proceedings of the 2nd National INDIAcom Conference*, New Delhi, India, 2008.

[132] Roger Wolter. XML Web Services Basics, June 2001. Retrieved 27 April 2008, `https://msdn.microsoft.com/en-us/library/ms996507.aspx`.

[133] World Wide Web Consortium (W3C). Web Services Addressing (WS-Addressing) specification, August 2004. `http://www.w3.org/Submission/ws-addressing/`.

[134] World Wide Web Consortium (W3C). Web Services Metadata Exchange (WS-MetadataExchange) specification, December 2011. `http://www.w3.org/TR/ws-metadata-exchange/`.

[135] Web Services Standards Overview poster, 2007. Retrieved 27 December 2007, `https://www.innoq.com/soa/ws-standards/poster/innoQ%20WS-Standards%20Poster%202007-02.pdf`.

[136] Organization for the Advancement of Structured Information Standards (OASIS). Web Services Resource Framework: Resource Properties (WSRF-RP) specification, April 2006. `http://docs.oasis-open.org/wsrf/wsrf-ws_resource_properties-1.2-spec-os.pdf`.

[137] Organization for the Advancement of Structured Information Standards (OASIS). Web Services Resource Metadata (ws-resourceMetadataDescriptor) specification, November 2006. `http://docs.oasis-open.org/wsrf/wsrf-ws_resource_metadata_descriptor-1.0-spec-cs-01.pdf`.

[138] Nicolai Josuttis. *SOA in Practice: The Art of Distributed System Design*. O'Reilly Media, Inc., 2007.

[139] Organization for the Advancement of Structured Information Standards (OASIS). Web Services Notification (WSN) specifications, October 2006. `http://www.oasis-open.org/committees/wsn`.

[140] Thomas Erl. *SOA Principles of Service Design*. The Prentice Hall Service-Oriented Computing Series from Thomas Erl. Prentice Hall PTR, 2007.

[141] Thomas Erl. *SOA Design Patterns*. The Prentice Hall Service-Oriented Computing Series from Thomas Erl. Prentice Hall PTR, 2008.

[142] Vincenzo De Florio, Hong Sun, Jonas Buys, and Chris Blondia. On the Impact of Fractal Organization on the Performance of Socio-technical Systems. In *Proceedings of the 2013 International Workshop on Intelligent Techniques for Ubiquitous Systems*, pages 672–677, Vietri Sul Mare, Italy, 2013. IEEE Computer Society Press.

[143] BEA, Intalio, Individual, Adobe Systems, Systinet, IBM, Active Endpoints, Inc., JBoss, Inc., SAP, Microsoft, and Oracle. Web Services Business Process Execution Language (WS-BPEL), version 2.0, January 2007. `http://docs.oasis-open.org/wsbpel/2.0/CS01/wsbpel-v2.0-CS01.html`.

[144] Nick Russell and Wil M.P. van der Aalst. Work Distribution and Resource Management in BPEL4People: Capabilities and Opportunities. In Z. Bellahsène and M. Léonard, editors, *Proceedings of the* 20^{th} *International Conference on Advanced Information Systems Engineering*, volume 5074 of *Lecture Notes in Computer Science*, pages 94–108, Montpellier, France, 2008. Springer-Verlag Berlin Heidelberg.

[145] Adobe, BEA, Oracle, IBM, Active Endpoints, and SAP. WS-BPEL Extension for People (BPEL4People), version 1.0, June 2007. `http://docs.oasis-open.org/bpel4people/bpel4people-1.1.html`.

[146] Adobe, BEA, Oracle, IBM, Active Endpoints, and SAP. Web Services Human Task (WS-HumanTask), version 1.0, June 2007. `https://docs.oasis-open.org/bpel4people/ws-humantask-1.1-namespace-cs-01.html`.

[147] Matjaz B. Juric. *Business Process Execution Language for Web Services BPEL and BPEL4WS*. Packt Publishing, 2^{nd} edition, 2006.

[148] L. K. Singh and Riktesh Srivastava. Memory Estimation of Internet Server Using Queuing Theory: Comparative Study between M/G/1, G/M/1 and G/G/1 Queuing Model. *World Academy of Science Engineering and Technology*, 1(9), July 2007.

[149] Roberta S. Russell and Bernard W. Taylor. *Operations Management*. Prentice Hall PTR, 1999.

[150] Vincenzo De Florio. Antifragility = Elasticity + Resilience + Machine Learning: Models and Algorithms for Open System Fidelity. *Procedia Computer Science*, 32:834–841, January 2014.

[151] Denis V. Gruzenkin, Alexey S. Chernigovskiy, and Roman Yu Tsarev. *n*-Version Software Module Requirements to Grant the Software Execution Fault Tolerance. In *Cybernetics Approaches in Intelligent Systems (Proceedings of the 2017 Computational Methods in Systems and Software Conference)*, volume 661 of *Advances in Intelligent Systems and Computing*, pages 293–303, Szczecin, Poland, 2017. Springer-Verlag Berlin Heidelberg.

[152] Vincenzo De Florio. On Ambients as Systemic Exoskeletons: Crosscutting Optimizers and Antifragility Enablers. *Journal of Reliable Intelligent Environments*, 1:61–73, August 2015.

[153] Denis V. Gruzenkin, Anton S. Mikhalev, Galina V. Grishina, Roman Yu. Tsarev, and Vladislav N. Rutskiy. Using Blockchain Technology to Improve *n*-Version Software Dependability. In *Computational and Statistical Methods in Intelligent Systems (Proceedings of the* 2^{nd} *Conference on Computational Methods in Systems and Software)*, volume 859 of *Advances in Intelligent Systems and Computing*, pages 132–137, Szczecin, Poland, 2018. Springer-Verlag Berlin Heidelberg.

[154] Alberto Goffi, Alessandra Gorla, Andrea Mattavelli, and Mauro Pezzè. *Intrinsic Redundancy for Reliability and Beyond*. Springer-Verlag Berlin Heidelberg, 2017.

[155] Andrea Mattavelli. *Software Redundancy: What, Where, How*. PhD thesis, Università della Svizzera italiana, 2016.

[156] Denis Gruzenkin, I.A. Yakimov, Aleksandr Kuznetsov, Roman Tsarev, Grishina Viktorovna, Alexandr Pupkov, and N Bystrova. Algorithm Diversity Metric for n-Version Software. *Journal of Physics: Conference Series*, 1333, October 2019.

[157] Fumio Machida. On the Diversity of Machine Learning Models for System Reliability. In *Proceedings of the* 24^{th} *Pacific Rim International Symposium on Dependable Computing*, pages 276–286, Kyoto, Japan, December 2019. IEEE Computer Society Press.

[158] Mark van den Brand and Jan Friso Groote. Software Engineering: Redundancy is Key. *Elsevier Science of Computer Programming*, 97(1):75–81, January 2015.

[159] Antonio Carzaniga, Andrea Mattavelli, and Mauro Pezzè. Measuring software redundancy. In *Proceedings of the* 37^{th} *International Conference on Software Engineering*, Firenze, Italy, May 2015. IEEE Computer Society Press.

[160] Denis V. Gruzenkin, Galina V. Grishina, and Mustafa S. D. Üstoğlu. Compensation Model of Multi-attribute Decision Making and its Application to n-Version Software Choice. In *Software Engineering Trends and Techniques in Intelligent Systems (Proceedings of the* 6^{th} *Computer Science On-line Conference)*, volume 575 of *Advances in Intelligent Systems and Computing*, pages 148–157, Prague, Czech Republic, 2017. Springer-Verlag Berlin Heidelberg.

[161] Ashraf Armoush, Falk Salewski, and Stefan Kowalewski. Design Pattern Representation for Safety-critical Embedded Systems. *Journal of Software Engineering and Applications*, 2(1), April 2009.

[162] Sheheryar Malik and Fabrice Huet. Adaptive Fault Tolerance in Real Time Cloud Computing. In *Proceedings of the IEEE World Congress on Services*, pages 280–287.

[163] Vincenzo De Florio and Giuseppe Primiero. A Framework for Trustworthiness Assessment based on Fidelity in Cyber and Physical Domains. *Procedia Computer Science*, 52:996–1003, June 2015.

[164] Ailec Wu, Abu Rubaiyat, Chris Anton, and Homa Alemzadeh. Model Fusion: Weighted n-Version Programming for Resilient Autonomous Vehicle Steering Control. In *Proceedings of the* 29^{th} *IEEE International Symposium on Software Reliability Engineering Workshops*, pages 144–145, Memphis, TN, USA, October 2018. IEEE Computer Society Press.

[165] Valeria Cardellini, Emiliano Casalicchio, Kalinka Castelo Branco, Júlio Estrella, Francisco Monaco, Yuhui Chen, Anatoliy Gorbenko, Vyacheslav Kharchenko, and Alexander Romanovsky. *Measuring and Dealing with the Uncertainty of SOA Solutions*, pages 265–294. IGI Global, January 2012.

[166] Katerina Goseva-Popstojanova and Aksenti Grnarov. Performability Modeling of n-Version Programming Technique. In *Proceedings of* 6^{th} *International Symposium on Software Reliability Engineering*, pages 209–218, Toulouse, France, January 1995. IEEE Computer Society Press.

[167] Pece Mitrevski, Katerina Goseva-Popstojanova, and Aksenti Grnarov. Per-Run Reliability Assessment of the n-Version Programming Technique Using Generalized Stochastic Petri Nets. In *Proceedings of the* 4^{th} *World Multiconference on Systemics, Cybernetics and Informatics*, pages 457–462, Orlando, FL, USA, 7 2000.

[168] Marco Marsan, Gianfranco Balbo, and Gianni Conte. A Class of Generalised Stochastic Petrinets for the Performance Evaluation of Multiprocessor Systems. In *Proceedings of the International Conference on Measurements and Modeling of Computer Systems*, pages 198–199, Minneapolis, MN, USA, 01 1983.

[169] Tadao Murata. Petrinets: Properties, Analysis and Applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

[170] William H. Sanders and John F. Meyer. Stochastic Activity Networks: Formal Definitions and Concepts. In *Lectures on Formal Methods and Performance Analysis*, volume 2090 of *Lecture Notes in Computer Science*, pages 106–115, Innsbruck, Austria, May 2008. Springer-Verlag Berlin Heidelberg.

[171] B. Kaczer, Jacopo Franco, Pieter Weckx, Philippe Roussel, Marko Simicic, Venkata Putcha, Erik Bury, Moonju Cho, R. Degraeve, Dimitri Linten, Guido Groeseneken, Peter Debacker, Bertrand Parvais, P. Raghavan, F. Catthoor, G. Rzepa, Michael Waltl, Wolfgang Goes, and T. Grasser. The Defect-centric Perspective of Device and Circuit Reliability — From Gate Oxide Defects to Circuits. *Solid-State Electronics*, 125:52–62, July 2016.

[172] Tushar Deepak Chandra and Sam Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[173] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley Publishing Company, Boston, MA, USA, 2^{nd} edition, 1994.

[174] Pierre Reldin and Peter Sundling. Explaining SOA Service Granularity — How IT Strategy Shapes Services. Master's thesis, Linköpings Universitet, Dept. of Management and Engineering Industrial Economics, Sweden, March 2007.

[175] Michael Stal. Using Architectural Patterns and Blueprints for Service-Oriented Architecture. *IEEE Software*, 23(2):54–61, March–April 2006.

[176] Stefano Modafferi, Enrico Mussi, and Barbara Pernici. SH-BPEL: a Self-healing Plug-in for WS-BPEL Engines. In *Proceedings of the* 1^{st} *workshop on Middleware for Service Oriented Computing*, pages 48–53, Melbourne, Australia, 2006. Association for Computing Machinery, Inc.

[177] H. A. Reijers and S. Liman Mansar. Best Practices in Business Process Redesign: An Overview and Qualitative Evaluation of Successful Redesign Heuristics. *Omega*, 33(4):283–306, August 2004.

[178] Frank Leymann Oliver Kopp, Rania Khalaf. Reaching Definitions Analysis Respecting Dead Path Elimination Semantics in BPEL Processes. Technical Report 2007/04, Stuttgart University, Germany, April 2007.

[179] John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., 1990.

[180] Anis Charfi, Benjamin Schmeling, and Mira Mezini. Transactional BPEL Processes with AO4BPEL Aspects. In *Proceedings of the* 5^{th} *IEEE European Conference on Web Services*, pages 149–158, Halle, Germany, 2007. IEEE Computer Society Press.

[181] A. Stephen. Using BPMN to Model a BPEL Process, April 2005. Retrieved 1 May 2018, `https://www.bptrends.com/publicationfiles/03-05%20WP%20Mapping%20BPMN%20to%20BPEL-%20White.pdf`.

[182] Glen Dobson. Using WS-BPEL to Implement Software Fault Tolerance for Web Services. In *Proceedings of the* 32^{nd} *EUROMICRO Conference on Software Engineering*, pages 126–133, Cavtat, Dubrovnik, Croatia, 2006. IEEE Computer Society Press.

[183] Jan Mendling, Karsten Ploesser, and Mark Strembeck. Specifying Separation of Duty Constraints in BPEL4People Processes. In *Proceedings of the* 11^{th} *International Conference on Business Information Systems*, volume 7 of *Lecture Notes in Business Information Processing*, pages 273–284, Innsbruck, Austria, May 2008. Springer-Verlag Berlin Heidelberg.

[184] H. Sun, V. De Florio, N. Gui, and C. Blondia. Towards Longer, Better, and more Active Lives - Building Mutual Assisted Living Community for Elder People. In *Proceedings of the 47th European FITCE Congress*, London, England, United Kingdom, September 2008. FITCE.

[185] Grigoris Antoniou and Frank van Harmelen. *A Semantic Web Primer*. MIT Press, Cambridge, MA, USA, 2004.

[186] David Sprott and Lawrence Wilkes. Understanding Service Oriented Architecture. *The Architecture Journal*, (1):10–17, January 2004.

[187] Organization for the Advancement of Structured Information Standards (OASIS). Universal Description, Discovery and Integration (UDDI) specification, October 2004. `http://www.uddi.org/pubs/uddi_v3.htm`.

[188] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. The Addison-Wesley Object Technology Series. Addison-Wesley Publishing Company, Reading, MA, USA, 1^{st} edition, 1998.

[189] Internet Engineering Task Force (IETF). RFC 3986: Uniform Resource Identifier (URI): Generic Syntax, January 2005. `http://www.ietf.org/rfc/rfc3986.txt`.

[190] World Wide Web Consortium (W3C). Web Services Policy specification, September 2007. `http://www.w3.org/2002/ws/policy/`.

[191] World Wide Web Consortium (W3C). XML Path Language (XPath) specification, November 1999. http://www.w3.org/TR/xpath/.