TDMA on commercial of-the-shelf hardware : fact and fiction revealed

Reference:
Torfs Wim, Blondia Christian.- TDMA on commercial of-the-shelf hardware : fact and fiction revealed
International journal of electronics and communications - ISSN 1434-8411 - 69:5(2015), p. 800-813
DOI: http://dx.doi.org/doi:10.1016/j.aeue.2015.01.010

TDMA on commercial of-the-shelf hardware: Fact and fiction revealed

Wim Torfs[a,*], Chris Blondia[a]

[a]University of Antwerp - iMinds, Middelheimlaan 1, B-2020 Antwerp, Belgium

Abstract

The proliferation of multimedia applications making use of 802.11n networks, and therefore requiring the accompanying QoS, has made it imperative to improve the QoS guarantees. A well-known method is to employ a TDMA access scheme instead of the standard CSMA access scheme on commodity hardware. A considerable number of related works have focused on this issue, however, many assume the manipulation of commodity hardware to be limited to QoS parameters and none of them did a thorough analysis of one of the most crucial elements to make such a system work, that is, the timer source. The goal of this article is twofold; first we discuss a detailed performance analysis of possible timer sources in different environments and stressed by several methods. Second, we discuss the issues that developers face when using commodity hardware in a TDMA access scheme. As a result we present a successful slotted transmission scheme on commodity hardware where less than 0.1% of the received packets exhibit a jitter larger than 10 μs, while transmitting a packet every 256 μs.

Keywords: TDMA access mode, Linux timers, commodity hardware

1. Introduction

Wi-Fi networks have become increasingly popular in multimedia applications, which can be attributed to their low cost, constantly increasing bandwidth and relative ease of deployment. The IEEE802.11n standard enables multimedia streaming thanks to its higher bandwidth and is now being deployed in applications where previously proprietary radio systems were required.

However, in situations where strict QoS guarantees are required, the standard IEEE802.11 has no answers. Although it can adjust some of the QoS parameters, it is still not possible to provide any guarantees. Using the Wi-Fi cards in TDMA access mode would be able to solve this QoS guarantee issue. Manufacturing such chip that allows for TDMA access mode is an incredibly expensive undertaking that would only be profitable in huge volumes. However, since the MAC layer is completely embedded in the Linux kernel and a huge part of the configuration of the physical layer is also located in software, thanks to open source drivers, it would be possible to substitute the existing code, which is now specific for 802.11, and allow another MAC protocol to take control over the wireless network card.

A considerable number of works have focused on devising a method where TDMA access mode can be used in combination with commodity Wi-Fi cards. Most of these works focus on Atheros hardware and are either based or have some connection to the MadWiFi Linux driver, which is only valid for 802.11b/g hardware. The more recent Atheros 802.11n hardware is controlled by the ath9k driver and was the focus of but a few works. Note that in order to allow TDMA access mode on commodity Wi-Fi cards, the default CSMA/CA operation needs to be circumvented, which is not always clearly specified in the related work. Although none of the related work discusses this, additional care should be taken in the avoidance of inadvertent receptions since they could jeopardize the timing schedule of the transmissions. Moreover, in order for TDMA to work, there needs to be some kind of time reference and timer source, upon which the TDMA functionality can rely to schedule any new packets. This has rarely been studied in the related works,

even while it can have such a huge impact. The precision and reliability with which a timer works, defines the upper bound of the performance for the whole TDMA functionality.

The goal of this article is twofold, first, we study the performance and stability of several potential timer sources to investigate the effect of the usage of the timer source for TDMA access mode. We perform the analysis for the Linux kernel 3.2 and 3.13.9, with different preemption models, different *HZ* configurations [1], with and without the RT-patch [2], on different hardware systems and different system loads, such as timers that are run in parallel or a high throughput network operation. In addition to this analysis, we also compare the obtained results with the performance of an actual real-time operating system (RTOS) such as eCos [3]. Our second goal is to control the Atheros Wi-Fi card such that reliable TDMA is possible and with a resolution of one packet every 250 μs, where the timing should not exhibit a jitter larger than 10 μs. This article goes into the details of achieving a slotted transmission, since the focus is on the accuracy of the scheduled transmissions and not on the actual TDMA scheme. Thereby we elaborate on the issues that need to be faced and provide several possible solutions, even when working in a noisy environment.

The rest of the paper is organized as follows. Related works

*Corresponding author

Email addresses: wim.torfs@uantwerpen.be (Wim Torfs), chris.blondia@uantwerpen.be (Chris Blondia)

are described in Section 2. Section 3 describes the different possible timer sources, while Section 4 presents the results of the analysis of previously discussed timer sources. Section 5 provides an overview of the relevant parts of the hardware and the Linux kernel for our slotted transmission scheme, whereas Section 6 describes our methodology. The obtained results are presented in Section 7 and Section 8 concludes this article.

2. Related work

The interesting properties, that low cost commodity Wi-Fi cards provide, are clear from the number of researchers that have considered this for a customized operation that deviates from the IEEE802.11 standard, for example a TDMA based operation. However, TDMA poses strict timing requirements, which is a topic that is rarely discussed in detail. In order to discover problems regarding the latency of the high resolution timer of Linux, the authors in [4] propose the KTAS methodology, which allows developers to determine the cause of timer latencies. The authors provide an example analysis throughout the paper, analyzing the performance of the hrtimer in terms of latency. Although it is not our goal to determine the cause of the latency, we analyze the functionality of the hrtimer and its related functionality and create several scenarios where some form of stress, such as a heavily loaded network, could result in a higher latency. The resulting measurements should give an indication whether the considered timers are sufficient for the task at hand. Tools such as KTAS can be helpful in determining issues and improving the performance of the high resolution timers when required.

The works that use commodity hardware to create a variation of a TDMA access based network can differ greatly in methodology, some use predominantly a userspace approach, while others prefer to have all features in the kernel and a final category forms hybrids that have part of the functionality in userspace and part in the kernel. One of the works that formed an inspiration to many others is SoftMAC [5], where the authors clearly identified the properties of 802.11 that should be disabled when working with commodity hardware in order to create a precise control over the timing of the wireless transmissions and receptions.

Soft-TDMAC [6] [7] is a protocol where all pairs of network clocks are said to be synchronized to within microseconds of each other. By considering the network as a collection of pairs, they achieve a network wide synchronization. The protocol consists of three parts, a small kernel module, a userspace library and a userspace application. The kernel module mainly forwards incoming and outgoing packets to and from the userspace library and configures the physical network card to abandon its default CSMA/CA behavior by relying on the 802.11 QoS features of the driver. The relevant functionality of the MAC is contained in the Soft-TDMAC application, which is linked to the userspace library. The library itself is responsible for the starting of timers and the triggering of Soft-TDMAC upon the elapse of such timer. One of the arguments for this split functionality is the avoidance of difficult kernel programming, since most lies in userspace. However, since most of the functionality lies in userspace, means also that it runs in process context, where it is susceptible to the scheduling of both other user processes and kernel processes. Such design could deteriorate easily when the load of the system increases. To avoid such issues, the authors decided to use the Linux RT patch [2], which allows tweaking the real-time priorities of the running processes, including kernel processes.

On the other hand, MadMAC [8] is a protocol that is entirely implemented in the kernel and operates on top of the MadWiFi driver. The goal is to have a configurable MAC, which transmits packets at a configurable time without triggering the CSMA functionality of the hardware, such that the packet transmissions do not experience delays from neither contention nor backoff. In order to disable some of the CSMA functionality, the hardware is placed in monitor mode. FreeMAC [9] can be deemed as an extension to MadMAC, where FreeMAC focuses not only on TDMA, but also on other MAC protocols that require strict control over the timing. Moreover, it extends its applicability to multichannel MAC protocols by providing the ability to switch between different frequency channels. The protocol makes use of the timer which is embedded in the Wi-Fi card.

All previous works were in some way making use of the MadWiFi driver, which is a driver for the older generation of 802.11 cards, i.e. 802.11n is not supported. The implementation of JaldiMAC, described in [10], refers to 802.11n enabled Wi-Fi cards, thereby opening the possibilities to use high throughput hardware. The protocol is partitioned in two parts, the first a click module, which resides in userspace and the second a kernel driver for the wireless network card that is used. The authors claim that the ath9k driver is considered to be unstable and therefore have written a proprietary driver for the adopted network card. The precise timing is done in the kernel module, while a more crude, relative timing is defined in in the click module, thereby making sure that the timing is in relative values and not in absolute timestamps in order to cope with timer jitter or unexpected transmit latencies.

The authors of [11] developed a protocol, RT-WiFi, where the 802.11 MAC and network card driver are replaced by the proposed implementation of a MAC. Although it is not clearly mentioned, we believe that the timer which is embedded in the Wi-Fi card is used. By using the timer as a reference for the link scheduler, tasks and data can be reliably scheduled. While the protocol seems to work fine in an isolated area, the authors noticed that there is an issue when operating in a non-isolated area, where other Wi-Fi devices are available, which is an issue that is not discussed in any of the other papers, whereas it jeopardizes, for example, the synchronization of the whole network.

The framework proposed in [12] takes the reuse of commodity hardware to the next level. The framework allows the dynamic and flexible configuration of a MAC protocol, either being TDMA or CSMA based. The firmware of commodity hardware is rewritten by means of the open firmware source code and accepts new MAC configurations which can be executed. Because of such methodology, it can only be applied to hardware where the firmware is allowed to be replaced, i.e. it depends whether the method is public knowledge. To current date,

this is only possible with Broadcom devices and although such an implementation would require a significant amount of work, the results would be very accurate since there is no dependency on the timings of the host system, while the adjusted firmware could consider real-time algorithms.

3. Timer analysis

A common known fact is that Linux is not a real time operating system, although many try to use it for such purpose. Linux does not provide any guarantees whatsoever regarding timer expirations. However, in some use cases a hard real-time system is not required and only the overall performance needs to be considered sufficient. We discuss in this section several possible timer sources that can be used for our use case, which is, adjusting the network card operation such that TDMA is possible. Note that for this purpose, we are considering the timer facilities at kernel level and not the timers that are available at user level. First we give a short overview of time related topics of the Linux kernel and what could be considered as timers, after which we go into detail regarding possible timer sources. In the next section, we will compare the performance measurements of each of the discussed timers.

The timer sources that we identified as potential trigger inputs for the TDMA access mode are: Software Beacon Alert, General Purpose timers, Linux timer wheel, Linux hrtimer and a proprietary high priority timer. The first two timers are embedded in the Atheros hardware, while the last three timers are Linux software timers.

3.1. Overview

It is a well known fact that Linux uses ticks as a time reference [1] [13] [14]. The scheduling of processes is for example dependent on the elapse of ticks. One tick of the system clock has a duration of one jiffy, which is $1/HZ$ seconds, with HZ being a kernel configuration parameter that can be set to 100, 250, 300 or 1000. The system clock is nothing more than a counter, which gets updated on every tick. The kernel is programmed to activate a software timer that fires every $1/HZ$ seconds to update the tick count and the wall clock time among other things.

Linux provides an abstraction of the hardware that is used for timing by means of clocksources and clockevents. Upon boot, all possible clocksources and clockevents are registered, that is, all configured timer drivers detect whether their respective hardware is available and register themselves as possible clocksources or clockevents. A hardware timer that operates as a clocksource is a free running counter with interrupts disabled. A hardware timer that is used as a clockevent is a timer without reload operation that can be programmed to signal an interrupt at a certain deadline.

Only a single clocksource is used by the Linux system at a time, whereas multiple clockevents are available. At boot time, each CPU selects the most appropriate clockevent for the CPU, taking into account CPU-affinity, resolution and stability.

In a single processor system, it comes down to two timers. The clocksource is used as a reference and the clockevent is used to schedule a timer interrupt. A software timer schedules a clockevent by comparing the earliest timer deadline with the current system time and loads the overflow value minus the remaining time into the hardware timer. All time critical system timers and the tick counter make use of this interface. To update the system jiffies counter for example, a timer is scheduled every $1/HZ$ seconds. The scheduler also requires access to such timers, as well as some networking subsystems and some of the network drivers. All these timers are scheduled on that single clockevent.

Since version 2.6.21 of the Linux kernel, dynamic ticks are introduced, thereby eliminating the requirement of having periodic timer ticks [15]. A CPU that is going into idle mode is exempt from having to generate timer ticks every $1/HZ$ seconds. Such a provision allows CPUs to keep sleeping until woken up , thereby not forcing the CPU to wake up every time to handle the tick timer and go to sleep again. Upon leaving the idle state, the periodic tick is resumed and the number of elapsed jiffies during the idle time is updated. This ensures a better and more efficient energy usage.

3.2. Software Beacon Alert (SWBA) and General Purpose timers

There are eight General Purpose timers that are accessible in the AR9220 network card. The Software Beacon Alert timer (SWBA, later also called bcntimer) is one of them and has a precision of 128 μs. In normal operating conditions, this timer is used to signal the arrival times of the beacons on stations and initiates the creation of a new beacon packet on the Access-Point (AP) side. Since all eight timers use the same hardware core, analyzing the performance of just the SWBA timer is sufficient.

The expiration of this timer leads to the generation of a PCI interrupt, which is delivered to the ath9k driver, since it has registered itself to capture interrupts from this PCI resource. The original code of the ath9k driver schedules a tasklet upon reception of this interrupt. We modified this code such that upon reception of the interrupt, the timer performance monitoring function is called (defined in Section 4.1).

3.3. Linux 'Timer wheel'

The Linux kernel timer wheel is a software structure that triggers timer handlers based on their expiry time in jiffies [16]. Therefore, the resolution of the timer is limited by the HZ configuration at kernel compile time. The timer wheel consists of five categories in which future timer expiries are placed. Each of the categories represents $256 * 64^n$ jiffies, with $n = 0 \ldots 4$, where n is the category index. The lowest category represents the earliest 256 jiffies, all sorted, while each of the following categories is subdivided in 64 buckets, which are coarsely sorted. A new timer expiry is placed into the associated bucket, based upon its expiry time. Each tick, the lowest category is checked for possible timers that should be triggered. After the elapsing of the time of a lower category, the next bucket of the higher category is cascaded down into the lower category. This approach results in a high processing overhead when timers are

cascaded down [17] [18]. However, if the timers are removed from the timer wheel before expiry, the processing overhead of cascading the timer is removed. Therefore, this type of structure is useful for timers that are needed for some kind of deadline, in case an unexpected event occurred.

Note that there is no reference towards any hardware timer source, since the timer wheels operation depends on the tick rate. The tick rate itself is depending on a hardware timer or another higher resolution timer, such as the hrtimer. We do not analyze the performance of the timer wheel since it depends on another timer, such as the hrtimer, and does certainly not perform better.

3.4. Linux high resolution timer (hrtimer)

The Linux kernel high resolution timer [19] [20] was merged into the 2.6.16 kernel. It provides a RB-tree structure per processor, where timers are sorted according to their expiry time. The high resolution timer should be able to provide a nanosecond resolution according to the documentation. Note that the hrtimer is a software structure, where the next expiry time is checked regularly and as soon as the expiry time comes within range of the current time, a clockevent is scheduled. Upon expiry of the hrtimer the *hrtimer_interrupt* function is called, which calls the handler function of the timer, which was provided during initialization. During our measurements, the *timer performance monitoring function*, which resides in our timer performance test kernel module, is selected to be the handler function of the timer. Note that during the short time the *timer performance monitoring procedure* is active, system interrupts can be triggered, since by the design of the hrtimer, it is run outside of the critical section of the timer interrupt.

When scheduling a hrtimer, two selections need to be provided. First, the timer deadline can be expressed as absolute, HRTIMER_MODE_ABS, or as a time relative to the current time, HRTIMER_MODE_REL. The latter is converted to an absolute time in the future by the hrtimer initialization code and therefore, both methods give similar results for the same timer interval. Second, the time base should be selected, which is the time line that should serve as a reference when deciding whether the timer should fire or not. The best known time bases are CLOCK_REALTIME and CLOCK_MONOTONIC. The former time base represents the time-of-day time, which can jump back and forth as the system time-of-day clock is changed, including by NTP. The latter represents the wall clock time, which is the elapsed time since some point in time, frequently the system boot time unless some persistent battery backed up clock is available. The CLOCK_MONOTONIC time base is unaffected by the system clock changes and is therefore best suited to use in a performance analysis.

Since the most critical parts of the hrtimer operate in interrupt or tasklet and softIRQ context, there are not many configuration options that could influence its performance. Under normal operating conditions, that is, making use of one of the available configuration values, the value of *HZ* is not going to have a huge influence on the performance of the hrtimer. Basically, the *HZ* value determines the rate of a single hrtimer that is used to update the tick count. The maximum value of *HZ*, which is defined in the standard kernel configuration for x86 and ARM, is 1000, which is the equivalent of just an extra timer running with an interrupt interval of 1ms.

Likewise, the dynamic tick configuration is not going to have any influence on the performance of the hrtimer, since the tick counter is just one of the timers that make use of the hrtimer. Moreover, the dynamic tick configuration is only applicable when the CPU goes idle and the idle time is longer than the configured tick rate, which is not the case in our measurements, since our timer intervals are equal or faster than the tick rate. Nonetheless we do run performance tests which indicate that the dynamic tick configuration is to be discarded from the list of possible influence sources.

The preemptiveness of the kernel is also a parameter that is not going to have a huge influence, even when the kernel is fully preemptive. Preemption is enforced in the process context, while everything of the hrtimer handling, even the timer specific handler function, occurs in interrupt context, which is executed immediately unless interrupts are disabled. Both the timer specific handler function as the disabling of interrupts in other drivers should be kept to the bare minimum, otherwise this would be perceived as bad coding. To prove our point, we performed performance tests with both a preemptible kernel (Preemptible Kernel, Low-Latency Desktop (CONFIG_PREEMPT)) and a non-preemptible kernel (No Forced Preemption, Server (CONFIG_PREEMPT_NONE)) with kernel version 3.13.9.

3.5. Linux clockevents

The whole timer system within Linux employs a single clocksource, which is mostly used for time comparison and getting the exact elapsed time. Furthermore, there is a single clockevent for each CPU, which is programmed every time to fire the next timer deadline. The hrtimer Red-Black-Tree contains all hrtimer structures that accommodate, amongst others, a handle to the specific timer and its deadline. Only a single Red-Black-Tree is constructed per CPU in Linux and all planned timer events, such as the periodic tick timer, some of the networking subsystem and all other mechanisms that required a precise timing are all stored in this single tree. It is imaginable that it can happen that the deadlines of some timers are thus close to each other such that the time to start the hardware timer is too small in order for the timer to fire in time. The hrtimer code solves this by defining a soft deadline, which can be lower than the actual deadline. Such case is therefore resolved by searching for timers whose actual deadline is close and whose soft deadline has already expired. However, we would like to have a timer interrupt that is very precise and which does not fire around the time it should fire.

Therefore, we designed a customized timer functionality, which we call hptimer, which at its core is based upon the hrtimer functionality, but otherwise differs significantly. The hptimer functionality is linked to a single CPU and tries to allocate the best possible clockevent for its purposes upon boot time, different from the clockevent allocated to the hrtimer. For this, the clockevent structure needed some adjustment, as well as the platform specific code to free some of the hardware

timers that were otherwise not made available. When an hptimer is created, it is assumed that the timer is a periodic timer, unless otherwise stated. The hptimer functionality will make sure that the timer is repeatedly inserted again into the timer storage structure, which could be a Red-Black-Tree or even a simple list. The expectation is that not a huge amount of timers is going to use this structure, it is only intended to be used for the timers with the highest priority for which it is of the utmost importance that they provide a precise timer interrupt.

Upon the creation of such hptimer, the hptimer functionality checks whether the timer is divergent with all other timers that are already stored. If this condition is not fulfilled, the timer tries again at a different CPU, until either a fit CPU has been found or not found at all. This is to make sure that there is sufficient time to program the clockevent for firing the next deadline. The actual timer interrupt routine is kept as short as possible and at every interrupt a pointer is made to the next deadline, to minimize the time needed to update the hardware timer when required.

3.6. RTOS

There exists an extensive set of real-time operating systems that can operate on a wide variety of hardware. Some of them can be considered as Linux derivatives, such as Xenomai [21] and RTAI [22], where an additional layer, a micro-kernel, is introduced in the original Linux kernel. However, most of these systems focus on the timer system, the process scheduling, the interrupts etc., but not on the real-time performance of the drivers, such as the ath9k driver for wireless Atheros devices. The ones that do provide drivers for Wi-Fi, do not yet support 802.11n chipsets. In order to exhibit real-timer performance in the ath9k driver, it would need to be ported to the respective RTOS. The Linux RT-patch project [2] tries to follow a different path in order to improve the responsiveness of the kernel, that is, it provides a set of patches that influence the global operation of the original Linux kernel. We perform measurements with the RT-patch and eCos [3], a real-time OS which can be deployed both on ARM and x86 devices, on the same platform as we tested the regular Linux kernels, in order to have an idea what to expect from an actual real-time OS.

4. Timer performance results

4.1. Timer analysis methodology

In order to measure the performance of timer interrupts, the timer interrupt handling procedure stores the current time at the kernel level, obtained by *ktime_get*, as raw data in a buffer at every timer interrupt. We refer to this procedure as the *timer performance monitoring function*. After the requested number of measurements is reached, that is, half a million measurements, the measurement stops and the buffer with the stored timing data is converted to a string buffer, which is used as a source for a debugfs file. As a result, the timing data can be accessed in userspace and stored into a file, on which the timing analysis and processing is done. The method is designed such that as little as possible measurement artifacts are possible by reducing the time in the interrupt routine to a bare minimum, which is why no string functions or *printk* are used in the interrupt routine. The usage of *printk* during the whole measurement is inhibited since it locks the CPU where it is running on and disables interrupts until it has finished its job. On multiprocessor computer boards, this would not be such a huge issue, however, when working with a single CPU processor board, this means that no further timer interrupts are being processed until *printk* has finished its job, which could mean that around 5ms no interrupts are being received, depending on the length of the message.

The performance of the previously discussed software timers is largely defined by the interrupt handling speed, the quantity of concurrent interrupts and the efficiency of the timer deadline storage facility, which is inherent to the structure of the software timer. As a consequence, the system load will not result in any deterioration of the timers, unless the higher load also leads to a significant increase in the number of interrupts. However, the hardware timers based on the Atheros wireless card can also degrade due to excessive communication on the PCI bus.

In order to emulate an actual system with some activity, not only measurements have been performed with just the timer under test, but also with a significant network load or other timers at different rates running in parallel. The latter is achieved by using a userspace program called cyclictest [23]. Having timers running in parallel allows investigating the influence of having multiple timer deadlines stored in the same software construction and therefore also scheduled on the same hardware timer. Moreover, the upsurge of timer activity also increases the frequency of interrupts. The parallel timers consist of 10 timers, of which the timer interval increases with 100 μs for every timer, of the interfering timers which start with an interval of 100 μs. The timer interval of the interfering timers which start with an interval of 1000 μs stays fixed.

We used two different platforms with a different architecture to perform this analysis. The first platform is an x86 compatible processor, ALIX3D3, 500MHz AMD Geode LX800 CPU, mini-PCI slot. The second platform is an IGEPv2 board with a 1GHz ARM CortexA8 processor.

To ensure that the measurements provide a realistic view of the to be expected performance, every measurement run collects timestamps from at least half a million timer interrupts and the test is repeated 10 times, after which a statistical analysis is done on the results.

4.2. Timer results

All figures depict the statistical analysis of measurement results, where the 95% confidence interval is indicated by the top and bottom values, while the average is indicated by a point. The different measurement scenarios regarding interfering timers are depicted at the x-axis, where nonoise is the case where no extra timers are employed. The other cases relate to the interfering timer characteristics, where the first number indicates the requested starting interrupt interval, the second number indicates the real-time priority and the last number indicates the number of interfering timers. Note that the default

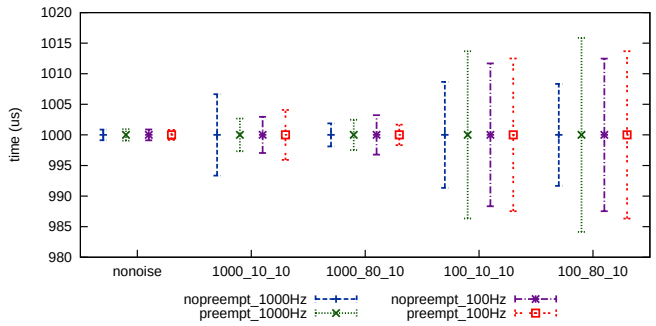

Figure 1: Hrtimer of kernel version 3.13.9, influence of preemption and *HZ* value, 1000 μs interval

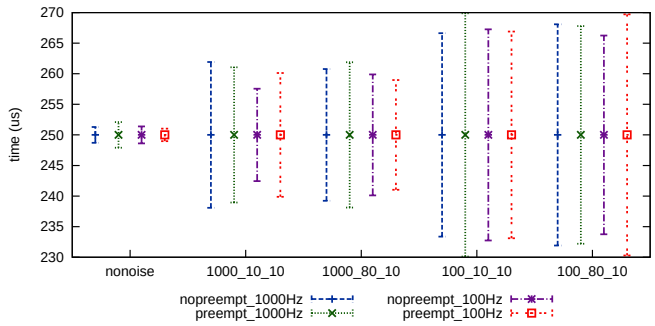

Figure 2: Hrtimer of kernel version 3.13.9, influence of preemption and *HZ* value, 250 μs interval

system timers are also running in parallel for all use cases. In the case of eCos there is no priority for the interfering timers and therefore, only two numbers identify the interfering timers, the interval and the number of timers.

First, we show the measured performance of the hrtimer of kernel 3.13.9, for all extreme combinations of preemptiveness (non-preemptive and preemptive) and *HZ* values (1000HZ and 100HZ). Second, we make the comparison between the hrtimer and the bcntimer (Atheros Software Beacon Alert(SWBA) timer), between kernel 3.13.9 against kernel 3.2 kernel, with and without network load or dynticks. Third, a proprietary timer, which makes use of a dedicated clockevent is compared to the performance of the hrtimer. Fourth, the Linux RT-patch is considered to verify whether this would improve the timer performance and last, the timer performance of a real-time OS is validated.

Unless otherwise mentioned, all measurements are performed on the Alix platform while using the timestamp counter (tsc) as clocksource.

The Figures 1 and 2 depict the obtained measurement results for a 1000 μs and 250 μs interval respectively with different preemptiveness configurations and *HZ* configurations. The average value is in every case almost exactly the requested timer interval with a fixed deviation in the order of nanoseconds. It can be noticed that an increase in the number of timers running in parallel increases the jitter, which is the deviation from the average value. The maximum deviation that can be observed occasionally is limited to 40 μs for a 1000 μs interval and 45 μs for a 250 μs interval. Note that a deviation of more than 2 μs from the average value is observed only seven times in half a million samples. Although that performance is impressive, it is possible to detect an outlier in some of the test runs when extra timers are running in parallel at an interval of 100 μs, thereby blocking interrupts once over a duration of three to four milliseconds.

As was expected, there is no noticeable influence from the *HZ* configuration or from the preemptiveness configuration. The preemptiveness reflects itself on the process management level, while the *HZ* configuration is dependent on the hrtimer and is basically nothing more than an extra timer running in parallel. On the other hand, the hrtimer is a low level timer, which is mostly influenced by the interrupt handling delay and the

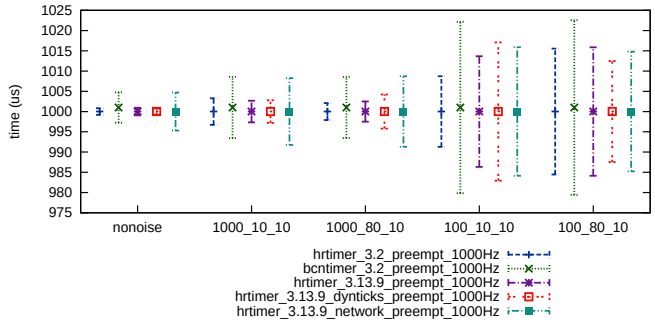

Figure 3: SWBA timer (bcntimer) vs hrtimer of the 3.2 and 3.13.9 kernel, 1000 μs interval

structure of the hrtimer that determines the speed with which the next timer deadline can be programmed. None of those things are influenced by the preemptiveness or the *HZ* configuration.

Another observation that can be made is that the priority of the interfering timers has no influence, there is no direct relation between the timer priority and the jitter. This is to be expected, since the timer priority is related to the process context, which should not influence the basic behavior of the kernel timers on a regular Linux kernel, which operate in interrupt context.

The following figures, Figs. 3 and 4, depict the comparison of the performance of the hrtimer vs. the bcntimer, kernel 3.13.9 vs. kernel 3.2, the usage of periodic ticks vs. the usage of dynticks and the effect of having a heavy network load over the wireless link, for a 1000 μs and 250 μs interval respectively. Since from the previous paragraph could be deduced that neither the preemptiveness nor the *HZ* configuration deemed to have any influence on the results, we selected only the results with preemption (Desktop model) and a *HZ* value equal to 1000.

The average measured interval of all hrtimer measurements is almost the same as the requested timer interval, with a fixed deviation in the order of nanoseconds. The SWBA timer (bcntimer) is less precise, since its average value has a fixed deviation towards the requested timer interval in the order of microseconds. In all cases results the stress of several timers running in parallel in a larger jitter. It can be noticed that the performance of the hrtimer of kernel 3.2 and kernel 3.13.9 is very similar, which can be expected since no structural changes

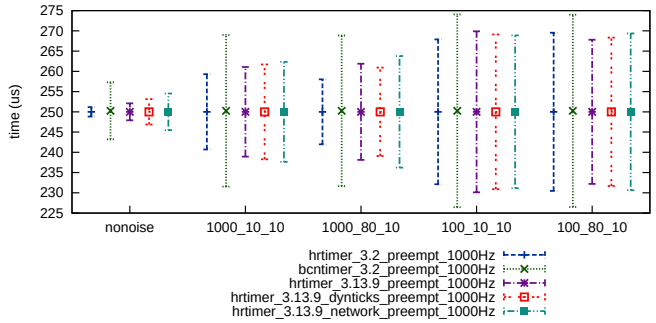

Figure 4: SWBA timer (bcntimer) vs hrtimer of the 3.2 and 3.13.9 kernel, 250μs interval

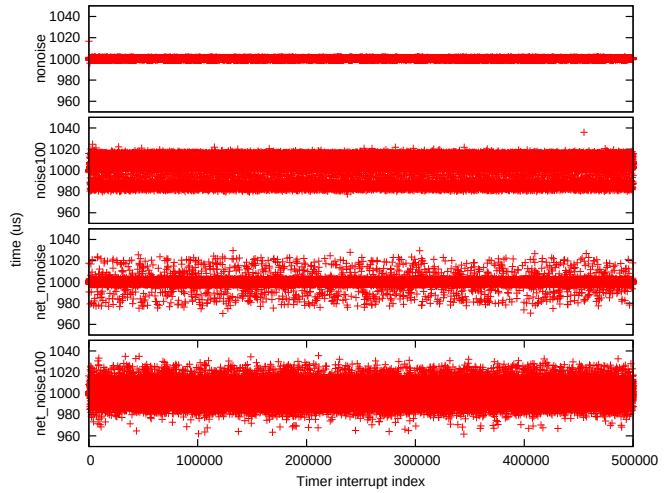

Figure 5: hrtimer of the 3.13.9 kernel, 1000 μs interval, with and without noise, with and without network operation

occurred between the two kernel versions. Likewise, the figure clearly indicates that the hrtimer performance with periodic tick configuration and with dynamic tick configuration is very similar. As we mentioned before, the dynamic tick rate is not expected to influence the operation of the hrtimer.

The less precise average interrupt interval of the bcntimer can be partially attributed to the coarser granularity of the timer. However, from the figures can also be observed that the jitter is larger when using the bcntimer compared to the hrtimer. The absolute maximum jitter of the hrtimer is around 32 μs for the hrtimer at an interval of 1000 μs and between 32 μs and 46 μs for the hrtimer at an interval of 250 μs (for both kernel versions 3.2 and 3.13.9), while the absolute maximum jitter of the bcntimer is around 77 μs and 79 μs for an interval of 1000 μs and 250 μs respectively. This maximum jitter excludes the single case where during three to four milliseconds no interrupt was handled and therefore resulted in an outlier. Those outliers occur once or twice during a complete test run, which is once every 5 million samples, especially when the interrupts are heavily stressed by running several high rate timers in parallel. The occurrence of these outliers is indifferent of which timer is being monitored and can be therefore attributed to the system and not some artifact of one of the timers.

Note that the performance of the bcntimer is mainly determined by the PCI bus and the interrupt system in the kernel, since the remainder of the timer interrupt is embedded in hardware in the Atheros network card and its delay is insignificant to the delays experienced by the software. The interference of the parallel timers does not reside in the timer software construction and scheduling, however in the interrupt handling of the timers. Since we only verified the influence of parallel timers, further performance degradation could result from a heavily loaded PCI bus and interrupts.

In order to verify the performance of the hrtimer in combination with a significant network load, which was said to be the main cause for timer latencies in [4] due to the SoftIRQs, we performed a timer analysis of the hrtimer with kernel version 3.13.9, where during the timer analysis data was being sent as fast as possible over a wireless link. The average transmission speed resulted to be between 45Mbps and 98Mbps. In both Figs. 3 and 4 it can be noticed that there is indeed a negative influence from the network load compared to the results without network load, while there are no timers running in parallel. If the load from the timers starts to increase, such as when running ten timers in parallel at intervals of 100 μs and rising, the effect from the network load is not noticeable. In order to highlight this, Fig. 5 depicts the hrtimer of kernel 3.13.9 without network load in the top two figures for both no noise and noise with timers at intervals of 100 μs. The bottom two figures depict the hrtimer with network load for both no noise and noise with timers at intervals of 100 μs. From the figures where there are timers in parallel, the distinction is not so clear. However, when comparing the figures where no timers are running in parallel, there is a clear difference between the case where there is no network load and the case where there is. We do not analyze the cause of the performance drop, however, besides the SoftIRQ mentioned in [4], when receiving this much data, the wireless network card is generating quite a lot of interrupts over the PCI bus, around 8000 per second when no aggregation is used. This is equivalent to a timer running at an interval of 125 μs. Therefore, it is not necessarily the cause of SoftIRQs that degradation can be observed. However, it is clear that heavy network operation does influence the timer performance, if not in such a high order as timers that are running in parallel.

In order to verify the influence of having all system timers on the same timer structure, we constructed a customized timer functionality. Since there were more hardware timers available on the IGEPv2 ARM platform and the code is more accessible, we modified the OMAP2 architecture specific code to support our hptimer functionality. The Figs. 6 and 7 depict the comparison between the regular hrtimer and the custom made high priority hptimer. The ARM platform provides two types of hardware timers, the gp timer and the more stable, but with a lower resolution, 32k timer. We did performance tests with both hardware timers. The hrtimer and hptimer results with prefix *ARM* indicate that they made use of the higher resolution gp timer, while the results with as prefix *ARM_32k* made use of the 32k hardware timer. For the timer tests with a 250 μs interval, we do not show the results with the 32k timer, since it is not sufficiently precise to give us any meaningful information.

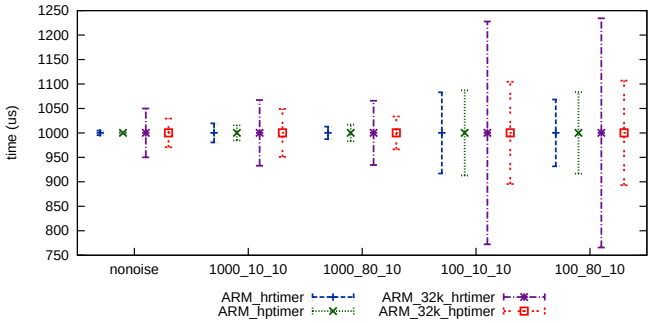

Figure 6: Hrtimer and hptimer of kernel 3.2 on ARM-Cortex A8, 1000 μs interval

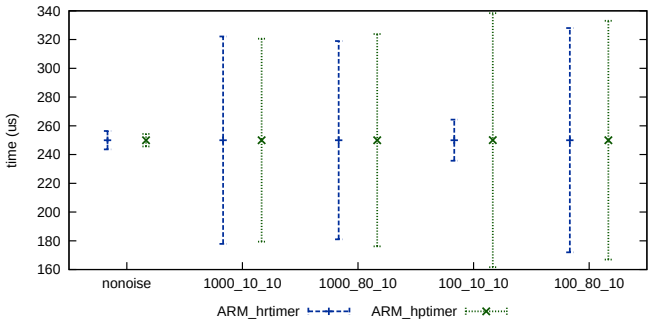

Figure 7: Hrtimer and hptimer of kernel 3.2 on ARM-Cortex A8, 250 μs interval

A very similar performance of the hrtimer and hptimer can be observed from the figures. There are some variations, sometimes the hrtimer shows a better performance, other times the hptimer performs better, however, this is due to random events upon which we cannot excise control. When using the 32k hardware timer, it seems that the hptimer is performing slightly better, however, the performance improvement is marginal compared to the performance of the hrtimer and not worth the effort of implementing another type of timer functionality in the kernel. These results show us that in a system with a single CPU, we need not fear that the multitude of timers that are scheduled concurrently will deteriorate the timer performance. Any experienced deterioration is not due to the hrtimer structure.

The Linux RT patch [2] is a Linux project that focuses on providing patches for the kernel in order to enhance the real-time behavior. The enhanced kernel does not provide a hard real-time Linux system, however, it aims to improve the performance of certain tasks that have a higher real-time priority. One of the changes that the patch employs is to convert most interrupt handlers into preemptible kernel threads. In such manner, the time spent in interrupt context is shorter, allowing a faster irq handling. Since the hrtimer interrupt handling is not converted to a kernel thread, it could be interesting to investigate the performance of the hrtimer of the patched kernel.

However, note that when people talk about real-time Linux, they often refer to the responsiveness of user processes. Not only placing the interrupt handlers in kernel threads, but also the accompanied manipulation of spin_locks, the addition of priority inheritance and other real-time mechanisms ensures a com-

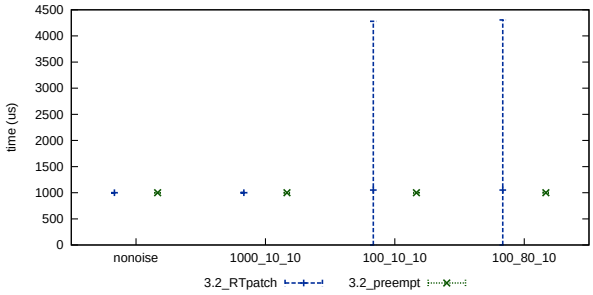

Figure 8: Hrtimer interrupt interval time, RT-patch, 1000 μs interval

plete different behavior of the system. This is also noticeable in the results (Fig. 8), where it is clear that the system modifications result in extreme jitter experienced when high rate timers are running in parallel. We did not do performance measurements with an interval of 250 μs, since the performance with 1000 μs already showed us that there is a significant degradation compared to the original kernel.

In order to get a reading on the efficiency that can be expected when working with an RTOS, Fig. 9 depicts the results of a timer test that has been run on eCos, both for a 1000 μs and 200 μs timer interval. Unlike with the Linux based test runs, the timers running in parallel consist of parallel threads that fire a timer at regular intervals. Since there are no real-time priorities associated to the interfering timers, only the number of interfering timers is indicated at the x-axis. It is clear that the performance of an actual RTOS can significantly outperform that of the hrtimer performance within the Linux kernel, even in the most stressed situations. The maximum deviation within the 95% confidence interval ranges from 400 ns to 2.5 μs. Note that, since eCos does not provide such extensive hardware support, a less efficient hardware timer has been selected to provide the test results compared to the one used with the Linux kernel performance evaluations. Although the difference between the results is significant, some perspective is in order. The Linux system provides interactive multi-user, multi-application support, running around 100 processes in parallel, while eCos is a single application OS, where in our test application only eleven parallel threads were running. The goal of the operating systems is different, where Linux tries to support everything that is required and more, while eCos lacks a lot of that support and requires some extra work to enable some functionality.

In conclusion, we can say that the timers within a real-time OS outperforms significantly the Linux kernel timers, however, unfortunately, hardware support is limited and Atheros wireless devices are not supported within eCos. Furthermore, the Linux hrtimer can be deemed sufficiently efficient, even although all system timers make use of the same timer structure, since in comparison with an isolated timer structure, which was specifically designed for high priority timers, the performance of the hrtimer is similar. The beacon timer of the Atheros card itself is less precise, but it is hard to say whether this is a result of the accuracy of the hardware or the variable delays that occur during the signaling of the interrupt over the PCI bus. Also the coarser granularity of the timer might have an influence. Neither the

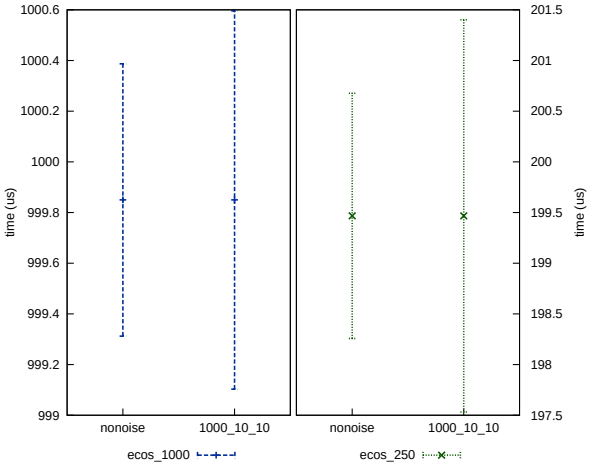

Figure 9: Ecos interrupt interval time, 1000 μs interval

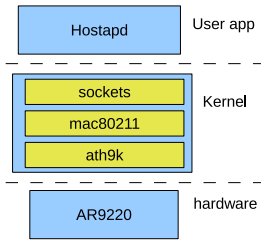

Figure 10: Access point architecture on Linux

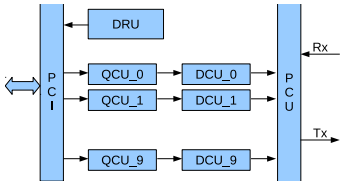

Figure 11: Queue structure of AR9220

preemptiveness, nor the *HZ* configuration, nor the dynamic tick configuration has a significant influence on the performance of the timers, as was expected. The influence of a heavy network load degrades the performance of the hrtimer, however, several high speed hrtimers running in parallel cause a much larger degradation in the performance. Note that these results are related to the system on which they were performed. The type and speed of the CPU can have a significant influence. An SMP machine will for example exhibit a better performance, since timers can be spread out over all the processors. However, in order to do a performance analysis of the timer itself and not of the combined system, a single processor system should be selected, such as we have done.

5. Reliable Tx on commodity Wi-Fi cards

While the performance analysis of the different possible timer sources was depicted in previous sections, this section depicts our second goal, which is to provide a reliable and precise slotted transmission arrangement with commodity hardware. The Wi-Fi cards that we decided to use are based upon the IEEE802.11n Atheros AR9220 chip. In the following sections, we give an overview of the relevant parts of the Linux kernel regarding the wireless network operation with the AR9220 card, along with common misconceptions and things to keep attentive of. In the sections afterwards, the methodology how to adjust the relevant parts is disclosed, after which the resulting implementation and performance results are presented.

At the time of conceiving this implementation, Linux 2.6.38.2 is the most recent stable kernel and AR9220 is one of the 802.11n chipsets that is commercially available on wireless cards. AR9220 interfaces to the host PC by means of a mini-PCI interface. The AR9280 chipset is the same chipset as the AR9220, except that the AR9220 provides an interface through a mini-PCI bus and the AR9280 provides a PCI-Express interface. Both hardware devices provide an interface to all its configuration registers through the 'host and peripheral interface'. The actual control and interrupt handling needs to be implemented on the host side, that is, the driver of the operating system, which is ath9k for Linux kernel version 2.6.38.2.

While the AR9002U chipset, which provides a USB interface, is composed out of the AR9280 and the AR7010 chipsets, it is not possible to directly access the registers of the embedded AR9280 chipset, as it is possible when having the AR9280 in combination with a PCI interface. The integrated AR9280 is controlled directly by the integrated AR7010. Firmware, that provides the same functionality as the ath9k driver, needs to be loaded into the AR7010, which provides only a command interface towards the host. Since this firmware is not open source software, no modifications can be made to this code, hence the exclusion of the USB version from our tests.

Because of the nature of our intended use case, that is, a multimedia conferencing system, we decided to make use of the Access-Point functionality of a typical Linux system. The architecture to setup a Linux based system as Access-Point can be seen in Figure 10. The individual parts of the figure are discussed in the next subsections.

5.1. AR9220

Starting from the bottom, the AR9220 is the Atheros chipset which resides on the mini-PCI wireless 802.11n card. The chipset provides the physical and baseband functionality of IEEE802.11n. The baseband can be configured by manipulating the memory which contains configurable parameters through the 'host and peripheral interface', that is, the mini-PCI interface.

The internal operation of the AR9220 chipset, more specifically, the transmission and reception architecture is depicted in Figure 11. The DRU (DMA Receive Unit) is the module responsible for the transferring and signaling of received packets. It expects to contain a pointer to some memory in the DMA range of the host, such that any incoming packet can be stored in that area and an interrupt is fired upon complete reception of the packet. The ath9k driver should handle the interrupt and initialize a new pointer to some memory in the DMA range.

The transmission queue system is somewhat more complex due to the provision of QoS. There are ten QCUs (Queue Control Unit), of which *QCU9* has the highest priority, that are linked to their dedicated DCU (Distributed Coordination Function(DCF) Control Unit). Data that needs to be sent is encapsulated in a Tx descriptor, together with per packet meta information for the baseband. A QCU stores a pointer to the first Tx descriptor of a linked list of Tx descriptors, which are located in the memory of the host PC. The QCU transfers the packet to its associated DCU depending on whether the conditions of the scheduling policy are fulfilled (the queue is triggered if necessary). The QCU provides several possible scheduling policies, that is, event-based, time-based or ASAP. In the case a trigger or timer is required, it needs to originate from the chipset itself, external triggers are excluded.

The DCU (Distributed Coordination Function Control Unit) provides the DCF functionality (Distributed Coordination Function), such as backoff, waiting for an idle channel, etc. When a packet is ready to be transmitted, the DCU signals the DCU arbiter that a packet is waiting to be transmitted. The DCU arbiter decides which DCU is allowed to send its packet to the PCU (Protocol Control Unit), according amongst others their respective priority. After the actual transmission, the DCU writes the status in the Tx descriptor, by accessing the pointer to the host memory. The PCU is responsible for buffering Tx and Rx frames, automatic acknowledgement transmission, RTS, CTS, carrier sensing, etc. The AR9220 chipset also provides a diagnostic register, where parts of the PCU can be manipulated, such as disabling the transmission of ACKs, forcing a clear channel assessment, etc.

5.2. ath9k

The driver that implements the control of the AR9220 chipset is ath9k in the Linux kernel. This driver controls the wireless card by writing to its registers which are accessible through the mini-PCI bus. In the most simplified way, it could be said that the driver accepts packets, puts them in a linked list, places the address of the first element of that list in the appropriate QCU and signals the hardware that new packet information has been written. Data is transferred to a queue matching its QoS type in QCU 0 to 4, that are configured to send as soon as data is available. In reality, there is a lot of configuration and preparation to be done by the ath9k driver, such as allocating memory, manipulation of several linked lists, combining packets that should be aggregated by the hardware, calculating the background noise, storing physical parameters that are measured during transmission and reception for diagnostic purposes, manipulation of encryption keys, etc. The ath9k driver also accepts configuration values from the upper layers, where global configuration is performed through a kernel API, while per packet parameters can be set in the control buffer, which is part of the packet coming from the upper layer.

Besides sending and receiving data, ath9k is responsible for configuring the hardware and is required to capture any interrupts that are triggered by the hardware and handle them accordingly. Interrupts could signal the transmission or reception of a packet, accompanied with its error codes, but it could also signal other events. For example, in AP mode the hardware signals an interrupt at the expiry of the SWBA (SW Beacon Alert) timer, upon which the ath9k driver prepares the next beacon packet and places it into the beacon queue, which is *QCU9*. This queue is configured such that when a packet is ready to be sent, it blocks all lower priority queues from entering new packets. Since *QCU9* has the highest priority, all queues are blocked until the beacon packet has been sent. The beacon queue is timer triggered and waits for the occurrence of the DBA timer (DMA Beacon Alert), which is an internal timer of the AR9220 chipset, upon which the DCU will try to send the beacon.

5.3. mac80211

The mac80211 is a kernel module that provides standard IEEE802.11 MAC related functionality. It provides a configuration interface that enables network device configuration, such as manipulation of the IP address, changing the operational mode (Ad hoc, station, AP,...), configuring the WMM parameters (cwmin, cwmax, aifs, txop,...), etc. Software encryption is provided in case the hardware does not support hardware encryption, while otherwise only the encryption key management is used. The module also provides a rate control mechanism that can be used, unless the wireless card driver implements its own rate control and is configured as the preferred method during kernel compile time. A part of the Block Ack management is also handled by this kernel module, while the more specific part of Block Acks is handled by the respective driver. Management packet handlers, such as probe response handlers, authentication response handlers, association response handlers, etc., are provided for Ad hoc, mesh and station mode, however, there are no management handlers foreseen for Access Point mode. In other words, the mac80211 kernel module is a universal module which supports various functionalities of the IEEE802.11 MAC layer. The cfg80211 kernel module, which provides configuration functionality which is specific to IEEE802.11 wireless communication, interacts directly with the mac80211 module. Most of the known drivers use the mac80211 kernel module to offload the driver itself from this MAC IEEE802.11 specific operation, however, some drivers prefer to keep their own implementation of the IEEE802.11 MAC and instead link directly to the cfg80211 module.

5.4. hostapd

Since the mac80211 module does not provide any management handlers for the Access-Point, nor does any other kernel module, a userspace application is required, named hostapd. It provides methods to configure the network interface in infrastructure mode, provides management handlers for this mode, as well as encryption, association, authentication and timings. The program is used at the Access-Point side and configures the respective layers by means of netlink and ioctl calls, for our system, which are basically socket operations. Hostapd also provides other types of interfaces for operation in various environments, however, we do not elaborate on the other interfaces.

The management packets are received through a monitor interface, which is setup during initialization. The transmission

of management response packets is also performed through the same monitor interface. Hostapd keeps a strict control on the timing of incoming responses and discards any responses that are too late.

6. Reliable Tx methodology

6.1. Common issues and misconceptions

It is common in the related works that it is assumed possible to disable the CSMA/CA functionality of commodity hardware, the modus operandi is left to the imagination of the reader or they claim to be able to exert TDMA access mode by relying solely on the manipulation of QoS parameters. Unfortunately, the 802.11 commodity hardware is specifically designed for CSMA/CA operation and does not provide, until so far, a switch that could disable this mode and only manipulating the QoS parameters does not suffice. However, it is possible to reduce the influence of the standard behavior of the commodity hardware. The authors in [5] list some of the actions that need to be taken in order to be able to transmit reliably at certain intervals:

1. Eliminate automatic ACK and retransmission
2. Eliminate RTS/CTS exchange
3. Eliminate virtual carrier sense (NAV)
4. Control PHY Clear Channel Assessment (CCA)
5. Control transmission backoff

The previously mentioned authors resolve the first two action points by using the hardware in monitor mode, while the other actions are handled by register manipulation. Because the paper refers to 802.11b/g hardware, or because the authors work with monitor mode, there was no need yet to enumerate certain other actions that need to be taken when discussing 802.11n hardware or when working in access point mode. Some of these actions are the disabling of Block Acks, disabling of rate control, manipulating the beacon queue properties, etc. However, a most important issue is the need to disable receptions when no receptions are expected. This is an issue that none of the related work discusses, except in [11], where the authors noticed in their performance evaluations that the proposed protocol exhibits a decreased performance when collocated with other Wi-Fi devices, although it is performing as expected in an isolated environment. The issue was also detected in [7], where it was described as an anomaly, called 'hiccup', attributed to some unknown maintenance function of the hardware device. However, the problem is not related to some maintenance function, nor does it need to be related to frequent foreign data transmissions, the problem is already perceivable due to the multiple probe requests that are broadcast. Probe requests are low rate transmissions and therefore more lengthy transmissions, even though the actual packet length is smaller than most data packets. If our hardware is receiving such a message, all scheduled transmissions within the QCU or DCU have to wait upon the completion of the packet.

Some of the works, such as [10], provide extra diagnostic functionality, which is a valuable addition to their work. However, care should be taken to avoid any *printk* or any other string operations. On a SMP (Symmetric Multi-Processing) architecture, this might not be noticeable, however, on a single processor system, this would imply a blocking action of around 4ms, depending on the length of the string and the nature of the string manipulation.

6.2. Hardware controlled operation

One of the options that the Atheros AR9220 hardware provides is the usage of the QCU's scheduling policies, where by means of the time triggered scheduling policy packets can be sent out at a regular interval through the use of the DMA beacon alert (DBA) timer. The DBA timer is a timer with a precision of 128 μs and therefore has a sufficiently precise resolution for our purpose. The timer triggers the queue system such that available packets are marked ready to transmit and the transmission continues until the end of the queue is detected. In combination with a ReadyTime limit, which is an Atheros configuration parameter that limits the duration of activity, slots can be defined in which transmissions are allowed. Such method is employed by the authors of [24], where the detail of control is even extended towards the application layer such that by opening an interface in TDMA mode, all packets sent through that interface is using this access mode. Although the timer provides a means to slotted operation, there is no control regarding the slot allocations, neither is it possible to know where the slot boundaries are located. The method allows sending packets in a CSMA manner during specific time slots, which is fine for some applications. However, for some protocols, such control is too limited and therefore require an additional control mechanism that enqueues packets at regular intervals if management and data packets are to be sent in specific slots. This would require some extra timer source and thus redundant functionality.

6.3. Software controlled operation

Depending on the operational mode that the network device is working in, different modifications are in order to implement a controlled slotted transmission scheme. Since the targeted environment is infrastructure based, it is more convenient to modify the AP mode than any other mode, since it already provides a lot of functionality that is required, but which in some cases needs to be modified.

The goal is to control the transmissions as much as possible with an 802.11n based device. Since the DCF functionality is embedded in the wireless network device, it is required to provide a reliable time triggered packet transmission towards the wireless network device in the form of a leaky bucket. At the time of conceiving this implementation, Linux 2.6.38.2 is the most recent stable kernel, of which we have analyzed the possible timers superficially, that is, we did a comparison of the hrtimer and the SWBA timer on a regular PC with an SMP architecture for all possible configurations regarding preemptiveness or *HZ* value, however a detailed comparison, including with a custom timer functionality, on different (single CPU) platforms, with different improvements to achieve real-time etc. was done only for kernels 3.2 and 3.13.9, which is discussed in Section 3. Based upon those first impressions, we decided to use the hrtimer as a timer source.

Due to the integrated functionality of the network device, the jitter exhibited by the timer source will be augmented by the default network functionality of the device when transmitting. To elevate the effects of the functionality of the chipset, some of this functionality needs to be bridged by disabling certain features such as the automatic acknowledgement transmission, the carrier sensing, RTS/CTS handshaking, minimizing backoff before transmissions, etc. The disabling of power saving and Block Acks also enhances the precision of the packet transmission, by eliminating variable factors. Automatic rate control is also of course out of the question when predictability is required.

The keyword of the system is reliability. Therefore, management packets also should need to be sent in a controlled way. Most management handlers for AP mode are located in the hostapd user program. The management handlers provide the functionality for the association procedure. For our use case, which is to validate the manageability of a slotted transmission over commodity hardware, there are no stringent requirements on the association procedure, since it can be considered as the initialization phase of the system. A system beyond this use case needs to make sure that the stations first listen for the beacons sent by the AP, instead of immediately starting to send probe requests, since the stations should follow the slot allocations of the system. Since we are developing a new type of protocol, where such an extensive information exchange during the association is not required, the association procedure itself as well as the contents of the management packets can be simplified.

7. Reliable Tx modifications

It is already mentioned before that a controlled slotted transmission is not supported in the ath9k driver. Moreover, the Atheros AR9220 wireless network card does only support 802.11n medium contention access. Therefore, it is required to optimize the network card for a low latency transmission and perform most of the packet transmission triggering in software. Since most data is originating from the AP for our case study, the current validation procedure allows the AP to transmit in a slotted manner, while the STA simply listens. In a further stage, this needs to be extended to a full TDMA algorithm, where the stations also send their own data back. This implies that most changes need to be done at AP side and just a few at the station side.

Due to the stringent timing requirements that hostapd enforces, association with the modified slotted transmission ath9k driver is hard or even impossible in the 5GHz band under certain circumstances. Due to the slotted transmission, system delays and the manner in which hostapd handles notifications, the association procedure is able to complete successfully, however, due to a timeout notification a disassociation message is sent nevertheless by hostapd even after the association has been completed. Therefore, the first step into the direction of a custom association procedure is required. Moreover, due to the fact that hostapd is a userspace application, it needs to use a variety of interfaces towards the kernel in order to provide the required functionality. All these facts combined led to the decision to create a simplified kernel module that provides sufficient functionality such that it can replace hostapd. The next subsection clarifies the migration of the hostapd user program towards the OmusAp kernel module, while the following subsection goes into detail regarding the modifications to the ath9k driver.

7.1. hostapd

The standard messaging and interface structure of the kernel that hostapd needs is depicted at the top of Figure 12. Our simplified kernel module, the OmusAp kernel module, depicted at the bottom of the figure, initializes the network card into AP mode and can handle management requests. Neither encryption nor authentication is supported. The module provides a file system through which ioctl calls can be made from a user application. Through the supported ioctl calls, the AP can be started with the passed parameters and stopped. In our future work, we might consider changing this into a sysfs entry, through which the kernel module can be manipulated.

The OmusAp module interacts directly with the network device software structure and *cfg80211_registered_device*, thereby limiting the function call stack. By means of hooks, placed into the mac80211 module, not unlike the monitor mode hooks, the OmusAp module receives the management packets that were not handled by the mac80211 module, as well as transmission acknowledgements.

The management packets currently have the same format and functionality as the standard 802.11n management packets in order to be able to test with standard stations. In a later stage, the management format and the number of management packets can be modified to support a more simplistic association procedure.

7.2. ath9k

Currently, the ath9k driver places all data it receives from the upper layers into a queue, whose selection depends on the QoS parameters, after processing the metadata and creating the Tx descriptor. Afterwards, the hardware is triggered to notify that data is available to be sent. Our modifications make sure that all packets, coming from the upper layers, are processed and as much as possible preparation to send is performed, before placing it into a buffer. The packet is transferred to the transmission queue in the wireless network card only according to a certain interval, that is, according to the Leaky Bucket algorithm. Since the requirement is to have at least an effective bandwidth of 36Mbps, the hrtimer is scheduled to give an interrupt every 256 μs at which point a packet of 1500 bytes will be sent. This results in a total available bandwidth of 48Mbps.

The AR9220 supports QoS by providing several queues of which the parameters can be modified individually. While the original ath9k driver makes use of this feature, our modifications use only a single queue. Therefore, all transmissions exhibit the same transmission properties due to the medium access control. All control regarding QoS can be performed in software by first allocating slots for higher priority transmissions. However, QoS control is implementation dependent of the TDMA algorithm, which is considered future work.

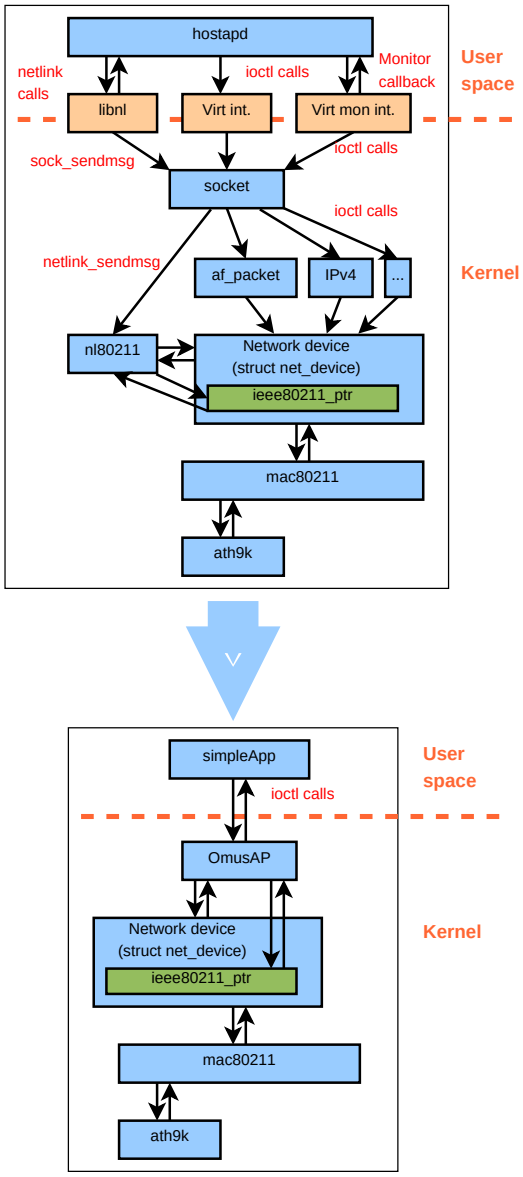

Figure 12: Migration from the hostapd application(top) to OmusAP kernel module(bottom)

In order to prevent the blocking of all data transmissions, due to the fact that beacons are sent in the highest priority queue which blocks all lower priority queues, beacons are scheduled to be transmitted in the same queue as data. Beacons are sent every 102.4ms, that is, every 400 timer interrupts a slot is allocated to send a beacon packet instead of data.

The association procedure is similar to the original one, except that the OmusAp kernel module is used instead of hostapd. However, the ath9k driver is ignorant of the association procedure, which implies that all management packets are considered as regular data packets and thus sent out according to the timer interrupts. This means that when sending data, management packets are interleaved with the data and occupy slots that would otherwise have been used for data packets. Since the current setup uses a single AP and a single station, all management packets are sent during the association procedure, which is the first phase, after which data can be sent during the second phase. In a later stage, this should be modified and management packets should be allocated in specific management slots and thus not interferes with the data transmissions.

In order to prevent the wireless network card being busy with receiving packets while a transmission should be scheduled, the reception of packets is turned off after the association procedure. In future work, this should be made more flexible and either a series of slots need to be considered as a transmission period where reception is not allowed, or the reception needs to be turned off (and cut off in case a reception is ongoing) each time a transmission needs to be scheduled.

In order to enhance the timing control, certain features of the 802.11n protocol are disabled, such as Block Acks, power saving capability, aggregation, etc. Those functionalities are disabled by removing the sections from the ath9k that enable such operation. While creating the Tx descriptor, certain per packet configuration values are overwritten. The rate selection is overwritten with a single fixed value, MCS5, since it is required to know how much time the radio will need to send a packet of a certain size. Even beacon packets are transmitted at MCS5, otherwise it would take longer than 256 μs to transmit a beacon at the basic rate. Of the range of possible transmission rates, only a single rate is specified, while retransmission with a lower rate is not allowed. Even more, retransmissions are disabled by manipulating the retransmit count. The RTS/CTS bit is set such that no RTS/CTS is required and the acknowledgement bit in the QoS data packet is set such that the station knows it does not need to send Acks.

All former modifications still do not ensure a timely transmission. For this, the internal registers of the AR9220 need to be modified. The PCU diagnostic register allows disabling the ACK transmission system, disabling the virtual carrier sensing and always assuming that the medium is clear for diagnostic reasons, which can be exploited. Making the hardware assume that the medium is always clear is crucial to send packets out in a controlled and timely manner. This also implies that packets could get lost due to collisions if there is a lot of traffic on the medium. Other parameters, such as the HT mode, the channel and the queue parameters (QoS parameters) are set by means of the configuration of OmusAp, such as the cwmin, cwmax, aifs and txop values and the channel and HT operation.

Since most changes occurred at the AP side, there are but a few modifications done at the STA side. Some features need to be disabled at the STA side however, because their capability for these features could influence the transmission pattern at the AP side. The functionality that needs to be disabled is amongst others, power saving capability, Block Acks and aggregation.

Since the reception procedure of the ath9k driver sets the timestamp to the system time at which the packet was processed in the packet, tcpdump is not able to see the actual time the packet was received. Therefore the actual mac timestamp at the reception is placed in the packet data in order to be able to analyze the packet arrival times, however, this is just for validation purposes and has no effect on the operation itself.

Deviation range	Number of deviations
[1, 2[	0
[2, 4[	1486019
[4, 6[	23279
[6, 8[	2408
[8, 10[	1046
[10, 20[	4901
[20, 30[	7010
[30, 40[	502
[40, 50[	40
[50, 100[	113
[100, 150[	0
[150, 200[	0
[200, 300[	1

Table 1: Deviations larger than 1 μs for the first test

7.3. Reliable Tx performance results

Our test setup consists of two regular pc's (AMD Sempron(TM) 2400+), both equipped with a mini-PCI wireless network card based upon the Atheros AR9220 chipset. One of the nodes is configured as an AP, while the other serves as station. As soon as the association phase is completed, we perform a tcpdump at the client side and start a program at the AP side which sends data as fast as possible as raw data, that is, the data is sent directly towards the MAC layer and does not contain any IP information, it is directed towards the MAC address of the station. The captured data is afterwards processed to extract the arrival times that are embedded at the head of the received data. All tests are performed with HT+, MCS 5 while sending 1500 bytes per packet every 256 μs.

Based upon the obtained results we can conclude that all experiments exhibit a similar behavior and therefore, we show only the three most interesting test cases. The first test is a test at channel 36 (5GHz band) and we captured data during an hour. In total we received 14027005 packets and missed 517 packets, which amounts to a loss of 0.003%. The average time between two packets is 255.995 μs. The minimum time between two packets is 194 μs and the maximum time between two packets is 535 μs. The standard deviation of the time between two packet arrivals is equal to 1.19356 μs and therefore 99.7% of the samples fall within the mean value +/- 3.6 μs. Of those that have a larger deviation than 4 μs, most of them exhibit a value lower than 30 μs. All deviations in text form can be found in Table 1. In total there are 12186 packets that exhibit a deviation which differs more than 10 μs compared to the average deviation, which amounts to 0.087%.

The second test we show, is a test with the same parameters as above, but with fewer samples, it ran only 60 seconds. Table 2 shows the obtained results in a textual manner. In total 233938 packets have been received, and only 17 packets have been missed, the average inter-arrival time is 255.995 μs, the minimum is 203 μs and the maximum is 307 μs. The standard deviation of the time between two packet arrivals is equal to 1.28414 μs and therefore 99.7% of the samples fall within the mean value +/- 3.85 μs. In total there are 137 packets that exhibit a deviation which differs more than 10 μs compared to the average deviation, which amounts to 0.059%

Deviation range	Number of deviations
[1, 2[	0
[2, 4[	39847
[4, 6[	1427
[6, 8[	22
[8, 10[	9
[10, 20[	134
[20, 30[	23
[30, 40[	2
[40, 50[	0
[50, 100[	2

Table 2: Deviations larger than 1 μs for the second test

The last test we would like to show is a test of only five seconds in the 2G4 band, where there is a lot of interference. We used channel 6 and HT, rate and packet size settings as above. The result is that 17019 packets have been received and 2632 packets have been lost, which amounts to a loss of 15.4%. The average inter-arrival time is 255.997 μs, while the minimum inter-arrival time is 228 μs and the maximum is 283 μs. The standard deviation is 1.16553 μs.

It can be noticed by the results that using the slotted transmission scheme in an environment with a lot of interference results in a vast amount of missed packets. However, the stability of the inter-arrival time of the packets is quite satisfactory, the standard deviation is around 1.2 μs, and hence 99.7% of all inter-arrival times are within a deviation of 3.6 μs. Even of the remaining 0.3%, most of them are positioned within a deviation of less than 30 μs. Since the transmission time of a data packet of 1500 bytes at MCS 5, HT+ is around 180 μs and the available slot time is 256 μs, this is still within acceptable boundaries.

8. Conclusion

From the timer analysis section, we can conclude that the hrtimer exhibits an impressive performance, even when multiple timers are running in parallel and stressing the interrupt system. Even a timer system that is dedicated to only some high priority timers shows a similar performance as the hrtimer. A heavy network load also stresses the performance of the hrtimer somewhat, however its influence is not that large as the timers running in parallel. When compared to the embedded timer, SWBA, of the Atheros hardware, the hrtimer is clearly more stable. Kernel configurations such as dynticks, *HZ*, or preemption do not influence the performance of the hrtimer in a significant way. It needs to be stressed however that a Linux system is not a hard real time system and there are no guarantees that a timer interrupt will occur in time.

Our second goal was to create a reliable slotted transmission mechanism based on commodity hardware. The hostapd application was transformed into a kernel module and the ath9k driver was adjusted such that transmissions occurred in a Leaky Bucket manner. A lot of the standard 802.11 operational functionality that was accessible was disabled or circumvented, such

as the disabling of Block Acks, power saving, ACKs, carrier sense, etc. The backoff window was minimized, all data and control packets use a single queue, the reception of packets was disabled after the association phase, etc. The performance of this system indicates that only 0.09% of the received packets exhibit a deviation larger than 10 μs from the expected arrival time. Even in the case where there is a lot of noise on the channel, the deviation stays bounded. However, there is no guarantee, since there are occasions where the data was received a whole slot too late.

In general, the results seem to be satisfying, a reliable TDMA system can be worked out on commodity hardware, however, care should be taken to construct a flexible protocol, where the slot boundaries are flexible and which is able to cope with data that is sent a slot too late. There are no hard guarantees that data will arrive on time.

Acknowledgment

This research is partly funded by the Flemish research institute iMinds through the OMUS project (grant number 10500-108). Companies and organizations involved in the project are Televic, Technicolor, Streamovations and Excentis, with project support of IWT. The research was also partly funded by the FWO project G016112N: "Stochastic modeling and performance analysis of MAC protocols for wireless networks

References

[1] J. Corbet, G. Kroah-Hartman, A. Rubini, Linux Device Drivers, 3rd edition, O'Reilly, 2005.
URL http://www.makelinux.net/ldd3/chp-7

[2] Real-Time Linux Wiki, accessed September 4, 2013.
URL https://rt.wiki.kernel.org/index.php/Main_Page

[3] eCos, accessed September, 2013.
URL http://ecos.sourceware.org/

[4] K.-D. Kwon, M. Sugaya, T. Nakajima, Ktas: Analysis of timer latency for embedded linux kernel, International Journal of Advanced Science and Technology 19 (2010) 59–70.

[5] M. Neufeld, J. Fifield, C. Doerr, A. Sheth, D. Grunwald, Softmacflexible wireless research platform, in: HotNets, 2005.

[6] P. Djukic, P. Mohaptra, Soft-tdmac: A software tdma-based mac over commodity 802.11 hardware, in: Proceedings of IEEE INFOCOM'09, Brazil, 2009.

[7] P. Djukic, P. Mohaptra, Soft-tdmac: A software-based 802.11 overlay tdma mac with microsecond synchronization, IEEE Transactions on Mobile Computing 11 (2012) 478–491.

[8] A. Sharma, M. Tiwari, H. Zheng, Madmac: Building a reconfigurable radio testbed using commodity 802.11 hardware, in: Proceedings of WSDR '06: First IEEE Workshop on Networking Technologies for Software Defined Radio Networks, USA, 2006.

[9] A. Sharma, E. M. Belding, Freemac: framework for multi-channel mac development on 802.11 hardware, in: Proceedings of the ACM workshop on Programmable routers for extensible services of tomorrow (ACM SIGCOMM PRESTO), 2008.

[10] Y. Ben-David, M. Vallentin, S. Fowler, E. Brewer, Jaldimac taking the distance further, in: Proceedings of the 4th ACM Workshop on Networked Systems for Developing Regions NSDR'10, USA, 2010.

[11] Y.-H. Weiy, Q. Lengy, S. Hany, A. K. Moky, W. Zhangz, M. Tomizuka, Rt-wifi: Real-time high-speed communication protocol for wireless cyber-physical control applications, in: Proceedings of IEEE Real-Time Systems Symposium (RTSS), Canada, 2013.

[12] G. Bianchi, P. Gallo, D. Garlisi, F. Giuliano, F. Gringoli, I. Tinnirello, Maclets: Active mac protocols over hard-coded devices, in: Proceedings of ACM CoNEXT 2012, France, 2012.

[13] R. Love, Linux Kernel Development, 2nd Edition, Sams Publishing, 2005.

[14] Linux man page about time, accessed August 15, 2013.
URL http://linux.die.net/man/7/time

[15] The impact of a tickless kernel, accessed April 8, 2014.
URL http://www.phoronix.com/vr.php?view=8963

[16] Kernel apis, part 3: Timers and lists in the 2.6 kernel, accessed September 3, 2013.
URL http://www.ibm.com/developerworks/linux/library

[17] Lklm.org ingo molnar on timer.c design, accessed Dec 9, 2013.
URL http://lkml.org/lkml/2005/10/19/46

[18] Lwn.net ingo molnar on timer.c design, accessed Dec 9, 2013.
URL http://lwn.net/Articles/156329/

[19] T. Gleixner, D. Niehaus, Hrtimers and beyond: transforming the linux time subsystems, in: Proceedings of Linux Symposium, Volume One, Canada, 2006.

[20] High resolution timers and dynamic ticks design notes, accessed April 8, 2014.
URL https://www.kernel.org/doc/Documentation/timers/highres.txt

[21] Xenomai, accessed December 8, 2013.
URL http://www.xenomai.org/

[22] RTAI, accessed December 18, 2013.
URL https://www.rtai.org/

[23] cyclictest rt wiki, accessed September 2, 2013.
URL https://rt.wiki.kernel.org/index.php/Cyclictest

[24] S. Leffler, Tdma for long distance wireless networks, in: Computer Laboratory Seminar, University of Cambridge, Cambridge, 2009.

Wim Torfs obtained his Master in Industrial Engineering Science from Group-T in Leuven (Belgium) in 2000. In 2000 he joined Philips Leuven (Belgium), where he was a hardware researcher between 2000 and 2005. He obtained his master degree in Computer Science in 2005 from the Free University of Brussels (VUB - Belgium). From 2006 till the current moment he is working in the PATS research group, which was recently renamed into the MOSAIC research group, at the University of Antwerp as a Ph.D. student and project collaborator. His main research interests include wireless network protocols, network architectures, wireless sensor, real-time systems and Operating Systems.

Chris Blondia obtained his Master in Science and Ph.D. in Mathematics, both from the University of Ghent (Belgium) in 1977 and 1982 respectively. In 1983 he joined Philips Belgium, where he was a researcher between 1986 and 1991 in the Philips Research Laboratory Belgium (PRLB) in the group Computer and Communication Systems. Between August 1991 and end 1994 he was an Associate Professor in the Computer Science Department of the University of Nijmegen (The Netherlands). In 1995 he joined the Department of Mathematics and Computer Science of the University of Antwerp (UA), where he is currently a Full Professor, head of the research group "Performance Analysis of Telecommunication Systems" (PATS -

www.pats.ua.ac.be), which was recently renamed into the MOSAIC research group, and head of the Department of Mathematics and Computer Science. He is lecturing Telecommunication and Performance Evaluation courses. He has also been lecturing at the University of Brussels. His main research interests are related to mathematical models for performance evaluation of computer and communication systems and the impact of the performance on the architecture of these systems. The systems that are studied are related to medium access control in both wired and wireless access networks, including satellite networks, MAC and routing in ad hoc networks and sensor networks. He has published over 250 papers in international journals and conferences on these research areas. He has been member of many program committees of international conferences and has been and is currently involved in many National and European Research Programs. He is editor of the Journal of Network and Computer Applications and of the International Journal of Electronics and Communications. The PATS research group, which was recently renamed into the MOSAIC research group, is member of the Flemish Research Institute iMinds (www.iminds.be).