A tool for the expression of failure detection protocols

A Tool for the Expression of Failure Detection Protocols

Vincenzo De Florio and Chris Blondia
University of Antwerp
Department of Mathematics and Computer Science
Performance Analysis of Telecommunication Systems group
Middelheimlaan 1, 2020 Antwerp, Belgium, *and*
Interdisciplinary institute for BroadBand Technology
Gaston Crommenlaan 8, 9050 Ghent-Ledeberg, Belgium

Abstract

Failure detection protocols—an important building block of fault-tolerant distributed systems—are often described by their authors making use of informal pseudo-codes of their own devising. Often these pseudo-codes use syntactical constructs that are not available in COTS programming languages such as C or C++. This translates into informal descriptions that require ad hoc interpretations and implementations. Being informal, these descriptions cannot be tested by their authors, which may translate into insufficiently detailed or even faulty specifications. Being non-standard, they require the reader to understand each time a different lingo. This paper tackles the above problem introducing a formal syntax for the expression of failure detection protocols and a C library that implements that syntax: a toolset to express and reason about failure detection protocols. The resulting specifications are more diffuse but non ambiguous and eligible for becoming a standard form among researchers and practitioners.

1 Introduction

Failure detection constitutes a fundamental building block of fault-tolerant distributed systems. Many researchers have devoted their efforts to this subject during the last decade. Failure detection protocols are often described by their authors making use of informal pseudo-codes of their conception. Often these pseudo-codes use syntactical constructs such as repeat periodically [4, 1, 3], at time t send heartbeat [5, 3], at time t check whether message has arrived [5], or upon receive [1], together with several variants (see Table 1). We observe that such syntactical constructs are not often found in COTS programming languages such as C or C++. Furthermore, the lack of a formal, well-defined, and standard form to express failure detection protocols often leads their authors to insufficiently detailed descriptions. Those informal descriptions in turn complicate the reading process and exacerbate the work of the implementors, which becomes time-consuming and error-prone.

Several researchers and practitioners are currently arguing that failure detection should be made available as a network service [10, 14]. No such service exists to date. Lacking such tool, it is important to devise methods to express in the application layer of our programs failure detection protocols in a simple and natural way.

In the following we introduce one such method—a class of "time-outs", i.e., objects that postpone a certain function call by a given amount of time. This feature allows to convert time-based events into non time-based events such as message arrivals and easily express the constructs in Table 1 in standard C. In some cases, our class allows to get rid of concurrency management requirements such as coroutines or thread management libraries. The formal character of our method allows to rapid-prototype the algorithms with minimal effort. This is proved through a Literate Programming [11] framework that produces from a same source file both the description meant for publication and a software skeleton to be compiled in standard C or C++.

The rest of this article is structured as follows: Section 2 introduces our tool. In Sect. 3 we put our tool to work and use it to express three classical failure detectors. Our conclusions are finally drawn in Sect. 4.

2 A Time-outs Management System

This section briefly describes the architecture of our time-out management system (TOM). The TOM class appears to the user as a couple of new types and a library of functions. Table 2 provides an idea of the client-side proto-

Construct	NFD-E [5]	φ [9]	FD [3]	GMFD [13]	$\mathcal{D} \in \Diamond\mathcal{P}$ [4]	$\mathcal{HB}$ [1]	$\mathcal{HB}$-pt [1]
Repeat periodically	no	no	yes	no	yes	yes	yes
Upon $t =$ current time	yes	no	yes	yes	no	no	no
Upon receive message	yes	yes	yes	yes	yes	yes	yes
Concurrency management	yes	yes	no	no	yes	yes	yes

Table 1. Syntactical constructs used in several failure detector protocols. φ is the accrual failure detector discussed in [9, 10]. $\mathcal{D}$ is the eventually perfect failure detector of [4]. $\mathcal{HB}$ is the Heartbeat detector of [1]. $\mathcal{HB}$-pt is the partition-tolerant version of the Heartbeat detector. By "Concurrency management" we mean coroutines, threading or forking.

```
1. /* declarations */
   TOM *tom; timeout_t t1, t2, t3; int my_alarm(TOM*), another_alarm(TOM*);
2. /* definitions */
   tom ← tom_init(my_alarm);
   tom_declare(&t1, TOM_CYCLIC, TOM_SET_ENABLE, TIMEOUT1, SUBID1, DEADLINE1);
   tom_declare(&t2, TOM_NON_CYCLIC, TOM_SET_ENABLE, TIMEOUT2, SUBID2, DEADLINE2);
   tom_declare(&t3, TOM_CYCLIC, TOM_SET_DISABLE, TIMEOUT3, SUBID3, DEADLINE3);
   tom_set_action(&t3, another_alarm);
3. /* insertion */
   tom_insert(tom, &t1), tom_insert(tom, &t2), tom_insert(tom, &t3);
4. /* control */
   tom_enable(tom, &t3);
   tom_set_deadline(&t2, NEW_DEADLINE2);
   tom_renew(tom, &t2);
   tom_delete(tom, &t1);
5. /* deactivation */
   tom_close(tom);
```

Table 2. An example of usage of the TOM class. In 1. a time-out list pointer and three time-out objects are declared, together with two alarm functions. In 2. the time-out list and the time-outs are initialized, and an alarm differing from the default one is attached to time-out `t3`. Insertion is carried out at point 3. At 4., some control operations are performed on the list, namely, time-out `t3` is enabled, a new deadline value is specified for time-out `t2` which is then renewed to activate the changing, and time-out `t1` is deleted. The whole list is finally deactivated in 5.

col of our tool.

To declare a time-out manager, the user needs to define a pointer to a `TOM` object and then call function `tom_init`. Argument to this function is an "alarm," i.e., in our terminology, the function to be called when a time-out expires:

```
int alarm(TOM*); tom = tom_init(alarm);
```

The first time function `tom_init` is called a custom thread is spawned. That thread is the actual time-out manager.

At this point the user is allowed to define his time-outs. This is done via type `timeout_t` and function `tom_declare`; an example follows:

```
timeout_t t; tom_declare(&t,TOM_CYCLIC,
TOM_SET_ENABLE,TID, TSUBID, DEADLINE).
```

In what above, time-out `t` is declared as:

- A cyclic time-out (renewed on expiration; as opposed to `TOM_NON_CYCLIC`, which means "removed on expiration"),
- enabled (only enabled time-outs "fire", i.e., call their alarm on expiration; an alarm is disabled with `TOM_SET_DISABLE`),
- with a deadline of `DEADLINE` local clock ticks before expiration.

A time-out `t` is identified as a couple of integers, in the above example `TID` and `TSUBID`. This is done because in our experience it is often useful to distinguish instances of

classes of time-outs. We use then `TID` for the class identifier and `TSUBID` for the particular instance.

Once defined, a time-out can be submitted to the time-out manager for insertion in its running list of time-outs. From the user viewpoint, this is managed by calling function

```
tom_insert( TOM *, timeout_t * ).
```

Note that a time-out might be submitted to more than one time-out manager.

After successful insertion an enabled time-out will trigger the call of the default alarm function after the specified deadline. If that time-out is declared as `TOM_CYCLIC` the time-out would then be re-inserted.

Other control functions are available: a time-out can be temporarily suspended while in the time-out list via function

```
tom_disable( TOM *, timeout_t * )
```

and (re-)enabled via function

```
tom_enable( TOM *, timeout_t * ).
```

Furthermore, the user is allowed to specify a new alarm function via `tom_set_action` and a new deadline via `tom_set_deadline`; he can delete a time-out from the list via

```
tom_delete( TOM *, timeout_t * ),
```

and renew[1] it via

```
tom_renew( TOM *, timeout_t * ).
```

Finally, when the time-out management service is no longer needed, the user should call function

```
tom_close( TOM * ),
```

which possibly halts the time-out manager thread should no other client be still active.

2.1 Requirements

A fundamental requirement of our model is that processes must have access to some local physical clock giving them the ability to measure time. The availability of means to control the priorities of processes is also an important factor to reducing the chances of late alarm execution. Note how these are commodity features in most of today's real-time kernels. We also assume that the alarm functions are small grained both in CPU and I/O usage so as not to interfere "too much" with the tasks of the TOM. Finally, we assume the availability of asynchronous, non-blocking primitives to send and receive messages.

[1] Renewing a time-out means removing and re-inserting it.

3 Discussion

In this section we show that the syntactical constructs in Table 1 can be expressed in terms of our class of time-outs. We do so by considering three classical failure detectors and providing their time-out based specifications.

Let us consider the classical formulation of eventually perfect failure detector $\mathcal{D}$ [4]. Its basic structure is that of a coroutine with three concurrent processes, two of which execute a task periodically while the third one is triggered by the arrival of a message:

Every process p executes the following:
$output_p \leftarrow 0$
for all $q \in \Pi$
  $\Delta_p(q) \leftarrow$ default time interval
cobegin
  —— *Task 1:* **repeat periodically**
    send "p-is-alive" to all
  —— *Task 2:* **repeat periodically**
    for all $q \in \Pi$
      if $q \notin output_p$ and p did not receive "q-is-alive" during the last $\Delta_p(q)$ ticks of p's clock **then**
        $output_p \leftarrow output_p \cup \{q\}$
  —— *Task 3:* **when** receive "q-is-alive" for some q
    if $q \in output_p$
      $output_p \leftarrow output_p - \{q\}$
      $\Delta_p(q) \leftarrow \Delta_p(q) + 1$
coend.

We call the **repeat periodically** in *Task 1* a "multiplicity-1" repeat, because indeed a single action (sending a "p-is-alive" message) has to be tracked, while we call "multiplicity-q" repeat the one in *Task 2*, which requires to check q events.

Our reformulation of the above code is as follows:

Every process p executes the following:
timeout_t t_{task1}, t_{task2}[NPROCS];
task_t p, q;
for (q=0; q¡NPROCS; q++) {
  $\Delta_p[q]$ = DEFAULT_TIMEOUT;
  $output_p[q]$ = TRUST;
}
/* "$\rightsquigarrow$" *is our symbol for the "address-of" operator* */
tom_declare($\rightsquigarrow t_{\text{task1}}$, TOM_CYCLIC,
    TOM_SET_ENABLE, p, 0, $\Delta_p[q]$);
tom_set_action($\rightsquigarrow t_{\text{task1}}$, action_Repeat_Task1);
tom_insert($\rightsquigarrow t_{\text{task1}}$);
for (q=0; q¡NPROCS; q++) {
  if ($p \neq q$) {
  tom_declare($t_{\text{task2}}+q$, TOM_CYCLIC,
      TOM_SET_ENABLE, q, 0, $\Delta_p[q]$);
  tom_set_action($t_{\text{task2}}+q$, action_Repeat_Task2);

1. Code of the $\mathcal{HB}$ failure detector for partitionable networks.
Aguilera, Chen and Toueg, Theoretical Computer Science n.1, 1999.

```
#define HEARTBEAT 1
#define ITTB 2
#define SOME_PERIOD 100000
#define FOREVER 1
#define PRESENT 1
#define ABSENT 0
⟨ Initialisation 2 ⟩
```

2. Every process p executes the following:

```
⟨ Initialisation 2 ⟩ ≡
  main( )
  {
    timeout_t τ_t2b;
    message_t m;
    for (q = 0; q < NPROCS; q++) {
      D_p[q] = 0;
      path[q] = ABSENT;
    }
    tom_declare(&τ_t2b, TOM_CYCLIC, TOM_SET_ENABLE, 1, 1, 1);
    tom_set_action(&τ_t2b, actionItsTimeToBroadcast);      /* sends ITTB */
    tom_set_deadline(&τ_t2b, SOME_PERIOD);      /* every 100000 ticks */
    tom_insert(&nextb);
    do {
      getMessage(&m);      /* sets m.date */
      switch (m.type) {
        ⟨ Task1 3 ⟩
        ⟨ Task2 4 ⟩
      }
    } while (FOREVER);
  }
```

This code is used in section 1.

3. Task 1

```
⟨ Task1 3 ⟩ ≡
case ITTB: D_p[p] = D_p[p] + 1;
  m.type = HEARTBEAT, m.path = p;
  for (q = 0; q < NPROCS; q++)
    if (isneighbor(q, p)) sendMessage(m, q);
  break;
```

This code is used in section 2.

4. Code of Task 2

```
⟨ Task2 4 ⟩ ≡
case HEARTBEAT:
  for (q = 0; q < NPROCS; q++)
    if (m.path[q] ≠ ABSENT) D_p[p] = D_p[p] + 1;
  m.path[p] = m.path[p] + 1;
  for (q = 0; q < NPROCS; q++)
    if (isneighbor(q, p) ∧ m.path[q] ≤ PRESENT) sendMessage(m, q);
  break;
```

This code is used in section 2.

5. Extra functions

```
int actionItsTimeToBroadcast( )      /* sends ITTB to caller */
{
  sendMessage(ITTB, p);
}
```

Figure 1. Reformulation of the $\mathcal{HB}$ failure detector for partitionable networks [1]. Special symbols such as τ and $\mathcal{D}_p$ are caught by cweb and translated into legal C tokens via its "@f" construct [12]. The expression $m.path[q] \leq$PRESENT means "q appears at most once in $path$".

```
  tom_insert(⇝t_task1);
  }
}
getMessage(⇝m);
switch (m.type) {
  TASK1; TASK2; TASK3;
}
```

where tasks and actions are defined as follows:

```
TASK1 ≡ case REPEAT_TASK1:
          sendAll(I_AM_ALIVE);
        break;
TASK2 ≡ case REPEAT_TASK2:
          q = m.id;
          if (output_p[q] ≡ TRUST)
            output_p[q] = SUSPECT;
        break;
TASK3 ≡ case I_AM_ALIVE:
          q = m.sender;
          if (output_p[q] ≡ SUSPECT) {
            output_p[q] = TRUST;
            Δ_p(q) = Δ_p(q) + 1;
          }
        break;

action_Repeat_Task1() {
        message_t m;
        m.type = REPEAT_TASK1;
        Send(m, p);
}
action_Repeat_Task2(timeout_t *t) {
        message_t m;
        m.type = REPEAT_TASK2;
        m.id = t->id;
        Send(m, p);
}
```

We can draw the following observations:

- Our syntax produces a longer listing with respect to the original specification, but our syntax is more formal. Indeed we have deliberately chosen a syntax very similar to that of programming languages like C or C++. Behind the lines, we assume also a similar semantics.

- Our syntax is more strongly typed: we have deliberately chosen to define (most of) the objects our code deals with.

- We have systematically avoided set-wise operations such as union, complement or membership by transforming sets into arrays as, e.g., in

  $$output_p \leftarrow output_p \cup \{q\},$$

  which we changed into

  $$output_p[q] = \texttt{PRESENT}.$$

- We have sistematically rewritten the abstract constructs `repeat periodically` as one or more time-outs (depending on their multiplicity). Each of these time-out has an associated action that sends one message to the client process, p. This means that

  1. time-related event "it's time to send p-is-alive to all" becomes event "message `REPEAT_TASK1` has arrived."
  2. time-related events "it's time to check whether q-is-alive has arrived" becomes event "message (`REPEAT_TASK2`, id=q) has arrived."

- Due to the now homogeneous nature of the possible events (they are all message arrivals now) a single process may manage those events through a simple multiple selection statement (in C lingo, a switch). The requirement for a coroutine has been removed.

Through the Literate Programming approach and a compliant tool such as cweb [12, 11] it is possible to further improve our reformulation. As well known, the cweb tool allows to have a single source code to produce a pretty printable TEX documentation and a C file ready for compiling and testing. In our experience this link between these two contexts can be very beneficial: testing or even simply using the code provides feedback on the specification of the algorithm, while the improved specification may reduce the probability of design faults and in general increase the quality of the code.

Figure 1 and Figure 2 respectively show a reformulation for the $\mathcal{HB}$ failure detector for partitionable networks [1] and for the group membership failure detector [13] produced with cweb. Those reformulations include simple translations for the syntactical constructs in Table 1 in terms of our time-out API. An interesting case is that of the group membership failure detector: here the authors mimic the availability of a cyclic time-out service but intrude its management in their formulation—which could have been avoided using our approach.

4 Conclusions

We have introduced a tentative *lingua franca* for the expression of failure detection protocols and in general any algorithm using the constructs in Table 1. TOM has the advantage of being simple, elegant and non ambiguous. Obvious are the many positive relapses that would come from the adoption of a standard, semi-formal representation with respect to the current Babel of informal descriptions—easier acquisition of insight, faster verification, and greater ability to rapid-prototype software systems.

Given the current lack of a network service for failure detection, the availability of standard methods to express failure detectors in the application layer is an important assets: a tool like the one described in this paper isolates and "conquers" a part of the complexity required to express failure detection protocols. This complexity becomes transparent to the designer—which saves development times and costs—and eligible for cost-effective optimizations.

In the future we are planning to make use of our tool in the framework of European Project "ARFLEX" (Adaptive Robots for FLEXible manufacturing systems, IST-NMP2-016880) [2] and IBBT Project "End-to-End Quality of Experience" [8]. A public domain portable version of our tool shall also be developed and validated on algorithms such as the ones introduced in [6] and [7].

References

[1] M. K. Aguilera, W. Chen, and S. Toueg. Using the heartbeat failure detector for quiescent reliable communication and consensus in partitionable networks. *Theoretical Computer Science*, (1):3–30, 1999.

[2] ARFLEX project web page, 2006. http://www.arflexproject.eu/.

[3] M. Bertier, O. Marin, and P. Sens. Implementation and performance of an adaptable failure detector. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN '02)*. IEEE Society Press, June 2002.

[4] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(1):225–267, 1996.

[5] W. Chen, S. Toueg, and M. K. Aguilera. On the quality of service of failure detectors. *IEEE Trans. on Computers*, 51(5):561–580, May 2002.

[6] V. De Florio. *A Fault-Tolerance Linguistic Structure for Distributed Applications*. PhD thesis, Dept. of Electrical Engineering, University of Leuven, October 2000. ISBN 90-5682-266-7.

1. Code of a group membership failure detector.
Raynal and Tronel, Distributed Systems Engineering **6** (1999) 95–102.

```
#define ITTS 1
#define ITTB 2
#define I_AM_ALIVE 3
⟨ Initialisation 2 ⟩
```

2. Every process p executes the following:

```
⟨ Initialisation 2 ⟩ ≡
  main( )
  {
    timeout_t τ_nextt, τ_bcast;
    date_t nextTimeout_i, timeout_i[NPROCS], nextBroadcast_i;
    boolean_t groupFailure_i;
    message_t m;
    task_t j;
    groupFailure_i = False;
    nextBroadcast_i = getCurrentDate( );
    for (q = 0; q < NPROCS; q++) {
      timeout_i[q] = MAX_DATE;
      r_i[q] = 0;
    }
    nextTimeout_i = min(timeout_i, NPROCS);
    tom_set_action(&τ_nextt, actionItsTimeToStop);      /* sends message ITTS */
    tom_set_deadline(&τ_bcast, T_r);
    tom_insert(&τ_nextt);
    tom_set_action(&τ_bcast, actionItsTimeToBroadcast);      /* sends message ITTB */
    tom_set_deadline(&τ_bcast, T_e);
    tom_insert(&τ_bcast);
    while (¬groupFailure_i) {
      getMessage(&m);      /* sets m.date */
      j = m.sender;
      switch (m.type) {
        ⟨ Task1 3 ⟩
        ⟨ Task2 4 ⟩
        ⟨ Task3 5 ⟩
      }
    }
  }
```

This code is used in section 1.

3. Task 1

```
⟨ Task1 3 ⟩ ≡
case ITTB: sendMessageAll(I_AM_ALIVE, b_i);      /* send "i is alive" to all */
  tom_insert(&τ_bcast);      /* a cyclic timeout could have been used here */
  b_i = b_i + 1;
  break;
```

This code is used in section 2.

4. Code of Task 2

```
⟨ Task2 4 ⟩ ≡
case ITTS: groupFailure_i = True;
  break;
```

This code is used in section 2.

5. Code of Task 3

```
⟨ Task3 5 ⟩ ≡
case I_AM_ALIVE: timeout_i[j] = m.date + T_r;
  nextTimeout_i = min(timeout_i, NPROCS);
  tom_set_deadline(&τ_bcast, nextTimeout_i);
  tom_insert(&τ_bcast);
  r_i[j] = r_i[j] + 1;
  break;
```

This code is used in section 2.

6. Ancillary functions.

```
int actionItsTimeToStop( )      /* sends message ITTS */
{
  sendMessage(ITTS, i);
}
int actionItsTimeToBroadcast( )      /* sends message ITTB */
{
  sendMessage(ITTB, i);
}
```

Figure 2. Reformulation of the group membership failure detector [13].

[7] V. De Florio, G. Deconinck, and R. Lauwereins. An algorithm for tolerating crash failures in distributed systems. In *Proc. of the 7th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS)*, pages 9–17, Edinburgh, Scotland, April 2000. IEEE Comp. Soc. Press.

[8] IBBT Project "End-to-end Quality of Experience" web page, 2006. http://www.ibbt.be/site/index.php?id=208&L=1.

[9] N. Hayashibara. *Accrual Failure Detectors*. PhD thesis, School of Information Science, Japan Advanced Institute of Science and Technology, June 2004.

[10] N. Hayashibara, X. Défago, R. Yared, and T. Katayama. The φ accrual failure detector. In *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems (SRDS'04)*, pages 66–78, Florianopolis, Brazil, October 2004.

[11] D. E. Knuth. Literate programming. *The Comp. Jour.*, 27:97–111, 1984.

[12] D. E. Knuth and S. Levy. *The CWEB System of Structured Documentation*. Addison–Wesley, Reading, MA, third edition edition, 1993.

[13] M. Raynal and F. Tronel. Group membership failure detection: a simple protocol and its probabilistic analysis. *Distributed Systems Engineering*, 6:95–102, 1999.

[14] R. van Renesse, Y. Minsky, and M. Hayden. A gossip-style failure detection service. In J. S. N. Davies, K. Raymond, editor, *Middleware '98*, pages 55–70, The Lake District, UK, September 1998. Springer.