Mälardalen University Doctoral Thesis
No.17

Adapting to Varying Demands in Resource Constrained Real-Time Devices

Tomas Lennvall

September 2005

MÄLARDALEN UNIVERSITY

Department of Computer Science and Electronics
Mälardalen University
Västerås, Sweden

ISSN 1651-4238
ISBN 91-88834-85-9
Printed by Arkitektkopia, Västerås, Sweden
Distribution: Mälardalen University Press

Abstract

In-home entertainment systems are becoming popular because they allow a variety of consumer electronic (CE) devices to be connected, using a wireless network, to form a system capable of handling multimedia content. Using such a system provide the end-users the possibility of transparently streaming multimedia content between devices of varying capabilities. This is possible because the system adapts the multimedia content to match the capabilities of the receiving device.

What looks simple to the end-user, is actually a very complex system that manages all existing multimedia streams and resources. It must manage all the varying resource demands, on all the constrained devices, in such a way that the resulting quality (video or audio playback) is acceptable to all the end-users of the system.

In this thesis we investigate two different, but still related, issues within the in-home entertainment network. First, we look at how we can adapt to the the capabilities of the nodes, which contains processors of varying capabilities and also operating systems which also provide different capabilities. Secondly, we have to adapt to the varying capabilities of the wireless network when it is used for video streaming in the presence of other network traffic.

For nodes, we present two scheduling methods that are extensions to the off-line scheduling paradigm.

The first method aims at improving the handling of soft aperiodic tasks in an off-line scheduled system, which are normally handled in the background resulting in long response times. The method creates space within an off-line schedule in order to allow a Total Bandwidth Server to use it during run-time in order to improve the response times.

The second method deals with overload caused by firm aperiodic tasks in an off-line scheduled system. The method deals with the overload by selecting which aperiodic tasks to execute, and which tasks to drop, without disturbing the execution of the more critical off-line scheduled tasks.

We also present a third scheduling related method that presents an plug-in based scheduling architecture with the purpose of allowing easy change of scheduling algorithm within operating systems.

In order to deal with the wireless network issues we present an architecture that decrease the network congestion in order to improve packet delivery reliability and decrease packet delays. In order to accomplish this, the architecture continuously predicts the available bandwidth, then uses this information to adapt the transmission rate of the node in order not to exceed what is available.

Dedicated to my family

Preface

First of all, thanks to my father, mother, and brother for encouraging me to take this path and continue until the end.

Thanks to my supervisor, Gerhard Fohler, for making this possible at all, helping me along the way, and for all the interesting research (gadget) discussions we have had during the years. I also wish to thank Professor Ivica Crnković for his co-supervision.

This has all been even more fun due to the people in the salsART research group, Radu, Damir, Larisa, and Pau. We have had a lot of interesting discussions ranging from research to cooking to movies to football to car repairs, all from which i have benefited.

Thanks Paolo, for taking care of me during my visits to Italy, "Peppe", Gerardo, Luigi, and Giorgio for taking care of me during my first visit to Pisa.

And, thanks to all my other colleagues at the department, who make it a fun and stimulating place to work in. Especially Jan Carlson, who collaborated with me at the beginning of my P.hD.

Finally, i want to thank Harriet, Monica, and Malin, who have all been a great help and pleasant company during my years here at the department.

This work has been supported by Volvo Research Foundation and the European Union (EU) Information Society Technologies (IST) project.

Tomas Lennvall
Västerås, September, 2005

Publications

I have authored or co-authored the following publications:

Book Chapters

- Gerhard Fohler, Tomas Lennvall, and Radu Dobrin: *A Component Based Real-Time Scheduling Architecture*, Architectures for Dependable Systems, editor(s): Rogerio de Lemos, Cristina Gacek, and Alexander Romanovsky, Springer Verlag, 2003.

Refereed Conferences and Workshops

- Gerhard Fohler, Tomas Lennvall: *Providing Adaptive QoS in Wireless Networks by Traffic Shaping*, Resource Management for Media Processing in Networked Embedded Systems (RM4NES), Eindhoven, The Netherlands, March, 2005.

- Jan Carlson, Tomas Lennvall, and Gerhard Fohler: *Enhancing Time Triggered Scheduling with Value Based Overload Handling and Task Migration*, 6th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, Hakodate, Japan, May, 2003.

- Tomas Lennvall, Gerhard Fohler, and Björn Lindberg: *Handling Aperiodic Tasks in Diverse Real-Time Systems via Plug-Ins*, 5th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, Washington D.C, USA, May, 2002.

- Gerhard Fohler, Damir Isovic, Tomas Lennvall, and Roger Vuolle: *SALSART - A Web Based Cooperative Environment for Offline Real-Time Schedule Design*, 10th Euromicro Workshop on Parallel, Distributed, and Network-based Processing (PDP), Gran Canaria, Spain, January 2002.
- Gerhard Fohler, Tomas Lennvall, and Giorgio Buttazzo: *Improved handling of Soft Aperiodic Tasks in Offline Scheduled real-Time Systems using Total Bandwidth Server*, 8th IEEE International Conference on Emerging Technologies & Factory Automation, Nice, France, October, 2001.
- Jan Carlson, Tomas Lennvall, Gerhard Fohler: *Value Based Overload Handling of Aperiodic Tasks in Offline Scheduled Real-Time Systems*, Work-in-progress Session, 13th Euromicro Conference on Real-Time Systems, Delft, The Netherlands, June, 2001.

Technical Reports

- Damir Isovic, Gerhard Fohler, and Tomas Lennvall: *Analysis of MPEG-2 Video Streams*, Mälardalen Real-Time Research Center (MRTC), Mälardalen University, August, 2002.
- Jan Carlson, Tomas Lennvall, Gerhard Fohler: *Simulation Results and Algorithm Details for Value based Overload Handling*, Mälardalen Real-Time Research Center (MRTC), Mälardalen University, May, 2002.
- Jan Carlson, Tomas Lennvall, Gerhard Fohler: *Value Based Overload Handling of Aperiodic Tasks in Distributed Offline Scheduled Real-Time Systems*, Mälardalen Real-Time Research Center (MRTC), Mälardalen University, May, 2001.
- Tomas Lennvall, Gerhard Fohler: *Integration of Off-line and Online Scheduling for Admission Control*, Mälardalen Real-Time Research Center (MRTC), Mälardalen University, September, 2000.

Licentiate Thesis

- Tomas Lennvall: *Handling Aperiodic Tasks and Overload in Distributed Off-line Scheduled Real-Time Systems*, Licentiate Thesis, Mälardalen University, May, 2003.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In-home entertainment systems are becoming more and more popular, because they provide the possibility to uniformly handle various consumer electronic (CE) devices with different capabilities. These kind of systems allow users to transparently stream multimedia content, i.e., video or audio, basically between any devices, without worrying about connection type or media format. The system will automatically adapt the multimedia stream according to the capabilities of the receiver device. Furthermore, most of the devices within the in-home entertainment system are connected using a wireless network. This permits the devices to be easily moved within the home and it also allows mobile devices to continuously receive streamed multimedia.

What looks simple from the user point of view is actually a very complex real-time system that transparently "handles" different resources with varying demands on real-time devices with very different capabilities.

Providing an architecture for such a real-time system requires a solution that is capable of efficiently handling varying resource demands in constrained real-time devices. Varying resource demands are, for example, task sets with varying utilization demands, or video/audio streams with variable bit rates. The problem is that these demands cannot always be met; which depends on the constrains of the devices, i.e. the processor power, operating system capabilities, and available bandwidth of the network. Thus, the architecture must adapt to the resource demands in order to match the constraints of the devices.

1.1 Real-Time Systems Background

Real-time systems are becoming more and more commonplace, and are used in applications ranging from cars, airplanes, and factory automation, to mobile phones, and multimedia systems. In [63] real-time systems are defined as systems where: *"not only the functional but also the timely correctness is important"*, i.e., not only the logical correctness of the computations performed are important, but also at the time these computations are performed.

In many cases, each computation has an associated *deadline*. If the computation does not complete before the deadline the system is considered to have failed, and depending on the category of the system, this can lead to serious consequences, i.e., for safety critical or hard real-time systems.

Real-time systems are usually divided into two categories: *hard* and *soft*. Hard real-time systems have stringent requirements on both functional and timely behavior. If either of these fail, the consequences can be catastrophic, such as damage to people or property. Soft real-time systems, on the other hand, are systems that can tolerate an occasional failure of the timing requirements without any severe consequences, e.g., only a possible degradation of performance or quality.

Real-time systems consist of applications and resources, i.e., where resources usually are one or several processors. Applications consist of tasks that cooperate to achieve the global goal of the application. To avoid contention and conflict between these tasks over the resources that exist in the system, i.e., the processors, a scheduling algorithm need to determine when to execute the tasks (in which order). There are two main scheduling algorithms paradigms, *off-line*, or *on-line* scheduling. Off-line scheduling takes place before run-time and provides predictability and support for general constraints at the cost of flexibility.

On-line scheduling, on the other hand, provides flexibility and dynamic run-time activities, but at the expense of less support for handling multiple constraints.

1.1.1 Constraints

Constraints originate from the demands of the application and impose requirements on the system. We define two classes of constraints as:

Simple can easily be handled by on-line scheduling algorithms. Examples of such constraints are periods, start-times, and deadlines.

Complex constraints, on the other hand, cause problems for on-line schedulers. However, some of them can be solved at the cost of a higher overhead. Here we give some examples of complex constraints:

Jitter causes the start or end of tasks to vary, which means that the interval between task instance invocations will vary. Some applications require the jitter between task instances to be constant or to have a small variation. In the extreme, periodic tasks can have invocations back-to-back or at the start of the first period and end of the second period.

Distribution cooperating tasks can execute on different nodes in a system, which can require synchronization. Many real-time systems are distributed by nature, requiring synchronization and communication between the parts. In order to provide determinism in such systems, the scheduler requires a global view of the whole distributed system.

Precedence means that a series of tasks must execute in a predefined order (also called a transaction). The basic example is the already mentioned sensor-actuator example, where the sensor measures some data, then computation take place, and finally data is sent to the actuator.

End-to-end deadline are deadlines for whole transactions of tasks, i.e., when the first task starts, a deadline is determined for the whole transaction of tasks. In the sensor-actuator example an end-to-end deadline exists for the whole transaction, from the sampling to the actuation.

1.1.2 On-Line and Off-line Scheduling

On-line scheduling provides flexibility for partially, or non specified, activities, i.e., for aperiodic and sporadic activities. Feasibility tests determine whether a given task set can be feasibly scheduled according to the rules of the particular algorithm applied. On-line scheduling allows to efficiently reclaim any spare time coming from early completions and allows to handle overload situations according to actual workload conditions. On-line schedulers are divided in two categories, dynamic priority schedulers, i.e., earliest deadline first (EDF), and fixed fixed priority schedulers (FPS) [46].

Off-line scheduling, also called *table driven scheduling*, is capable of constructing schedules for distributed applications with complex constraints, e.g., precedence, jitter, or end-to-end deadlines. The inclusion of additional constraints into an offline scheduler is typically straightforward, e.g., by including the constraints in a feasibility test applied during schedule construction. As a result, the off-line scheduler produces a table containing task execution positions. During run-time, the dispatcher simply performs a simple table lookup to execute tasks according to the schedule, resulting in a very simple run-time dispatching.

This approach has been shown to be suitable for critical hard real-time systems [39, 40]. By applying strict temporal control, critical activities can be performed in a deterministic way.

1.1.3 Event and Time Triggered Systems

Real-time systems are usually further classified as event or time-triggered, with respect to how the real-time activities are controlled.

In *event-triggered* systems, the activities happen in response to external events. The typical example of this is the sensor-actuator example: a sensor detects an external event and activates a task that reacts to this event (performs a computation), after which the task sends it's output to an actuator. This is an example of a system reacting and adjusting to an external event. One of the main issues with event-triggered systems is that external events can cause many tasks to be activated, thus, causing overload in the system, potentially leading to system failure. On-line scheduling is suitable for event triggered systems as it provides the ability to handle dynamic on-line events. SPRING [64] is an example of an event-triggered real-time operating system.

Time-triggered systems, on the other hand, require a priori knowledge about all activities. In distributed time-triggered systems, each node must have the same notion of time, implying that clock synchronization is needed. The main advantage of time-triggered systems is the predictable behavior they provide at the cost of low run-time flexibility. Time triggered systems are scheduled using off-line scheduling, which provides a time table containing task activation times, corresponding to the external events. Examples of a time-triggered real-time operating system are MARS [41] and TTP-OS [69] a time triggered operating system.

1.1.4 Overload

Overload is defined as a situation when there is not enough processor time available for the timely completion of all tasks, i.e., some tasks will miss their deadlines. Overload situations are usually sudden and transient, i.e., if a system reacts to a sudden event by activating many tasks. It is very hard to predict when the system will become overloaded.

Traditional on-line scheduling algorithms such as EDF or FPS behave poorly in overload situations, as shown in [47]. In the worst case they might even cause all tasks to miss their deadlines.

Off-line schedulers also handles overload poorly because of the lack of flexibility they provide. If possible overload situations have to be included as a consideration when creating an off-line schedule, the result, will in many cases, lead to in an over-constrained system. At the same time, because of the low flexibility, the number of allowed overload activities would usually be restricted over the system lifetime.

Since real-time system can also be distributed, it is possible that overload situations occur on a set of processing nodes although the system is globally underloaded. Such situations could be resolved by migrating tasks from overloaded nodes to underloaded nodes.

1.1.5 Network Scheduling

Real-Time communication can, just as real-time systems, be divided into hard or soft systems. Hard real-time systems consider a packet deadline miss as seriously as a task deadline miss, i.e the consequences can be catastrophic.

On the other hand, soft real-time systems may only experience a quality degradation as a result a late packet. Soft real-time communication is often based on general purpose communication networks, such as Ethernet [3]. We give more details on the issues of using Ethernet for real-time communication in section 2.3.

Furthermore, real-time communication can be classified into time triggered or event triggered, with the same distinction as for real-time systems. The same advantages and disadvantages as for real-time systems.

Controller Area Network

Controller Area Network (CAN) [29] is an example of an event triggered communication bus, where each message has a unique id that also represents the priority of that message. When a node wants to transmit a message, it starts by transmitting the id of the message, bit-by-bit. If several nodes want to transmit, they all start with the message id. A simple arbitration scheme determines which node that will be allowed to send its message. Basically the CAN bus works as a XOR gate, i.e., the node with most dominant bits will be allowed to send its message.

Time-Triggered Protocol

Time-Triggered Protocol (TTP) [42] is an example of a time-triggered communication bus that uses the Time Division Multiple Access (TDMA) algorithm. Each node is assigned at least one time slot where it can send its message. All these slots create a scheduling table that is repeated over and over again. There is no need for arbitration since the ordering of message transmissions are determined off-line.

Ethernet

The Ethernet [3] family is the most common network communication protocol in use today. It is widely used in environments ranging from industry, offices, and homes. The main advantages of Ethernet is: the low cost compared to the special protocols used for hard real-time communication, that it has been widely used and tested during a long period of time (decades), and that it provides very high data rates.

When using Ethernet for real-time communication, the problem is that it was not designed with real-time systems in mind. Ethernet was intended to be used for general purpose communication. Therefore, it has unbounded delays on message passing, no priority scheme, and no delivery guarantee. According to [10], the non-determinism of Ethernet increase if the load exceeds 30% of the total capacity.

Switched Ethernet is a simple way to achieve better real-time communication using Ethernet. The idea is that each node is connected to a full-duplex switch. The switch ensures that no collisions between packets can occur, thus allowing a reliable message passing.

1.2 Resource Constrained Real-Time Devices

1.2.1 Processor and Operating System Constraints

The nodes in our system contain both processors and operating systems, who are responsible for running the tasks that make up the applications. We give a short description of their constraints and limitations.

Desktop PCs are usually uniprocessor systems, but still with very powerful processors that are capable of a huge amount of computations per second. It is also possible to have multiple processors tightly connected within the same PC, a so called Symmetric Multi-Processing (SMP), which further increase the capability of the PC. Another possibility is to have a distributed system, where multiple processors are located on different PCs, but still cooperate when performing computations.

The processor is not solely responsible for the handling and running of tasks, the operating system (OS) is also a vital part.

The OS is responsible for handling most things during the life of a task. It manages when tasks are created, activated, deactivated, when they execute, and so on. It is within the OS that the task scheduling actually takes place, i.e., the decisions of in which order tasks will be executed on the processor.

Real-time OSs (RTOS) provide real-time scheduling algorithms for task scheduling, while general-purpose OSs (GPOS) provide algorithms that are fair, i.e., all tasks gets to run. Both types of OSs are viable to exist on different nodes within an in-home entertainment network.

Windows NT/2000 in Real-Time Systems

Real-time systems have somewhat stricter demands on tight and well specified deadlines. There are several reasons to why Windows NT/2000 (NT/2000) is not suitable for use in real-time systems "as is". Some of these reasons can be traced back to the fact that one of the goals with NT/2000 was to introduce a fairness in the system. NT/2000 is designed to prevent the potential starvation of low priority processes.

In [30, 52], thorough experimental investigations of the real-time capabilities of NT/2000 are made, and reasons for using NT/2000 in real-time systems are mentioned. One reason is that there are many competent programmers available that knows the NT/2000 programming environment. Contributing is also that the development environment around NT/2000 is quite large. There also exists a vast amount of inexpensive Commercial Off The Shelf (COTS) software for the Windows NT/2000 platform. If the underlying operating system can provide real-time support, it is possible that some of the COTS software can as-well.

1.2.2 Network Constraints

In in-home entertainment systems, nodes communicate using wireless networks in order to achieve more flexibility for the system. More specifically, we use the Ethernet standard network (IEEE 802.11). We give a short introduction to how it works, and present the problems that come with wireless Ethernet.

Wired Ethernet in Real-Time Systems

As mentioned in 1.1.5, Ethernet (IEEE 802.3) [3] is the most common and popular network communication protocol in use today.

The problem is that it was not designed with real-time systems in mind, thus lacking support for bounded message delays, message priorities, and delivery guarantees. In addition, the networks starts to become more and more non-deterministic as the load on the network increases.

Wireless Networks

Another, widely used variant of the IEEE 802 family is the wireless Ethernet (IEEE 802.11 [1]) protocol, which has gained popularity in the recent years. It is based on similar techniques as the wired version (IEEE 802.3), but there are also some significant differences.

A big advantage is that no wiring is necessary to connect devices, and it is even possible to have mobile devices roaming within the network which is a desirable feature in many cases. Depending on which part of the 802.11 standard (a,b,or g) that is used, where two different maximum communication speeds can be achieved, 11 or 54 Mbps.

Unfortunately, there are additional drawbacks to the ones from the wired variant (IEEE 802.3). The main problem is packet losses due to interference and disturbance of the radio signal. The IEEE 802.11 family of network protocols standards all use radio waves to transport the packets, they operate in different frequencies, 2.4 Ghz for 802.11b/g and 5 Ghz for the 802.11a standard.

The wireless network, since it uses radio waves, is subject to interference from various sources. Interference causes packets to disappear and makes it more problematic to handle the communication protocol.

There are many types of radio wave interference, here we shortly describe two of the most common.

Diffraction, Refraction, Reflection As described in [62], these three phenomena causes the radio wave to practically disturb itself. Diffraction occurs when the wave is split into multiple waves, Refraction occurs when the wave changes direction, and Reflection occurs when the wave bounces of an object (like a wall).

Unregulated frequency Another problem is that the operating frequency of the 802.11b/g standards is unregulated, which means that any device can use it. Bluetooth [14] and microwave owens both operate in the unregulated frequency causing interference if they operate during the same time as the the wireless network. Thus, packet losses increase and the performance of the network degrades. The same problems can also appear if many wireless networks that use the same frequency operate within the range of each other.

One advantage of the 802.11a standard is that it operates in a regulated frequency and, thus, is not subject to the same amount of interference. The drawback is that the higher frequency limits the range and penetration capability of the radio signal, making the standard less popular than the other two (b and g).

Other effects Walls, movement, and other physical effects also contribute as interference to the wireless network.

A well known issue with wireless communication is the packet overhead introduced in order to increase the reliability of the packet delivery. For each data packet transmitted, several "small" packets are also transmitted to ensure the delivery of the data packet. This has the effect that the actual efficient bandwidth available for transmitting data is significantly lower than the theoretical maximums (11 or 54 Mbps). Distance to the communication endpoint is also an important factor that decreases the actual speed. According to [72], typical peak throughput is more around $4-5$ Mbps for a 11 Mbps network, which our own measurements confirm. Typical data rates in home networks are more around $2-3$ Mbps. The same is true for a the 54 Mbps network, where speeds up to 50%, i.e 27 Mbps, might be reached.

1.3 Adapting to Varying Demands

1.3.1 Processor Scheduling Overview

In this section we present three processor scheduling solutions. We present two theoretical scheduling algorithms, both extensions to the slot shifting algorithm [27]. The first extension allows more efficient handling of soft aperiodic tasks, and the second extension allows overload situations to be handled.

The third solution proposes a plug-in based architecture to allow different scheduling algorithms to be incorporated into the operating system in a "plug-in" manner.

1.3.2 Related Work

Soft Aperiodic Task Handling

On-line scheduling is used to efficiently handle those activities that cannot be completely characterized off-line in terms of worst-case behavior, and, hence, cannot receive *a priori* guarantee. Examples of these activities include soft aperiodic tasks (e.g., multimedia tasks) whose computation time or, inter-arrival times, can have significant variation from instance to instance. Moreover, on-line scheduling is used to reclaim any spare time coming from early completion. A bandwidth reservation technique [20] is used to isolate the temporal behavior of the two schedules and prevent the event-driven tasks to corrupt the off-line plan.

The MARS system [41] is an example of a system with entire off-line planning of all activities. On the other side of the spectrum, SPRING [64] is using planning and global task migration [75] for handling a variety of constraints on-line. Its planning efforts are expensive; a dedicated scheduling chip is suggested. In our approach, the on-line scheduling is very simple as we only compute new deadlines.

The use of free resources in offline constructed schedules for aperiodic tasks has been discussed in [55]. The resulting flexibility is limited since aperiodic tasks are inserted into the idle times of the schedule only. Slot shifting [27] analyzes off-line schedules for unused resources and leeway, which is represented as execution intervals and spare capacities. This information is used at runtime to shift task executions, accommodate dynamic tasks, and to perform on-line guarantee tests. It provides increased flexibility, but focuses on hard and firm tasks only. In [34] the authors present a new and improved, with respect to complexity, method for on-line handling of aperiodic tasks.

From the on-line side, the integration of different scheduling paradigms in the same system requires a resource reservation mechanism to isolate the temporal behavior of each schedule. In [49], Mercer, Savage, and Tokuda propose a scheme based on processor capacity reserves, where a fraction of the CPU bandwidth is reserved to each task. This approach removes the need of knowing the worst-case computation time (WCET) of each task because it fixes the maximum time that each task can execute in each period. Since the periodic scheduler is based on the Rate Monotonic algorithm, the classical schedulability analysis can be applied to guarantee hard tasks, if any present.

In [22], Liu and Deng describe a two-level scheduling hierarchy which allows hard real-time, soft real-time, and non real-time applications to coexist in the same system. According to this approach, each application is handled by a dedicated server, which can be a Constant Utilization Server [23] for tasks that do not use non-preemptable sections or global resources, and a Total Bandwidth Server [61, 60] for the other tasks. At the lowest level, all jobs coming from the different applications are handled by the EDF scheduling algorithm. Although this solution can isolate the effects of overloads at the application level, the method requires the knowledge of the WCET even for soft and non real-time tasks.

The use of information about amount and distribution of unused resources for non periodic activities is similar to the basic idea of slack stealing [67], [21] which applies to fixed priority scheduling. Our method applies this basic idea in the context of offline and EDF scheduling. Chetto and Chetto [19] presented a method to analyze idle times of periodic tasks based on EDF. Our scheme analyzes offline schedules, which can be more general than strictly periodic tasks, e.g., for control applications.

Overload Handling

Value based overload handling has been thoroughly investigated. In [15], a number of methods that use values and deadlines to handle overload are compared. For a wide range of overload conditions, the best performance was achieved by EDF scheduling, extended with a value based overload recovery mechanism and resource reclaiming. An example of such an algorithm is RED [16]. For very high overloads, scheduling based on value density outperforms EDF based methods. In [6], task priorities are calculated dynamically from values and remaining execution times. They consider tasks with soft deadlines, i.e., values that decrease if the deadline is missed, rather than become zero or negative. In [11], an overload algorithm is presented for the special case when a minimum slack factor for every task is known. Also, tasks are assumed to be equally important.

These methods do not consider distributed scheduling, or overload handling in the presence of offline scheduled critical tasks.

Distributed overload handling is addressed in, e.g., [56], where an acceptance test is performed upon arrival of aperiodic tasks. If it fails, the node initiates an intricate bidding procedure in which nodes cooperate to decide where to migrate the task. The problem considered in this thesis requires an overload handling where values are taken into account. Another difference is that in our method migration is initiated by the receiving node rather than the current owner of the task, and that migration is integrated with resource reclaiming and the acceptance test of new aperiodic tasks.

Plug-in Scheduling

A number of aperiodic task handling methods have been presented [66, 67, 61], but within their respective packages only.

Instead of extending an existing scheduling package, we concentrate the functionality into a module, define the interface, and discuss its application to off-line and on-line scheduling methods as examples.

S.Ha.R.K [28] is an operating system where scheduling algorithms including aperiodic servers are created in a modular fashion. The interface between the system and the scheduler in S.Ha.R.K is more complex than the interface we propose in this paper.

Another operating system which provides the possibility of "plugging in" scheduling algorithms is the MaRTE OS [7]. MaRTE supports the *application-defined scheduling interface* proposed in [8] which is further generalized in [9]. Application-defined scheduling allows the application designer to construct his/her own scheduling algorithm and use it to schedule the task within the application. The application scheduler executes as a special thread thus allowing several different application schedulers to co-exist.

A problem with this approach is that since the scheduling algorithm acts as an extra layer, between the kernel scheduler and application, overhead will be introduced due to extra context switches occurring for each "normal" context switch.

1.3.3 Contribution

In this section we describe how the time triggered approach can be enhanced to suit distributed real-time systems where overload situations must be anticipated. We give a precise formulation of overload detection and value based task rejection in the presence of off-line scheduled tasks, and present a heuristic overload handling algorithm. Overload situations are detected immediately when the offending tasks arrive, and resolved by rejection of low value tasks.

The overload handling includes a task migration algorithm to benefit from the distributed setting, that integrates migration of rejected tasks with resource reclaiming and the acceptance test of newly arrived tasks.

We assume that the critical tasks are scheduled offline, but the schedule is handled in a flexible way at runtime to facilitate the inclusion of aperiodic tasks. This is achieved by including mechanisms from the slot shifting algorithm [27] that allow the planned execution of offline scheduled tasks to be shifted in time, while still ensuring that no critical constraints are violated.

This allows the designer to choose, for each activity individually, the trade-off between guaranteed timely execution, and less resource demanding non-guaranteed handling based on values.

Plug-in Scheduling

Scheduling algorithms have been typically developed around central paradigms, such as earliest deadline first (EDF) [46], rate monotonic (RM)[46], or off-line scheduling. Additional functionality, such as aperiodic task handling, guarantees, etc., is typically provided as extensions to a basic algorithm. Over time, scheduling packages evolved, providing a sets of functionality centered around a certain scheduling methodology.

EDF or FPS, for example, are chosen for simple dispatching and flexibility. Adding constraints, however, increases scheduling overhead [73] or requires new, specific schedulability tests which may have to be developed yet. Off-line scheduling methods can accommodate many specific constraints and include new ones by adding functions, but at the expense of runtime flexibility, in particular inability to handle aperiodic and sporadic tasks.

A similar approach dominates operating system functionality: implementation of the actual real-time scheduling algorithm, i.e., take the decisions which task to execute at which times to ensure deadlines are met, are intertwined with kernel routines such as task switching, dispatching, and bookkeeping to form a scheduling/dispatching module. Additional real-time scheduling functionality is added by including or “patching” this module. Replacement or addition of only parts is a tedious, error prone process.

Consequently, a designer given an application composed of mixed tasks and constraints has to choose which constraints to focus on in the selection of scheduling algorithm; others have to be accommodated as good as possible. Along with the choice of algorithm, operating system modules are chosen early on in the design process.

This contrasts actual industrial demands: designers want to select various types of functionality without consideration of which package they come from. They are reluctant to abandon trusted methods and to switch packages for the sake of an additional functional module only. Instead, there is a need to seamlessly integrate new functionality with a developed system, enabling designers to choose the best of various packages.

In this paper, we propose the use of a plug-in approach to add functionality to existing scheduling schemes and provide for easy replacement at the operating system level. In particular, we present an architecture to disentangle actual real-time scheduling from dispatching and other kernel routines with a small API, suited for a variety of scheduling schemes as plug-ins. We detail two plug-ins for aperiodic task handling and how they can extend two target systems, table-driven and EDF scheduling using the presented approach.

1.3.4 Packet Scheduling Overview

1.3.5 Related Work

Ethernet Communication

There are many attempts to modify Ethernet to make it more deterministic and useful in real-time communication. As mentioned above, we want a solution which uses the standard hardware and protocols, but for completeness we will discuss a few solutions that changes part of the standard.

In [48], the authors present a method to achieve hard real-time communication using standard Ethernet, using *Switched Ethernet*. The idea is to use switches (not hubs) and connect only one node to each port in the switch in order to avoid any contention, thus avoiding the use of the unpredictable contention resolution algorithm.

The switch is capable of full duplex transmission, i.e., sending and receiving simultaneously, thus allowing efficient communication. Due to the switch ensures there are no collisions. The only problem is that output queues can overflow if the input rate is to high, i.e., several nodes are transmitting to the same node with a combined rate higher than the switch's output rate. In this case packets will be dropped by the switch.

Another problem with switched Ethernet is that there are no priorities among packets, which is desirable for real-time communication. A solution to this problem is to use the IEEE 802.3p [4] protocol, which adds the possibility to prioritize packets.

In [59], the authors propose a change to the MAC layer, which will result in a non standard solution. The good thing with the proposed solution is that the modified MAC layer is only required on the nodes that want to use real-time communication. All other nodes can use the standard Ethernet protocol. The method relies on on the sensing and collision detection abilities of standard network cards, and it requires that the cards can send a jam signal (called black burst). Real-time traffic is prioritized using round-robin to determine which node is allowed to transmit.

Nodes transmitting real-time packets uses the normal Ethernet rules when transmitting the first packet, but for subsequent packets a special algorithm is used. The nodes schedule the next access attempt in the future. To seize the channel for the next packet transmission the node waits an amount of time before transmitting (Inter-frame spacing). If a collision is detected, the node immediately sends the jamming signal, i.e., the black burst, and continues doing so up to a predefined maximum time, or until no collision is detected. If several nodes transmit the black burst, the node with the longest burst gets access to the medium and immediately transmits its packet.

The length of the burst is is a direct function of the contention delay, measured from the time the access was scheduled to the time when the node perceives the medium to be idle, i.e., the node that has waited the longest time has the longest burst.

Bandwidth Estimation

Network bandwidth, especially the available bandwidth, is of major importance for QoS verification, streaming applications, and congestion control. Much research effort have therefore been dedicated to the problem of measuring bandwidth within the network communication area. However, there are few existing methods which are oriented toward the wireless environment. Although some adaptive bandwidth allocation schemes have been proposed, they all require some form of resource reservation, which is not included in the Ethernet standard. The adaptation of QoS is therefore strongly dependent on an end-to-end on-line measurement of the availability of bandwidth along the path from the sender to the receiver.

We give a short introduction to some methods that are based on probing the network path, which serve a basis for our own bandwidth prediction method, which is presented in chapter 4.

Probe Packet Probing is a method suited for determining end-to-end characteristics of a network path.

The basic idea is to probe the network by generating a traffic flow that is sent from one end node to the other. Information about the state and behavior of the network can then be extracted from the time delay of the probe packet.

The time delay of a probe packet across a single link consists of three components: the propagation delay, queuing delay, and transmission delay.

Propagation delay, also called latency, is due to the time it takes for the signal to traverse the link, i.e., the time it takes for each bit of the packet to traverse the link. It is independent of the packet size but depends on the geographical spread of the network, i.e., the distance between the nodes.

Queuing delay occurs inside nodes (such as routers) when there is contention at the input and output ports of these devices. All incoming packets are queued in an output buffer, and queuing delay is simply the waiting time for a packet before it gets services, i.e., transmitted, and occurs due to previous packets that have not yet been serviced. The amount of traffic passing through the node determines the delay, more traffic means longer delay.

Transmission delay is related to the underlying transmission hardware. It occurs due to the time it takes to physically transmit a packet onto the network.

One Packet Probing Technique The one packet probing technique, introduced in [12], is based on a deterministic model which is mathematically represented as shown in equation 1.1, where the path is assumed to consist of n links, the latency of link l is denoted by d_l , the bandwidth of link l is denoted by b_i , the size of probe packet k is s_k , the time when packet k has completely arrived on link l is denoted by t_l^k .

$$t_l^0 = t_0^0 + \sum_{i=0}^{l-1} \left(\frac{s^0}{b_i} + d_i \right) \tag{1.1}$$

In [12], the authors observe that in the absence of cross traffic, queuing delays can be assumed to be negligible; queuing caused by additional traffic can only increase the total delay time. This model also assume that the propagation delay remains constant for different packet sizes since the absolute signal frequency has a higher order relative to the size of the packet. Therefore, sending a large number of packets of different sizes ensures that the minimum value of their transmission times will approximate a line whose slope is the inverse of link bandwidth.

Packet Pair Probing Technique This technique, proposed in [38], is a well-known mechanism for measuring the capacity of a path. It was first introduced to measure the available bandwidth given in fair queuing networks.

A packet pair measurement consists of two packets of the same size L sent back-to-back from the source to the destination. Without any cross traffic in the path, the packet pair will reach the receiver with a time spacing, called dispersion, δ, that is equal to the transmission delay at the narrowest link in the end-to-end path.

The receiver can then estimate the capacity C of the bottleneck link from the measured dispersion δ, as $C = L/\delta$.

Similarly to the one-packet model, the packet pair technique can produce a wide variety of measurements and erroneous capacity estimates. The main reason is that cross traffic can distort the packet pair dispersion, leading to capacity underestimation or overestimation.

Pathload Pathload is another method proposed in [24]. This technique probes a network path by periodically sending a stream of UDP packets. The period is varied among different streams according to the probing information that is returned.

Pathload is based on the observation that, if there is an increasing trend in the one-way packet delays, the sending rate of a packet stream is higher than the available bandwidth. Otherwise, the sending rate is lower than the available bandwidth.

A third possibility, referred to as grey-region, is the case of ambiguous trend (the trend is neither increasing or decreasing). The grey-region is due to the fact that the available bandwidth varies around the sending rate during a probing stream.

As a result of the bandwidth estimation, a range containing the lower and upper bounds for the available bandwidth is reported to the sender. This approach has the advantage that it detects one-way delays, thus it does not require a symmetric network structure and the estimation can be more accurate for some single way communication. Another interesting aspect of Pathload is that it watches for the presence of the overall increasing trend during the entire stream instead of instantaneous performance measurement.

Pathload is based on an iterative algorithm to analyze the delay variation at the receiver side. The measurement latency is large since it needs a large quantity of the delay samples to produce a final estimation. The overhead is also relative high, which is not desirable for mobile devices such as pocket PCs and other handheld devices.

Traffic Shaping

In [44] and [43], the authors propose a method to achieve better results using Ethernet for real-time communication with respect to round-trip time and packet loss ratio.

The basic idea of applying traffic smoothing for network transmission is to smooth a bursty stream of data into a constant stream of data by using the leaky bucket algorithm. The rationale is that, by smoothing out bursts, the transmission is evenly spread out over time in order to reduce the probability of congestion and collisions on the network. It is also a way to control the rate of the transmitted traffic generated by each node.

The traffic smoother architecture resides on all nodes, and is inserted between the UDP, TCP/IP layers and the MAC layer, where it intercepts all outgoing IP packets. Traffic is divided into real-time or non real-time traffic. The idea is to only smooth non real-time traffic, and let the real-time traffic pass the smoother without interference, to eliminates contention within the node. By smoothing the non real-time traffic, collisions between real-time and non real-time packets are reduced.

The traffic smoother uses a credit bucket to regulate the burstiness of the non real-time traffic stream. The transmission rate for non real-time traffic is dynamically adjusted depending on the current network utilization. Adjustment is carried out by changing the refresh period of the credit bucket, i.e., how often new credits are added to the bucket. The transmission rates for all nodes are adjusted using a mechanism similar to the "slow start increase and multiplicative decrease" of TCP/IP congestion avoidance mechanism, called harmonic increase multiplicative decrease (HIMD). Periodically, the transmission rates of all nodes are harmonically increased (by decreasing the refresh period) until a collision occurs. When a collision is detected, the remaining credits are deleted from the bucket, and the refresh period is doubled.

In [18] propose an improvement to the above described traffic smoothing method. First, in order to have a better picture of the current network utilization, both throughput and collisions are monitored. The smoothing is dynamically adjusted using a fuzzy controller instead of the HIMD approach described above.

The fuzzy controller takes the number of collisions and throughput as input, and produces an output that determines the refresh period adjustment. One difference with the previous approach is that the adjustment of the refresh period is dynamic instead of static. Fuzzy control is used because of the non-linearity and complex behavior of the systems, which is difficult to model for a traditional controller.

The authors show that this approach gives shorter round trip times (RTT) and a better throughput than the previous work.

Wi-Fi Multimedia (WMM)

The Wi-Fi Alliance [70] started interoperability certification for WMM as a profile of the upcoming IEEE 802.11e QoS extension [2] for 802.11 networks in 2000.

WMM defines four access categories, voice, video, best effort, and background to be used for prioritizing of traffic. Legacy devices, i.e., without WMM support, are supported and such traffic is transmitted using the best effort priority.

Clients can get the permission to transmit a burst of data. They can actually transmit data for a certain amount of time, based on the access class of the traffic. A higher priority access class gets a longer time interval in which it may transmit packets. The length of the time frame, ranging from 0.2ms to 6ms, depends on the speed of the wireless network, i.e., 11 or 54 Mbps.

In the future, WMM will also support a feature called *scheduled access*, where applications are allowed to reserve network resources based on their traffic characteristics through requests sent to the AP.

1.3.6 Contribution

The traffic smoothing solutions presented in 1.3.5 are based on the assumption that you can detect collisions in the network, which is possible if wired Ethernet is used. But, the in-home entertainment systems will use wireless Ethernet, which, as mentioned earlier, uses a collision avoidance (CSMA/CA) scheme instead of collision detection (CSMA/CD). This is due to difficulties in detecting collisions on the wireless medium, that makes the traffic smoothing solutions not applicable in our scenario.

Instead, we need other methods to measure the currently available bandwidth of the network. Furthermore, because of the variation of the available bandwidth of the wireless network (it is not varying in wired networks), we need to continuously adapt the transmission rate according to these variations.

To accomplish this we try to come up with the answer to a simple question:

"is it possible to continue transmitting at the current rate?".

To determine the answer to this question we need to predict the available bandwidth and compare it to the transmission rate of the node. If the answer is "yes" we can continue, otherwise we have to adapt the transmission rate to the amount of bandwidth that will be available according to the prediction.

The purpose of the transmission rate adaptation is to avoid overloading the network with packets which will cause congestion. Congestion is bad since packets will collide triggering retransmission algorithms, in turn causing longer delays for the packets. In the worst case, congestion can also cause packets to be dropped, causing the network QoS to become poor.

We propose an architecture that provides network QoS by continuously adapting the transmission rate of nodes in order to match the currently available bandwidth of the wireless network.

The architecture consists of a *bandwidth predictor*, that first uses a simple probe-packet technique to measure the available bandwidth of the network. Then, prediction of the future available bandwidth uses both the probe packet measurement and the history of previous predictions in order to come with the current prediction. The predicted bandwidth is then fed into the *traffic shaping* part of the architecture, which adjust the transmission rate of the node accordingly.

It also prioritizes video streams traffic over other traffic when transmitting packets using a low level traffic shaper setup.

Bandwidth Prediction

Our bandwidth prediction uses the well known packet-pair probing technique presented in [38]. Shortly, the sender transmits two probe packets of identical size, back-to-back, to the receiver. The receiver measures the delay between the two packets, and returns this information to the sender.

This information indicates whether the network load is high or low, i.e., a long delay indicates high load and vice versa.

To make a more accurate prediction of the available bandwidth we also include the history of previous predictions in the current prediction. The reason for including the history in the calculation of the current prediction is to not react to "strongly" to temporary spikes. With the history of prediction we include the previous trend of the network bandwidth.

We have to repeatedly perform the bandwidth prediction in order to catch the behavior of the varying bandwidth of the wireless network. We perform measurements to determine the sampling frequency we need to use in order to properly predict the bandwidth. What we conclude from the measurements, using different sampling periods of 0.2, 0.5, 1.0, and 2.0 seconds, is that the prediction result is unaffected by the choice of period.

Traffic Shaping

The *Traffic Shaper* shapes the outbound traffic according to the portion of available bandwidth assigned to the node and the bandwidth prediction result.

We perform traffic shaping at two different levels: at the low level, where we shape all outgoing IP-packets, and at the application level, where the application itself can adapt the transmission rate.

At the low level, we have the possibility of controlling the transmission rate at a bit level, giving us a fine granularity control. Furthermore, this level allows us to prioritize video stream packets over other packets by installing a complex traffic shaping architecture, using network QoS features built into the Linux kernel. The architecture allows us to give the video stream a higher share of the rate assigned to the node, but, without starving the other traffic that also needs to be transmitted.

On the application level we have a coarse granularity control an only perform major transmission rate changes. If the application does not react we risk internal buffers to overflow causing packet to be dropped even before they are transmitted.

1.4 Outline of the Thesis

The rest of the thesis is organized as follows way: in chapter 2 we present what we consider to be resource constrained real-time devices within an in-home entertainment system. Chapter 3 contains the processor and operating system scheduling solutions, two theoretical algorithms and a plug-in scheduling architectural solution. In chapter 4 we present our solution for the wireless network issues presented in chapter 2 which is two part: we present our bandwidth prediction and traffic shaping methods.

In Appendix A, we present implementation details and simulation results from the solutions we presented in chapters 3 and 4.

Chapter 2

Resource Constrained Real-Time Devices

2.1 Overview

We look at two types of resources within the in-home entertainment system, namely the nodes, containing processors and operating systems, and the network.

In this chapter we describe the resources in more detail and we also present problems and issues related to each resource.

Nodes are varying devices ranging from full blown PCs, with powerful processors and lots of memory, to handheld devices, with more restricted processors and memory. The processors have different power (speed), that limits the amount of work than efficiently can be performed on them. Since it is not possible to simultaneously execute all existing threads, scheduling of the application threads (tasks) to efficiently utilize the processors is required.

In this chapter we look at both wired and wireless Ethernet. We discuss their general properties, and give a more specific description the problems that may be encountered when using wired and wireless Ethernet for real-time communication.

2.2 Processors and Operating Systems

Desktop PCs are usually uniprocessor systems, but still with very powerful processors that are capable of a huge amount of computations per second. It is also possible to have multiple processors tightly connected within the same PC, a so called Symmetric MultiProcessing (SMP), which further increase the capability of the PC. Another possibility is to have a distributed system, where multiple processors exists on different PCs, but still cooperate when performing computations. In this thesis we consider PCs with uniprocessor architectures.

The processor is not solely responsible for the handling and running of tasks: the operating system (OS) is also a vital part. The OS is responsible for managing the life of all tasks, i.e., task creation, activation, deactivation, when they should execute, and so on. Task scheduling takes place within the OS. Real-time OSs (RTOS) provide real-time scheduling algorithms for task scheduling, while general-purpose OSs (GPOS) provide algorithms that are fair, i.e., all tasks are scheduled to get share of the processor for execution. Both types of OSs are viable to exist on different nodes within an in-home entertainment network.

As mentioned in 1.3.3, one problem with RTOSs is that they usually only support one specific scheduling algorithm (or paradigm), for example FPS. All applications has to be tailored to fit the scheduling paradigm present within the OS, even if using another paradigm would be more suitable. Since most RTOSs also have a monolithic design, it is not trivial to change or replace the scheduling algorithm because of the tight coupling between the scheduler and the rest of the OS kernel.

GPOS, on the other hand, do not at all, or has very limited, support for real-time scheduling. Most of them use *Round-robin* algorithms, where each task is given a "slice" of time for execution, in a "round-robin fashion". GPOSs suffer from the same problem as RTOSs, it is not easy to separate the scheduling parts from the rest of the OS kernel if there is a desire to change the scheduling algorithm.

The theoretical scheduling algorithms we present in chapter 3 are all real-time algorithms, aimed at being used in RTOSs. Furthermore, we propose an architecture to disentangle the tight coupling between the scheduling and OS kernel routines, which will allow any scheduling algorithm to be easily added to the OS.

2.2.1 Windows NT/2000 in Real-Time Systems

Real-time systems have somewhat stricter demands on tight and well specified deadlines. There are several reasons to why Windows NT/2000 (NT/2000) is not suitable for use in real-time systems "as is". Some of these reasons can be traced back to the fact that one of the goals with NT/2000 was to introduce a fairness in the system. NT/2000 is designed to prevent the potential starvation of low priority processes.

There are also other reasons for not using NT/2000 in real-time systems. The fairness policy makes the operating system highly responsive, but it does so at the expense of determinism.

Deficiencies

It is not really feasible to incorporate NT/2000 into a real-time system without some adjustments and limitations. The issues that need adjusting according to [30, 52], are described below. The deficiencies of NT/2000 in the context of real-time systems are more observable under higher system load, when the system is stressed.

There is no priority tracking scheme in the interrupt handling [52]. Priority tracking is a scheme that ensures that all parts of a thread executes at the same priority. For example, in case of an interrupt, the interrupt will run with the same priority as the thread. Thus the interrupt will not affect the execution of any of the other threads in the system.

In NT/2000, the Interrupt Service Routine (ISR) is always executed at the highest priority. The execution of the ISR releases a Deferred Procedure Call (DPC). DPCs are handled in FIFO order and run on a priority higher than any user thread. This can cause priority inversion, and is, therefore, not compliant with priority inheritance.

Priority inversion is the name of the phenomenon when a lower priority thread blocks the execution of a higher priority thread by locking a shared resource for a unbounded time period. Because all ISRs run on higher priority than the DPCs, even lower priority interrupts, for example interrupts caused by mouse movement, affect the execution of all DPCs.

As mentioned earlier, scheduling is a central part in real-time systems. The scheduling scheme must allow an analysis that can determine whether the system will keep its timely restrictions or not. The scheduling scheme of NT/2000 is based on priority classes (with seven priority levels in each class).

There are four classes (presented in ascending priority order): IDLE, NORMAL, HIGH, and REALTIME. Threads that belong to the three lower classes are incorporated into the previously mentioned fairness policy, their priorities are raised if they get too little execution time.

Processes running on a priority in the REALTIME class is not included in that policy. There is no straightforward way to make an analysis when the scheduling is affected by the fairness policy, because lower priority threads can affect higher priority threads in unexpected ways. Thus, real-time systems should use the REALTIME class. Unfortunately, there are only seven priorities within that class and many applications require quite a lot more.

An additional drawback is that all ISRs and DPCs run on priorities higher than the REALTIME class, which may cause REALTIME threads to be blocked by non time-critical interrupts. In order to remedy the fact that most conventional uses of PCs require more RAM than is actually available in the system, Windows NT, and most other general purpose operating systems use page swapping. In cooperation with a virtual memory scheme and a secondary storage medium (hard-drive), page swapping allows the system to allocate more RAM than actually exists in the system. This is very useful and practically a must for general purpose operating systems. Unfortunately, it introduces unquantifiable delays, a property that is unacceptable in real-time systems. Threads running on kernel level have permission to disable interrupts, or raise the Interrupt ReQuest Level (IRQL). Thus, for unknown periods of time that the application has no control over (or even knowledge of), interrupts can be disabled. Also, the mapping of the IRQL is performed dynamically at system startup, and may differ between hardware architectures.

2.2.2 Windows CE

As stated in [71], Windows CE is designed as a general-purpose operating system for small devices, typically diskless systems with limited memory capacity, such as pocket PCs. The Hardware Abstraction Layer (HAL) is a thin layer of code that is adapted to specific hardware platforms.

Unlike other Windows operating systems, Windows CE does not represent one standard or identical piece of software that is common to all platforms. To support the varying need of a wide range of products, Windows CE is modular, i.e., it can be custom built for a product by selecting from a set of provided software modules.

Windows CE offers a subset of the of the same programming interfaces used to develop applications for other Windows operating systems.

Real-Time Support in Windows CE

Windows CE is a preemptive multitasking operating systems, and supports a maximum of 32 processes to run simultaneously. The number of threads a process can contain is limited only by available system resources.

Windows CE uses a priority-based time-sliced algorithm to schedule the execution of threads, and supports eight discrete priority levels. Level 0 and 1, the highest priorities, are intended for real-time applications and device drivers. Levels $2-4$ are intended to be used for kernel threads and normal applications, while $5-7$ should be used by applications that can always be preempted by other applications. Threads with higher priority are always scheduled to execute before threads with lower priority, and threads of the same priority run in a round-robin fashion where each thread gets a time slice for execution.

Unlike other Windows operating systems, Windows CE thread priorities are fixed and do not change. To avoid priority inversion, Windows CE allows the lower priority thread to inherit the more critical threads priority and run at the higher priority until it releases its use of the resource.

Windows CE offers a different set of "wait objects" for thread synchronization. These include critical section, event , and mutex objects, which allow a thread to block its own execution and wait until the specified object changes. The requests to the synchronization objects are processed in "FIFO-by-priority" order: a different FIFO queue is defined for each of the discrete priority levels. A request from a thread is placed at the end of that priority level queue, and the scheduler adjusts these queues when priority inversion occurs.

Windows CE splits interrupt processing into two steps: an interrupt service routine (ISR) and an interrupt service thread (IST). Each hardware interrupt request line (IRQ) is associated with one ISR, so, when an interrupt occurs, the kernel calls the registered ISR. Since the ISR is the kernel part of the interrupt processing, with the primary responsibility of directing the kernel to launch the appropriate IST, it is kept as short as possible. The ISTs are waiting for specific events that are generated by the kernel whit directions from the ISR. When this event occurs, the IST starts its execution, where it performs additional interrupt processing. ISTs run at the two highest priority levels, 0 or 1, ensuring that these threads run as quickly as possible.

2.2.3 Methods to Give GPOS Real-Time Support

There are several possible approaches on how to improve the real-time capabilities of a general purpose operating system.

There are also other approaches on how to improve real-time performance of GPOS [30], as described below: If the worst-case behavior of a system can be decided beforehand, it may be possible to use the environment "as is". The requirements should either be very modest real-time requirements, or every system property should be known beforehand. Then it could be possible to dimension the system in such a way that there are always sufficient resources available. It is also possible to make modifications to the kernel of the operating system. However, it seems that there are some issues that makes this approach questionable [53] . It is also a matter of following the future development of the platform.

Another possibility is to have two machines running two separate OSs, one with the GPOS and one with a RTOS. If a RTOS offered the same API as the GPOS, there is no need to incorporate the GPOS into the real-time system.

2.3 Networks

According to [5, 43], Ethernet (IEEE802.3) is one of the most successful local area networks ever implemented. It is widely used in real-life networks both in office and in industry applications.

Ethernet is based on a scheme called *Carrier Sense Multiple Access with Collision Detection* (CSMA/CD). The scheme allows all nodes to make one attempt to send a packet, as long as the network is not busy. If the first transmission attempt fails (because of a packet collision), the nodes involved in the collision will detect the collision and wait an arbitrary time before making a new sending attempt. The algorithm in use is called *Binary Exponential Backoff* (BEB), and it is used to resolve failed transmission attempts by randomly generating the time for the next transmission attempt.

This scheme yields very good performance in general purpose networks, but the CSMA/CD scheme is one of the reasons for Ethernets inability to give real-time guarantees. A node could potentially be blocked for an unlimited time if packet collisions occurs at inappropriate times. This non-prioritized competition for packet transmission and randomly generated times are undesirable in real-time systems.

2.3.1 Wireless Network Issues

Due to technical difficulties in detecting collisions on the wireless medium [1], a *Collision Avoidance* scheme is used instead of collision detection. Hence it belongs to the group of protocols that uses the *CSMA/CA* medium sharing mechanism.

The Collision Avoidance scheme is designed to reduce the probability of collisions when they are most likely to occur [37]. Because of the Carrier Sense protocol, most collisions occur when two nodes simultaneously makes attempts to access the medium. Simultaneous access to the medium is most likely to occur right after a busy medium becomes free, because there may be several nodes that have sensed the busy state, and are waiting to transmit. As a protection from these potential collisions, the Collision Avoidance scheme implements a random back-off agreement that decreases the probability of collisions in these situations. In resemblance with wired Ethernet, the CSMA/CA medium access protocol uses a random back-off time in the case where the medium is identified as busy.

However it is often not possible to determine whether a radio channel is busy or not. One of the reasons for this is that one node may be receiving information from second node that is out of range with respect to a third node that wants to know whether the channel to the second node is busy or not. This is referred to as the natural hidden terminal problem [74].

The standard that specifies the IEEE 802.11 protocol [1], defines the physical and medium access control layers. Wireless Ethernet communication is carried out over radio or infrared, and the *Medium Access Control* (MAC) supports communication between independent mobile nodes as well as communication via access points. Networks without access points are referred to as *ad-hoc* networks [1]. Such networks are often created spontaneously and no additional resources than the devices participating in the network are required. Routing in ad-hoc networks

[1]Radio modems are half-duplex, thus they cannot listen while they transmit. Even if they could, the relation between transmitted and received signal strength makes listening non-trivial.

is more complicated due to the limited temporal and spatial extent of the network. Important differences to wired media exist in the physical layer specified in the IEEE 802.11 standard [1] for Wireless Ethernet. There are no well-defined coverage areas, i.e., no absolute or observable bounds where stations are known to be unable to receive frames.

The following list shows some of the problems with wireless Ethernet:

- The physical layer is unprotected from interference.
- The media used for communication is significantly less reliable.
- The network topology is dynamic and may change over time.
- There is no full connectivity. Stations may be hidden from each other and it cannot be assumed that every station can listen to the other stations participating in the network.
- The propagation properties are varying over time and are asymmetric.

Yet another difference between wired communication and Wireless Ethernet is that the assumption that an address is the same as a destination (valid for wired communication) is invalid in Wireless Ethernet [1]. When using Wireless Ethernet the destination of a message is a station (abbreviated STA in the IEEE standard) but not necessarily a particular location. The above properties makes Wireless Ethernet less suitable for real-time systems, where timeliness and determinism are essential properties.

Point Coordination Function

Within the MAC layer there are two possible ways for medium access control, the *Distributed Coordination Function* (DCF) and the *Point Coordination Function*.

DCF is the "normal" way of accessing the medium, i.e., where all nodes try to access the medium using the CSMA mechanism.

PCF is an optional method where access control is handled in a centralized way, i.e., one node acts as a master. The master is called point coordinator and is responsible for polling all other nodes to control when they can transmit data. In this way the point coordinator can be used to control when time sensitive data will be sent. In order to guarantee the point coordinator access to the medium, it uses a shorter *Interframe Space* (IFS), compared to what is normally used, before transmitting its polling packet. Normally, a node uses an *Distributed Coordination Function IFS* (DIFS) as delay before contending for access. The point coordinator uses a *Point Coordination Function IFS* (PIFS) which is shorter in order to be sure that it gets access.

In order to prevent the point coordinator of monopolizing the network, a time interval known as *superframe* was introduced. Within a superframe, there are two time intervals: first an interval where PCF is used, this is where the time critical messages can be sent. The second interval is used for the "normal" DCF, allowing nodes to compete for access to the medium.

2.3.2 Network Congestion

Network congestion is a problem when using Ethernet networks. Network congestion occurs when the network is over loaded with traffic which leads to packet collisions and packet drops within routers. Collisions occur because sever nodes try to transmit packets at a high rate and at the same time, and in the end it is difficult to successfully transmit a packet. Routers drop packets if their internal queues are full further increasing the difficulty of successful packet delivery.

Network congestion causes packets to collide because many nodes are trying to transmit at the same time. These collisions will cause retransmissions, as nodes will repeatedly try to transmit their data. As an end result there will be no reliable network communication, the probability of getting a packet "through" is low.

In addition, packets can be dropped by the switches or access points (AP). Switches, or APs, drop packets if the internal queues become overloaded with packets. In the case of constant retransmissions and bursty traffic there is a high probability of packet overflow. If a 10 Mbit/s switch simultaneously receives 10 Mbit/s of traffic from several different nodes, with the same destination node, that outgoing port (within the switch) will become overflowed, causing packet drops.

In order to avoid, or minimize, congestion, the network traffic must be controlled. Nodes can not be allowed to transmit any amount of data at any time. Instead the packet transmission from the nodes must be strictly controlled.

2.4 Chapter Summary

In this chapter we presented two types of constrained real-time devices within in-home entertainment system; the nodes, containing processors and operating systems, and the network.

Processors comes in a variety of classes, from powerful desktop PC processors to less powerful processors used in handheld computers. And operating systems (OS), which run on the processors and are responsible for managing the life-cycle of the tasks, i.e., including the scheduling of tasks.

OSs can be divided into two classes; real-time operating systems (RTOS) and general-purpose operating systems (GPOS), both with different goals in mind. A RTOS is intended to be used in real-time systems, where timeliness is as important as correct functionality. GPOSs, on the other hand, are more focused on providing correct functionality and a fair share of the processor time to all tasks (threads) running in the system. Using GPOSs in real-time systems is problematic since they were not designed with real-time as a goal, typical problems come from lack of control of all resources and tasks leading to problems with blocking times, priority inversion, and so on. A common problem with both types of OSs is that they usually are quite monolithic, meaning that the scheduling algorithm is tightly coupled with the OS kernel, making it problematic to change algorithm within the OS.

The second resource we presented was both wired and wireless Ethernet. Using Ethernet for real-time communication is problematic because it cannot give any real-time guarantees. Wireless Ethernet is similar to the wired variant, but in addition has the problem of sensitivity to interference. For both types, wired and wireless, network congestion is a big problem.

Chapter 3

Processor Scheduling

3.1 Overview

In this chapter we look at our task scheduling solutions. We present two theoretical scheduling algorithms, both which are extensions to the slot shifting algorithm, which we also give a short introduction on.

The first algorithm extends the handling of soft aperiodic tasks in conjunction with the off-line scheduling methods used in slot shifting. It allocates processor bandwidth to be used, at run-time, for an aperiodic task handling server. Bandwidth is allocated in the off-line phase, where the scheduling table is created, to not interfere with the execution of the hard real-time guaranteed time triggered tasks.

The second extension deals with the case when there is overload in the system, i.e., the tasks request more processor capacity then what is available. The problem is that time triggered tasks with hard real-time guarantees cannot be skipped or dropped due to the overload, instead the algorithm deals with the firmly guaranteed aperiodic tasks. The method finds a suitable set of firm aperiodic tasks to remove, based on task values, without disturbing the execution of the time triggered tasks.

Finally, we propose a solution to a problem that exists in most real-time systems, namely that they contain a predefined scheduling algorithm. All applications has to be tailored to fit the paradigm of the algorithm present in the system, even though it might not be suitable. With our solution we propose to insert a plug-in scheduling architecture within the operating system kernel. The plug-in architecture has a small and simple interface allowing for easy creation and insertion of various scheduling algorithms, allowing the operating system to be tailored to the application and not vice versa.

3.1.1 Slot Shifting

As mentioned, both theoretical algorithms are extension to the slot shifting algorithm presented in [27], here we give a short summary of the principal ideas of the algorithm.

Slot shifting introduces flexibility into the off-line schedule by allowing off-line scheduled tasks to be shifted in time, but never in such a way that their timely execution is impeded.

Information about this flexibility, i.e., available resources and leeway in the off-line schedule, is represented as *spare capacity* of disjoint time intervals. This information is used by the runtime scheduler to decide for each slot whether to execute an aperiodic or an off-line task. In this thesis, the spare capacity of these fixed intervals are only considered as a way to determine the spare capacity of arbitrary future intervals when handling overload.

3.2 Soft Aperiodic Task Handling

The rationale of our method to provide for complex application constraints and efficient runtime flexibility is to concentrate all mechanisms to handle complex constraints in the *off-line* phase, where they are transformed into *simple constraints* suitable for earliest deadline first scheduling, which is then used for *on-line* execution. The off-line determined simple constraints serve as an "interface" between off-line preparations and on-line scheduling. Specifically, we use the off-line scheduler presented in [25] [1], start-time, deadline pairs as simple constraints, and EDF based Total Bandwidth Server [61] and Constant Bandwidth Server [20] as runtime algorithms. The amount of desired flexibility can be set in this step as well.

Our transformation technique can extract maximum flexibility. By tightening start time and deadlines of certain tasks, it is possible to constrain the execution of some tasks, e.g., for reasons of testing or reliability.

Our method works by reducing complexity (NP hard in the case of distributed, precedence constrained executions with end-to-end deadlines) off-line by instantiating a set of independent tasks with start-times, deadlines constraints on single processors which fulfill application constraints and guaranteed bandwidth requirements. The issues of allocation to nodes, subtask deadline assignment, fulfilling jitter requirements are resolved by the off-line scheduler. This allows the use of time intensive algorithms to resolve the constraints, since they are performed off-line, i.e., before the system is deployed, and flexible scheduling at runtime.

Once tasks with start-time, deadline constraints have been derived and analyzed, earliest deadline first scheduling is performed on single nodes individually at runtime; the original set of complex constraints, distribution, etc., remains hidden from on-line scheduling.

The resulting instance of simple constraints will not generally be optimum. Since it is performed off-line, however, additional analysis can be performed, possibly resulting in another instantiation with different simple constraints. Consider a subtask deadline assignment which induces tight constraint on one node. Performance analysis may show a different, more relaxed setting to be more appropriated.

As an additional requirement, the offline scheduler has to create a schedule such that a desired fraction U_s of the processor utilization (i.e., a desired bandwidth) is reserved for on-line aperiodic service. This means that, if a bandwidth U_s is reserved on a node, then for any interval

[1]This serves as example; a number of other off-line scheduling algorithm can be applied, e.g., the one presented in [54].

$[t_1, t_2]$, there must be at least $(t_2 - t_1)U_s$ time available for aperiodic processing.

A trivial approach is to replace the worst case execution time of each task with $\frac{C}{1-U_S}$. No modifications to the scheduler are required to guarantee a bandwidth of U_S. This method, however, does not consider spare capacities in the schedule for bandwidth reservation. Response times in the resulting scheduling can thus be prohibitively long.

Our bandwidth reservation method during offline schedule construction analyzes the schedule for idle resources and their distribution. It maximizes flexibility by considering the leeway in the schedule, as per the specification constraints. In particular, the offline scheduler contains a function with tests the feasibility of the schedule constructed so far; it is extended by testing the availability of the specified bandwidth as well.

3.2.1 Transformation Technique

The feasible schedule with guaranteed bandwidth is transformed into independent tasks with start-times, deadline pairs. Our method is based on the preparations for on-line scheduling in slot shifting [27].

The offline scheduler allocates tasks to nodes and resolves the precedence constraints. The scheduling tables list fixed start- and end times of task executions, that are not as flexible as possible. The only assignments fixed by specification are starts of first and completion of last tasks in chains with end-to-end constrains, and tasks sending or receiving inter-node messages. The points in time of execution of all other tasks may vary within the precedence order. We calculate earliest start-times and latest finish-times for all tasks per node based on this knowledge. As we want to determine the maximum leeway of task executions, we calculate the deadlines to be as late as possible.

Let $end(PG)$ denote the end and $start(PG)$ the start of a precedence graph PG according to the schedule. The start of an inter-node message transmission M is denoted $start(M)$, the time it is available at all receiving nodes $end(M)$. These are the only fixed start times and deadlines, all others are calculated recursively with respect to precedence successors.

These fixed constraints are derived first: The *deadline* of task T, d_T, of precedence graph PG in a schedule is:

- If T is exit task in PG: $d_T = dl(PG)$,
- if T sends an inter-node message M: $d_T = start(M)$.

The *earliest start time* of task T, r_T, of precedence graph PG in a schedule is calculated in a similar way:

- If T is entry task: $r_T = start(PG)$,
- If T receives an inter-node message M: $r_T = end(M)$.

Next, constraints of predecessors and successors of tasks with fixed constraints are derived:

- A predecessor P of a task T with fixed deadline is assigned a deadline so as to be executed before T with EDF, i.e., $d_P = d_T - C_T$.
- A successor S of a task T with fixed start-time is assigned the same start-time as T. An appropriate deadline and EDF with ensure P preceding T. $r_P = r_T$. This step is applied recursively.

3.2.2 On-line Scheduling

Once the transformation is performed off line and a bandwidth U_s is reserved on each processing node, on line scheduling of aperiodic tasks can be handle by a Total Bandwidth Server (TBS). This service mechanism was proposed by Spuri and Buttazzo [60, 61] to improve the response time of soft aperiodic requests in a dynamic real-time environment, where tasks are scheduled according to EDF.

The server works as follows: whenever an aperiodic request enters the system, the total bandwidth (in terms of CPU execution time) of the server, is immediately assigned to it. This is done by simply assigning a suitable deadline to the request, which is scheduled according to the EDF algorithm together with the periodic tasks in the system. The assignment of the deadline is done in such a way to preserve the schedulability of the other tasks in the system.

In particular, when the k-th aperiodic request arrives at time $t = r_k$, it receives a deadline

$$d_k = \max(r_k, d_{k-1}) + \frac{C_k^a}{U_s},$$

where C_k^a is the execution time of the request and U_s is the server utilization factor (i.e., its bandwidth). By definition $d_0 = 0$. The request is then inserted into the ready queue of the system and scheduled by EDF, as any periodic or sporadic instance. Note that the maximum between r_k and d_{k-1} is needed to keep track of the bandwidth already assigned to previous requests.

Figure 3.1 shows an example of schedule produced with a TBS. The first aperiodic request, arrived at time $t = 6$, is assigned a deadline $d_1 = r_1 + \frac{C_1}{U_s} = 6 + \frac{1}{0.25} = 10$, and since d_1 is the earliest deadline in the system, the aperiodic activity is executed immediately. Similarly, the second request receives the deadline $d_2 = r_2 + \frac{C_2}{U_s} = 21$, but it is not serviced immediately, since at time $t = 13$ there is an active periodic task with a shorter deadline (18). Finally, the third aperiodic request, arrived at time $t = 18$, receives the deadline $d_3 = \max(r_3, d_2) + \frac{C_3}{U_s} = 21 + \frac{1}{0.25} = 25$ and is serviced at time $t = 22$.

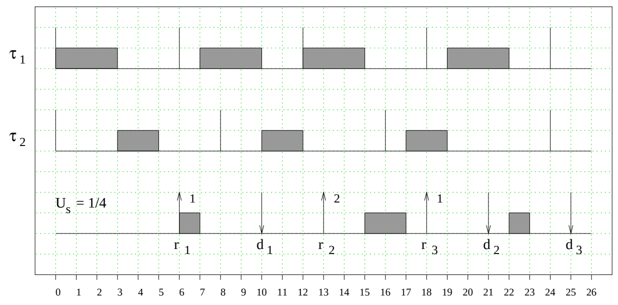


Figure 3.1: Total Bandwidth Server example.

Intuitively, the assignment of the deadlines is such that in each interval of time, the fraction of processor time allocated by EDF to aperiodic requests never exceeds the server utilization U_s. Since the processor utilization due to aperiodic tasks is at most U_s, the schedulability of a periodic task set in the presence of a TBS can simply be tested by verifying the following condition:

$$U_p + U_s \leq 1,$$

where U_p is the utilization factor of the periodic task set. This results is proved by the following theorem.

Theorem 1 (Spuri and Buttazzo, 96)**.** *Given a set of n periodic tasks with processor utilization U_p and a TBS with processor utilization U_s, the whole set is schedulable* if *and* only if

$$U_p + U_s \leq 1.$$

The implementation of the TBS is straightforward, since to correctly assign the deadline to a new request, we only need to keep track of the deadline assigned to the last aperiodic request (d_{k-1}). Then, the request can be inserted into the ready queue and treated by EDF as any other periodic instance. Hence, the overhead is practically negligible.

3.2.3 Simulation Results

In order to evaluate the method we implemented and simulated the described algorithm in a Real-Time simulator, presented in [58]. All the offline and soft aperiodic tasks were randomly generated, and the load of the offline tasks is varied between 0 and 0.9, and the soft aperiodic task load is varied between three different levels over the LCM.

The simulation length for each run was 10000 slots, and the soft aperiodic tasks arrived during that length.

We have studied the average response times of the soft aperiodic tasks, and compared our algorithm against background scheduling. We also check what happens with the response times of the soft tasks when the load of the offline tasks increases. When the offline load is equal to 0 only TBS is running.

Figure 3.2 shows the result of the simulation. The same task sets were run with both background scheduling and our method, and the response times are measured in slots.

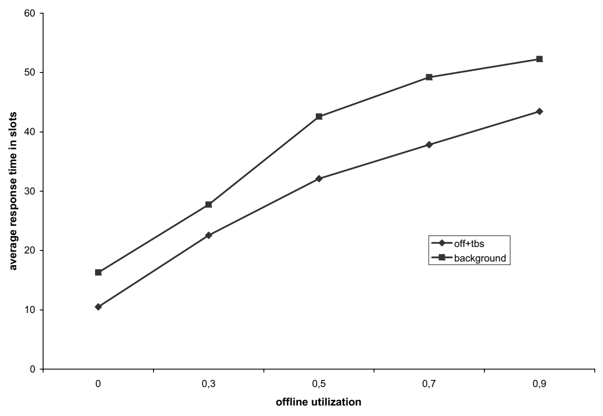


Figure 3.2: Response times for the soft aperiodic tasks.

Due to high offline load, many soft aperiodic tasks did not finish before the simulation ended and therefore the response times became higher with increased load. Even when the offline load was 0 some soft aperiodic tasks were not able to finish because the ready queue was overloaded.

3.3 Overload Handling

The overload handling algorithm presented in this chapter is developed in cooperation with Jan Carlson [2].

At runtime, local scheduling uses the slot shifting algorithm described in [27], except for the guarantee mechanism.

As outlined above, slot shifting is used to decide when aperiodic tasks can be allowed to run without causing an off-line scheduled task to miss its deadline. In addition, the scheduler must decide which aperiodic task to execute. In the proposed method, aperiodic tasks are served according to EDF once accepted by the overload detection mechanism.

To handle overload situations, each node keeps the *ready queue*, containing the aperiodic tasks ready to be executed on that node, constantly free from overload. When new aperiodic tasks arrive, they are inserted into the ready queue based on their deadlines. Then, the queue is processed to detect future overload situations and to resolve them to make the queue free from overload again.

All tasks removed from the ready queue due to overload are stored in a separate *maybe-later queue*, as long as they have positive laxity. This queue is similar to the *reject queue* in RED [16], but used for tasks migration as well as resource reclaiming.

The basis of the task migration algorithm is that selected tasks from maybe-later queues are retried, possibly on other nodes. Retrying tasks locally is required to reclaim resources when tasks finish in less time than WCET. If a task is accepted on the new node, it is immediately migrated. An important aspect of this scheme is that a task is only migrated if it has been found non-profitable for local execution, and if there is room for it on the new node, possibly after rejecting a number of lower valued tasks.

[2] P.hD. Student at the Department of Computer Science and Electronics, MĎlardalen University, Sweden.

3.3.1 Overload Handling

At run time, scheduling is performed locally via the slot shifting scheme, which decides for each slot if an aperiodic task can be allowed to execute without causing an offline scheduled task to miss its deadline.

Aperiodic tasks are served according to EDF, which gives good performance in non-overload situations. When the system is overloaded, two important issues must be addressed. In general, high valued tasks should be preferred over tasks with low value. Additionally, tasks should be removed as early as possible, rather than simply being allowed to miss their deadlines, since an early removal might allow the task to be stolen by another node in the system.

Our algorithm ensures an overload-free ready queue, i.e., all tasks in the queue can be executed without missing their deadlines, also in the presence of offline scheduled tasks. When new aperiodic tasks arrive, the algorithm checks if they cause overload, and if so, which tasks to reject in order to resolve this efficiently.

Problem Formulation

Detection and removal of overload can be formulated as a general binary optimization problem. This allows us to abstract on details, since the dynamic aspects of the rejection problem (e.g., that rejecting a task influences the finishing times of the others) are represented by static restrictions. This facilitates the development of a suitable algorithm.

Let $\tau_1, \ldots, \tau_n$ be the aperiodic tasks currently in the ready queue, including the ones that just arrived, sorted according to EDF. For each task τ_i we use a boolean variable x_i to represent whether the task should be kept in the ready queue ($x_i = 0$), or rejected ($x_i = 1$). These variables are the output of the overload algorithm, used by the Flea Market algorithm described in Section 3.3.3.

To explain the problem formulation, we first consider a simpler setting without offline scheduled tasks, and then proceed by showing the modifications needed to incorporate offline scheduled tasks as well.

Consider a single aperiodic task τ_i. To detect if there is a risk of this task missing its deadline, we need the expected finishing time, denoted ft_i. In a pure EDF setting, with no offline scheduled tasks to consider, this would be computed by adding the remaining execution times $c_1, \ldots, c_i$ to the current time.

However, detecting overload is not enough. To solve it efficiently we need to know the size of each deadline miss, so we denote by σ_i the overload amount of τ_i, defined in the simple setting as $ft_i - dl_i$. In order to ensure that τ_i does not miss its deadline, at least σ_i slots must be freed, by removing some of the tasks $\tau_1, \ldots, \tau_i$. This is represented by the following restriction:

$$c_1x_1 + c_2x_2 + \ldots + c_ix_i \quad \geq \sigma_i$$

Similar reasoning can be applied to each of the tasks in the ready queue, resulting in the following set of restrictions:

$$\begin{array}{ll} c_1x_1 & \geq \sigma_1 \\ c_1x_1 + c_2x_2 & \geq \sigma_2 \\ \quad\vdots & \\ c_1x_1 + c_2x_2 + \ldots + c_nx_n & \geq \sigma_n \end{array}$$

Note that these restrictions give a static formulation of the problem, since the σ-values are defined in term of the current ready queue, and do not depend on the x-values.

An assignment of the values 0 or 1 to the x-variables corresponds to a potential solution to the task rejection problem. Furthermore, any assignment that satisfies the restrictions corresponds to a solution that would result in a ready queue free from overload. However, we do not simply look for a solution (rejecting all tasks is always a valid possibility), we want a solution that gives as high value as possible to the system. This means that the summed values of the removed tasks should be minimized, which is represented as:

$$\min \quad v_1x_1 + v_2x_2 + \ldots + v_nx_n$$

So far, we have considered a simplified system that contains only aperiodic tasks. In order to construct similar restrictions when offline scheduled tasks also have to be considered, the definition of σ_i must be modified.

Let $sc[a, b]$ be the spare capacity of the interval from a to b, i.e., the number of slots in the interval that is not required to execute offline scheduled tasks in time. Now, σ_i can be defined as follows:

$$\sigma_i = sc[dl_i,\ ft_i]$$

This definition requires the expected finishing time to be computed, and now that the system contains offline scheduled tasks as well, this is not straightforward. Instead, we use the following definition, which is equivalent to the previous one except that it assigns negative values rather than zero to tasks that finish before the deadline. In this definition, t_c denotes the current time.

$$\begin{array}{rcll} \sigma_1 & = & c_1 - sc[t_c,\ dl_1] & \\ \sigma_i & = & \sigma_{i-1} + c_i - sc[dl_{i-1},\ dl_i] & \quad (1 < i \leq n) \end{array}$$

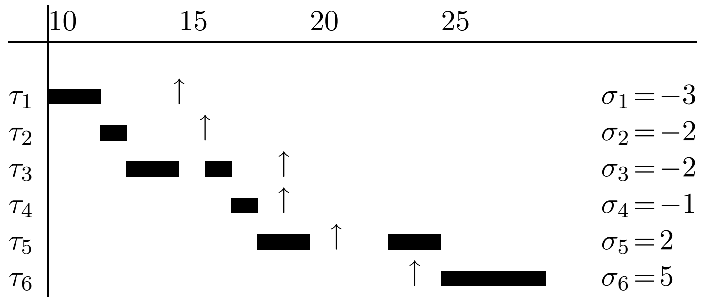
The modified definition of σ_i allows the same restrictions to be used as in the simplified setting, and the final representation of task rejection as a optimization problem is:

$$\begin{array}{lll} \text{min} & v_1x_1 + v_2x_2 + \ldots + v_nx_n & \\ \text{when} & c_1x_1 & \geq \ \sigma_1 \\ & c_1x_1 + c_2x_2 & \geq \ \sigma_2 \\ & \quad \vdots & \\ & c_1x_1 + c_2x_2 + \ldots + c_nx_n & \geq \ \sigma_n \\ & x_1, x_2, \ldots, x_n \in \{0, 1\} & \end{array}$$

Example: Let the ready queue contain the following aperiodic tasks at the beginning of slot 10, where (dl_i, c_i, v_i) represents τ_i.

$$\begin{array}{lll} \tau_1 : (15, 2, 20) & \tau_3 : (19, 3, 10) & \tau_5 : (21, 4, 20) \\ \tau_2 : (16, 1, 10) & \tau_4 : (19, 1, 5) & \tau_6 : (24, 4, 20) \end{array}$$

The tasks τ_3 and τ_6 have just arrived, and might have caused overload. If no more tasks were to arrive, the execution of the aperiodic tasks would look as follows. The arrows denote deadlines, and the gaps indicate slots needed to execute offline tasks. For simplicity, we assume that the offline schedule has a low load in the interval.



The corresponding optimization problem is:

$$\begin{array}{lll}
\text{min} & 20x_1 + 10x_2 + 10x_3 + 5x_4 + 20x_5 + 20x_6 & \\
\text{when} & 2x_1 & \geq -3 \\
 & 2x_1 + 1x_2 & \geq -2 \\
 & 2x_1 + 1x_2 + 3x_3 & \geq -2 \\
 & 2x_1 + 1x_2 + 3x_3 + 1x_4 & \geq -1 \\
 & 2x_1 + 1x_2 + 3x_3 + 1x_4 + 4x_5 & \geq 2 \\
 & 2x_1 + 1x_2 + 3x_3 + 1x_4 + 4x_5 + 4x_6 & \geq 5 \\
 & x_1, x_2, \ldots, x_6 \in \{0, 1\} &
\end{array}$$

The last two inequalities correspond to the overload at τ_5 and τ_6, and describe what must be done in order to resolve this.

3.3.2 Rejection Algorithm

Even when all restrictions except the last one are trivially satisfied ($\sigma_i \leq 0$ for $1 \leq i < n$), the problem is hard to solve. In fact, it has been reduced to the well known NP-hard binary knapsack problem, which indicates that an optimal algorithm is not feasible. Instead, our algorithm is based on heuristics that exploit properties of this particular problem.

One such property is that each restriction contains less variables than the subsequent ones. Furthermore, a good solution (with respect to the minimization criteria) to a single restriction is a reasonably good partial solution to all subsequent restrictions, since the variables are equally weighted in all restrictions.

Algorithm Description. Initially, all x_i variables are set to 0, which represents a solution where no tasks are removed. The rejection algorithm traverses the restrictions top-down, solving each of them individually.

The restrictions are solved by changing some of the variables from 0 to 1. Once a variable is set to 1, this variable is never changed during the solving of subsequent restrictions.

Each restriction, unless already satisfied by the current variable settings, is solved in three steps.

- First, we consider the variables of the left-hand side of the restriction that are currently set to 0, and would solve the restriction if set to 1. From these we select as our *best single candidate* the one with lowest v_i.
- Next, we construct the *collection candidates*. From the remaining left-hand side variables that are currently set to 0 (i.e., those that would not solve the restriction if set to 1), we collect variables from right to left until the restriction would be solved if all variables in the collection are set to 1.
- Finally the value of the best single candidate is compared against the summed values of the collection candidates (if a large enough collection was found), to decide what the final choice should be.

3.3.3 Remote Task Stealing

A distributed system with runtime task migration must somehow decide when and where to move tasks in order to maximize the total value of executed tasks. These decisions become increasingly important when the load, or the value of tasks, varies a lot between nodes. Ensuring optimal global scheduling is an NP-hard problem, and we therefore aim for a sub-optimal solution.

In order to cope with the complexity of the problem, scheduling is primarily handled locally on each node, as discussed in Section 3.3.1. Task migration is handled together with acceptance tests of new tasks, and local resource reclaiming. Further, task migration is always initiated by the node the task is to migrate to, and not the current owner. Therefore, we use the term *task stealing*, rather than migration.

To keep network usage low, and to simplify the algorithm by ruling out the possibility of conflicting thefts, only one node at a time is allowed to steal tasks. This is ensured by something similar to a conceptual token ring, where the owner of the token may steal tasks from any other node during one slot, before the token is passed to the next node in the ring.

By some arbitrary communication scheme, the maybe-later queues (or parts of them) are made visible to all nodes in the system. At the start of a slot, each node adds newly arrived aperiodic tasks to its ready queue. In addition, the node holding the token may add tasks from any maybe-later queue in the system, including its own. After adding tasks, each node applies the overload handling algorithm to resolve any overload situations.

Since only one node is allowed to steal tasks from any maybe-later queue at the start of each slot, and no additional data have to be sent over the network, the stealing node may execute one of the stolen task immediately (in the current slot).

The parameter *MaxTheft* is used to adjust the algorithm w.r.t. network capacity and system size. At the start of every slot, each node performs the following algorithm:

1. Let A be the set of all aperiodic tasks currently in the ready queue.
2. Add to A all aperiodic tasks that arrived to the node at this tick.
3. This step is only performed by the node currently holding the token. Gather tasks from the maybe-later queues of all nodes in the system. From the maybe-later queues of other nodes, consider only tasks that are movable. Add to A the tasks with highest value density, at most *MaxTheft* tasks.

4. Apply the overload algorithm to A. The result is a boolean value x_i for each $a_i \in A$, where 0 represents acceptance and 1 rejection. For each x_i, perform the following action depending on whether the task a_i was added during step 1, 2 or 3 of this algorithm.

x_i	step	action
1	1	Remove a_i from ready queue, and insert it in the maybe-later queue.
1	2	Insert a_i into maybe-later queue.
1	3	Do nothing.
0	1	Do nothing.
0	2	Insert a_i into ready queue.
0	3	Insert a_i into ready queue, and inform the current owner (possibly yourself) of the theft.

5. If the node holds the token, send it to the next node.

3.3.4 Node Communication

The algorithm is described as if the whole maybe-later queues are visible to all nodes, but this is actually not required. The node holding the token is interested only in the *MaxTheft* tasks with highest value density. By keeping maybe-later queues sorted according to value density, it is sufficient to make the *MaxTheft* first tasks in each queue visible. Also, since aperiodic tasks are assumed to reside on all nodes in the system, only tasks identifiers are sent over the network.

Furthermore, only one node uses the maybe-later queues each slot. Thus, the distribution of maybe-later queue information in a system of n nodes can be accomplished by a total of $n-1$ messages, each consisting of *MaxTheft* task identifiers and remaining execution time.

Communication is also required in order to migrate tasks. Since only one node may steal tasks from the maybe-later queues in each slot, the only communication needed in order to migrate a task is to inform the current owner of the theft. Thus, a stolen task may execute on the new node in the same slot as it is stolen. At most $n-1$ messages, each containing one task identifier, are sent each slot due to task migration.

The algorithm, as described above, assumes that the network is fast enough to permit the following communication during a single slot:

- The node holding the token sends theft messages to all nodes.
- When receiving the theft message, each node sends its new maybe-later queue information to the next token holder.

If the network does not permit this within a single slot, but within t slots, the algorithm can be modified so that the token is inactive for $t-1$ slots when it arrives to a node. Figure 3.3 and 3.4 show the communication between three nodes for $t = 1$ and $t = 3$. Ticks are denoted by vertical lines, and the scheduling performed in each slot is represented by a grid. Horizontal lines denote the token holder, and dashed lines represent that the token is inactive. Arrows starting in a grid are messages concerning stolen tasks, and those starting in the middle of a slot are messages containing maybe-later queue information.

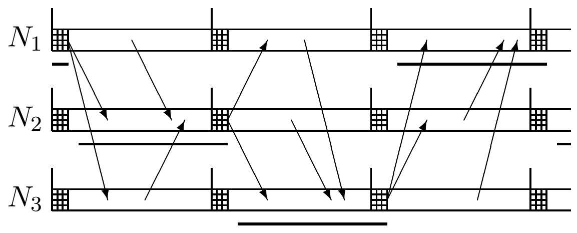


Figure 3.3: Node communication ($t = 1$).

3.3.5 Simulation Results

Simulations have been performed for various overload scenarios, showing how the algorithm behaves in terms of total accumulated value.

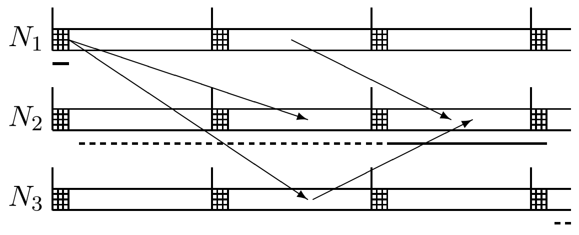


Figure 3.4: Node communication ($t = 3$).

In order to evaluate the efficiency of introducing task migration, i.e moving the tasks to other nodes for possible execution, we compare the total accumulated value for both scenarios. The simulation results are visible in tables 3.5 and 3.6.

In the figures we also show the results of a comparison of the following methods for overload handling:

1. The full method presented in the paper.
2. The overload handling algorithm, without task migration.
3. A basic algorithm that uses the offline schedule, assigning idle slots to the aperiodic tasks based on value density.
4. Same as 3, but aperiodic tasks are ordered by value.
5. Same as 3, but aperiodic tasks are ordered EDF.
6. Same as 3, but aperiodic tasks are serviced in order of arrival.

Methods 1 and 2 implement the efficiency improvements suggested in Section 3.3.2. Each point in the figures represents some 300 simulations.

In the first experiment, all nodes in the system are subject to the same amount of load. The result is presented in figure 3.5. Because all nodes are overloaded, the possibility of task migration does not provide any significant improvement. Compared to the basic methods, the proposed method performs better.

The second experiment, shown in figure 3.6, is a scenario of unevenly distributed load. Half of the nodes have no aperiodic tasks arriving, only offline scheduled tasks. Here, the task migration algorithm

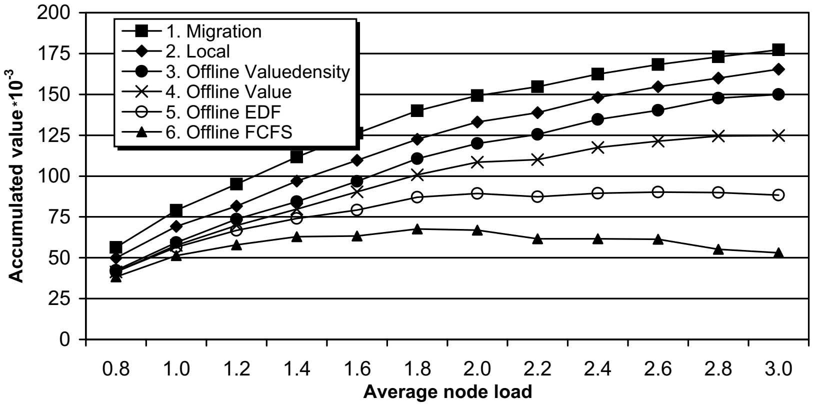


Figure 3.5: Even load distribution.

clearly increases the system performance, compared to overload handling without migration, because tasks can migrate to nodes with no aperiodic load.

Another important issues we investigated is how we can deal with the worst case complexity of the algorithm $O(n^2)$, where n is the length of the ready queue, i.e the number of tasks present in the ready queue.

We found that by reducing the number of tasks we look at in the ready queue, not necessarily all tasks, we can deal with the complexity and still get an acceptable behavior from the algorithm. Figure 3.7 shows the accumulated value for different *cutoff* values.

As we see in the figure, the difference between the different cutoff values are minimal, indicating that it is possible to efficiently deal with the complexity by looking at fewer tasks.

We have performed more simulation in order to evaluate the behavior of the algorithm, the interested reader can see these results in Appendix A.

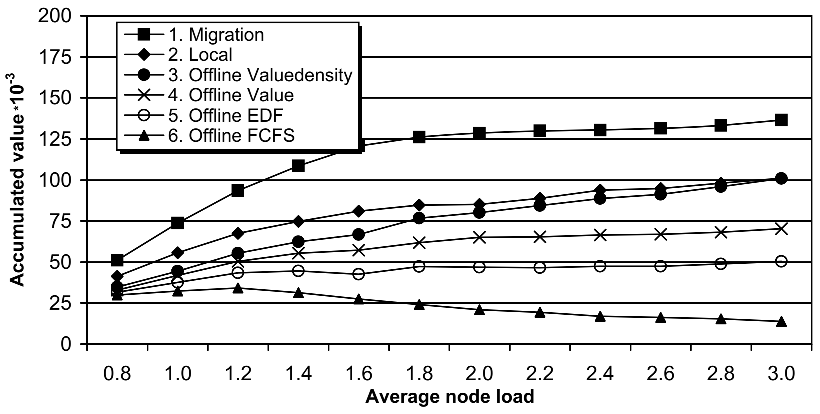


Figure 3.6: Uneven load distribution.

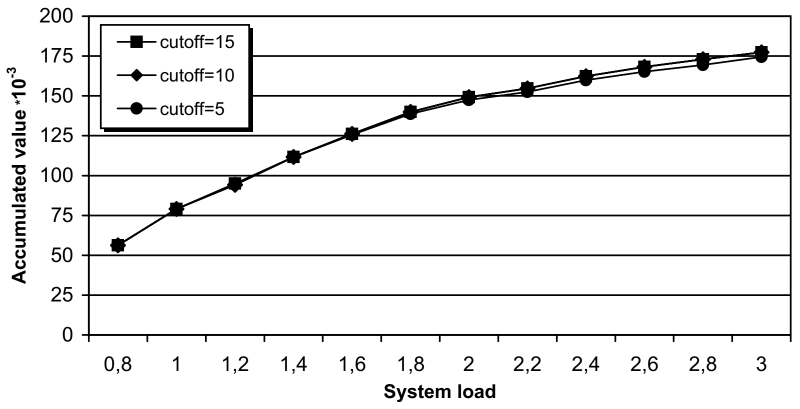


Figure 3.7: Accumulated value for different *cutoff* values.

3.4 Plug-in Scheduling Architecture

Functionality for various services of scheduling algorithms is typically provided as extensions to a basic algorithm. Aperiodic task handling, guarantees, etc., are integrated with a specific basic scheme, such as earliest deadline first, rate monotonic, or off-line scheduling. Thus, scheduling services come in packages of scheduling schemes, fixed to a certain methodology.

A similar approach dominates operating system functionality: implementation of the actual real-time scheduling algorithm, i.e., take the decisions which task to execute at which times to ensure deadlines are met, are intertwined with kernel routines such as task switching, dispatching, and bookkeeping to form a scheduling/dispatching module.

Consequently, designers have to choose a single scheduling package, although the desired functionality may be spread over several ones. Instead, there is a need to seamlessly integrate new functionality with a developed system, enabling designers to choose the best of various packages.

In this section, we propose the use of a plug-in approach to add functionality to existing scheduling schemes and provide for easy replacement on the operating system level. In particular, we present an architecture to disentangle actual real-time scheduling from dispatching and other kernel routines with a small API, suited for a variety of scheduling schemes as plug-ins. We detail two plug-ins for aperiodic task handling and how they can extend two target systems, table-driven and earliest deadline first scheduling using the presented approach.

A *plug-in* can be thought of as a hardware or software module that adds a specific feature or service to an existing system. The purpose of a plug-in is to add functionality without calling for redesign or extensive modifications. To accomplish this it must be clear what services the plug-in provides and an interface between the plug-in and the target system must be defined.

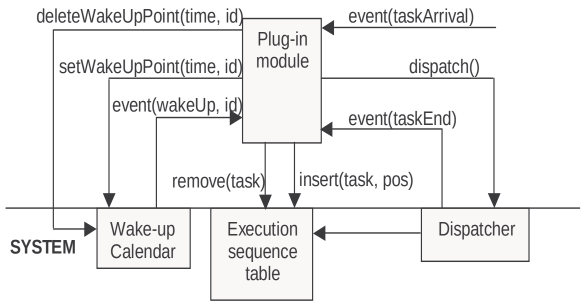


Figure 3.8: Plug-in and system architecture

Target System Architecture and Interface

Before we go into the details of the plug-in, we define the target system model that the plug-in will interact with. The model is presented in figure 3.8 and it consists of three separate modules as parts of the system:

Execution Sequence Table This is the table where the tasks are kept sorted in a certain order, depending on the plug-in module's scheduling algorithm. The plug-in module has exclusive modification rights on this table. To manipulate the table, the plug-in module uses the two methods *insert(task, pos)* and *remove(task)*.

Dispatcher It is responsible for taking the first task in the execution sequence table and execute it. The dispatcher has access to view the contents of the whole execution sequence table, but it cannot modify it. The plug-in module also has exclusive control over the dispatcher and it is activated by the *dispatch()* call. When the dispatcher is activated, it will check if there is an executing task and either preempt the task, if it exists in the execution sequence table, or else abort it.

Wake-up Calendar This calendar controls a set of watch-dog timers, all tasks will get entries set in the calendar corresponding to their deadlines (to catch deadline misses). The calendar will also hold other time critical points, such as the critical slots from [32]. To set or remove these wake-up points the plug-in module uses the *setWakeUpPoint(time, id)* and the *deleteWakeUpPoint(time)* methods. All the wake-up points are associated with an id. The id's represents deadlines, critical slots, and so on.

Plug-In Interface

The plug-in module encapsulates a scheduling algorithm for scheduling of user level tasks (not system level tasks), such that the rest of the system becomes completely decoupled from the scheduling. This means that the plug-in module is the only part of the system that knows about scheduling, and it is also the only part that needs to be changed, if the scheduling algorithm is being changed.

Therefore the interface to the plug-in module is kept small and simple such that it is clear how to write a new plug-in. This makes it easier for designers to create the scheduling package they want. The plug-in interface is used by the system, specifically the wake-up calendar and dispatcher, to activate the plug-in module at certain events or times. Thus each plug-in module that is implemented, is responsible for reacting correctly to the events that activates it.

The details of the plug-in module interface and the events it must react to follow below:

event(taskArrival) This event activates the plug-in when a new user level task has been activated. The plug-in is responsible for executing the appropriate acceptance test to either accept or reject the new task. If the task is accepted, the plug-in must insert it at the correct position in the execution sequence table and activate the dispatcher.

event(wakeUp, id) This event is sent by the wake-up calendar and it activates the plug-in at a certain point of time earlier set by the plug-in itself. Here, the plug-in must check what the wake-up activation corresponds to, by looking at the id, and take the appropriate actions.

event(taskEnd) The dispatcher sends this event to the plug-in when a task has finished its execution. The dispatcher does not care if the task is periodic (and should be reactivated later) or aperiodic, it's the job of plug-in module to make the correct decision based on this. Here, the plug-in should remove the task from the execution sequence table and activate the dispatcher.

System and Plug-In Interaction

In figure 3.8, we can see the interface the system and the plug-in uses to interact with with each other. In this section we will describe in more detail how this interaction works for some of the events that can happen during system execution.

Task arrival when a new user task is activated, *event(taskArrival)* is called to activate the plug-in module. The module executes its acceptance test to either accept or reject the task. If the task is accepted, the plug-in calls *setWakeUpPoint(dl, id)* to set a watchdog on the deadline of the task. Then, the task is inserted into the execution sequence table, using *insert(task, pos)* to set it at the correct position according to the scheduling algorithm. Finally the plug-in activates the dispatcher, by calling *dispatch()*, and then it suspends itself. The dispatcher is activated, looks at the front of the execution sequence table, picks that task for execution and then it suspends.

Task finishing execution when a task has finished its execution in a timely manner, the dispatcher gets activated and activates the plug-in module by calling *event(taskEnd)*, then the dispatcher suspends. The plug-in removes the wake-up time for the task deadline with *removeWakeUpPoint(dl, id)*, then it removes the task from the execution sequence table by *remove(task)*. The plug-in also calls

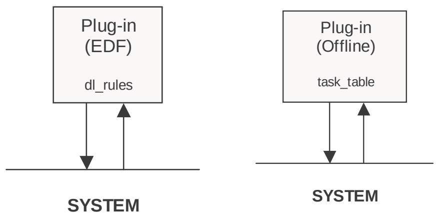


Figure 3.9: Example plug-ins

dispatch() again to activate the dispatcher. The dispatcher looks at the front of the execution sequence table and picks that task for execution, then it suspends.

Task deadline miss if a task has not finished execution before its deadline, the wake-up calendar will be activated by a timer interrupt. It will then use *event(wake-up, id)* to activate the plug-in module. The plug-in module sees that the *id* indicates a deadline miss and removes the task from the execution sequence table, and, if necessary takes other actions to handle a deadline miss. Then the plug-in calls the dispatcher, using *dispatch()*, to activate it. The dispatcher checks if the executing task exist in the execution sequence table. When it discovers that the task has been removed by the plug-in it will abort the task. The dispatcher also checks for the first task in the execution sequence table, picks it for execution, and suspends itself.

3.4.1 Target System Diversity and Plug-In Applicability

The plug-in module design makes it possible to hide the differences between scheduling algorithms behind a common interface. We will discuss how this architecture would be applied to the different scheduling paradigms that exist, and detail what the functions in the interface would do. Figure 3.9 shows the plug-ins.

Earliest Deadline Scheduled System

In an event-triggered system using the EDF scheduling algorithm, the tasks are characterized by start times, worst case execution time (WCET), and deadlines. The tasks can also be either periodic, and have the period as an additional attribute, or aperiodic. Before the start of the system, the plug-in sorts any existing tasks in the execution sequence table in EDF order. It also sets the wake-up events for the deadlines of the tasks in the wake-up calendar.

When the system is started, the plug-in activates the dispatcher and suspends itself. The dispatcher does it's job and suspends. If no new task arrives, the executing task will continue until it finishes its execution and then the dispatcher will activate the plug-in module again. The plug-in will see that it has been activated by a task-end event and remove that task from the execution sequence table. Then it activates the dispatcher again. This is how the plug-in and the system would interact if no new tasks would arrive or no deadline misses would occur.

If a new task arrives, the plug-in is activated and executes the acceptance test. If the task is accepted, it will be inserted into the execution sequence table at the correct position. The plug-in then activates the dispatcher and suspends, and the interaction continues as normal.

If a deadline miss occurs and activates the plug-in, the task will be removed from the execution sequence table. The plug-in then activates the dispatcher and the execution continues.

Off-line Scheduled System

A target system using an off-line generated [54] schedule usually has more stringent task requirements, such as precedence constraints, than an on-line scheduled, event-triggered counterpart. In an off-line generated schedule, tasks have fixed starting and finishing times. In off-line scheduled systems there are only off-line scheduled task and no new task will dynamically arrive during the runtime of the system.

Before the execution of the system, the plug-in prepares the execution sequence table to correspond to the task table internally stored in the plug-in. The on-line execution of this plug-in will therefore be simpler with an EDF plug-in module. As with the EDF plug-in, wake-up points will also be set for the deadline of the tasks in the off-line schedule. The plug-in also sets wake-up points for every time slot, like the MARS system described in [40].

When the system is activated, the plug-in immediately sets a wake-up point at the next time-slot. If no task has a start time equal to the current time, it suspends. The plug-in will be activated at the start of the next time-slot and repeat what it did in the previous time-slot.

If there is a task with the start time equal to the current time, the plug-in activates the dispatcher, then it suspend. The dispatcher activates the execution of the next task and suspends.

The plug-in will be activated every slot, and it will also get events when tasks end or if tasks miss their deadline. If a task finishes execution in a timely manner, the dispatcher activates the plug-in, which removes the task from the execution sequence table and then checks if there is a task ready.

3.4.2 Plug-Ins for Aperiodic Task Handling

Below we present two plug-ins that handles aperiodic tasks. These plug-ins are meant to be “plugged into” a scheduling module that makes scheduling decisions based on earliest start times and deadlines. The plug-ins work independently of the scheduling module and can be seen as a layer on top of it.

At all times, the scheduling module schedules task that are ready to execute, that is, tasks that are present in the ready-queue. The plug-in deals with the aperiodic tasks and places them in the ready-queue of the scheduling module, which then processes the aperiodic tasks as it would any other tasks in the system.

The mechanism for the two plug-ins for aperiodic task handling is based on the slot shifting [27], taking advantage of resources not needed by non-aperiodic tasks and using them to schedule aperiodic tasks.

We have named the different plug-ins, plug-in A and plug-in B to distinguish between the two different algorithms. Plug-in A focuses on guarantees and handling of single aperiodic tasks with fixed demands, e.g., execution time, while plug-in B is geared toward large number of aperiodic tasks with changing requirements.

Aperiodic tasks have *unknown* arrival times. The earliest start time of an aperiodic task is equal to its arrival time. Aperiodic tasks are considered independent. We assume that task dependencies are resolved in the off-line phase.

Known WCET Aperiodic tasks with known worst case times and deadlines are termed *firm* aperiodic. If accepted, which is determined by a guarantee test, these tasks must be completed before their deadlines.

Unknown WCET Aperiodic tasks without deadlines and possibly without known maximum execution times are termed *soft* aperiodic. These are executed in a best effort fashion at lower priority than guaranteed tasks such that the timely execution of guaranteed tasks is not impaired.

Off-line Preparations - Slot Shifting

We propose to use the off-line transformation and on-line management of the slot-shifting method [27], and further extended in [33]. We don't give a full description here, but confine to salient features relevant to our new algorithms. More detailed descriptions can be found in [26], [27], [33]. It uses standard off-line schedulers, e.g., [54], [26] to create schedules which are then analyzed to define start-times and deadlines of tasks.

After off-line scheduling, and calculation of start-times and deadlines, the deadlines of tasks are sorted for each node. The schedule is divided into a set of *disjoint execution intervals* for each node. *Spare capacities (sc)* to represent the amount of available resources are defined for these intervals.

Each deadline calculated for a task defines the end of an interval I_i, $end(I_i)$. Several tasks with the same deadline constitute one interval.

Note that these intervals differ from execution windows, i.e. start times and deadline: execution windows can overlap, intervals with *sc* are disjoint. The deadline of an interval is identical to that of the task. The start, however, is defined as the maximum of the end of the previous interval or the earliest start time of the task. The end of the previous interval may be later than the earliest start time. Thus it is possible that a task executes outside its interval, i.e., earlier than the interval start, but not before its earliest start-time.

Obviously, the amount of unused resources in an interval cannot be less than zero, and for most computational purposes, e.g., summing available resources up to a deadline are they considered zero, as detailed in later sections.

Negative values are used in the spare capacity variables to increase runtime efficiency and flexibility. In order to reclaim resources of a task which executes less than planned, or not at all, affected intervals only need to be update with increments and decrements, instead of a full recalculation. Which intervals to update is derived from the negative spare capacities. The reader is referred to [26] for details.

Thus, it is possible to represent the information about amount and distribution of free resources in the system, plus on-line constraints of the off-line tasks with an array of four numbers per task. The runtime mechanisms of the first version of slot shifting [26] added tasks by modifying this data structure, creating new intervals, which is not suitable for frequent changes as required by sporadic tasks. The improved [33] method briefly described in this section only modifies spare capacity.

On-line Activities

Runtime scheduling is performed locally for each node. If the spare capacities of the current interval $sc(I_c) > 0$, EDF is applied on the set of ready tasks. $sc(I_c) = 0$ indicates that a guaranteed task has to be executed or else a deadline violation in the task set will occur. It will execute immediately. Since the amount of time spent is known and represented in *sc*, guarantee algorithms include this information.

After each scheduling decision, the spare capacities of the affected intervals are updated. If, in the current interval I_c, an aperiodic task executes, or the CPU remains idle for one slot, current spare capacity in I_c is decreased. If an off-line task assigned to I_c executes, spare capacity does not change. If an off-line task T assigned to a later interval $I_j, j > c$ executes, the spare capacity of I_j is increased - T was supposed to execute there but does not, and that of I_c decreased. If I_j "borrowed" spare capacity, the "lending" interval(s) will be updated. This mechanism ensure that negative spare capacity turns zero or positive at runtime. Current spare capacity is reduced either by aperiodic tasks or idle execution and will eventually become 0, indicating a guaranteed task has to be executed. See [27] for more details.

Guarantee Algorithm A

Assume that an aperiodic task T_a is tested for guarantee. We identify three parts of the total spare capacities available:

- $sc(I_c)_t$, the remaining sc of the current interval
- $\sum sc(I_i),\ c < i \leq l,\ end(I_l) \leq dl(T_A) \wedge end(I_{l+1}) > dl(T_A),\ sc(I_i) > 0$, the positive spare capacities of all *full* intervals between t and $dl(T_A)$
- $min(sc(I_{l+1}), dl(T_A) - start(I_{l+1}))$, the spare capacity of the last interval, or the execution need of T_A before its deadline in this interval, whichever is smaller

If the sum of all three is larger than $wcet(T_A)$, T_A can be accommodated, and therefore guaranteed. Upon guarantee of a task, the spare capacities are updated to reflect the decrease in available resources. Taking into account that the resources for T_A are not available for other tasks. This guarantee algorithm is $O(N)$, with N being the number of intervals.

Guarantee Algorithm B

This plug-in uses a newer version of slot shifting, presented in [33], as guarantee test and the basic idea behind it is based on the standard EDF guarantee. EDF is based on having full availability of the CPU, so we have to consider interference from the non-aperiodic tasks in S and pertain their feasibility.

Assume that at time t_1 there is a set of guaranteed aperiodic tasks G_{t_1} and a set of non-aperiodic tasks S. At time t_2 where $t_1 < t_2$, a new aperiodic A arrives to the plug-in module. Meanwhile, a number of tasks of G_{t_1} may have executed; the remaining task set at t_2 is denoted G_{t_2}. A test if $A \cup G_{t_2}$ can be accepted, considering tasks in S is performed. If so, A is added to the set of guaranteed aperiodic tasks, G.

The finishing time of a firm aperiodic task A_i, with an execution demand of $c(A_i)$, is calculated with respect to the finishing time of the previous task, A_{i-1}. Without any off-line tasks, it is calculated the same way as in the EDF algorithm:

$$ft(A_i) = ft(A - i - 1) + c(A_i) \tag{3.1}$$

Since firm aperiodic tasks are guaranteed together with tasks in S, the formula above is extended with a new term that reflects the amount of resources reserved for these tasks:

$$ft(A_i) = c(A_i) + \begin{cases} t + \mathrm{R}[t, ft(A_1)] & , i = 1 \\ ft(A_{i-1}) + \mathrm{R}[ft(A_{i-1}), ft(A_i)] & , i > 1 \end{cases} \tag{3.2}$$

where $R[t1, t2]$ stands for the amount of resources (in slots) reserved for the execution of the tasks in S between time t_1 and time t_2 . It is possible to access $R[t1, t2]$ via spare capacities and intervals at runtime:

$$R[t_1, t_2] = [t_2 - t_1] - \sum_{I_c \in (t_1, t_2)} max(sc(I_c), 0) \tag{3.3}$$

As $ft(A_i)$ appears on both sides of the equation, a simple solution is not possible. But in [33] an algorithm, with a complexity of $O(N)$, for computing the finishing times of hard aperiodic tasks is presented.

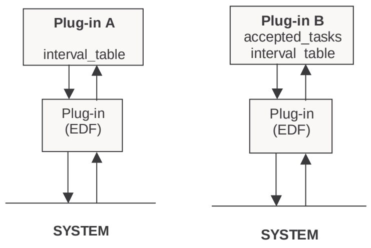


Figure 3.10: Plug-in A and Plug-in B

In this plug-in module no explicit reservation of resources is done, which would require changes in the intervals and spare capacities, as done in the plug-in A module. Rather, resources are guaranteed by accepting the task only if it can be accepted together with the previous tasks in G and S. This enables the efficient use of rejection strategies, and simplifies the handling of the intervals and *sc*.

Guarantee Plug-Ins

When a plug-in is activated, it updates the intervals in conformity with the last task execution and checks if there are any pending aperiodic tasks. If so, it processes them and puts one or more of them into the ready-queue of the scheduler. Figure 3.10 show the two plug-ins and the data structures they contain.

Plug-In A The plug-in keeps a table consisting of the intervals and their attributes (start, end, sc, and so on) that was created in the off-line phase. It must also keep track of which task executed last, when it started its latest execution, and how much time it consumed, to be able to update the intervals table. Using this information, the plug-in updates interval spare capacities and possibly also wake-up points.

Plug-In B Plug-in B also needs information about the last task execution to be able to update spare capacities and wake-up points in the intervals table it keeps locally. It focuses on handling large numbers of aperiodic tasks with changing requirements, therefore accepting tasks is done with explicit guarantees via modifying intervals and spare capacities. Rather, guarantees are including implicitly, by keeping a list of the so far accepted task. Should a task finish early, it is removed from the list and the resources reserved for it are freed without further provisions. It is well suited for efficient overload handling, since task removals do not require changes in intervals and spare capacities as in plug-in A.

After each scheduling decision, the spare capacities of the affected intervals are updated as for plug-in A.

3.4.3 Example

In this section we will use an example to illustrate how the two plug-in modules we defined earlier, plug-in A and plug-in B, work and interact with the rest of the system. We assume that there are three periodic tasks scheduled by the EDF algorithm, and the task-set is the following: $A = (1, 4)$, $B = (1, 6)$, $C = (2, 12)$, where (C, T) represents WCET and period. Deadline is assumed to be equal to the end of the period $(D = T)$. The tasks have harmonic periods to make the example simple. Firm aperiodic tasks have the format: $Ta_f = (C, D)$, and soft aperiodic tasks have the following format: $Ta_s = (C)$.

Off-line In the off-line phase the plug-ins create a table that contains all the interval start and end points, the length of the interval, the *sc* and total execution time in an interval, and lastly the wake-up (wu) point of the interval. This table is stored within the plug-in and it will be updated during runtime to reflect the correct state. Both plug-ins create identical tables as shown in table 3.1. The table is created with a length equal to the least common multiple (LCM) of the periods of the tasks. This table will be restored and repeated when time t is equal to a multiple of the LCM.

Interval	*start(I)*	*end(I)*	$\mid I \mid$	*sc(I)*	*wu(I)*
I_0	0	4	4	3	3
I_1	4	6	2	1	5
I_2	6	8	2	1	7
I_3	8	12	4	0	8

Table 3.1: The original interval table.

The execution sequence table (ES-table) contains the following periodic tasks from the start: ES-table$=\{A_0, B_0, C_0\}$.

On-line The on-line behavior of the two models differs so we will show step by step how each of them behave, and what happens with the interval table at different times. Below we will see the actions taken during each step by the system and the plug-ins.

Time	System actions	Plug-in actions

This shows how the actions by the different parts will be represented. At each point time we can see the system's dispatcher, wake-up calendar actions, and the plug-ins actions.

$t = 0$	dispatch A_0	*setWakeUpPoint(*3*), dispatch()*

No new aperiodic tasks have arrived so the plug-in sets a wake-up point and suspends.

$t = 1$	dispatch Ta_f	*remove(A_0),* Guarantee-test, *deleteWakeUpPoint(3,critical-slot), setWakeUpPoint(4), insert(Ta_f,dl-pos), dispatch()*

ES-table$=\{B_0, C_0\}$ and a firm aperiodic task has arrived,
$Ta_f = (1, 4)$.

Plug-in A The absolute deadline of Ta_f is 5, so $I_c = I_0$ and $I_f = I_1$ and the available sc in this interval is 4 ($sc(I_c) + sc(I_f)$), which is larger than Ta_f execution requirement, so Ta_f will be guaranteed. Since Ta_f 's deadline, 5, is not equal to $end(I_f)$, I_1 will have to be split. The *sc* is

Interval	*start(I)*	*end(I)*	$\mid I \mid$	*sc(I)*	*wu(I)*
I_0	0	4	4	3	3
I_{1a}	4	5	1	0	4
I_{1b}	5	6	1	0	5
I_2	6	8	2	1	7
I_3	8	12	4	0	8

Table 3.2: Updated interval table for plug-in A.

also updated after the split and the interval table for plug-in A is shown in table 3.2.

Plug-in B In this plug-in the set of guaranteed aperiodic tasks (G) is empty. The plug-in tests if Ta_f can be accepted together with the periodic tasks. This is done by calculating the finishing time of Ta_f, which is 2 in this case (according to formula 3.2). No interval split will occur in this plug-in, nor any change to the *sc* of the intervals table because an aperiodic task was accepted.

Both plug-ins will set an updated wake-up point. The wake-up point has been changed because task A_0 has executed one slot, and then suspend.

$t = 2$	*event(taskEnd)*, dispatch B_0	*remove(Ta_f)*, Internal-work, *dispatch()*

ES-table= $\{B_0, C_0\}$. No new aperiodic tasks has arrived, Ta_f has finished. The plug-ins will be activated by this task-end event, *plug-in A* will modify the wake up point of the interval Ta_f belonged to in the intervals table, $wu(I_1a = 5)$, and then suspend again. *Plug-in B* takes no action and suspends.

$t = 3$	*event(taskEnd)*, dispatch C_0	*remove(B_0)*, Internal-work, *dispatch()*

ES-table= $\{C_0\}$. No new aperiodic tasks have arrived. C_0 will execute. B_0 has finished, the wake up point is not modified because B_0

belongs to a later interval (but the wu in that interval is modified, so $wu(I_1) = 6$).

$t = 4$	*event(wakeUp),* dispatch Ta_s	*insert(Ta_s,first-pos), insert(A_1,pos), setWakeUpPoint(5), setWakeUpPoint(6), dispatch()*

ES-table= $\{A_1, C_0\}$. Next instance of task A is ready. C_0 has finished executing and it belongs to a later interval, so the wu of that interval is modified ($wu(I_3) = 9$).

A soft aperiodic task $Ta_s = (4)$ has arrived. Both plug-ins will behave in the same manner: since $sc(I_c) > 0$, task Ta_s will be inserted first in the ready-queue. *Plug-in B* will set the next wake up point and suspend. *Plug-in A* will set the wake up to 5 even though the original $wu(I_c) = 4$, this has changed because Ta_f executed in an earlier interval and thus the $sc(I_c)$ increased to 1.

$t = 5$	*event(wakeUp),* dispatch Ta_s	*setWakeUpPoint(6), dispatch()*

ES-table= $\{A_1, C_0\}$. Plug-in A is activated by the wake-up point event. Normally this means that the execution of the soft task must be stopped in favor of a periodic task. But in this case we have only an interval change, and the $sc(I_c) > 0$, so the soft task can continue to execute ($sc(I_c) > 0$ because B_0 executed in an earlier interval). *Plug-in A* resets the wake up point and suspends itself. *Plug-in B* is not activated.

$t = 6$	*event(wakeUp),* dispatch Ta_s	*insert(B_1,EDF-pos), setWakeUpPoint(7), dispatch()*

ES-table= $\{A_1, B_1, C_0\}$. The second instance of task B is activated. Both plug-ins are activated by wake up points, this means that the execution of the soft task must be stopped in favor of a periodic task. Once again, there is only an interval change and a new wake up point can be set, and since the $sc(I_c) > 0$, Ta_s can continue to execute. Both the plug-ins suspend.

$t = 7$	*event(wakeUp),* dispatch A_1	*remove(Ta_s), setWakeUpPoint(8), dispatch()*

ES-table= $\{A_1, B_1, C_0\}$. The plug-ins are activated due to the wake up point. Ta_s must be interrupted so A_1 won't miss it's deadline. The plug-ins set the next wake up point and suspend.

$t = 8$	*event(wakeUp)*, *event(taskEnd)*, dispatch Ta_s	*remove(A_1)*, *insert(A_2,pos)*, *insert(Ta_s,first-pos)*, *setWakeUpPoint(9)*, *dispatch()*

ES-table= $\{A_2, B_1, C_0\}$. The next instance of task A is activated. Since the $sc(I_c) > 0$, Ta_s will be put first in the ready queue and executed. The plug-ins set the next wake up point and suspend.

$t = 9$	*event(wakeUp)*, *event(taskEnd)*, dispatch A_2	*remove(Ta_s)*, *setWakeUpPoint(10)*, *dispatch()*

ES-table= $\{A_2, B_1, C_0\}$.
Ta_s has finished executing, the plug-ins set the next wake up point and suspend.

$t = 10$	*event(wakeUp)*, *event(taskEnd)*, dispatch B_1	*remove(A_2)*, *setWakeUpPoint(11)*, *dispatch()*

ES-table= $\{B_1, C_0\}$. A_2 has finished it's execution, B_1 is executed. The plug-ins set the next wake-up point and suspend.

$t = 11$	*event(wakeUp)*, *event(taskEnd)*, dispatch C_0	*remove(B_1)*, *setWakeUpPoint(12)*, *dispatch()*

ES-table= $\{C_0\}$. B_1 has finished executing, the plug-ins set the next wake up point and suspend.

After this, because $t =$ total length of the interval tables, the plug-ins recreate the original intervals table by restoring the *sc* and *wu* of the intervals. If an aperiodic task arrives and has a deadline longer than the end of the interval table, the table will be extended by repeatedly adding the original table to the end of the extended table, until it is longer than the deadline. All the interval information (start, end, sc, and so on) of

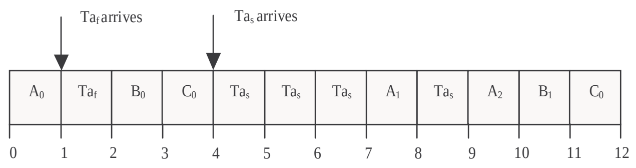


Table 3.3: Example execution trace

the extended table is adjusted to represent a larger table, and thus later time points.

3.4.4 Results

Our proposed plug-in architecture addresses the need for adding functionality to systems, in particular scheduling algorithms, without need for abandoning trusted methods or major revisions.

We presented a plug-in approach for aperiodic task handling, in two different plug-in modules, and showed their applicability to two scheduling schemes, EDF, and off-line scheduling. Our method concentrates the aperiodic task functionality into a software module with a defined interface.

The purpose of the architecture is to disentangle actual real-time scheduling from dispatching and other kernel routines with a small API, suited for a variety of scheduling schemes as plug-ins. As the functionality of the plug-in is independent of the basic scheduling scheme and the interface is very small, we can insert and apply the aperiodic-plug-ins to both off-line and on-line scheduling methods. The API is kept small in order to keep it simple when adding/writing a new scheduling module, basically the plug-in module must react to three events from the surrounding system. When task arrives for execution, wake-up points in time, and when a task ends its execution. Task arrival is where any acceptance tests can be performed in order to decide if the task can be allowed to execute or not. Wake-up points are used to wake the scheduling module at certain time points, for instance if time is slotted, at preemptions, and so on. Finally, the task end event allows the module to handle tasks when their execution has finished, periodic tasks should be reactivated at a later time, aperiodic can be removed, and so on.

3.5 Chapter Summary

In this chapter we presented scheduling solutions for various scenarios. Two solutions are theoretical extensions to the slot shifting algorithm presented in [27], the third solution deals with operating system kernel scheduling architectures.

We presented a solution for handling soft aperiodic tasks in off-line scheduled systems. The method allocates processor bandwidth in the off-line phase of scheduling, which is then supposed to be used by a soft aperiodic task scheduling server, such as the Total Bandwidth Server. This allows for a more efficient handling than originally proposed in slot shifting.

The second theoretical solution deals with overload in a off-line scheduled distributed system. Overload must be removed without disturbing the execution of the time-triggered tasks which has hard real-time guarantees. Thus, the method removes the overload by removing firm aperiodic tasks. The algorithm uses task values to compute the set of tasks that is most appropriate to remove in order to solve the situation. Removed tasks can be reinserted later for execution, if there are more resources available due to execution being less than WCET, or they can be migrated to other nodes for possible execution there.

The final solution within processor scheduling deals with a common real-time operating system problem. Most RTOSs uses a fixed scheduling algorithm, such as fixed priority, which is highly intertwined with the rest of the OS kernel in order to have efficient overhead. The problem is that all applications written for that OS must be tailored to the specific scheduling paradigm, even though it might not be suitable at all. We propose a way to disentangle the scheduling algorithm from the rest of the kernel by implementing a plug-in module. The plug-in module has a Small and simple interface allowing easy implementation and insertion of any scheduling algorithm. Thus, allowing the OS to be tailored to the applications and not the opposite.

Various simulations have been conducted to show the results of the algorithms presented above. All results are collected and presented in appendix A.

Chapter 4

Network Packet Scheduling

4.1 Overview

In this chapter we look at the wireless network part of the in-home entertainment system. Wireless Ethernet (IEEE 802.11) is used when streaming video between devices.

We will give an overview of our architecture where we state the QoS parameters we are interested in, and we also give a short overview of the problems that exist when using a wireless network.

Finally, we present our architecture in detail, first how we perform the available bandwidth prediction using probe packets. Secondly, we present how the prediction information is used to adapt the transmission rate of the node by using low level traffic shaping and application level rate changes.

4.1.1 Streaming using Wireless Ethernet

Streaming of video and audio is a major part of in-home entertainment systems. As mentioned earlier, the basic idea with these systems is to be able to wirelessly stream the video or audio from any device capable of streaming to any device capable of receiving and handling the stream. The problem is that not all devices are capable of handling any stream, for instance, a handheld PC is not powerful enough to handle a full quality DVD movie. Because the available bandwidth of the wireless network varies, there are no guarantees that a there is enough bandwidth available to stream the full quality DVD movie, resulting in long delays or even packet losses.

In order to achieve proper streaming, i.e for the user to perceive a good video or hear good audio quality, the in-home entertainment system needs to provide some kind of QoS.

For the in-home entertainment system we envision in this thesis, we are interested in the following network QoS:

- A more reliable delivery of streaming packets compared to standard Ethernet.

- A lower average latency for the streaming packets compared to standard Ethernet.

We compare our solution to what unmodified wireless Ethernet standard can provide, in terms of the network QoS we defined above.

In this chapter we present an architecture that provides network QoS, as defined above, for streaming of video using a wireless network.

Unfortunately, as we have presented in section 2.3, the wireless Ethernet (IEEE 802.11) types of network, was not designed with QoS as a goal. Furthermore, since the network uses an unregulated bandwidth spectrum, it can experience disturbance from other devices, such as cordless phones, microwave owens, Bluetooth devices, and other wireless networks, all operating within the same spectrum. Walls, movement, and other physical effects also affects the performance of the network. All of this, combined with the radio wave phenomenon called fading, where the signal basically disturbs itself, makes guaranteeing reliable network communication difficult to achieve.

As presented in section 1.3.5 bandwidth prediction is a big research area. Most bandwidth prediction research is aimed at Internet, where the purpose is to try to find bottleneck links and then reroute the traffic around those links, and temporal aspects are not as important as in our case.

Traffic smoothing is a way to deal with network congestion, by limiting the transmission rates from all nodes to a certain limit, congestion can be decreased. The traffic smoothing methods presented in [44, 43, 18] was all intended to be used in a wired Ethernet, where the available bandwidth is constant, which is not the case in wireless Ethernet. The methods also rely on specific information, such as the number of collisions occurring on the network, which is hard to get in wireless networks, thus rendering the methods not directly applicable to our in-home entertainment system.

It is not enough to shape the traffic at the low level presented in [44, 43, 18], because if the application continues to transmit at a high rate, there is a chance of internal buffer overflow. Thus, it is also important that the application is aware of the varying bandwidth, and adapts appropriately.

4.2 Basic Idea

The architecture we propose provides QoS for video streaming over the wireless network, by adapting the transmission rate to the varying bandwidth, and by prioritizing the transmission of the video stream packets.

In order for the architecture to provide QoS, it performs two functions: it predicts the available bandwidth of the network and adapts the transmission rate accordingly.

A major problem with bandwidth prediction in wireless networks is that there is no easy way to know what the actual bandwidth is. Thus it is not possible to compare the prediction to the real value in order to find the error term, i.e. *actual bandwidth* − *predicted bandwidth*.

What we get from our bandwidth prediction is the average bandwidth that will be available during the time interval until we make the next prediction, which is also the bandwidth we adapt to. But it can actually look like what is shown in figure 4.1.

If our prediction of the available bandwidth during the time interval is lower than the bandwidth that is actually available, we are safe, i.e. we will not try to transmit more than what is available. Otherwise, we will

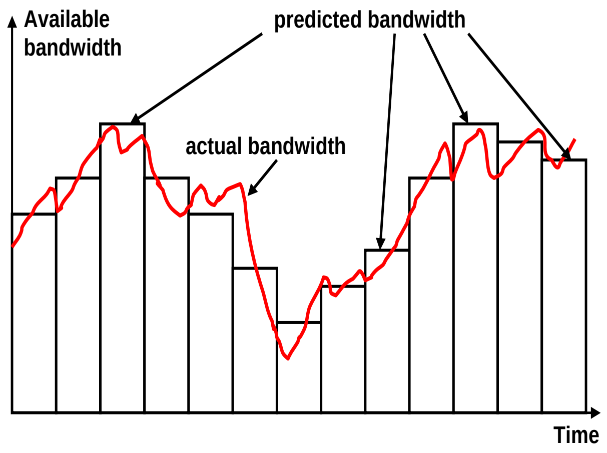


Figure 4.1: Difference between actual and predicted bandwidth for each time interval.

try to transmit more than what is available, which will result in network congestion. network congestion is bad since it will cause packets to collide, which in turn causes retransmissions, which in turn causes the packet to become delayed for a longer time, or even dropped. In the end, this can cause important streaming packets to miss deadlines or even disappear, causing the quality of the displayed video to be bad. So, this is the major issue we are addressing with our proposed architecture.

Actually, instead of trying to predict the exact bandwidth, we will try to answer a very simple question:

"is it possible to continue transmitting at the current rate?".

There are two answers to this question: "yes" or "no". A "no" means that is it not possible to continue and we have to adapt the transmission rate appropriately. A "yes" answer means we can continue, or maybe even increase the rate we have if so desired. In order to answer this question we are no longer interested in the exact amount of available bandwidth, instead we want to know if it is higher or lower than the currently requested transmission rate. This shifts our interest, from an exact bandwidth prediction, to a quick prediction that is a just part in the process of determining an answer to the question above. We are more interested in the end result, i.e. that the streaming is done properly, which in our case means the timely arrival of the streaming packets at the receiver node.

Once we know the answer to the question posed above, we take the appropriate action: continue with the current transmission rate or change to a lower rate. This approach introduces two new important questions we need to answer:

- How often should we check if we can continue transmitting with the current rate?
- With which granularity should we react to a change?

1st Question The first question deals with how often we need to sample the wireless network in order to "catch" the variability of the available bandwidth. If it varies rapidly, we need to predict, i.e., sample, the bandwidth often, otherwise we will cause congestion since we might try to transmit at a higher rate than what is available. On the other hand, if the variations occur more seldom we don't need to sample and predict that often.

This is the problem of determining the sampling interval we need to predict the available bandwidth. Normally, the Nyquist-Shannon Sampling Theorem[1] would be used to determine how often we should predict the bandwidth. In order for us to use the sampling theorem, we need to have a model representing the "behavior" of the available bandwidth of wireless network. Unfortunately, there are no proper models, and a model is difficult to create since the behavior varies depending on the environment in which the wireless network is deployed.

A problem with sampling often is the overhead introduced by our prediction method, since we send probe packets in the network and perform computations at the sending node. The question is how big is the

[1]The Sampling Theorem states that the sampling rate should be at least $2x$ the bandwidth of the signal to be sampled

Maximum time	43.9 ms
Minimum time	15.5 ms
Average time	17.5 ms

Table 4.1: The measured maximum, minimum, and average overhead for the bandwidth prediction.

overhead is. Table 4.1 below shows the overhead of performing the bandwidth prediction in our architecture.

As we seen in the table it takes about 15 ms to perform a complete bandwidth prediction, sending the probe packets and receiving the time difference result and performing the calculation. At a maximum, it takes about 43 ms to perform the prediction. From these numbers we conclude that frequent sampling should not be a problem. Furthermore, since the code is not optimized we expect the possibility of improving these numbers with more efficient code.

2nd Question The second question, at which granularity we should react, deals with when we should react to the answers we get from the question of whether we can continue transmitting at the current rate or not. It is a question of whether we should react with a small granularity, i.e., to changes on the bit level, or with a large granularity, i.e., when the answer indicates a big change in rate. For example, we could have a small set of different bandwidth levels, such as Low, Medium, or High bandwidth. As long as the rate changes stays within the same level, we do not react, but when the rate changes to another level we react.

Even if the answer is "no", the result can be that we adjust the rate only to discover that the next answer says we can switch back to the previous rate again.

If we react to often the end result might be a bad experience for the user, if the video quality is fluctuating because of the varying bandwidth of the network the perceived quality of the displayed video is bad.

Our architecture has the possibility of adjusting the transmission rate

Maximum time	55.5 ms
Minimum time	22.8 ms
Average time	38.8 ms

Table 4.2: The measured maximum, minimum, and average overhead for updating the traffic shaper parameters.

simultaneously on both a low and the application level within the operating system.

On the low level it is possible to catch all IP packets and adjusts their rate before they are inserted into the Network Interface Control (NIC) layer. At the low level we get a very fine granularity for control, basically we can adjust the rate in terms of one single bit if we wish. This is where the traffic smoothers presented in [44, 43, 18] are positioned.

The problem is that it is not enough to only adjust at this level, the applications also needs to adapt their transmission rates according to the node limit. Otherwise we risk internal buffers to overflow, so we may drop packets locally even before they are transmitted.

By only performing traffic shaping on the application level, the problem will be that the communication layers below the application does not respect the rate transmitted by the application, and thus can create bursty traffic at lower levels. Therefore, application level traffic shaping it is not a viable solution by itself, but in combination with the low level shaping it is viable.

The architecture we present performs traffic shaping at both levels to avoid the problems that otherwise can occur.

We are again interested in the overhead of switching, it might be worth switching at the low level every time there is a change, and the high level switches occur more seldom.

Table 4.2 below shows the maximum, minimum, and average times it takes to change the low level traffic shaper in our architecture.

As the table show it takes some time to change the traffic shaper at the low level, this limits the probing intervals we can use. As we see in the table the maximum time a change of the traffic shaper parameters took around 55 milliseconds.

4.3 Architecture Details

The bandwidth prediction and traffic shaping architecture should be present on all nodes within the in-home entertainment system. In this way all nodes try to adapt according to the available bandwidth of the network, by repeatedly checking if they can continue transmitting at the current rate or not. It is important to note that the architecture works without global control, each node uses our architecture locally without consideration of the other nodes within the system.

If there are legacy nodes, i.e., nodes that for some reason cannot use our architecture, the system will still function but with the possibility of those uncontrolled nodes of causing network congestion when they transmit data.

To answer the question we need to determine if the available bandwidth is higher or lower than the rate we are transmitting at. We look at two parameters to determine the available bandwidth; first we measure the delay between probe packets that we send into the network. This delay indicates the amount of congestion between the sender and receiver, i.e the whole path through the network. Secondly, we look at the history of predictions in order to catch any trends of the varying bandwidth, i.e., is it going down, up, or staying the same?

These two parameters are combined to give us the answer to the question, i.e. if we can keep transmitting at the current rate or not.

If the answer is yes, we can continue, or maybe even increase the transmission rate if we want or need to.

Otherwise, if the answer is no, we have to adjust (lower) the transmission rate to match what we can actually transmit. In order to know what rate we can transmit at, we use the two parameters, probe packet delay and actual data transmitted from the node, to predict what the new rate will be.

This is repeated periodically, and all this activity occurs on every node connected to the network, so each node detects and, if needed, adjusts it's rate accordingly.

Every node is also allocated a proportional share of the available bandwidth of the wireless network. The share is an upper bound on how much traffic the node is allowed to transmit into the network. By carefully assigning these shares, we can prioritize specific traffic from a node, i.e. the video stream from the streaming server. In our system we consider the node containing a streaming server most important, and therefore we assign it the largest share of the available bandwidth. Note that, because of the previously mentioned fluctuation of available bandwidth, the assigned share for each node will also fluctuate over time.

Figure 4.2 shows how this can look, where a streaming server is assigned a higher bandwidth share than the other nodes.

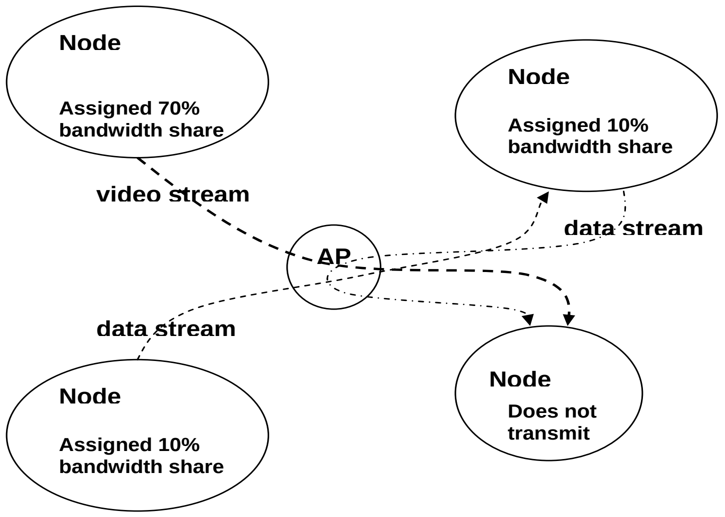

Node
Assigned 70% bandwidth share
Node
Assigned 10% bandwidth share
video stream
data stream
AP
data stream
Node
Assigned 10% bandwidth share
Node
Does not transmit

Figure 4.2: Example architecture: a set of nodes with different bandwidth shares, note that the streaming server has the highest share.

The problem with this approach is that bandwidth can be wasted unless there is a possibility to dynamically adjust these shares according to how they are used by the nodes. If a node would not use all of it's share of the bandwidth, no other node would have the possibility of using it instead.

In [57] the authors present an architecture, called the Matrix, that controls all the available resources within an in-home entertainment system. Within the Matrix architecture there is a global resource manager that is responsible for all decisions regarding resource usage.

The parts presented in this chapter of the thesis would fit into the Matrix architecture as a local resource manager, which takes orders from the global resource manager. The global manager would be responsible for the proportional bandwidth share of all nodes, and because it has a global view of how every node uses its share it could adapt them accordingly.

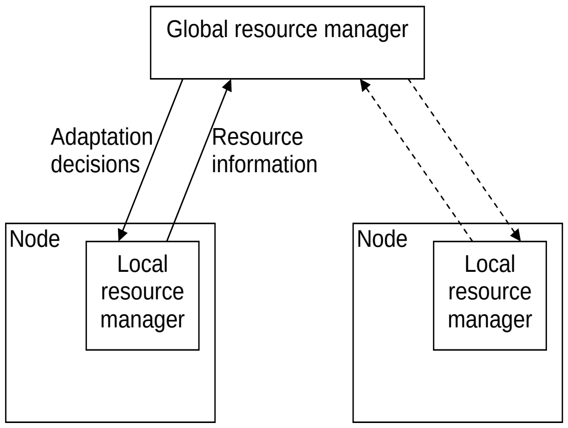


Figure 4.3: Overview of how the data flows between the local resource manager and the global resource manager, using the Matrix architecture.

4.3.1 Adjustment Level

There are two levels where we can adjust the traffic. The low level, where we have a fine granular control and can adjust the traffic on bit level, and the application level where have a coarse granularity of control because of the underlying communication protocol will interfere with our control.

We do not want to change the rate at the application level to often. In the Matrix architecture, [57], the authors suggest that a video stream can be divided into a small number of quality levels, such as: Low, Medium, and High. Each level of quality indicates a different bit rate, so the Low level quality video has a lower bit rate than Medium, and so on. The idea is to switch between these different streams when the available bandwidth of the network varies.

The reason for this abstraction over the fluctuations in [57] is to avoid overloading the scheduling if it has to react to every small change.

Again, tying in to the MATRIX architecture presented in [57], the decision to switch between the different the different quality level streams is a major decision and would therefore be taken by the global resource manager. This is because a switch of the video stream quality affects several resources within the in-home entertainment system, i.e. a change to higher quality implies that the stream demands more resources for the whole path between server and client.

The local resource manager would handle all low level traffic rate adjustment, which occur more frequently and are too small to involve a global decision. The global resource manager of course still has knowledge about the resource availability at the local node, even though it does not intervene.

In order to properly adjust the transmission rate, the first part of our architecture, the *bandwidth predictor*, predicts the future bandwidth based on the current available bandwidth and previous bandwidth predictions.

The bandwidth prediction is then fed into the low level part of our architecture, the information is also if necessary conveyed to the global resource manager suggested in [57] so that the application can be notified if needed. It shapes the traffic according to the available network bandwidth. By separating the handling of streaming and non-streaming traffic by using different shapers, we prioritize the streaming packets by giving it a larger share of the bandwidth assigned to the node.

Available bandwidth is predicted between one sender and one receiver, i.e if there are multiple receivers, available bandwidth must be predicted for each, because the results can be different depending of the fluctuations of the wireless network.

Furthermore, for this thesis we assume that each sender only transmits to one receiver, i.e only unicast communication, but this also works for multiple receivers.

4.3.2 Bandwidth Prediction

Common for most of the bandwidth prediction presented in the literature is that they do not easily apply to wireless networks. The reason is that the available bandwidth fluctuates an measurements needs to be repeated often, thus the measurements need to be simple. The methods presented above are mainly intended to find bottleneck links within a more static environment. And the rate of change is on a generally measured on a larger timescale that what we are interested in, as for instance presented in [35].

What we want is a fairly simple method that repeatedly probes the wireless network and predicts the available bandwidth until the next prediction takes place. These predictions are repeated in the sub second range in order to give us a consistent picture of the fluctuating available bandwidth.

Bandwidth Prediction Method Details

Our bandwidth prediction uses a packet-pair probing technique presented in [51], because of the advantage of being a method independent of which protocol that is used, i.e., TCP or UDP. We determine the network state by measuring the time delay between the probe packets. In [17], the authors investigate other methods to determine the network state, the signal strength, and signal strength variation, but found that the probe packet delay is the most reliable method.

Interval of 0.2 s	3427 kbps
Interval of 0.5 s	3411 kbps
Interval of 1.0 s	3363 kbps
Interval of 2.0 s	3364 kbps

Table 4.3: Different sampling intervals has little effect on the average predicted bandwidth.

In order to dynamically react to the varying bandwidth of the network, we periodically repeat the bandwidth prediction. Depending on the time for the period of prediction we can get different results on the measurements, so we have performed measurements with different periods to try to determine which period that is most suitable.

As we can see in table 4.3, the average bandwidth predicted using different sampling intervals is almost identical.

More results from the bandwidth prediction is available in Appendix A.

In our method, the sender transmits two probe packets (of identical length), back to back, to the receiver. The receiver measures the delay between the probe packets and returns this information, in another packet, to the sender. This delay gives an indication of the current network load, a high delay indicates a high network load and vice versa.

Formula 4.1, taken from [51], shows our simple calculation for the measured bandwidth.

$$BWT = L/\Delta_T \tag{4.1}$$

Where *BWT* is the resulting bandwidth, *L* is the probe packet length, and Δ_T is the measured delay between the probe packets ($T_2 - T_1$).

But this is not enough, we want to look at the history of measurements so we don't react wildly to a freak measurement that is not at all consistent with the overall behavior of the network.

So, in order to predict the future available bandwidth we use *exponential averaging*, which is a technique used to examine and average a sequence of values along a time series, which enables us to make a prediction based on previous predictions as well as the current network load.

Formula 4.2, also taken from [51], shows how we predict the bandwidth including history predictions:

$$P_k = \alpha BWT_k + (1 - \alpha)P_{k-1} \tag{4.2}$$

Where P_k is the future prediction, P_{k-1} is the previous prediction, BWT_k is the current bandwidth measurement, and α is a constant used to determine how important the history vs. the current measurement is to the current prediction.

More results are presented in Appendix A.

4.3.3 Traffic Shaping

The second part in our architecture is the traffic shaper. It is responsible for controlling the rate of packets transmitted from a node, and it uses information from the traffic prediction to properly adjust the transmission rate.

Normally, packets are transmitted in FIFO order, without any consideration to the data contained within the packet. Traffic shaping is a way to enforce prioritization policies on the packet transmission over a network link.

Originally, traffic shaping was intended as a tool to divide bandwidth between different applications. Making it possible to prioritize interactive traffic over other traffic, such as file transfers, in order to experience shorter delays.

Traffic Shaping Architecture Details

The traffic shaping architecture we propose shapes the outgoing IP-packets. It is positioned at a low level within the network communication stack, below the IP level, where it has access to all IP packets being sent.

A requirement we have for our shaper is that it must prioritize data, i.e packets, of different classes. In our solution we define three classes of data; probe packets, video streams, and other data. The reason is that we want to have as low latency as possible for the probe packets, i.e we perform probing it should experience as little latency as possible in order to reflect a fresh network state. Video streams are also important, since we envision an in-home entertainment network. Other data is just a class used for all other transmissions, such as http, ftp, and so on.

Our traffic shaper is based on the token bucket[2] algorithm. The token bucket algorithm has a very widespread use in the context of traffic shaping.

Each token bucket has three parameters; tokens, bucket depth, and refresh period.

Tokens indicates the number of tokens presently in the bucket, n tokens are consumed when a packet of size n bytes is transmitted. Packets are only transmitted if there are enough tokens available.

Bucket depth indicates how many tokens a bucket can contain at any time.

Refresh Period is the time between each refill of tokens, up to the bucket depth, into the bucket.

In [44, 43, 18] they all sue a token bucket solution to smooth the traffic, the difference is that they do not smooth the real-time packets, which is something we want to do. The reason for this is that they assume that real-time packets are not sent very often, the video stream is composed by real-time packets, which in this case will be sent very often. So, in the pure form, one token bucket is not enough for our needs, since we want packet prioritizing.

[2]The leaky bucket algorithm is similar, with the difference in that it has a constant output rate.

Token Bucket Solution

A way to prioritize the different packets is to use several buckets, and filter the different packet classes into different buckets. By giving a guaranteed transmission rate to each of the buckets, some kind of prioritizing occurs. For instance, the video stream class requires a higher transmission rate than the probe packet class, but it should not starve the probe packet class.

Figure 4.4 shows the architecture of the traffic shaper when token buckets are used.

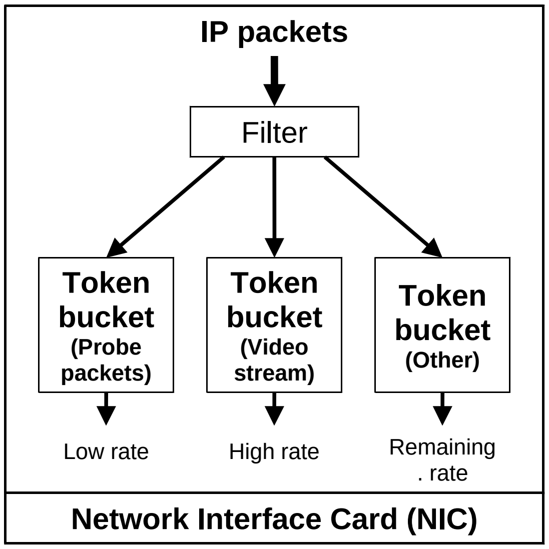

IP packets
Filter
Token bucket (Probe packets)
Token bucket (Video stream)
Token bucket (Other)
Low rate
High rate
Remaining . rate
Network Interface Card (NIC)

Figure 4.4: Architecture using 3 token buckets for traffic shaping

There are three buckets, one for each class of packets, in this architecture the video stream is given the highest transmission rate, probe packets the lowest, and other traffic gets what is remaining.

In our implementation of this method we use a queuing discipline described in [45] called the *Hierarchical Token Bucket* (HTB), which is implemented in the Linux kernel, instead of the normal token buckets described above. HTBs allow for borrowing of bandwidth from one bucket to another, thus, if there is no video stream transmission at the moment, the other buckets have the possibility of using that, otherwise wasted, bandwidth.

The problem with this solution is that no actual prioritizing takes place, thus it does not exactly fulfill our requirement.

Priority Queue Solution

A solution that fulfills our prioritizing requirement is to use a token bucket to limit the total transmission rate from the node, and then filter the three packet classes into different priority queues.

Packets from a higher priority are always transmitted before lower priority packets, with the possibility of starvation. Thus, packets from lower priority queues are transmitted as "background traffic" to the higher priority queues, which is not something we desire.

Figure 4.6 shows the architecture of the traffic shaper when priority queues are used instead of token buckets.

In this solution we have 3 priority queues, the probe packets have the highest priority in order to guarantee that they are transmitted when they are sent. The video stream has middle priority, and the other packets has the lowest priority. In this solution the video stream can, and probably will, starve the lowest priority packet class because it sends a large amount of data.

Combined Solution

The best solution would be if we could combine the two solutions presented above, i.e we both prioritize the classes and give guaranteed rates to them. We want the probe packets to be transmitted without any hindrance from other packets, and since they are small this will not have a sever performance impact on the video stream. Furthermore, we also want the video stream to have a higher transmission rate than the other traffic.

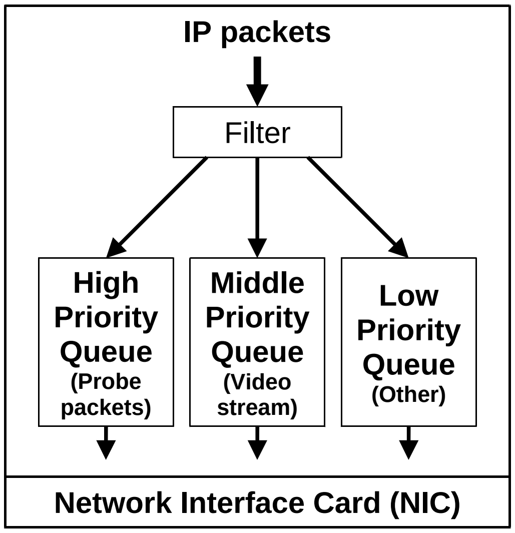

IP packets
Filter
High Priority Queue (Probe packets)
Middle Priority Queue (Video stream)
Low Priority Queue (Other)
Network Interface Card (NIC)

Figure 4.5: Architecture using 3 priority queues for traffic shaping

In this solution we first have two priorities, in order for us to prioritize the probe packets over the video stream and other packets. This allows the probe packets to be transmitted when the need to, without disturbance from the video stream or the other traffic. Then the video stream and other traffic is separated into two different buckets, with a high transmission rate set for the video stream bucket, and a low rate set for the other traffic. In this setup the video stream will be on the same priority as the other traffic and therefore cannot starve it.

Again we measure the round-trip times to see evaluate the performance of our architecture. We can see the measured RTT in figures 4.7 and 4.8, for ease of comparison we also included the RTT for standard wireless Ethernet in the figures. The two figures show the RTT results for two different payloads, 0.375 and 3.0Mbps respectively.

As we can see, with our architecture the RTT stays fairly constant, and around 13 microseconds, while for the standard Ethernet it can it varies all the way up to $3000 - 4000$ microseconds.

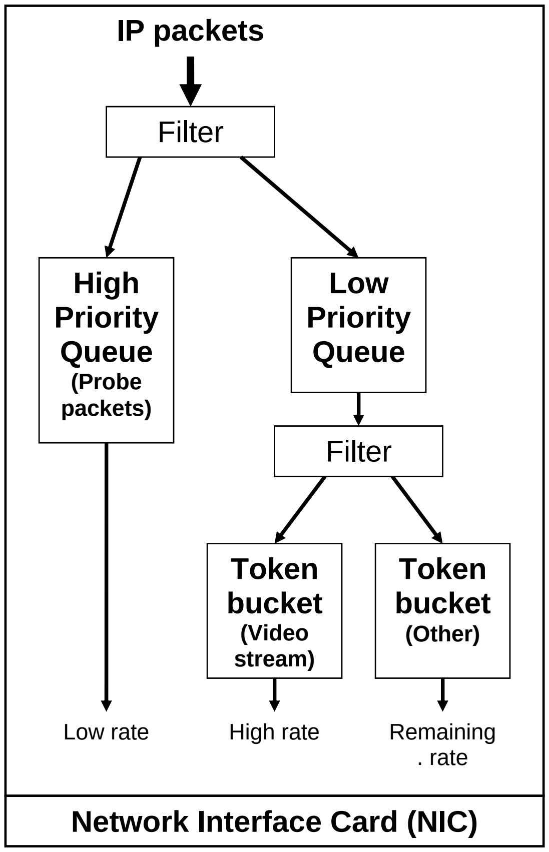


Figure 4.6: Architecture using a combination of priority queues and token buckets for traffic shaping

Clearly, using our architecture we will get a much lover latency than what standard Ethernet can provide, and thus, a high likelihood of the timely arrival of the streaming packets at the receiver which is a goal we set out to accomplish.

We implemented the traffic shaping architecture using features within the Linux 2.6 kernel, more details are available in Appendix A.

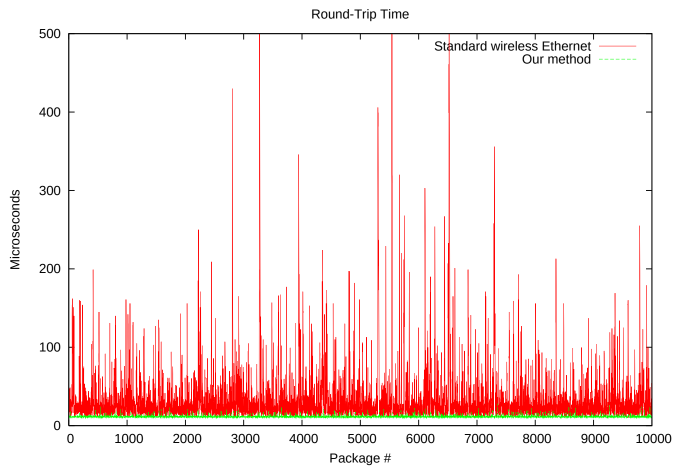


Figure 4.7: Round-Trip Times using our method compared to standard wireless Ethernet, 0.375Mbps of payload.

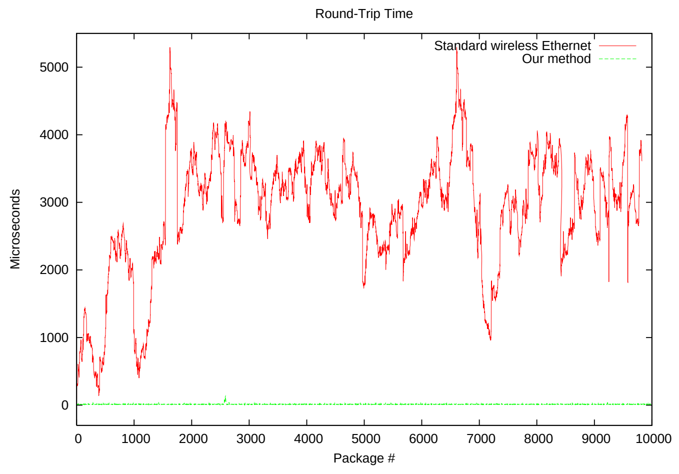


Figure 4.8: Round-Trip Times using our method compared to standard wireless Ethernet, 3.0Mbps of payload.

4.4 Streaming

Streaming of video and audio is the major reason for having an in-home entertainment system. Video and audio stream will consume most of the resources of the system, of course there are other data streams, such as control information, and normal computer usage data, i.e. web browsing, ftp transfers, telnet sessions, and so on, present as well.

For the work presented in this thesis we have focused on the streaming on video.

In our in-home entertainment system we give priority to the video streams over any other type of data, with the exception of the probe packets used for bandwidth prediction. We prioritize the transmission of the video stream packets from the nodes, and if possible we also give nodes that will stream video a large enough network bandwidth share to accomplish the streaming. We want to have as good a result as possible for the video stream.

There are many issues related to streaming that is not the focus of our work, and we will not discuss it in this thesis, but we will mention a very important issue. It is the issue of which network communication protocol to use for streaming? We have looked at both TCP/IP and UDP/IP as possible options for our streaming applications. Both options has their share of advantages and disadvantages, an in the table below we describe them in more detail:

TCP/IP The Transmission Control Protocol (TCP) is a connection oriented protocol, intend to be used over a unreliable network. TCP provides a reliable end-to-end delivery of packets through the use of retransmissions of lost packets.

From a streaming point of view this is not suitable, retransmissions will cause longer delays of packets, which also transfers to delays in the video. Buffering is a method to avoid delays, the problem is that it is not trivial to determine a suitable size for the buffer in order to avoid the delays. And, for live video streaming buffering is not an option, because delays are not tolerated at all.

TCP video streaming will result in a good picture quality, because no packets are lost, but retransmissions can result in delays for the displayed video.

UDP/IP The User Data Protocol (UDP) is a connectionless protocol that provide a simple datagram delivery with no guarantees of packet delivery at all. UDP is a simple protocol compared to TCP, it add a small 8-byte header to a raw IP-datagram.

The advantage of using UDP for streaming is its simplicity, there is almost no overhead added by the protocol, not adding to any packet delays. The drawback is that there are no guarantees for a reliable packet delivery, forcing the applications to be capable of handling data losses.

UDP video streaming can result in a bad picture quality, because packets can be lost, but the video will be displayed without any delay.

The choice of which protocol to use, TCP or UDP, boils down to a chaise between picture quality and delay. This depends on the type of video that is streamed, live or a DVD movie, and also on the device that will display the video. A small handheld device might not have enough memory to handle the buffering needed when using TCP, so UDP becomes the only viable choice. End user preference is of course also important, i.e. do the user prefer a bad picture quality or delays?

The reason we choose to use UDP for our streaming applications, is because of the cases when we are dealing with live video streams. We do not want the communication protocol to create delays by retransmitting packets. Furthermore, if TV is broadcasted, it has a TV tableau with fixed times for when each program i showed, if the communication protocol delay the program, it will no longer be according to the TV tableau.

In the following sections we present two different video streaming applications that we have developed. The exact details of how they differ is presented in each section of the applications, but what they have in common is the use of UDP for the streaming.

4.4.1 Two Streaming Application Architectures

In this section we present two applications that uses the packet scheduling architecture we presented in above. Both applications are aware of the fact that the network bandwidth is fluctuating, i.e, both applications adapts the rate of sending data.

The applications are two different variants of a streaming server, i.e they both stream MPEG-2 video to a receiver. The difference lies in the sources they have for the MPEG-2 video, the first application streams the video from file, and the second application streams the video from a web camera. Thus, both the applications adapts the MPEG-2 stream to the fluctuating bandwidth in different ways.

Furthermore, we also present a Middleware used for network communication, which we modified to accommodate our plug-in architecture, presented in section 3.4. The Middleware schedule the data transmission using a fixed scheduling algorithm, but our plug-in architecture now allows us to implement and insert any algorithm we desire.

4.4.2 MPEG-2

The Moving Picture Coding Experts Group (MPEG) [50] was established in January 1988 with the mandate to develop standards for coded representation of moving pictures, audio and their combination. It is a working group of ISO, the International Organization for Standardization.

MPEG is an encoding and compression system for digital multimedia content. To understand why video compression is so important, one has to consider the vast bandwidth required to transmit uncompressed digital movie. The basic idea is to transform a stream of discrete samples into a bit-stream of tokens, which takes less space, but is just as filling to the eye or ear (perceptive coding).

A MPEG-2 video stream consists of picture frames, of which there are three different types: I-frame, P-frame, and B-frame.

I-frame (Intra coded)I-frames contains a compete picture, very similar to a JPEG picture, which can be decoded into a picture without information from any other frame.

P-frame (Predicted)P-frames needs information from a previous I-frame when being decoded into a picture.

B-frame (Bi-directional)B-frames needs information from both a previous frame and a future frame in order to be decoded into a picture.

Group-Of-Picture (GOP) is a collection of frames of all types of arbitrary length, the GOP starts with an I-frame and is followed by a sequence of P- and B-frames.

4.4.3 Video Stream Server

The first application is a streaming server capable of dynamically switching between a small number of differently sized versions of the same MPEG-2 stream, as described in [57]. Synchronization of the streams is based on *Groups Of Pictures* (GOP), i.e. a switching between streams always occurs at a the start of a GOP. The server switches the transmission between these streams according to the currently available bandwidth predicted by our method.

Application Details

For the video streamer, we must prepare different version of a MPEG-2 video stream before the streaming starts. These different versions of the video only differ in the bit rate, there is no changes, additions, or removal of headers.

The number of different streams created and used depends on the granularity of switching that is wanted. If many streams are used, with only a small difference in bit rate, a lot of switching will probably occur, and vice versa.

In our solution we use a small number of different sized versions of the same stream, we consider bandwidth to also be divided into a small set of levels, as described in [57], how bandwidth are mapped into these bounds are not within the scope of this thesis.

The server parses these streams and sends data in chunks only from the stream currently selected for transmission to the receiver. At the same time the server also parses all different versions of the streams on a GOP by GOP basis.

If there is an indication of enough bandwidth change to warrant a stream switch, and depending on the predicted bandwidth, the server needs to determine which stream version to switch too.

Since we switch streams only at GOPs starts, an if, as is the worst case, the order to switch comes 1 bit after the GOP header has been transmitted, we have to discard all data until the next GOP start. This will result in a lot of data, a GOP can be half a second of video, being lost, which will result in poor picture quality at the receiver.

4.4.4 Web-cam Server

The second application, developed by Enrico Viero and Daniel Lundberg [3], is also a streaming server, but it uses a web camera to capture pictures, which are then encoded into a video stream with the possibility to control the stream size by adjusting a quality parameter of the encoding. This encoding is accomplished by the JPEGConverter program[4] The quality parameter is determined based on the bandwidth prediction we perform. This stream is then transmitted onto the network adapted so that it fits with the available bandwidth.

JPEG

JPEG is a standardized image compression mechanism [36], and stands for Joint Photographic Experts Group, after the original name of the committee that wrote the standard. JPEG is designed for compressing either full-color or gray-scale digital images of "natural", real-world

[3]As part of their Master Thesis.

[4]JPEGConverter was kindly provided by Joe Woelfel and Chia Shen at Mitsubishi Electric Research Lab (MERL).

scenes, it does not work very well on non-realistic images, such as cartoons or line drawings.

JPEG does not handle black-and-white (1-bit-per-pixel) images, nor does it handle motion picture compression, where MPEG is used instead.

Regular JPEG is "lossy", meaning that the image you get out of decompression isn’t quite identical to what you originally put in. The algorithm achieves much of its compression by exploiting known limitations of the human eye, notably the fact that small color details aren’t perceived as well as small details of light-and-dark.

Application Details

The web camera application works in three steps, first it communicates with the camera, grabs a image in bitmap format, 24 times per second. Second, it converts the grabbed bitmap into a JPEG picture, in which we can tune the size (quality) of the picture using a parameter in the JPEG encoder, from the Independent JPEG group [31]. The parameter can be set in a range between 0 to 100, where 100 represents lossless compression. In the third step, the JPEG picture is converted to an MPEG-2 frame, using the Berkley MPEG encoder [13], frames are then collected in a buffer, up to a certain data chunk size, which is then sent to the receiver. The server only creates I-frames, not P- or B-frames, in order to simplify the whole encoding process.

The JPEG encoding step is necessary in order to dynamically adapt and change the size of the pictures, which is something the MPEG encoder is not capable of.

Depending on the available bandwidth we decide the parameter value when encoding the next JPEG picture. Again, finding an optimal solution for the available bandwidth to size parameter match is not within the scope of this thesis.

The web camera server can be "faster" in its response to bandwidth change, if the buffer being transmitted is small, say one frame, the "new" frame, encoded with a different size, will almost immediately be transmitted.

4.5 Chapter Summary

In this chapter we presented an architecture that provides QoS for video streaming, using wireless Ethernet, within the in-home entertainment system. To achieve a better QoS than standard Ethernet we adapt the transmission rate according to how much bandwidth that currently is available on the network. Each node within the in-home entertainment system is assigned a proportional share of the available bandwidth. In our system the streaming serve gets th highest share of all nodes since we consider it to be very important that streaming can occur without problems.

The architecture we propose has the goal of improving these two QoS parameter, which we define as QoS for the wireless network in this thesis:

- A more reliable delivery of streaming packets.
- A lower average latency for the streaming packets.

Conceptually we try to answer the following question: can we continue to transmit at the current rate? The architecture comes up with an answer to this simple question and takes the appropriate action, depending on the answer it can lower or continue with, or maybe even increase, the transmission rate.

In order to answer the question, the architecture performs two tasks. First, it predicts the future available bandwidth using probe packets to measure the currently available bandwidth, and the history of previous predictions. This prediction is performed periodically.

The results of each prediction, a bit/s answer, is then fed into the second part, the traffic shaper. The traffic shaper is responsible for changing the transmission rate of the node, and does so by changing Linux QoS kernel parameters to set the new maximum allowed rate.

The kernel level adjustment takes place at a very low level, it is positioned so it can shape all IP packets that will be transmitted from the node. At this level the granularity of control is very fine, the traffic shaper works on a bit level.

The problem is that this is not enough, if the streaming application continues to stream at a previous level, there is a chance of internal buffer overflow. Thus, the application must also be aware of bandwidth fluctuation, and have the possibility to adapt its streaming rate according to the bandwidth prediction.

We provide to example streaming applications that adapts the transmission rates, according to the available bandwidth prediction, in two different ways. Both applications stream MPEG-2 video, the first one switches between different size version of the same stream when ordered to change the transmission rate. The other application is a live video streaming web cam, where the size of the resulting video stream can be set with a parameter, which is changed when the transmission rate needs to be adjusted.

Finally, in order to evaluate our architecture we performed a set of simulations aimed at verifying an increased QoS compared to what the standard wireless Ethernet can provide.

We have seen that the round-trip times we get when using our architecture is basically the same with or without crosstraffic, which is around 14 microseconds. For the standard Ethernet, the round trip times varies alot as the crosstraffic increases, with times up to around 4000 microseconds.

This good result is because the crosstraffic is controlled by our traffic shaper, and adjusted to what is available and to what bandwidth share the node is assigned.

To overhead for perform the bandwidth prediction, i.e., packet probing and calculation, is on average around 1.7 milliseconds, which is a good result considering the code is not optimized at all. And finally, the overhead for changing the traffic shaper parameters is on average 38 milliseconds.

Chapter 5

Conclusions

In this thesis we have presented various solutions for a in-home entertainment system. Within these type of systems, different consumer electronic (CE) devices are interconnected using a wireless network. The idea is that multimedia content could be shared between any device in a transparent way, i.e., it should be possible to view the DVD movie, streamed from the DVD player, on any device, such as a handheld PC.

In order to accomplish this, the system needs to adapt to the varying resource demands in the constrained real-time devices, i.e., the DVD movie needs to be adapted in order for the pocket PC to display the video.

In chapter 2, we presented the constrained real-time devices which we are interested in. Processors comes in a variety of classes, from powerful desktop PC processors to less powerful processors used in handheld computers. And operating systems (OS), which run on the processors and are responsible for managing the life-cycle of the tasks, i.e., including the scheduling of tasks.

OSs can be divided into two classes; real-time operating systems (RTOS) and general-purpose operating systems (GPOS), both with different goals in mind. A RTOS is intended to be used in real-time systems, where timeliness is as important as correct functionality. GPOSs, on the other hand, are more focused on providing correct functionality and a fair share of the processor time to all tasks (threads) running in the system. Using GPOSs in real-time systems is problematic since they were not designed with real-time as a goal, typical problems come from lack of control of all resources and tasks leading to problems with blocking times, priority inversion, and so on.

The second resource we presented was both wired and wireless Ethernet. Using Ethernet for real-time communication is problematic, since it was not intended to provide real-time guarantees. Network congestion is a common problem that occurs if the network is overloaded with traffic. Congestion leads to collisions and re-transmissions, which in turn leads to more collisions, and so on. In the worst case, congestion leads to packets drops.

Wireless Ethernet is similar to the wired variant, but, in addition has the problem of being sensitive to interference. Interference is the phenomenon where the radio signal is disturbed, by itself, physical effects (walls, movement, ...), and other devices (Microwave owens, Bluetooth devices, cordless phones, ...). Thus, the reliability is even lower for wireless- compared to a wired Ethernet.

In chapter 3 we presented two theoretical and one practical solution for processor scheduling.

We presented a solution for handling soft aperiodic tasks in off-line scheduled systems. The method allocates processor bandwidth in the off-line phase of scheduling, which is then supposed to be used by a soft aperiodic task scheduling server, such as the Total Bandwidth Server. This allows for a more efficient handling than originally proposed in slot shifting.

The second theoretical solution deals with overload in a off-line scheduled distributed system. Overload must be removed without disturbing the execution of the time-triggered tasks which has hard real-time guarantees. Thus, the method removes the overload by removing firm aperiodic tasks. The algorithm uses task values to compute the set of tasks that is most appropriate to remove in order to solve the situation. Removed tasks can be reinserted later for execution, if there are more resources available due to execution being less than WCET, or they can be migrated to other nodes for possible execution there.

The final solution within processor scheduling deals with a common real-time operating system problem. Most RTOSs uses a fixed scheduling algorithm, such as fixed priority, which is highly intertwined with the rest of the OS kernel in order to have efficient overhead. The problem is that all applications written for that OS must be tailored to the specific scheduling paradigm, even though it might not be suitable at all. We propose a way to disentangle the scheduling algorithm from the rest of the kernel by implementing a plug-in module. The plug-in module has a Small and simple interface allowing easy implementation and insertion of any scheduling algorithm. Thus, allowing the OS to be tailored to the applications and not the opposite.

In chapter 4 we propose an architecture to deal with the unreliability of the wireless network.

Our bandwidth prediction is based on a well known, and used, method: packet probing. We use packet pair probing to gather information about the current state of the network, i.e., the available bandwidth can be derived from the probe packets. This is used in an exponential averaging to predict the future available bandwidth, including both the result from the probe packets and the history of previous prediction. The result, which indicate the future available bandwidth, is used fed into the second part, the traffic shaper.

our traffic shaper works on the assumption that we have three classes of traffic; probe packets, video streams, and other traffic. Probe packets are the two packets we use for the bandwidth prediction, and they need as low latencies as possible when being transmitted because we want the measurement to be as fresh as possible. Secondly, the video stream is the main form of data in the in-home entertainment system. Other traffic is a collection class for all other kinds of traffic, such as http, ftp, telnet, and so on.

Probe packets have the highest priority for transmissions, then both the video stream and other have the same priority. The video stream ant the other traffic is differentiated by the data transmission rates they are allowed to use, the video stream gets a high rate, and other traffic gets the remainder of what is available.

Appendix A

Implementation Details and Measurement Results

In this Appendix we present simulation results for the overload scheduling algorithm, implementation details the network packet scheduler, and the results from the measurements we performed in order to evaluate the network packet scheduling.

The network packet scheduling is implemented in Linux, and uses built in kernel QoS features. The measurements we performed for the network packet scheduling, i.e., bandwidth prediction and traffic shaping, were performed using a wireless network.

A.1 Overload Handling Results

We have implemented the described method, and have run simulations for various scenarios. The simulated system consists of 8 processing nodes, connected via a network where all necessary messages can be sent during one time slot.

Each simulation has a length of 2000 slots. The offline schedules are created from randomly generated precedence graphs, an offline scheduler transforms the precedence graphs to offline schedules. Each node has one offline schedule with a load of 0.4 and a length between 300 and 1000 slots.

Worst case computation time for both offline and aperiodic tasks varies uniformly in the range 1–10. Aperiodic tasks are assigned an actual execution time uniformly distributed between 0.5 and 1.0 of its WCET, and relative deadlines varying between 1–3 times WCET.

Arrival times of aperiodic tasks are distributed over the simulation length, with the restriction that no task have a deadline exceeding the simulation length. Finally, values of aperiodic tasks vary uniformly in the range 1–100.

The average node load varies between 0.8 and 3.0, the offline load of 0.4 included. The load parameter is based on WCET, and thus represents the load as perceived by the overload algorithm. The actual system load is lower[1], since execution time is less than WCET.

A.1.1 Experiment 1: Method Comparison

We have studied the total accumulated value of aperiodic tasks that finished in time, and the following methods have been compared:

1. The full method presented in the paper (*Migration*).
2. The overload handling algorithm, without task migration (*Local*).
3. A basic algorithm that uses the offline schedule, assigning idle slots to aperiodic tasks based on value density (*Offline Valuedensity*).
4. Same as 3, but aperiodic tasks are ordered by value (*Offline Value*).
5. Same as 3, but aperiodic tasks are ordered according to EDF (*Offline EDF*).
6. Same as 3, but aperiodic tasks are serviced in order of arrival. (*Offline FCFS*).

[1]The actual system load varies approximately between 0.7 and 2.35 in the experiments, based on the distribution of actual execution times

Methods 1 and 2 implement the efficiency improvements suggested in 3. Each point in the figures represents some 300 simulations.

In the first part of the experiment, all nodes in the system are subject to the same amount of load. The result is presented in Figure A.1. Here, the possibility of task migration does not provide any significant improvement. Compared to the basic method, the performance of the proposed method is significantly higher.

The second part of the experiment, shown in Figure A.2, is a scenario of unevenly distributed load. Half of the nodes have no aperiodic tasks arriving, only offline scheduled tasks. Here, the task migration algorithm clearly increases the system performance, compared to overload handling without migration, because tasks can migrate to nodes with no aperiodic load.

A.1.2 Experiment 2: Restrictions

The theoretical worst case time complexity of the overload algorithm, for a ready queue of length n, is $O(n^2)$. This experiment shows how the execution time is affected by system load, and the impact on performance from restricting the algorithm as suggested in 3 to deal with complexity issues.

The parameter *cutoff* denotes the maximum length of the ready queue. Tasks that are inserted at a position greater than *cutoff* are automatically rejected, which means that they are placed in the maybe-later queue (if they just arrived, or if they were in the ready queue during the previous slot), or not stolen (if they were from a maybe-later queue).

We have measured the total accumulated value of aperiodic tasks that finished in time (similar to experiment 1) for different *cutoff* values. Execution time has been approximated by the number of arithmetic, comparison and assignment operation performed in the overload algorithm, including the computation of σ-values.

The parameters are the same as in experiment 1, with the load evenly distributed over the nodes, and using the full method from the paper (*Migration*). In Figure A.3, the average number of operations for a single call to the overload algorithm is presented. Figure A.4 gives the

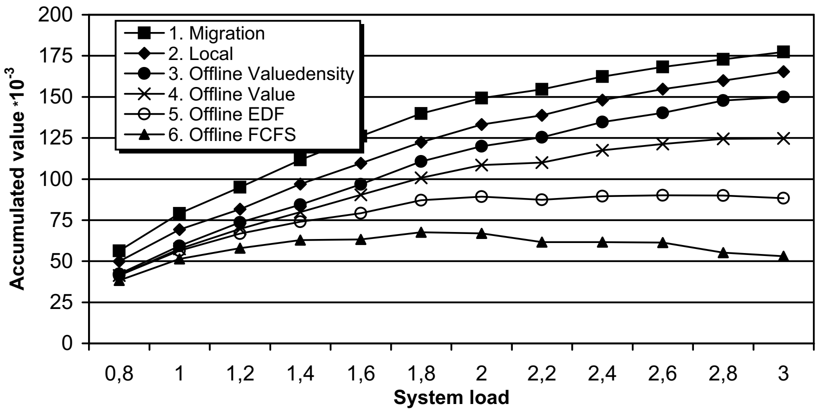


Figure A.1: Accumulated value for even load distribution.

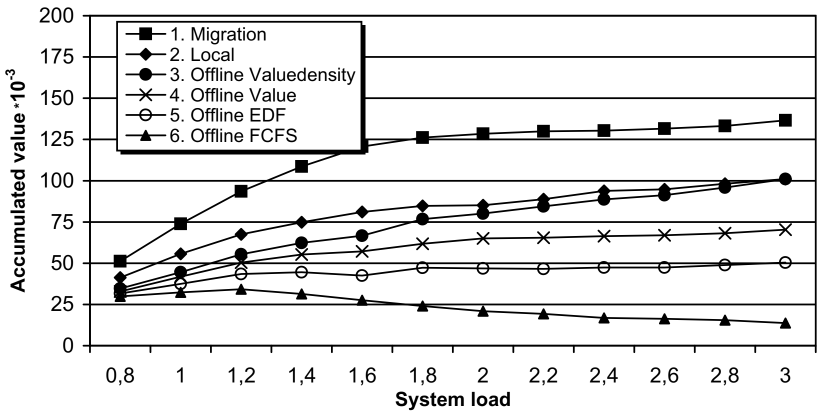


Figure A.2: Accumulated value for uneven load distribution.

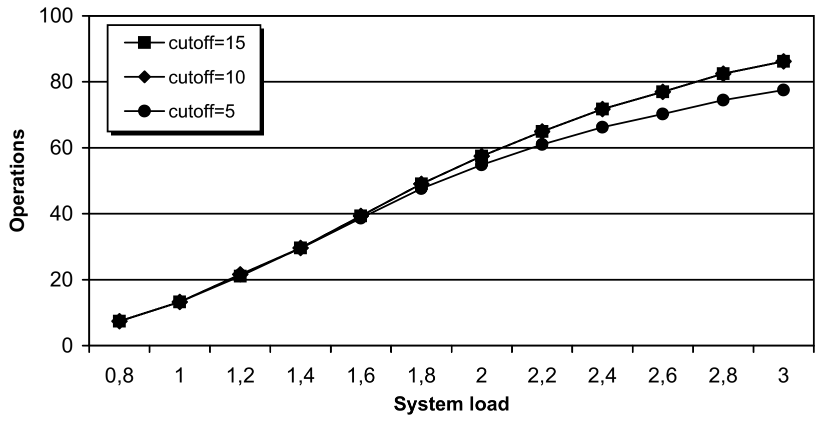


Figure A.3: Average number of operations for different *cutoff* values.

maximum number of operations performed during a single call to the overload algorithm. Each point in the figures represents some 300 simulations. Thus, in figures A.3 and A.4, each point represents over 4 million calls to the overload algorithm (8 nodes, and a simulation length of 2000).

In practice, the execution time is not as big an issue as the theoretical complexity suggests. None of the 57 million calls to the overload algorithm made during simulations needed more than 720 operations to be performed.

This is partly because the ready queue size (which is the parameter used in the complexity analysis) is not proportional to system load. Also, the worst case assumes that none of the restrictions are trivially solved by the solution to the previous ones, which is highly unlikely when the queue is long.

The simulations show that restricting the length of the ready queue significantly reduces worst case execution time, with only a moderate performance decrease.

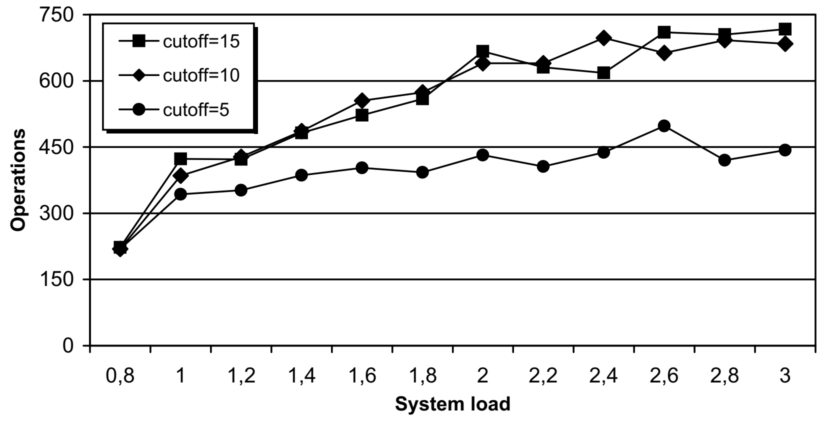


Figure A.4: Maximum number of operations for different *cutoff* values.

A.1.3 Algorithm for Computing Overload Amount

Given the deadlines and remaining execution times of the aperiodic tasks, and the spare capacity (slots not reserved for offline scheduled tasks) of consecutive intervals, this algorithm computes the overload amount of each aperiodic task.

Let $\tau_1 \ldots \tau_n$ be a sequence of aperiodic tasks sorted by increasing deadline. Also, assume a sequence of consecutive, non-empty, time intervals, each associated to a number of offline scheduled tasks as defined by the slot shifting algorithm [27]. The following additional notation is used in the algorithm.

dl_x	the deadline of τ_x
c_x	the remaining execution time of τ_x
end_x	the end time of interval number x
sc_x	the spare capacity of interval number x
oa_x	will be assigned the overload amount of τ_x

Algorithm

Let ct be the current time, and ci the number of the interval that ct belongs to. Further, assign $oa_1 := c_1$. If the algorithm is called with compute-oa$(ct, 1, ci, sc_{ci})$, then oa contains the overload values for τ_1 to τ_n, upon termination.

```
**function** compute-oa$(t, d, i, c)$
**if** $d \leq n$ **then**
    **if** $dl_d < end_i$ **then**
        $tmp :=$ min$(c, dl_d - t)$
        $oa_d := oa_d - tmp$
        **if** $d < n$ **then** $oa_{d+1} := oa_d + c_{d+1}$
        compute-oa$(dl_d, d + 1, i, c - tmp)$
    **else**
        $oa_d := oa_d - c$
        compute-oa$(end_i, d, i + 1, sc_{i+1})$
```

Note that the function is tail-recursive and thus can be implemented with bounded memory, e.g., as a standard imperative loop.

Complexity

Before considering the complexity of the algorithm, we formulate an invariant, i.e., a proposition that is true every time the function is called. For this, we define $in(x)$ to be the number of the interval containing the time x. This allow us to formulate the invariant as $i \leq in(dl_d)$.

The correctness of the invariant is proven as follows. For the initial call to the function, we have $i = ci \leq in(dl_1)$ since no task in the sequence has already violated its deadline. Next, we assume that the invariant holds for one call, and show that this implies that it must hold for the next recursive call as well.

If the first branch of the if-then-else statement is selected, i is unchanged and $d$is increased by one in the next recursive call. Since $in(dl_d) < in(dl_{d+1})$, and since $i \leq in(dl_d)$ by assumption, we have

$i \leq in(dl_{d+1})$ so the invariant holds for the next call as well.

If, instead, the else branch is selected, we must have $dl_d \geq end_i$. Assume further that the invariant does not hold for the next call. Than, since it holds for the current call, we must have $i = in(dl_d)$. This implies that $end_i \geq dl_i$, which leads to a contradiction and thus proves that the invariant must hold for the next call.

By induction, we have now shown that the invariant holds each time the function is called.

Since we have $d \leq n$, the invariant implies $i \leq in(dl_n)$. Also, we know that d and i are never decreased, that one of them is increased in each recursive call, and that they are initialized to 1 and ci respectively. This implies that the total number of calls to the function can be no more than $n + m$, where m is the number of intervals between the current time, and the deadline of τ_n. Thus, the worst case time complexity of the algorithm is in $O(n + m)$.

A.2 Bandwidth Prediction in Linux

We implemented the bandwidth predictor as a user level application running on Linux. The bandwidth predictor uses UDP to transmit the two probe packets, i.e., it is a connectionless transmission.

In order to catch the two probe packets on the receiver side, we use the *libpcap* [65] library. Libpcap allows us to catch the IP-packets arriving to the node. A time stamp is taken at the arrival of each of the probe packets, the difference is calculated, then sent back to the sender node, using a UDP packet. The time difference between the two probe packets occurs due to network induced delays, i.e., due to crosstraffic. By catching the probe IP-packets we avoid UDP protocol overhead in the time delay (even if it it low). The time difference is used to calculate the predicted bandwidth, as we described in 4.3, for ease of reference, we show the equation we use, again:

$$BWT = L/\Delta_T \tag{A.1}$$

Where L is the length of the probe packets (in bytes), Δ_T is the time difference of the probe packets, and BWT, is the resulting measurement of the available bandwidth. But, as described in 4.3, this is not enough for our bandwidth prediction, we also want to include the history of predictions.

Thus, we use the following formula to calculate the predicted bandwidth:

$$P_k = \alpha BWT_k + (1 - \alpha)P_{k-1} \tag{A.2}$$

BWT_k is the bandwidth measurement we just performed (from equation A.1, α is a weight, and is a number between $0 - 1$, that is used to determine how much of the current bandwidth measurement, and how much of the history of predictions we use when calculating the current prediction, P_k. This result, P_k is used to update the traffic shaper.

A.3 Traffic Shaping in Linux

We implemented our traffic shaper in Linux, using features built into kernel. Linux has very advanced routing, filtering, and traffic shaping options, which has been present in various forms since the 2.2 version of the kernel. These features where intended to be used in various ways, according to [45]:

- Throttle bandwidth for certain computers.
- Throttle bandwidth to certain computers.
- Help to fairly share bandwidth.
- Restrict access to computers.
- Do routing based on user id, MAC address, source IP address, port number, type of service, ...

We are interested in the filtering and traffic shaping capabilities of the Linux kernel.

A.3.1 Bandwidth Management

To control the way in which data is sent, i.e., bandwidth management, Linux contains methods called queuing disciplines. It is much more difficult to control the data being received, so within Linux most queuing disciplines work only for sending data. Each network device has a queuing discipline attached to it. This makes it possible to simultaneously perform different traffic control on different devices.

There are many queuing disciplines (called qdiscs) available in the kernel, but they can be classified into two types., *classless* and *classfull*.

classless qdiscs basically accepts packets, then only reschedules, delays, or drops them. A token bucket is an example of a classless queue, since it only delays or drops the packets it receives.

The other type is the *classful* qdiscs, which is useful if different kinds of traffic should receive different treatment. Within classful qdiscs packets are "filtered" based on one or several conditions, for example, source or destination port, source or destination IP address, protocol type, and so on. Filtering allows packets to be separated, and put into different qdiscs, which uses different methods for further processing of packets.

A.3.2 Our Implementation

We use the traffic control (tc) tool [68], in order to set up our traffic shaping architecture. As presented in 4 we want to prioritize the probe packets, give the video stream a high transmission rate while not starving any other traffic. Figure A.5 show the resulting architecture within the Linux kernel.

Tc is used as a QoS control too within Linux, and it acts as an interface for both setting and changing the traffic shaping parameters within kernel. For us to set up the traffic shaping architecture we follow the guidelines on [45].

First we use a priority qdisc in order to prioritize the probe packets. Then, we add two hierarchical token buckets (HTB) in order to separate the video stream from the rest of the traffic (called other). Filters (in our case filtering on destination ports) are used to control the flow of packets into the correct qdisc.

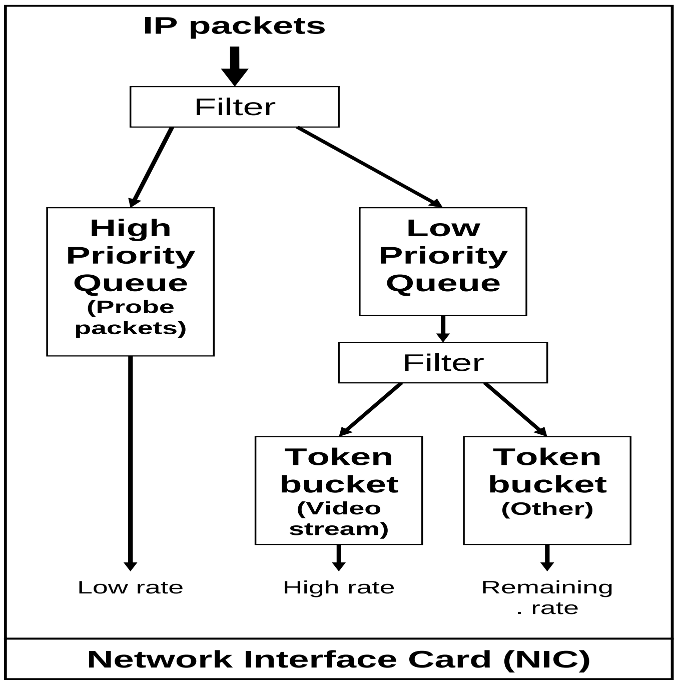


Figure A.5: Architecture using a combination of priority queues and token buckets for traffic shaping

A.4 Network Packet Scheduling Results

In order to evaluate the network packet scheduling architecture we proposed in 4, we need to see how it performs in terms of the network QoS requirements, which we defined in 4.1, compared to standard wireless Ethernet. First we present the results of the bandwidth prediction, then the results for the packet loss, and finally latency results for the video stream packets.

For ease of reference we repeat the network QoS requirements we defined earlier:

- A more reliable delivery of streaming packets.
- A lower average latency for the streaming packets.

First, to measure the reliable delivery of packets we try to stream packets to simulate a video stream, and count how many of those packets that manage to reach the receiver during various scenarios. Secondly to measure the latency experienced by a message we measure the round-trip time of all the messages that simulate a video stream.

In order to perform the measurements described above we have set up a Infrastructure wireless network. It is a IEEE 802.11b, 11 Mbit/s network, all nodes are connected using the Infrastructure method, i.e., all packets pass through the access point (AP). The nodes are: one 600 MHz, Pentium III laptop (P3-laptop), and a 233 MHz, Pentium II laptop (P2-laptop), and one 350 MHz, Pentium II desktop PC (P2-desktop). The P3-laptop acts as the streaming server, and the P2-desktop acts as the receiver of the stream. The P2-laptop is used to generate crosstraffic in the network, which is sent to the P3-desktop.

To simulate various demands of the video stream, we use four different traffic rates: 0.375, 0.75, 1.5, and 3.0 Mbit/s.

Crosstraffic is generated as a constant stream with four different rates: 0.5, 1.0, 2.0, and 3.0 Mbit/s.

A.4.1 Bandwidth Prediction

In this section we present the results we get from our bandwidth prediction method. We already know [72] that the average bandwidth we could expect from a 11 Mbps wireless network, which due to protocol overhead only is around 3 - 4 Mbps, with possible peaks up to $6 - 7$ Mbps.

Furthermore, we are also interested in the prediction results we get when we use different prediction intervals. The four intervals we test are: 0.2, 0.5, 1.0, and 2.0 seconds.

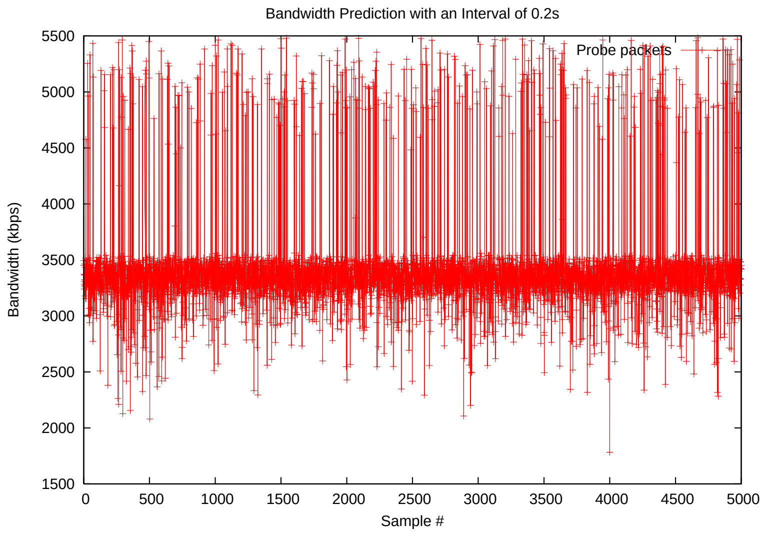


Figure A.6: Bandwidth prediction results using a 0.2s interval.

For this experiment we use the P3-laptop and the P2-desktop described above, where the P3-laptop acts as the sender and the P2-desktop as receiver. We perform 5000 predictions, where each one includes sending two probe packets and calculating the available bandwidth.

The results we get is shown in figures A.6, A.7, A.8, and A.9.

As we see in the figures, the interval does not have a big impact on the bandwidth prediction result. We can also see that the average bandwidth which we predict is around 3.5 Mbps, which is as expected.

We also perform experiments on order to evaluate the effect different α values has on the bandwidth prediction result. The second reason is the weight factor we use in our bandwidth prediction formula, called α, which determines how much of the current measurement vs. how much of the previous predictions we should use for the current prediction. For these experiments alpha was set to be 0.5, so we look equally at the current prediction and the history of previous predictions.

For ease of reference, we show the bandwidth prediction formula

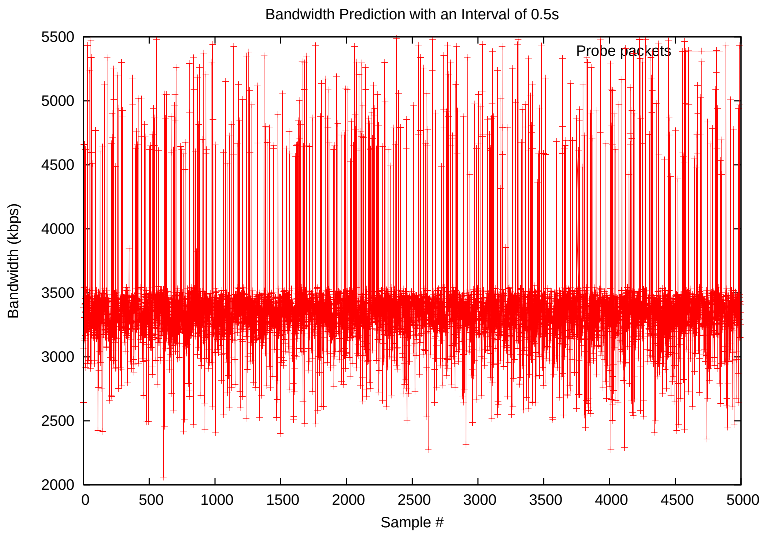


Figure A.7: Bandwidth prediction results using a 0.5s interval.

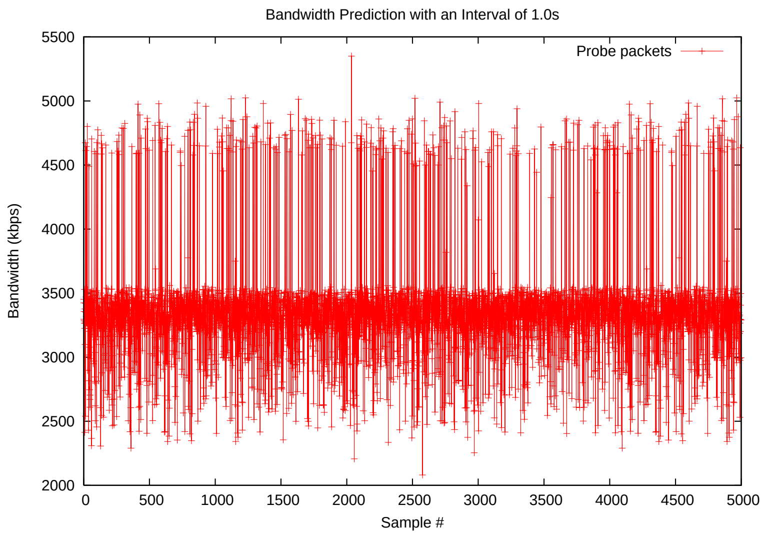


Figure A.8: Bandwidth prediction results using a 1.0s interval.

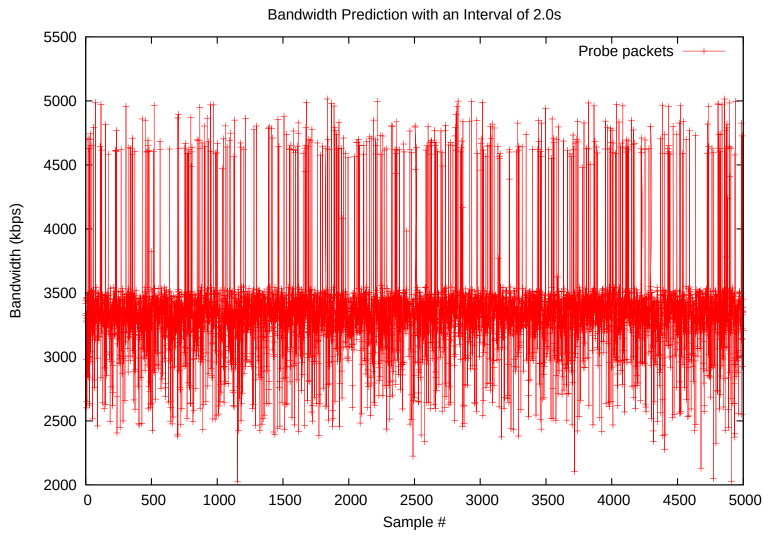


Figure A.9: Bandwidth prediction results using a 2.0s interval.

again:

$$P_k = \alpha BWT_k + (1 - \alpha)P_{k-1} \tag{A.3}$$

If α is low, i.e., closer to 0, we will use more of the previous predictions in our calculation which has the result that we react very slowly to any changes in bandwidth. Thus, the predictions will be quite similar to the previous prediction, unless the measured bandwidth is extremely high or low, and therefore it is a more "careful" prediction, i.e., it does not react extremely to extremely high or low bandwidth measurements. Thus, since we started from 0 for these experiments, the predictions will slowly If we, on the other hand, have a high α, the results will give an even more fluctuating predictions, since we care mostly about the measurements and not the history.

We performed a series of experiments with different α values in order to determine which value would be the most appropriate to use. The different α values we measure are: 0.0, 0.25, 0.5, 0.75, and 1.0.

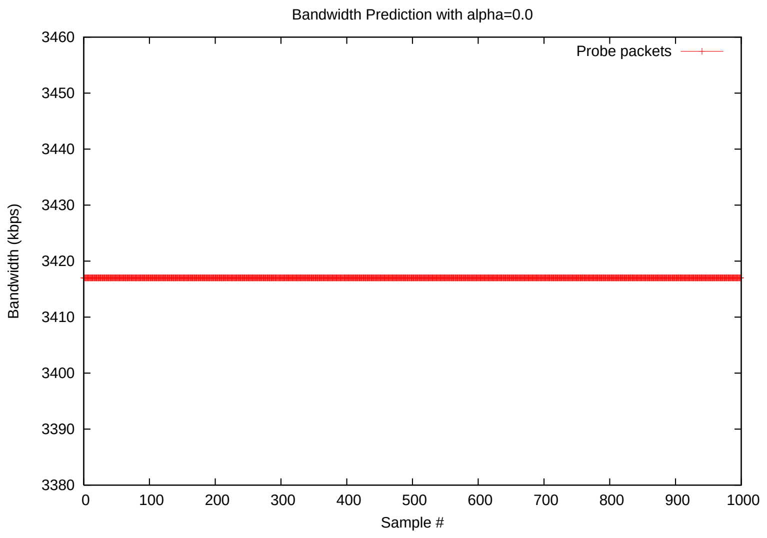


Figure A.10: Bandwidth prediction results an alpha value of 0.0.

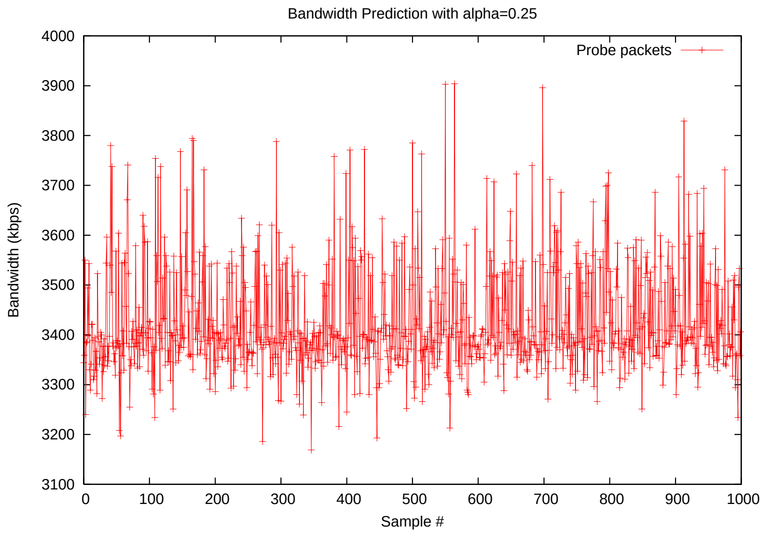


Figure A.11: Bandwidth prediction results an alpha value of 0.25.

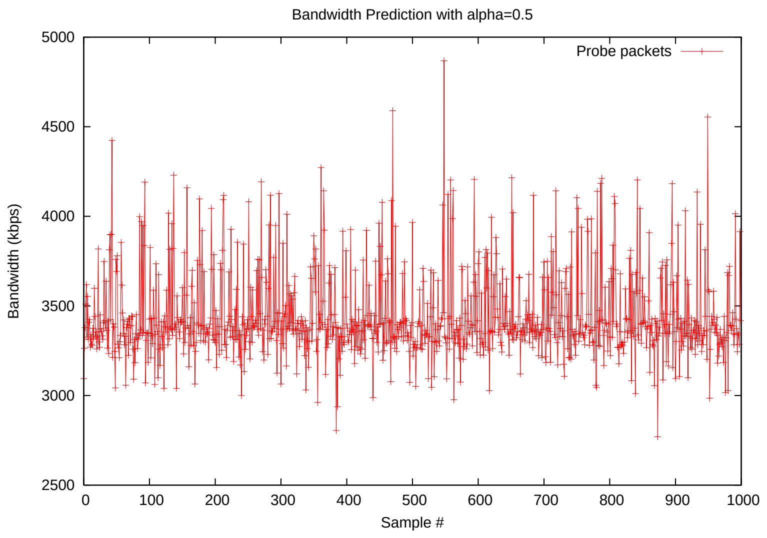


Figure A.12: Bandwidth prediction results an alpha value of 0.5.

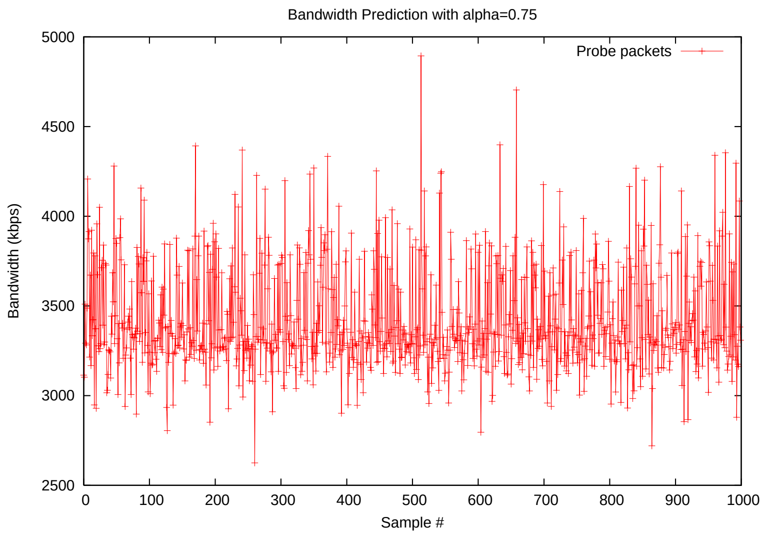


Figure A.13: Bandwidth prediction results an alpha value of 0.75.

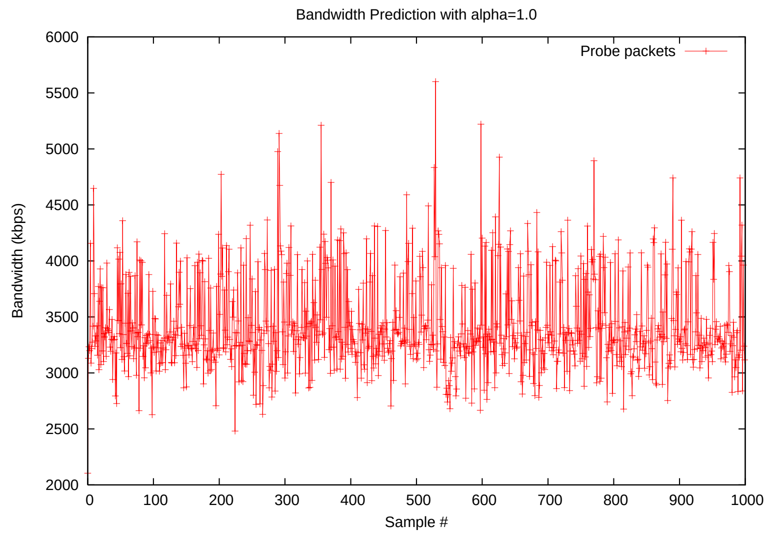


Figure A.14: Bandwidth prediction results an alpha value of 1.0.

As we see int the figures, $\alpha = 0$ give a static result, which is because it only looks at the history of predictions and does not consider any measurement results. As α increase, so does the fluctuations in the bandwidth prediction, which is because the measurement influences the prediction. It is difficult to see any big difference between values of 0.5, 0.75, and 1.0. For the rest of the experiments we will use an α of 0.5, which looks at both the history and current measurement in a fair way.

A.4.2 Round Trip Time

To measure the Round-Trip Time (RTT) we used the ping utility, which is provided in most operating systems. Ping is a user level application that was created to measure network performance, such as RTT. It uses the Internet Control Message Protocol (ICMP) to send its messages, instead of TCP or UDP. ICMP contains a special ECHO_REQUEST message that automatically generates a

ECHO_REQUEST_RESPONSE response from the receiver, which is how ping measures RTT.

To measure the RTT we send 10000 packets to simulate our video streaming traffic, i.e., our streaming packets. The RTT packets have the maximum IP packet size, i.e., 1500 bytes, and are sent with different periods to simulate different network loads. The size was set to the maximum because we assume that when streaming the amount of data is so large that the IP packets will almost always be of the maximum size. The streaming application will periodically send large chunks of data which will translate into large IP packets.

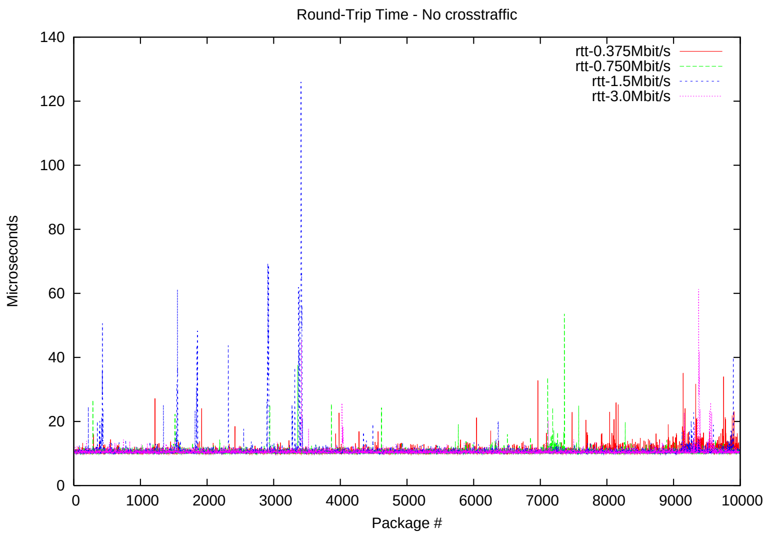
The periods varies between $0.0005s$, $0.001s$, $0.01s$, and $0.1s$, representing a traffic rate (payload) of 3 Mbit/s, 1.5 Mbit/s, 0.75 Mbit/s, and 0.375 Mbit/s, respectively.

We also generate crosstraffic at different rate to see the effects on the RTT. Crosstraffic is generated with a user level application that streams data at specific rates into the network. We send crosstraffic with the following traffic rates: no crosstraffic, 0.5 Mbit/s, 1.0 Mbit/s, 2.0 Kbit/s, and 3.0Kbit/s.

Furthermore, if we combine our method to the experiment, i.e., bandwidth prediction and traffic shaping, the cross traffic should only have a minimal effect on the RTT. The reason is because we limit the available bandwidth of the crosstraffic generating node to a low value, 20% of what is available. We consider our streaming node to be most "important" and thus we give it a high bandwidth share, actually 80% of what is available, even though it might not use all of it.

Figures A.1, A.2, A.3, A.4, and A.5 shows the RTT for both our method and standard Ethernet. In the figures we also see the RTT for streaming rates between 0.375 and 3.0Mbit/s, and with crosstraffic rates between 0.5Mbit/s and 3.0Mbit/s.

As we can see in all the figures, our architecture adapts the crosstraffic according to the predicted bandwidth and the assigned bandwidth share of the node, resulting in low RTT compared to standard Ethernet. Standard Ethernet does not perform this adjustment resulting in RTT that varies depending on the amount of crosstraffic. Standard Ethernet



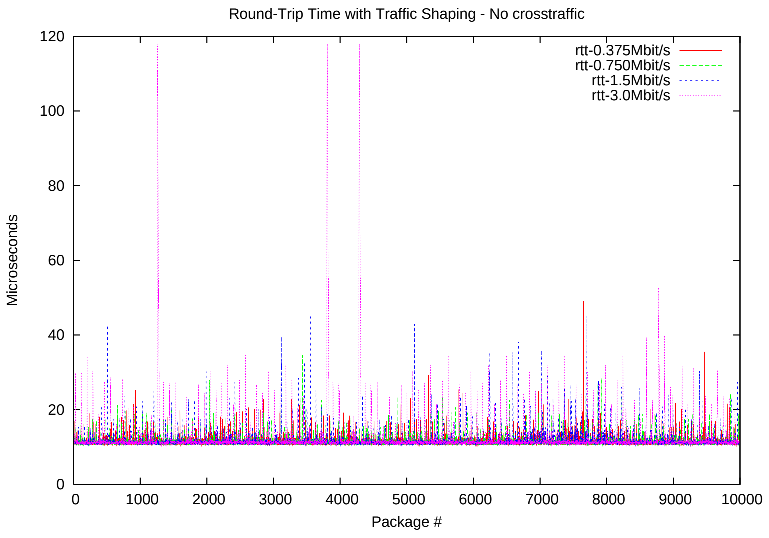


Table A.1: RTT with our method and standard Ethernet, without any crosstraffic.

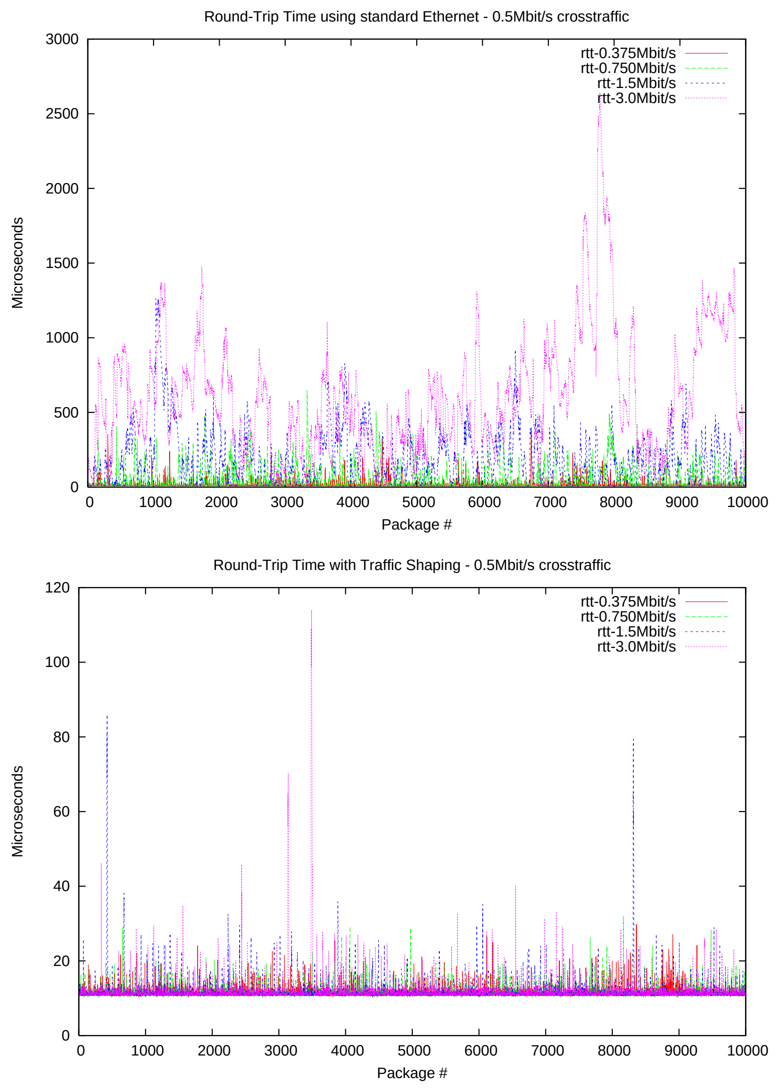
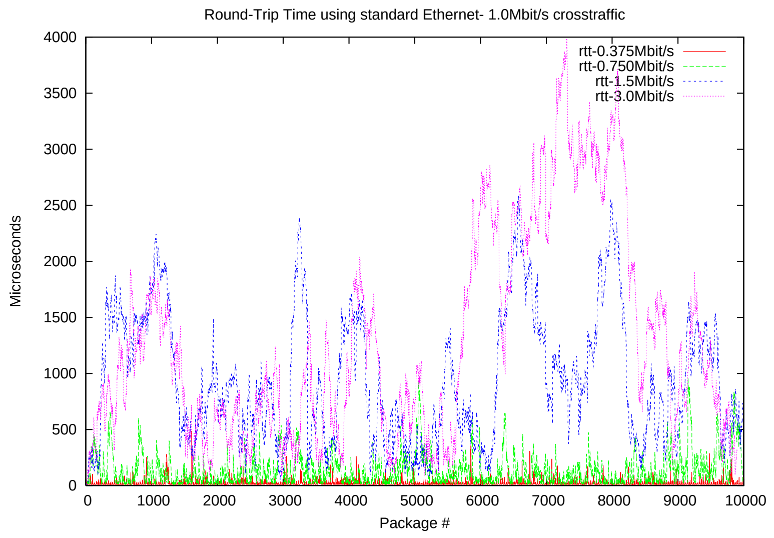


Table A.2: RTT with our method and standard Ethernet, with 0.5 Mbit/s of crosstraffic.



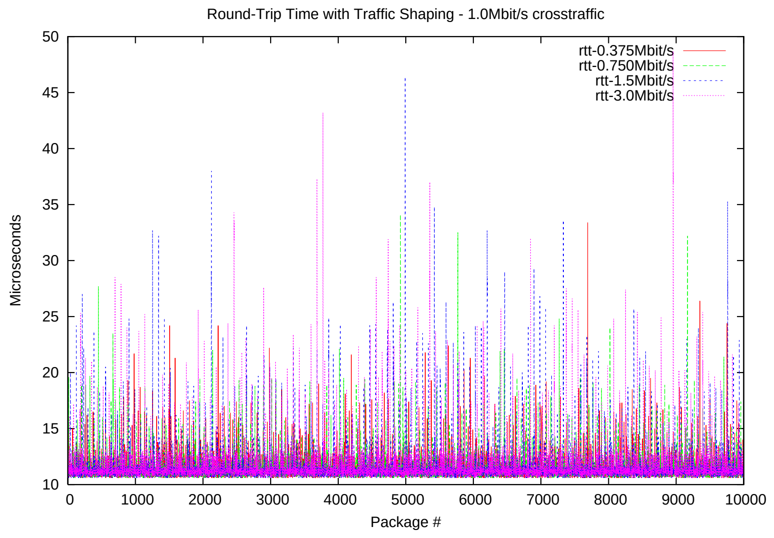
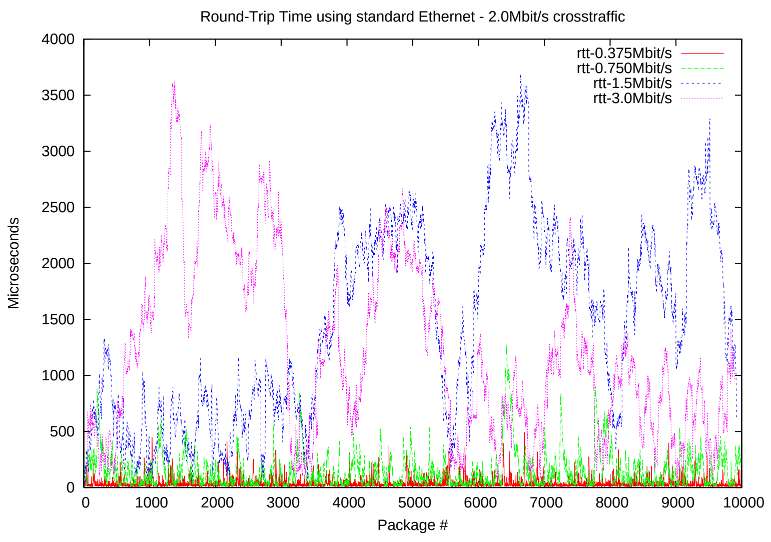


Table A.3: RTT with our method and standard Ethernet, with 1.0 Mbit/s of crosstraffic.



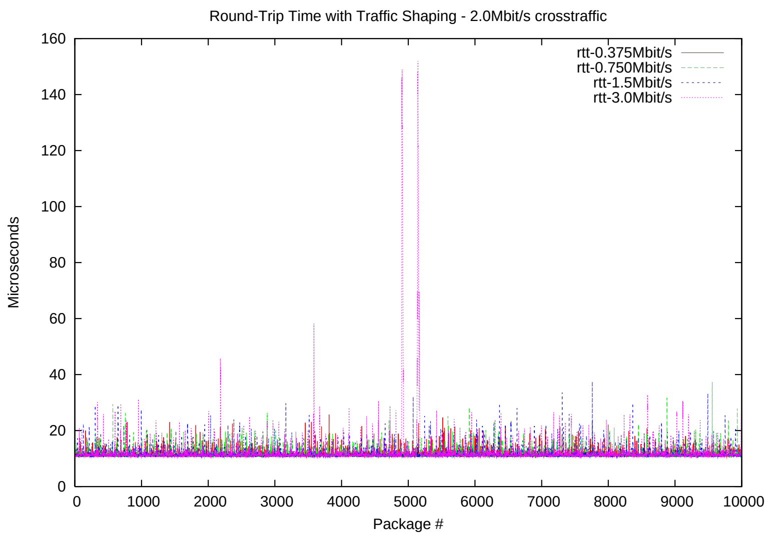
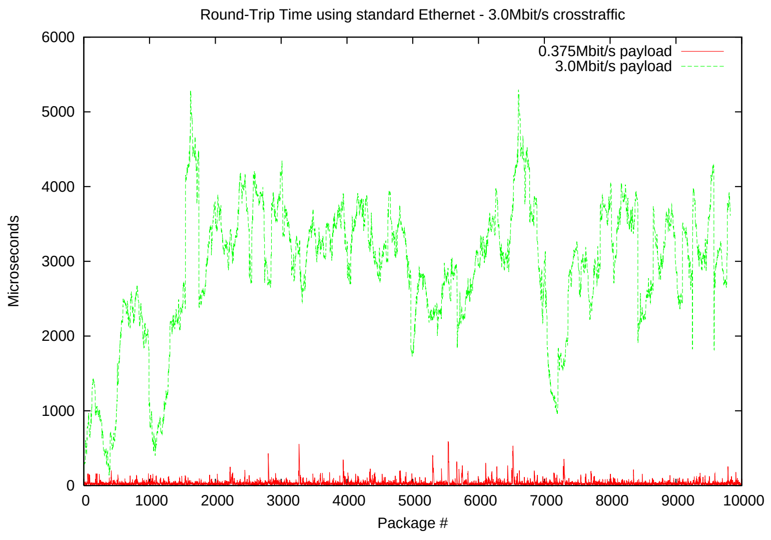


Table A.4: RTT with our method and standard Ethernet, with 2.0 Mbit/s of crosstraffic.



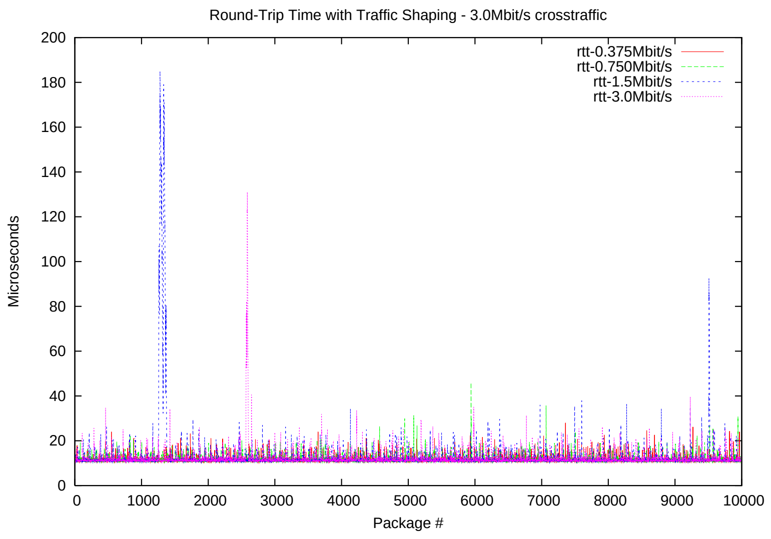


Table A.5: RTT with our method and standard Ethernet, with 3.0 Mbit/s of crosstraffic.

Standard Ethernet	Packet loss
0.375Mbps	13%
0.75Mbps	10%
1.5Mbps	0%
3.0Mbps	0%

Our architecture	Packet loss
0.375Mbps	0%
0.75Mbps	0%
1.5Mbps	0%
3.0Mbps	0%

Table A.6: Packet losses with standard Ethernet and our architecture, with varying payloads and 3.0Mbps of crosstraffic.

achieves round-trip times up to 30, 40, and even 50 milliseconds, which is not desirable from a streaming point of view.

A.4.3 Packet Loss

When we performed the RTT measurements described above, we also measured the packet loss ratio, comparing standard Ethernet with our architecture.

Table A.6 shows the packet losses using our shaper architecture. For the payloads that are lower than 3.0 Mbps, the packet loss is 0% for both standard Ethernet and our architecture.

A.4.4 Rate Change Overhead

We also performed measurements to see the overhead introduced in the various parts of our architecture. First we look at the time it takes to perform the full bandwidth prediction algorithm, then we measure the time it takes to update the low level traffic shaper.

We use the same setup as described above, but in this case we only

Maximum time	43.9 ms
Minimum time	15.5 ms
Average time	17.5 ms

Table A.7: Overhead for the packet probing.

transmit between the P3-laptop, and the P2-desktop, acting as sender and receiver respectively. We use the same setup as described above, but in this case we only transmit between the P3-laptop, and the P2-desktop, acting as sender and receiver respectively.

Bandwidth Prediction Overhead

In this experiment we measure the time it takes for a complete bandwidth prediction, from sending the probe packets, receiving the packet delay from the receiver node, to performing the calculation.

We take timestamps before and after the function calls for sending probe packets and calculating the new bandwidth, and repeatedly run the experiment. We did not insert any crosstraffic into the network in order to get times that are as free as possible from any delays caused by network interference. We performed 10000 measurement.

As we seen in table A.7, it takes about 15 milliseconds to perform a complete bandwidth prediction, sending the probe packets and receiving the time difference result and performing the calculation. At a maximum, it takes about 43 milliseconds to perform the prediction.

With these numbers we see that there should be no problem with sampling quite often, and since the code is not optimized we expect the possibility of improving these numbers with a more efficient code.

Traffic Shaper Overhead

In the second experiment we were interested in finding out the time it takes to change the setting of the Linux kernel traffic shaper.

Maximum time	55.5 ms
Minimum time	22.8 ms
Average time	38.9 ms

Table A.8: Overhead for updating the traffic shaping parameters.

In our implementation we call the Linux traffic control (tc) interface using the *system()* call provided by the kernel, which adds to the overhead.

When performing this experiment, we don't send any probe packets. Instead, we repeatedly call the function that updates the traffic shaper with the new numbers. We performed 10000 measurements.

As table A.8 show it takes some time to change the traffic shaper at the low level, limiting the probing intervals we can use. As we see in the table the maximum time a change of the traffic shaper parameters took around 55 milliseconds.

Bibliography

[1] IEEE Std 802.11-1997 - Wireless LAN Medium Access Control and Physical Layer Specifications, 1997.

[2] IEEE 802.11 WG. Draft Supplement to International Standard for Information Technology-Telecommunications and Information Exchange Between Systems LAN/MAN Specific Requirements. IEEE 802.11e/D2.0, 2001.

[3] IEEE Std 802.3-1985 - Local Area Networks: Carrier Sense Multiple Access with Collision Detection (CSMA/CD) - (ETHERNET), 1985.

[4] IEEE Std 802.3p-1993 - Suppements to Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications, 1985.

[5] M. A. Aboelaze and A. Elnaggar. The Performance of Ethernet Under a Combined Data/Real-Time Traffic. In *In Proceedings of the 25th Annual IEEE Conference on Local Computer Networks*, Tampa, USA, November 2000.

[6] S. A. Aldarmi and A. Burns. Dynamic Value-Density for Scheduling Real-Time Systems. In *Proceedings 11th Euromicro Conference on Real-Time Systems*, Dec 1999.

[7] M. Aldea and M. González Harbour. MaRTE OS: An Ada Kernel for Real-Time Embedded Applications. In *In Proceedings of*

the International Conference on Reliable Software Technologies - Ada-Europe'2001, Leuven, Belgium, May 2001.

[8] M. Aldea and M. González Harbour. POSIX-Compatible Application-Defined Scheduling in MaRTE OS. In *In Proceedings of the 14th Euromicro Conference of Real-Time Systems*, Vienna, Austria, June 2002.

[9] M. Aldea and M. González Harbour. A New Generalized Approach to Application-Defined Scheduling. In *In Proceedings of the 16th Euromicro Conference of Real-Time Systems*, Catania, Italy, June 2004.

[10] C. Baek-Young, S. Song, N. Birch, and J. Huang. Probabilistic Approach to Switched Ethernet for Real-Time Control Applications. In *In Proceedings of the 7th International Conference on Real-Time Computing Systems and Applications*, Cheju Island, South Korea, December 2000.

[11] S. Baruah and J. Haritsa. Scheduling for Overload in Real-Time Systems. *IEEE Transactions on Computers*, September 1997.

[12] S. M. Bellovin. A Best-Case Network Performance Model. http://www.research.att.com/smb/papers/netmeas.ps, February 1992.

[13] Berkley MPEG Tools - http://www.bmrc.berkley.edu/research/mpeg/.

[14] The Official Bluetooth Website - http://www.bluetooth.com/.

[15] G. Buttazzo, M. Spuri, and F. Sensini. Value vs. Deadline Scheduling in Overload Conditions. In *In Proceedings of the 16th Real-Time Systems Symposium*, Pisa, Italy, December 1995.

[16] G. Buttazzo and J. Stankovic. RED: A Robust Earliest Deadline Scheduling Algorithm. In *In Proceedings of the 3rd International Workshop on Responsive Computing Systems*, September 1993.

[17] F. Carone and R.P.O. Guerra. Available bandwidth Measurements on Wireless Networks. Technical report, Mälaradlens Högskola, Västerås, Sweden, 2002.

[18] A. Carpenzano, R.Carponetto, L. LoBello, and O. Mirabella. Fuzzy Traffic Smoothing: An Approach for Real-Time Comunication over Ethernet Networks. In *IEEE International Workshop on Factory Communication Systems*, Västerås, Sweden, August 2002.

[19] H. Chetto and M. Chetto. Some Results on the Earliest Deadline Scheduling Algorithm. *Transactions on Software Engineering*, 15, October 1989.

[20] L. Davis and G. Buttazzo. Integrating Multimedia Applications in Hard Real-Time Systems. In *In Proceedings of the 19th Real-Time Systems Symposium*, Madrid, Spain, December 1998.

[21] R.I. Davis, K.W. Tindell, and A. Burns. Scheduling Slack Time in Fixed Priority Pre-emptive Systems. In *In Proceedings of the 14th Real-Time Systems Symposium*, Raleigh-Durham, USA, December 1993.

[22] Z. Deng and J. W. S. Liu. Scheduling Real-Time Applications in Open Environment. In *In Proceedings of IEEE Real-Time System Symposium*, San Francisco, USA, December 1997.

[23] Z. Deng, J. W. S. Liu, and J. Sun. A Scheme for Scheduling Hard Real-Time Applications in Open System Environment. In *In Proceedings of the 9th Euromicro Workshop on Real-Time Systems*, Toledo, Spain, June 1997.

[24] C. Dovrolis and M. Jain. Pathload: A Measurement Tool for End-to-end Available Bandwidth. In *In Proceedings of the 3rd Passive and Active Measurements Workshop*, Fort Collins, USA, March 2002.

[25] G. Fohler. Analyzing a Pre Run-Time Scheduling Algorithm and Precedence Graphs. Technical report, Technishe Universität, Wienna, Austria, 1992.

[26] G. Fohler. *Flexibility in Statically Scheduled Hard Real-Time Systems*. PhD thesis, Technishe Universität, Wienna, Austria, 1994.

[27] G. Fohler. Joint Scheduling of Distributed Complex Periodic and Hard Aperiodic Tasks in Statically Scheduled Systems. In *In Proceedings of the 16th Real-Time Systems Symposium*, Pisa, Italy, December 1995.

[28] P. Gai, L. Abeni, M. Giorgi, and G. Buttazzo. A New Kernel Approach for Modular Real-Time Systems Development. In *Proceedings of the 13th Euromicro Real-Time Systems Conference*, Delft, Netherlands, June 2001.

[29] R. Bosch GmbH. Contoller Area Network (CAN), 1986.

[30] O. J. González. *Building Distributed Real-Time Systems with Commercial-Off-The-Shelf Components*. PhD thesis, University of Massachusetts, Amherst, USA, 2001.

[31] Independent JPEG Group - http://www.ijg.org/.

[32] D. Isovic and G. Fohler. Handling Sporadic Tasks in Statically Scheduled Distributed Real-Time Systems. In *Proceedings of the 11th Euromicro Real-Time Systems Conference*, York, England, June 1999.

[33] D. Isovic and G. Fohler. Efficient Scheduling of Sporadic, Aperiodic, and Periodic Tasks with Complex Constraints. In *Proceedings of the 21st IEEE Real-Time Systems Symposium*, Orlando, Florida, USA, November 2000.

[34] D. Isovic and G. Fohler. Online Handling of Firm Aperiodic Tasks in Time Triggered Systems. In *Proceedings of the 12th EUROMICRO Conference on Real-Time Systems*, Stockholm, Sweden, November 2000.

[35] Andreas Johnsson, Bob Melander, and Mats Björkman. Bandwidth Measurement in Wireless Networks. In *Mediterranean Ad Hoc Networking Workshop*, 6 2005.

[36] Joint Photographic Experts Group - http://www.jpeg.org/.

[37] A. Kamerman and G. Aben. Net Throughput with IEEE 802.11 Wireless LANs. In *In Proceedings of the IEEE Wireless Communications and Networking Conference*, Chicago, USA, September 2000.

[38] S. Keshav. A Control-Theoretic Approach to Flow Control. In *Conference of Communications Architecture and Protocols*, Zürich, Switzerland, September 1991.

[39] H. Kopetz. Time-Triggered Model of Computation. In *In Proceeedings 19th Real-Time Systems Symposium*, Madrid, Spain, December 1998.

[40] H. Kopetz, A. Damm, Ch. Koza, M. Mulazzani, W. Schwabl, Ch. Senft, and R. Zainlinger. Distributed Fault-Tolerant Real-Time Systems: The MARS Approach. *IEEE Micro*, February 1989.

[41] H. Kopetz, G. Fohler, G. Grünsteidl, H. Kantz, G. Pospischil, P. Puschner, J. Reisinger, R. Schlatterbeck, W. Schütz, A. Vrchoticky, and R. Zainlinger. The Distributed, Fault-Tolerant Real-Time Operating System MARS. *IEEE Operating Systems Newsletter*, 6(1), 1992.

[42] H. Kopetz and G. Grünsteidl. TTP - A Time-Triggered Protocol for Fault-Tolerant Real-Time Systems. In *In Proceedings of the 23rd IEEE International Symposium on Fault Tolerant Computing*, Tolouse, France, September 1993.

[43] S-K. Kweon, K. Shin, and G. Workman. Achieving Real-Time Communication over Ethernet with Adaptive Traffic Smoothing. In *IEEE Real-Time Technology and Applications Symposium*, Washington DC, USA, May-June 2000.

[44] S-K. Kweon, K. Shin, and Q. Zheng. Statistical Real-Time Communications over Ethernet for Manufacturing Automation Systems. In *IEEE Real-Time Technology and Applications Symposium*, Vancouver, Canada, June 1999.

[45] Linux Advanced Routing and Traffic Control - http://www.lartc.org/.

[46] C.L. Liu and J.W. Layland. Scheduling Algorithms for Multiprogramming in Hard Real-Time Environment. *Journal of the ACM, 20, 1*, January 1973.

[47] C. D. Locke. *Best-Effort Decision Making for Real-Time Scheduling*. PhD thesis, Carnegie-Mellon University, Pittsburgh, USA, 1986.

[48] J. Loeser and H. Haertig. Low-Latency Hard Real-Time Communication over Switched Ethernet. In *In Proceedings of the 16th Euromicro Conference on Real-Time Systems*, Catania, Italy, June 2004.

[49] C. W. Mercer, S. Savage, and H. Tokuda. Processor Capacity Reserves for Multimedia Operating Systems. In *In Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, Bostin, USA, May 1994.

[50] Moving Pictures Expert Group - http://www.chiariglione.com/.

[51] P. Ni. Bandwidth Estimation for Adaptive Multimedia over Wireless Networks. Technical report, Mälardalen University, Västerås, Sweden, 2003.

[52] K. M. Obenland, J. Kowalik, T. Frazier, and J. S. Kim. Comparing the Real-Time Performance of Windows NT to an NT Real-Time Extension. In *In Proceedings of the 5th IEEE Real-Time Technology and Applications Symposium*, Vancouver, Canada, June 1999.

[53] D. Pattengale and D. Rohn. NT for Soft Real-Time Control. In *In Proceedings of IEEE Workshop on Advanced Process Control Applications for Industry*, Vancouver, Canada, April 1999.

[54] K. Ramamritham. Allocation and scheduling of complex periodic tasks. In *In the 10th International Conference on Distributed Computing Systems*, Paris, France, May 1990.

[55] K. Ramamritham, G. Fohler, and J.-M. Adan. Issues in the Static Allocation and Scheduling of Complex Periodic Tasks. In *In Proceedings of the 10th IEEE Workshop on Real-Time Operating Systems and Software*, New York, USA, May 1993.

[56] K. Ramamritham, J.A. Stankovic, and W. Zhao. Distributed Scheduling of Tasks with Deadlines and Resource Requirements. *IEEE Transactions on Computers*, August 1989.

[57] Larisa Rizvanovic and Gerhard Fohler. The MATRIX: A QoS Framework for Streaming in Heterogeneous Systems. In *RTMM - International Workshop on Real-Time for Multimedia*, Catania, Sicily, Italy, July 2004.

[58] RETIS Lab, RTSIM - Real-Time Simulator - http://retis.sssup.it/.

[59] J. L. Sobrinho and A. S. Krishnakumar. EQuB - Ethernet Quality of Service using Black Bursts. In *IEEE Local Computer Networks Conference*, Lowell, USA, October 1998.

[60] M. Spuri and G. Buttazzo. Efficient Aperiodic Service under Earliest Deadline Scheduling. In *In Proceedings of the 15th IEEE Real-Time Systems Symposium*, San Juan, Puerto Rico, December 1994.

[61] M. Spuri and G. Buttazzo. Scheduling Aperiodic Tasks in Dynamic Priority Systems. In *In Proceedings of the IEEE Real-Time Systems Symposium*, Washington D.C., USA, December 1996.

[62] W. Stallings. *Wireless Communications and Networks*. Prentice Hall, 2002.

[63] J. A. Stankovic and K. Ramamritham. *IEEE Tutorial: Hard Real-Time Systems*. IEEE Computer Society Press, Washington, D.C., USA, 1988.

[64] J. A. Stankovic and K. Ramamritham. The Spring Kernel: A New Paradigm for Real-Time Operating Systems. *IEEE Software*, May 1991.

[65] tcpdump/libpcap, Packet Dumping/Capturing Libraries - http://www.tcpdump.org/.

[66] S. R. Thuel and J.P. Lehoczky. On-Line Scheduling of Hard Deadline APeriodic Tasks in Fixed-Priority Systems. In *In Proceedings of the 14th Real-Time Systems Symposium*, Raleigh-Durham, USA, December 1993.

[67] S. R. Thuel and J.P. Lehoczky. Algorithms for Scheduling Hard Aperiodic Tasks in Fixed-Priority Systems using Slack Stealing. In *In Proceedings of the 15th Real-Time Systems Symposium*, San Juan, Puerto Rico, December 1994.

[68] tc - traffic control, Linux QoS Control Tool - http://www.lartc.org/howto/.

[69] TTP-OS: Time-Triggered Operating System with TTP Support.

[70] Wi-Fi Alliane Webpage - http://www.wi-fi.org.

[71] Real-Time Systems with MS Windows CE - http://www.microsoft.com/technet/prodtechnol/wce/plan/realtime.mspx.

[72] What is the Actual Speed of a Wireless Network? - http://compnetworking.about.com/od/wirelessfaqs/f/maxspeed80211b.htm.

[73] V. Yodaiken. Rough notes on Priority Inheritance. Technical report, New Mexico Institute of Mining, 1998.

[74] A. Zahedi and K. Pahlavan. Natural Hidden Terminal and the Performance of Wireless LANs. In *In Proceedings of the 6th IEEE International Conference on Universal Personal Communications Record*, 1997.

[75] W. Zhao, K. Ramamritham, and J. A. Stankovic. Preemptive Scheduling Under Time and Resource Constraints. *IEEE Transactions on Computers*, August 1987.

Populärvetenskaplig svensk sammanfattning

"Adapting to Varying Demands in Resource Constrained Real-Time Devices"

Inom en nära framtid kommer så kallade digitala underhållningssystem att bli mer och mer populära i hemmen. En stor fördel med dessa system är att apparaterna, t.ex. TVn, DVDn, digital boxen, och datorn, kommer att kunna kopplas ihop trådlöst, vilket gör det möjligt att slippa sladdar mellan dem. Detta innebär också att flera apparater (TVn, PCn, eller mobiltelefonen) kommer att ha möjlighet att spela upp en DVD film som sitter i DVD spelaren, oavsett vart i huset apparaterna finns. Det blir till och med möjligt att flytta de mobila apparaterna medans filmen spelas upp, t.ex. kan man gå runt i huset samtidigt som man tittar på filmen på en liten handdator eller mobiltelefon. Detta är inte möjligt idag eftersom man kopplar en sladd direkt mellan DVD spelaren och TVn.

De digitala underhållningssytemen måste klara av att hantera både ljud (musik) och bild (film) på ett sätt som är tillfredställande för den som använder systemet, d.v.s, kvaliteten på ljudet som spelas upp eller bilden som visas måste vara minst lika bra som den är idag. Detta kommer att ställa höga krav på apparaterna, som måste hålla de kvalitetskrav som ljud (musik) och bildströmmar (film) har.

Tack vare flexibiliteten, d.v.s, möjligheten att ha mobila apparater, i dessa system, och kraven på systemen, uppstår nya problem som inte funnits innan. T.ex. så har inte handdatorer eller mobiltelefoner den tekniska möjligheten att spela upp en DVD film med samma kvalitet som på en TV.

Ett annat problem är den varierande kapaciteten i det trådlösa nätverket som knyter samman alla apparater. På grund av att nätverket är väldigt känsligt för störningar (t.ex. från mikrovågsugnar) så varierar kapaciteten för nätverket hela tiden, något som inte är fallet när man har ett nätverk med sladdar.

Detta leder till problem med den trådlösa överföringen mellan olika apparater. Eftersom en DVD film kräver en hel del kapacitet från nätverket, och man inte kan vara säker på vad som finns tillgängligt så vet man inte om överföringen av filmen kommer att gå bra.

I vår forskning presenterar vi två olika sätt att hantera problemen med olika apparaters tekniska begränsningar och det trådlösa nätverkets varierande kapacitet.

För att hantera problemet med olika apparater och dess varierande tekniska möjligheter så har vi tagit fram metoder som möjliggör för oss att anpassa uppgifterna (t.ex. spela musik eller film) som utförs på apparaten beroende på den tekniska kapaciteten de har.

Vi föreslår även en metod där vi mäter hur mycket kapacitet som det finns tillgängligt i det trådlösa nätverket. Eftersom kapaciteten på nätverket varierar repeterar vi mätningen med jämna mellanrum så att vi hela tiden har en aktuell bild av vad som finns tillgängligt. Vi använder sedan den informationen för att anpassa det som skall överföras (musik eller film) för att hela tiden göra det bästa möjliga av situationen.