Low-Latency Hard Real-Time Communication over Switched Ethernet

Dissertation

zur Erlangung des akademischen Grades Doktoringenieur (Dr.-Ing.)

vorgelegt an der
Technischen Universität Dresden
Fakultät Informatik

eingereicht von

Dipl.-Inform. Jork Löser

geboren am 20. Oktober 1974 in Leipzig

Gutachter:

Prof. Dr. rer. nat. Hermann Härtig
Prof. Dr. Gerhard Fohler
Prof. Dr. rer. nat. habil. Dr. h. c. Alexander Schill

Tag der Verteidigung: 31. Januar 2006

Abstract

With the upsurge in the demand for high-bandwidth networked real-time applications in cost-sensitive environments, a key issue is to take advantage of developments of commodity components that offer a multiple of the throughput of classical real-time solutions.

It was the starting hypothesis of this dissertation that with fine grained traffic shaping as the only means of node cooperation, it should be possible to achieve lower guaranteed delays and higher bandwidth utilization than with traditional approaches, even though Switched Ethernet does not support policing in the switches as other network architectures do.

This thesis presents the application of traffic shaping to Switched Ethernet and validates the hypothesis. It shows, both theoretically and practically, how commodity Switched Ethernet technology can be used for low-latency hard real-time communication, and what operating-system support is needed for an efficient implementation.

Acknowledgments

First, I would like to thank my supervisor, Prof. Hermann Härtig. Despite scare resources, he has grown the operating-systems group at TU Dresden to an internationally respected research institution and an enjoyable and inspiring place to work at. Without his support and his wit, this thesis would not exist.

Many members of the operating-systems group at TU Dresden have contributed work that has helped making this thesis a reality. I am indebted to Claude-Joachim Hamann, Michael Hohmuth, and Jean Wolter, whose comments have opened my eyes more than once; and Ronald Aigner, Adam Lackorzynski, Frank Mehnert, and Lars Reuther, for their continuing effort to improve the group's operating system DROPS.

I am thankful to many people who have read draft versions of this document or parts of it. Their valuable comments have helped improving the thesis tremendously. I thank Gerhard Fohler, Markus Fidler, Jane Liu, and Alexander Schill.

Finally, I would like to thank a person that is often forgotten, simple because it is her job to make us feel her less: Angela Spehr, our group's secretary. Without her, I would not have had the time to write this thesis.

Contents

1. Introduction **1**

2. Background **4**
2.1. Switched Ethernet — QoS at the hardware level 4
2.2. DROPS — QoS at the software level 5
2.2.1. DROPS 5
2.2.2. DROPS scheduling models 6
2.2.3. Kernel scheduling model 9

3. Related work **14**

4. QoS at the hardware level **17**
4.1. Network calculus 17
4.1.1. Definitions and theorems 17
4.1.2. Application to Switched Ethernet 18
4.2. Bounding network delays 19
4.2.1. Definitions 19
4.2.2. Modeling network traffic 20
4.2.3. Delay and burstiness increase at NICs 21
4.2.4. Delay and buffer calculation of switches 22
4.3. Burstiness increase at the switch 24
4.3.1. Theoretical background 24
4.3.2. Application to Switched Ethernet 25
4.3.3. Remark 25
4.4. Traffic reservation 26
4.4.1. Established traffic reservation techniques 26
4.4.2. Traffic reservation technique for a Switched Ethernet network 26
4.5. Networks with multiple switches 27

5. Traffic shaping in the nodes **29**
5.1. Application network model 29
5.2. Traffic shaping implementation aspects 30
5.2.1. Performance of shapers 31
5.2.2. CPU utilization 31
5.2.3. Shaper versions 31
5.3. Strictly periodic shaper 32
5.4. Periodic shaper with data dependency 33
5.5. Strictly periodic shaper with long periods 34
5.6. Token-bucket shaper 34
5.7. Token-bucket shaper for best-effort connections 36
5.8. Comparison 37
5.9. Summary 39

6. Implementation **40**
6.1. Local network manager . . . 41
6.2. Design issues . . . 42
6.3. Real-time send path . . . 44
6.3.1. Client interface abstraction . . . 45
6.3.2. Connection-specific TX thread . . . 46
6.3.3. CPU usage of the TX thread . . . 47
6.4. Real-time notification path . . . 47
6.4.1. Device-specific IRQ thread . . . 49
6.4.2. Software-based interrupt coalescing . . . 55
6.4.3. Connection-specific notifier thread . . . 57
6.4.4. Blocking client model . . . 60
6.4.5. Polling client model . . . 66
6.4.6. Interrupt coalescing with the polling client model . . . 69
6.5. Summary . . . 73
6.5.1. Analyzed execution models . . . 73
6.5.2. Resource usage dependencies . . . 73
6.5.3. Alternative scheduling schemes . . . 74
6.6. Real-time connection setup . . . 74
6.7. Real-time client API . . . 76
6.8. Best-effort communication . . . 77
6.8.1. Best-effort send path . . . 78
6.8.2. Best-effort notification path . . . 80
6.9. Best-effort client network stacks . . . 81
6.9.1. L^4Linux . . . 81
6.9.2. Flips . . . 82
6.9.3. Routing between multiple IP-Stacks . . . 83
6.10. Outlook: Offloading traffic handling to network cards . . . 83

7. Experimental evaluation **84**
7.1. Network hardware analysis . . . 84
7.1.1. Measurement setup . . . 84
7.1.2. Measuring inter-node µ-second delays . . . 85
7.1.3. Achieving worst-case delays . . . 86
7.1.4. Switch multiplexing delays . . . 86
7.1.5. Switch buffer capacities . . . 87
7.2. Application-to-application effects . . . 88
7.2.1. Application-to-application test packet transmission delays . . . 88
7.2.2. Fast Ethernet with DROPS . . . 88
7.2.3. Gigabit Ethernet with DROPS . . . 91
7.3. Sharing a network with non–real-time nodes . . . 92
7.4. Dynamic real-time system performance . . . 94
7.4.1. Measuring CPU costs . . . 94

7.4.2. Measurement setup . . . 94
7.4.3. Transmit CPU costs . . . 95
7.4.4. Receive CPU costs . . . 100
7.4.5. Costs for multiple connections . . . 102
7.4.6. Application scenario . . . 104
7.4.7. L^4Linux integration . . . 107

8. Conclusion **110**
8.1. Main contributions . . . 110
8.2. Auxiliary contributions . . . 111
8.3. Suggestions for future work . . . 111
8.4. Concluding remarks . . . 112

A. Derivations **A-1**
A.1. Bounds for stair-case arrival curve and service curves . . . A-1
A.1.1. Buffer bound . . . A-2
A.1.2. Delay bound . . . A-3

B. Abbreviations **B-1**

C. Glossary **C-1**

List of Figures

1. Buffering inside an output-queueing Switch . . . 4
2. The DROPS architecture . . . 6
3. Quantiles of a distribution . . . 7
4. Standard L4-type fixed-priority scheduling . . . 10
5. Capacity-reserves scheduling . . . 10
6. Fiaso's thread scheduling . . . 11
7. Fiasco's minimal inter-release time scheduling . . . 12
8. Illustration of the arrival curve $\alpha(t)$ as the sum of two flows . . . 23
9. Shaping the traffic at the sending nodes . . . 29
10. Multiplexing of multiple shaped connections to one NIC. . . . 30
11. Maximum delay of a conforming packet at the strictly periodic traffic shaper. . . . 32
12. Burstiness parameter of the flow generated by the strictly periodic shaper . . . 33
13. Maximum delay induced by the periodic shaper with data dependency . . . 33
14. Obtaining the burstiness parameter of the flow generated by the token-bucket shaper. 35
15. Obtaining the burstiness parameter of the flow generated by the best-effort shaper. 38
16. Setup for comparing the effect of the different traffic shapers. . . . 39
17. Classical network node architecture. . . . 40
18. DROPS real-time capable network node architecture . . . 41
19. Thread structure of the network driver and real-time applications . . . 44

20. Transmission: data sharing between the client and the RT-Net server 45
21. Architecture of the notification path at the RT-Net server 48
22. Interaction between the RT-Net server code and the native network Linux driver code within an *IRQ thread*. 49
23. Arrival curve $\alpha_{i,rxring}$ of traffic received by the *IRQ thread* 55
24. Reducing the *IRQ thread* frequency by minimal inter-release time scheduling . . 56
25. Bound $R_i(t)$ of traffic received by the *IRQ thread* for a connection 56
26. Arrival curve $\alpha_{i,rxring}$ of traffic received by the *IRQ thread* with interrupt coalescing for the receive ring of connection i. 57
27. IPC communication structure of the connection-specific notifier thread. 58
28. Arrival curve of *IRQ thread* with the clients service curve 64
29. Arrival curve of the coalescing *IRQ thread* with the blocking clients service curve and their replacements. 66
30. Application model of the client with coalescing notifier thread. 67
31. Arrival and service curves $\alpha_{i,rxring}$ and $\beta^{poll}_{i,client}$ of the receive ring of connection i . 68
32. Arrival curve of the coalescing *IRQ thread* with the clients service curve 70
33. Client API for communication with the network driver in real-time mode. 76
34. Thread structure of the network driver and best-effort applications 78
35. General measurement setup 85
36. Precise time synchronization 86
37. One period of a symmetric burst. 87
38. Delay–CPU trade-off with different traffic shaping intervals 90
39. Setup for Section 7.4 95
40. Per-period CPU usage of the *TX threads* for 399-Byte packets 96
41. Per-period CPU usage of the *TX threads* for 1472-Byte packets 97
42. Per-period CPU usage of the *TX threads* for 1 ms periods 97
43. Per-period CPU usage of the *IRQ thread* for different packet sizes 99
44. Distribution of application-to-application packet latencies 106
45. Setup for Section 7.4.7 107
46. CPU usage of plain L^4Linux and L^4Linux with the RT-Net stub, data reception . . 108
47. CPU usage of plain L^4Linux and L^4Linux with the RT-Net stub, data transmission 109
48. Analysis of stair-case arrival and service curves A-1

Listings

1. Example of a strictly periodic real-time thread 13
2. Strictly periodic shaper, one packet per period 32
3. Token-bucket shaper 35
4. Best-effort shaper 37
5. Receive path demultiplexing pseudo code 50
6. Connection-specific receive function that enqueues a packet into the receive ring . 50
7. Waking the notifier threads 51

8. Connection-specific notifier thread 59
9. Waiting for data from the *IRQ thread* 60
10. Transmitting data to the client 61
11. Waiting for data from the *IRQ thread* 81
12. Transmitting data to the client 81

1. Introduction

With the upsurge in the demand for high-bandwidth networked real-time application in cost-sensitive environments, a key issue is to take advantage of developments of commodity components that offer a multiple of the throughput of classical real-time solutions at competitive prices. Ethernet as defined in the IEEE 802.3 standard is *the* commodity network since decades, and has undergone a number of changes in its existence. It is used for hard real-time communication already, and demanding applications continue to emerge.

A typical example is factory automation, where Ethernet replaces proprietary fieldbusses for performance and cost reasons. In the context of professional audio mastering (audio-LAN) Ethernet is experimented with: Multiple nodes generate samples for hundreds of instruments in parallel and send them to a central mixer node. The process is interactively controlled and delays are expected to be less than 10 ms. The bandwidth requirement for such a scenario is ten to hundred megabytes a second. Another application from the audio domain is DMIDI [Ker03], an attempt to use Ethernet LANs for MIDI control commands. Although the bandwidth demands are moderate, the delays are expected to be a few milliseconds too. The automotive industry uses Ethernet for in-car, soft real-time multimedia communication as well as for diagnostic purposes — in addition to CAN, LIN, and TTP networks for hard real-time control.

Ethernet originally has been defined as a bus-based protocol, using the *carrier sense, multiple access/collision detection* (CSMA/CD) mechanism to achieve coordination among the connected nodes. The CSMA/CD mechanism, however, results in nondeterministic delays caused by collisions when multiple nodes access the medium at the same time. Hence, higher-level protocols are needed to achieve an exclusive bus access for bounded communication delays. Real-time approaches using the bus-based Ethernet basically fall in three categories: token-based medium access control protocols, time-slot-based protocols, and statistical approaches. Time slots and token passing techniques are used by cooperating nodes to achieve both: to avoid collisions and to obey the limit of bandwidth allocated to the participating nodes. Intuition indicates that the use of such techniques to avoid collisions limits the achievable utilization and increases the CPU load of the nodes much more than using more relaxed forms of cooperation that only control bandwidth allocation. Related research supports that intuition on the high cost for collision avoidance by node cooperation (see Section 3).

Later in its development, Ethernet has been extended by intelligent switches, forming the *Switched Ethernet* technology. It is a star-based topology providing a private collision domain to each of the ports of a switch. If only one node is connected to each port of the switch, collisions do not occur. Consequently, node cooperation is needed only for bandwidth control, not any more to avoid collisions. It was my starting hypothesis that with fine grained traffic shaping as the only means of node cooperation, it should be possible to achieve lower guaranteed delays and higher bandwidth utilization than time-slotted and token-passing approaches, even though Switched Ethernet does not support policing in the switches as for example in ATM switches.

In this dissertation, I present the application of traffic shaping to Switched Ethernet and validate the hypothesis as stated previously. I show how commodity Switched Ethernet technology can be used for low-latency hard real-time communication, provided the right operating-system

support is available. The main contribution of this dissertation is the identification and analysis of this operating-system support. As a prerequisite and further contribution, I formally model the Switched Ethernet network and the traffic flowing through it to the extent needed by the operating-system analysis.

I apply well-established and recently developed networking scheduling theory such as the network calculus to Switched Ethernet to obtain bounds on the characteristics of traffic as it traverses the network. The analysis also contains intermediate worst-case delays at the network elements (network interface cards, switches) and bounds on the resource needs (i.e., buffer memory) therein. Putting aside the details unrelated to the main goal of my dissertation, the network analysis concentrates on networks with one switch. Once the operating-system requirements have become clear and well understood when traffic shaping is applied to small networks, an extension to networks with multiple switches mainly requires extending the mathematical network model. The principal operating-system requirements however do not change.

I propose various traffic-shaper implementations that ensure a cooperation of the nodes according to previously acknowledged traffic contracts. I analyze the scheduling needs of these implementations and estimate their performance with respect to achievable network delays and buffer needs and estimate the implementation costs with respect to CPU utilization.

I developed and present an implementation of Switched Ethernet-based networking that gives guarantees for traffic handling in the network and at the attached nodes and that provides hard application-to-application real-time communication. The implementation uses the real-time operating system *DROPS* that features a capacity-reserves-like CPU reservation scheme. DROPS is a priority-based, dynamic real-time system that allows to start and execute real-time applications and non–real-time applications in parallel. I derive the network node architecture for that dynamic real-time system and detail the task and thread model, the real-time/non–real-time traffic classes and the application interaction. I give a complete analysis of the DROPS-scheduler effects to the traffic as it travels through the network, such as to scheduler-induced delays and resulting memory requirements to buffer data. I present the developed admission process and the integration of legacy operating systems running on DROPS into the real-time communication. I further discuss the consequences of an implementation on operating systems with different scheduling schemes.

I present elaborate measurements on the practical applicability of traffic shaping for Switched Ethernet-based real-time networking. These measurements analyze principal characteristics of Ethernet hardware, show the general bounds on the delays and resource requirements that can be achieved on this hardware and provide detailed results on the costs and practical limits of the implementation on DROPS.

This dissertation is structured as following: Section 2 presents the key concepts of real-time communication on Switched Ethernet and introduces the DROPS real-time system. Section 3 on page 14 surveys other work in the area of real-time communication and relates it to this dissertation. Section 4 on page 17 reviews the theory on network calculation, defines the network-traffic model used throughout this document, applies the general network theory to Switched Ethernet, and derives an appropriate traffic-reservation scheme. It also outlines the main aspects of using multi-switch networks. Section 5 on page 29 details the steps necessary to shape traffic to give hard and tight guarantees on the achievable delays at the network system. It presents possible

traffic-shaper implementations for a priority-based real-time system, and quantitatively analyzes them. Section 6 on page 40 describes the implementation of Switched-Ethernet-based real-time networking on DROPS and discusses consequences of an implementation using other scheduling schemes. Section 7 on page 84 provides the experimental analysis, and Section 8 on page 110 summarizes this dissertation.

2. Background

Hard real-time communication means to give real-time guarantees for application-to-application data transfer. This is to provide a certain quality of service (QoS) on the transmission of packets from an application on one node to an application on another node. The QoS to be provided is the assurance of data delivery, a delay bound for the delivery of data, and a guaranteed communication bandwidth. This section introduces the key concepts of fundamental infrastructures used to provide QoS on packet transmission: The network hardware and the operating system at the nodes.

2.1. Switched Ethernet — QoS at the hardware level

This section shows the principal operation of a typical switch of the Ethernet technology as defined in the IEEE 802.3 standard. The section clarifies how lossless communication with bounded delays is done, and is fundamental to the traffic-shaping approach used in this dissertation.

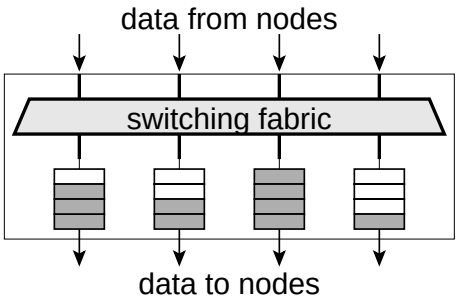

Figure 1: Buffering inside an output-queueing Switch. If queueing a frame is necessary, memory is allocated from a shared memory pool and assigned to the corresponding queue.

Figure 1 shows a typical Ethernet switch. The switch has four receive ports, control logic, buffer space and four queued transmit ports. The queues are operating in first-in first-out (FIFO) mode. When a frame arrives at the switch, the control logic determines the transmit port and tries to transmit the frame immediately. If the port is busy because another frame is already being sent, the frame is stored in the transmit ports queue. The memory to store pending frames is obtained from a shared memory pool. If no more memory is available, the received frame is dropped. In a real-time system, this dropping must be prevented. Also, the delay of a packet at the switch must be bounded.

Real-Time communication

If a packet needs to be queued, its delay within the switch depends on the current length of the queue: All packets within the queue must be transmitted before the current packet. Thus, guarantees on the maximum delay of packets within the switch caused by queueing can only be given if

the length of the queues is bounded. To bound the queues in length, the *input* traffic to the switch must be bounded. In Section 4 I detail how the traffic is described to allow a formal analysis of the queue-length bounds. I also derive the packet-delay bounds and switch-buffer bounds depending on the input traffic at the switch.

As Ethernet lacks any policing mechanism, the only way to bound the input traffic at the switch is to bound the *output* traffic of all nodes connected to the switch. Therefore, these nodes must cooperate to prevent flooding the switch with data. This cooperation means that all sending nodes shape their traffic to conform to a previously acknowledged traffic specification, in detail described in Sections 4 and 5.

Other Ethernet capabilities

Another class of Ethernet switches is *input buffered*. In these switches, packets are queued at the input ports, and a packet is removed from the queue only if the according output port is free. Input-buffered switches are vulnerable to the *head-of-line blocking* phenomenon. Head-of-line blocking means that packets back in the input queue are blocked when the first packet in the queue must wait for its output port to become free. Today, input buffered switches are rarely used because of the head-of-line blocking effect, and I do not consider them in my dissertation.

The IEEE 802.3 extension 802.1p allows to assign priorities to individual network frames. Switches supporting frame priorities map these priorities to an internal set of up to 8 priority levels. Each transmit port is assigned a set of transmit queues, one per mapped priority level. Upon transmission, frames from higher prioritized queues are sent before those of lower prioritized queues. However, the prioritized input traffic still must be bounded to prevent overload situations and mutual interactions at the high-priority queues. Further, PEDREIRAS and others report in [PLA03] that lower prioritized traffic may lock switch memory, which cannot be used for higher prioritized traffic then. Thus, there is no real isolation between the different priorities, and I do not consider prioritizing frames in my dissertation.

2.2. DROPS — QoS at the software level

To provide real-time guarantees for application-to-application data transfer, not only the network must provide a certain QoS level. Also the operating systems executing these applications must ensure timely execution of its drivers, intermediate network stacks, and the applications. The target system I used to implement, analyze, and verify the real-time networking approach is the Dresden Real-Time Operating System DROPS [HBB+98].

2.2.1. DROPS

The design objectives for DROPS are to execute real-time programs and best-effort programs in parallel, and to start and stop these programs dynamically. Therefore real-time applications reserve the resources they need for proper operation at *resource managers*. Spare resources, including

CPU cycles, memory, network and disk bandwidth, may be consumed by non–real-time (best-effort) applications.

DROPS is a mikrokernel-based system on top of Fiasco [HH01], an implementation of the L4 mikrokernel interface [Lie95]. The L4 mikrokernel interface defines a minimal set of kernel abstractions, that is address spaces with multiple threads and synchronous IPC-based communication between all threads. Drivers accessing hardware devices run in user space, just as resource managers and normal user applications do. In addition to the standard L4 abstractions, Fiasco provides a real-time scheduling interface [Ste04], described in Section 2.2.3.

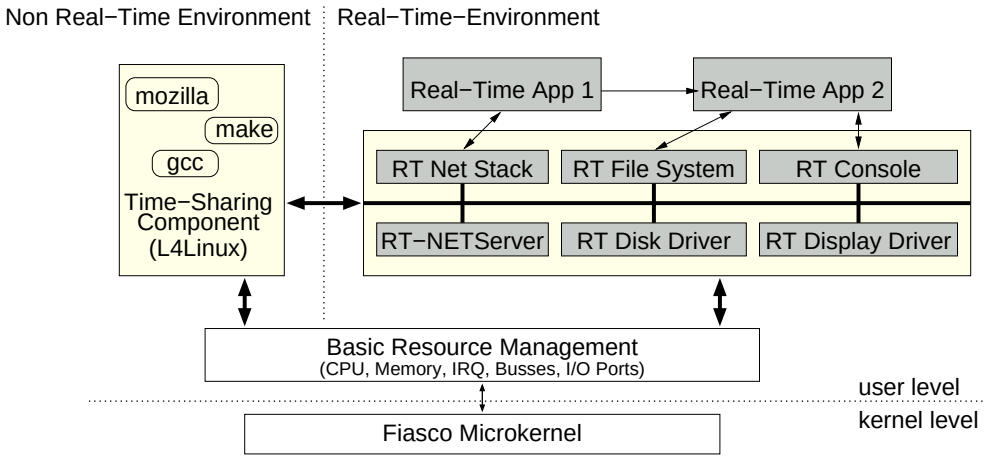

Figure 2: The DROPS architecture

Figure 2 shows the architecture of DROPS. Lower-level resource managers provide abstractions such as memory, CPU, or hardware-device access. Other abstractions such as networks, disks, or displays are provided by resource managers using the service of lower-level resource managers. Network stacks, file systems or consoles are even higher abstractions provided by higher-level resource managers. To guarantee a quality-of-service level of a certain resource, the corresponding manager maps that resource to the required lower-level resources and reserves them at their corresponding managers. This separated architecture provides an effective isolation of resource usage.

2.2.2. DROPS scheduling models

To give execution time guarantees in a dynamic environment, real-time programs reserve all the resources they need for proper operation before committing a certain quality of service. It is well known that reservations based on worst-case assumptions achieve a 100% quality but result in poor resource utilization. However, applications often do not need such a strong quality guarantee. If their execution resource profile can be statistically described, an appropriate scheduling model provides statistical guarantees to these applications.

The CPU resource model for DROPS applications is based on *task*s as the periodic execution of *job*s. Each job consists of one *mandatory* and a number of *optional parts*. While the mandatory parts must be executed completely under all circumstances, only a certain percentage of the optional parts (the quality level) needs to be scheduled and needs to be executed in time. The admission and CPU reservation uses statistical distribution data of the execution times of the job parts to determine priorities and reservation time quantities for these parts. On execution, the parts are scheduled by a fixed priority scheduler that enforces the reservation time quantities.

An important parameter for the DRPOS scheduling models is the *quantile* of a distribution of execution times:

Definition: The *quantile q(p)*, or *p-quantile* ($0 \leq p \leq 1$) of a cumulative distribution function $F(x) = P(X > x)$ of a random variable X is that value that is assigned a probability of p by F. In other words, part p of all values of X are less than the quantile. More formal: $q(p) = (x : P(X < x) = p)$. The definition for a discrete frequency distribution, or just *distribution*, of a series X of measured values is analogous:

$$q(p) = min(x : P(X < x) \geq p).$$

Figure 3 illustrates the definition of the quantile:

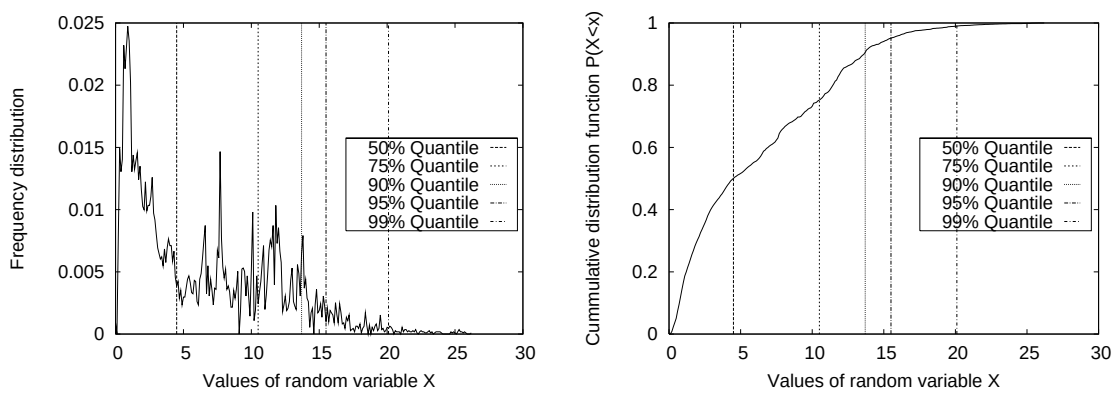

Figure 3: The 50%, 75%, 90%, 95% and 99% quantiles of P(X<x). The quantiles are 4.5, 10.5, 13.7, 15.5 and 20.1.

Quality-Assuring Scheduling model (QAS)

The Quality-Assuring Scheduling model (QAS) presented in [HLR+01] is one of the scheduling models supported by DROPS. It can handle preemptible as well as nonpreemptible jobs and allows a 100% resource utilization. However, the periods of all tasks must have the same length for the analysis given in [HLR+01]. A newer model (not published yet) relaxes this restriction and allows harmonic task periods.

A task $\tilde{T}_i$ in QAS consists of periodic jobs J_i with an assigned period of T_i. A job J_i is described by its parts P_i and a desired quality level Q_i. The parts of a job J_i depend on each other, this means part P_i^j of job J_i is only scheduled if all previous parts $P_i^{k:k<j}$ were successfully executed. Each part description P_i^j consists of an execution time distribution $\check{C}_i^j$ of that part. The part execution times described by $\check{C}_i^j$ must be independent of each other.

The admission of QAS in done in two phases: The first phase assigns scheduling priorities to the parts of all jobs, based on quality-monotonic scheduling. Therefore, mandatory parts are assigned the highest priority and optional parts are assigned priorities corresponding to their desired quality level: The lower the quality level is, the lower is the assigned priority. The second step verifies that (1) all mandatory parts can be successfully executed, and (2) at least a fraction of Q of all optional parts P_i^j can be successfully executed in the average.

Whether or not arbitrary periods can be handled with the QAS model is an open problem at the time of this writing. The admission processes of both the periodic and the harmonic QAS models are currently implemented in an off-line version.

Quality Rate Monotonic Scheduling model (QRMS)

The operating system group at TU Dresden currently develops another scheduling model — the Quality Rate Monotonic Scheduling (QRMS, not published yet). In contrast to QAS, priorities and a CPU quantity is assigned to jobs as a whole instead of to the parts therein. Subject to the admission process is the maximum of (1) the quantile q_i of the execution time distribution $\check{C}_i$ of a job J_i that corresponds to the desired quality level Q_i, and (2) the worst-case execution time (WCET) C_i of the mandatory part therein. Priorities are assigned using the RMS scheme. The admission can be done based on the overall CPU utilization or by using time-demand analysis.

QRMS allows arbitrary task periods, but allows no 100% resource utilization. In the presence of multiple optional parts, it is outperformed by the QAS scheduling model — QAS achieves a higher CPU utilization. Further, QRMS is limited to preemptible jobs. At the time of this writing, an admission test for this model has not been implemented.

Task model for drivers

Some hardware drivers, in contrast to applications, are not allowed to miss deadlines, because they need to respond to device activities in time. Further, for some devices it is impossible to statistically describe the execution time profile of drivers: A variety of conditions that heavily change with dynamic client applications and environmental influences such as incoming network packets determine the drivers execution time. These drivers use a worst-case based scheduling analysis. Further, drivers might need especially low execution jitters.

Therefore a driver task model consists only of mandatory parts and allows to specify priorities and relative deadlines shorter than the length of a period. The tasks are described by classical static priorities p_i, WCET C_i, deadlines D_i and periods T_i, with $D_i < T_i$. The tasks are denoted

by (p_i, C_i, D_i, T_i) quadruples. Although this model cannot guarantee other quality levels than 0% and 100%, it is well-understood and is an appropriate abstraction for hardware drivers with strict deadlines. An admission based on time-demand analysis can be efficiently done on-line, and has been implemented within the work for this dissertation.

Note that a real-time driver that uses this strict task model does in general not affect, which scheduling models are used by its client applications. Applications are free to use any of the DROPS scheduling models, especially QAS or QRMS to achieve a better resource utilization.

Combining (p_i, C_i, D_i, T_i) tasks with QAS and QRMS

QRMS can directly be mapped to the (p_i, C_i, D_i, T_i) model once the execution time quantiles have been identified. To combine task sets of QAS with the (p_i, C_i, D_i, T_i) model, the (p_i, C_i, D_i, T_i) tasks must be included in the QAS analysis as additional mandatory tasks with their given WCET, deadline and period. In this case, the periods T_i must fulfill the restrictions of the QAS model, that is equal periods or harmonic periods. After assigning priorities with QAS, the deadlines of the (p_i, C_i, D_i, T_i) tasks must be checked with time-demand analysis. If the QAS task set only contains mandatory parts, the QAS tasks can be directly mapped to (p_i, C_i, D_i, T_i) quadruples and can be admitted online using time-demand analysis.

Summary: The open discussion on which of the scheduling models QAS or QRMS is going to be the standard model in DROPS requires applications to be flexible. If for instance drivers with real-time guarantees are to be used with applications using the QAS model, the drivers must be able to adapt their scheduling periods to external restrictions.

2.2.3. Kernel scheduling model

The Fiasco mikrokernel is used to execute the real-time and non–real-time applications of DROPS. A user-level scheduler determines thread priorities, periods, and time quanta and passes these parameters to the kernel. The actual scheduling of applications is then done by the kernel based on these parameters.

Best-effort threads

Fiasco uses the fixed-priority scheduling scheme of L4 to schedule best-effort threads. Threads of the highest priority level are scheduled until they voluntarily yield the CPU by blocking in a synchronous IPC. Figure 4 shows the execution graph of three threads with different priorities.

Real-time threads

In addition this, Fiasco provides a capacity-reserves–like reservation mechanism to schedule real-time threads. The idea of capacity reserves is to assign priority/quantum/period triples (p_i, w_i, T_i) and an budget to threads. The budget is replenished to the quantum at the begin of each period.

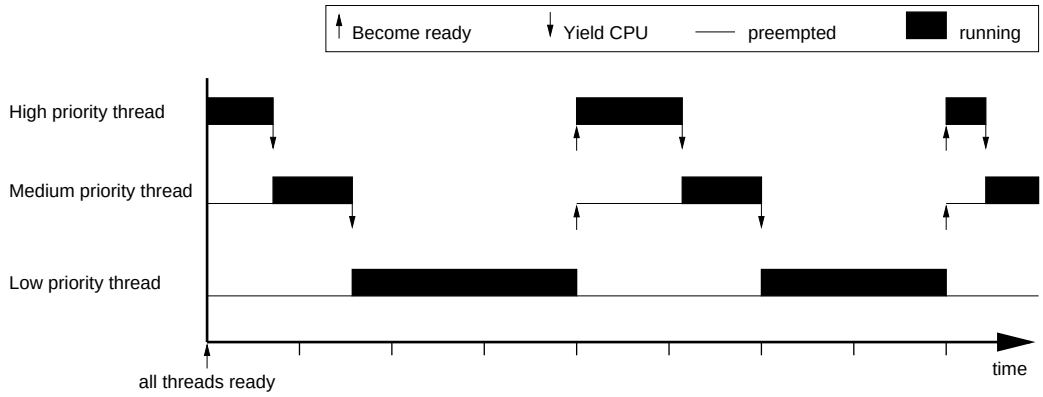

Figure 4: Standard L4-type fixed-priority scheduling. Low-priority threads run only if higher prioritized threads voluntarily yield the CPU.

Whenever a thread is executed, its budget is charged. If the budget becomes null, the thread will not be scheduled until the begin of its next period. Figure 5 illustrates the capacity reserves based scheduling.

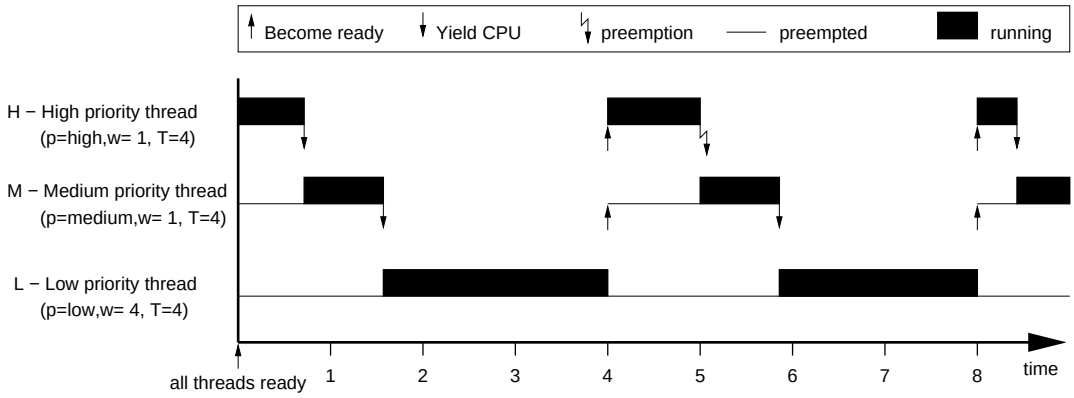

Figure 5: Capacity-reserves scheduling. Thread H yields the CPU at time 0.8. At time 4.0 it becomes ready with a replenished quantum. At time 5.0 the quantum is over and it is preempted by the kernel.

To support the concept of optional parts in QAS, Fiasco allows to assign multiple priority/quantum pairs (p_i^j, w_i^j) to a thread [Ste04]. Thus, each part of a QAS job corresponds to one pair, a job corresponds to the current period the thread is executing, and a QAS task corresponds to a Fiasco thread. Fiasco stores the thread scheduling information data in *timeslices*. Each timeslices consists of a (p_i^j, w_i^j) pair and a budget holding the CPU time left in the current period for that pair.

Thread scheduling At the begin of each thread's period, Fiasco replenishes all budgets of a thread and activates the first timeslice. This means, the effective priority of thread i is set to p_i^1.

The thread can execute on this priority for w_i^1 time units. After this time, the kernel activates the next timeslice, and so on. When all timeslices are used up, the thread falls back to its best-effort priority p_i^0. Before the end of a timeslice, a thread can voluntarily yield the CPU. To do so, it switches to the next timeslice. This may result in a lower effective priority, so the thread may be preempted by other threads. Alternatively, the thread can wait for the begin of the next period skipping all remaining timeslices.

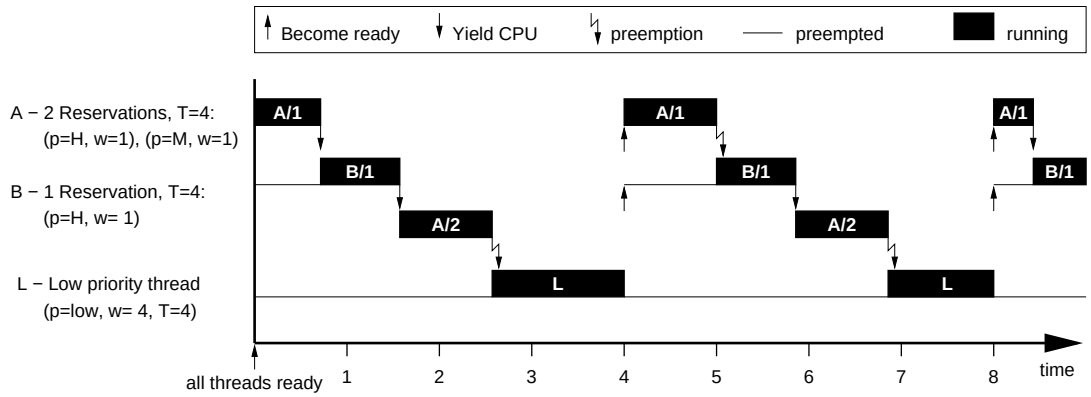

Figure 6: Fiaso's thread scheduling. At time 0.8, thread A voluntarily switches from timeslice 1 to timeslice 2, lowering its priority to M. Thread B is scheduled. At time 1.7 B waits for the begin of its next period. Thread A is scheduled at its second timeslice. At time 2.7, A's quantum is over and it is preempted by the kernel. Thread L is scheduled.

Application use To use this kernel mechanism for QAS scheduling, the mandatory parts of QAS are mapped to the first timeslice and are given a high priority. The optional parts are mapped to the following timeslices with lower priorities. Best-effort threads are assigned the lowest priorities. At the end of a each part, a real-time thread switches to the next timeslice, potentially yielding the CPU to another thread (Figure 6).

The admission of (p_i^j, w_i^j) pairs is done by a user-level admission controller, which can guarantee deadlines D_i^j using response-time analysis. This allows to implement the DROPS scheduling models QAS and QRMS after mapping the QAS and QRMS parameters to (p_i^j, w_i^j) pairs using the appropriate admission scheme. The (p_i, C_i, D_i, T_i) parameters of the driver task model can be admitted directly.

Minimal inter-release time scheduling Using the terminology of [Liu00], best-effort threads of Fiasco resemble aperiodic (sporadic) threads. The real-time threads of Fiasco described so far resemble strictly periodic threads. In addition to this, Fiasco supports also periodic threads in the terminology of LIU. While strictly periodic threads are always released at the begin of their period, periodic threads have a minimal inter-release time, whose release is coupled to an event. In Fiasco this event is the successful execution of a *next-period IPC* issued by the periodic thread.

When the next-period IPC is successfully executed, the thread is released again, but not before the time passed since its last release is at least its minimal inter-release time. Early wakeup events will be buffered, as illustrated in Figure 7. From a scheduling analysis perspective, strictly periodic threads and periodic threads are treated equally. As such, the term *period* is used to describe both the period of strictly periodic threads and to describe the minimal inter-release time of periodic threads.

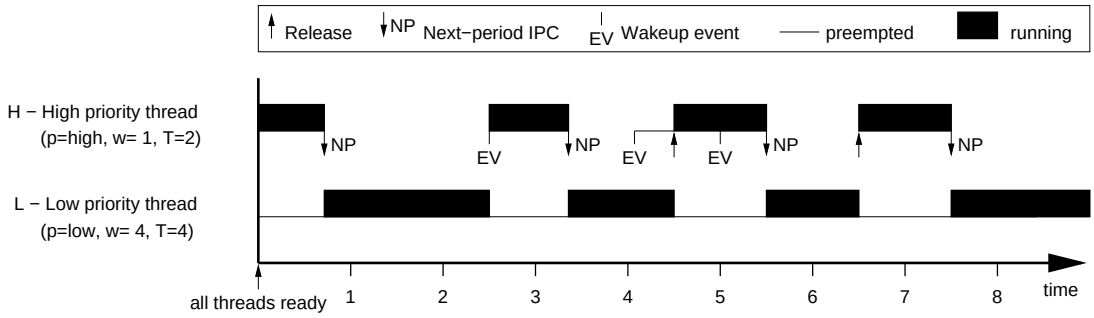

Figure 7: Fiasco's minimal inter-release time scheduling. At time 0.8, thread A issues a next-period IPC. At time 2.5, this IPC is executed, thus an event is successfully received from another thread. The kernel sets the earliest next release time to 4.5. The event at time 4.0 is queued until time 4.5. Analogously, the event at time 5.0 releases the thread at time 6.5.

Kernel scheduling API

As part of its real-time API the kernel provides a set of functions to control the scheduling of threads. The most important are the following:

void *l4_rt_set_period* (**l4_threadid_t** thread, **l4_kernel_clock_t** period);
sets the length of the period for the given thread.

int *l4_rt_add_timeslice* (**l4_threadid_t** thread, **int** prio, **int** wcet);
adds a timeslice of the length *wcet* and priority *prio* to the given thread.

int *l4_rt_begin_strictly_periodic* (**l4_threadid_t** dest, **l4_kernel_clock_t** clock);
starts strictly periodic execution of the given thread at time *clock*.

int *l4_rt_begin_minimal_periodic* (**l4_threadid_t** dest, **l4_kernel_clock_t** clock);
starts periodic execution with minimal inter-release times of the given thread at time *clock*.

int *l4_rt_next_reservation* (**unsigned** id, **l4_kernel_clock_t***clock);
voluntarily switch to the next timeslice and return the remaining CPU quantum of the current timeslice in **clock*.

int *l4_rt_next_period* (**void**);
wait for the next period, skipping all unused timeslices.

Code Example

Listing 1 shows the code of a strictly periodic real-time thread. It continuously executes a mandatory part *work_mandatory()* followed by an optional part *work_optional()*. To wait for the successful start of the periodic mode, *worker()* issues a *l4_rt_next_period()* system call at the begin. The setup of the scheduling parameters is done in *init()*. For the sake of a better readability, the example performs no error handling.

```
void worker(){
    l4_kernel_clock_t left;
    /* initial next-period call: wait for start of periodic mode */
    l4_rt_next_period();
    while(1){
        work_mandatory();                    /* do mandatory work */
        l4_rt_next_reservation(1, &left);    /* switch to the optional timeslice */
        work_optional();                     /* do optional work */
        l4_rt_next_period();                 /* wait for the begin of the next period */
    }
}
void init(){
    l4_threadid_t t = thread_new(worker);
    l4_rt_set_period(t, 1000);               /* period length is 1ms */
    l4_rt_add_timeslice(t, 80, 100);         /* add 100µs timeslice at priority 80 */
    l4_rt_add_timeslice(t, 50, 200);         /* add 200µs timeslice at priority 50 */
    l4_rt_begin_strictly_periodic(t, 0);     /* immediate start of periodic mode */
}
```

Listing 1: Example of a strictly periodic real-time thread with a period of 1 ms, an optional part of 100 µs and a mandatory part of 200 µs.

3. Related work

Various real-time transfer solutions exist for the original CSMA/CD Ethernet. They differ in the methods to control the access to the network. While hard real-time guarantees can only be given with exclusive network access, statistical guarantees allow for concurrent access.

A common method to achieve hard real-time guarantees is the **token-based** approach, where a circulating token represents the permission to transmit data. An advantage of token-based solutions is their flexibility, as they can be used with almost any network architecture. A disadvantage is that only one station owns the transmit right at a given time. This unnecessarily limits the performance on modern switched networks. Also, time and network bandwidth for the token management affects the overall performance. In [VcC94] Venkatramani and Chiueh present results from the "RETHER" project. They simulated a 10 MBit/s CSMA/CD Ethernet with a maximum token rotation time of 33 ms (thus delays bounds are $\geq$33 ms) and achieved a network utilization of 60%. With 100 MBit/s Ethernet they gained only a small throughput increase, which they attribute to the dominance of software overheads.

Another method for controlling access to the network is the **time-slotted** approach. Examples can be found in [KDK+89, Sch02]. There is a fundamental problem with this approach: the longer the time-slots, the higher is the worst-case delay of messages transferred. Therefore, time-slots should be as short as possible. On the other side, time-slots must be long enough to prevent overlapping of messages due to delays and jitter imposed by the network. This may lead to a severe performance cut, as the 4% overall network utilization for Switched Gigabit Ethernet reported by SCHWARZ in [Sch02].

A mixed approach is presented by PEDREIRAS and ALMEIDA in **FTT-Ethernet** [PAG02], where a central master periodically distributes tokens that allow to send data for a specific amount of time.

KWEON and SHIN describe in [KSZ99] a method to achieve **statistical real-time** guarantees. The idea of such a statistical approach is to keep the overall network traffic below a certain limit. As the probability of a collision is reduced with lower network load, a statistical guarantee for the transmission time and bandwidth can be given, provided the participating nodes keep their traffic below certain limits. In [KS00a, KS00b] KWEON and SHIN extended their work by adaptive traffic smoothing that automatically adapts to the current load situation on the network. Besides the probabilistic behavior of this approach, the overall network utilization decreases with stronger statistical guarantees. They report an experiment of 10 nodes connected by a 10 MBit/s CSMA/CD Ethernet, exchanging real-time traffic with a total bandwidth of 53 KBit/s and non–real-time traffic with a total bandwidth of 4.4 MBit/s. The deadline-miss ratio was 10^{-4} with a deadline of 129 msec.

LO BELLO and others extended the statistical approach by **fuzzy traffic smoothing** [CBM02]. They use the overall throughput together with the number of collisions as network load indicators and feed them into fuzzy traffic smoothers. This allows them to handle sporadic traffic more flexible.

Most of these methods developed for the shared CSMA/CD Ethernet are established today; the token-based and time-slotted approaches could also be used for Switched Ethernet. However,

none of them makes use of a collision-free switched architecture that allows higher utilization bounds and lower delay bounds.

Schedulability conditions for different packet scheduling methods in network switches are given by LIEBEHERR in [LWD96]. The schedulability conditions allow to guarantee **delay bounds depending on the input traffic characteristics** and the selected scheduling method in the switch. They apply their theory in examples to token-bucket shaped (leaky-bucket shaped) traffic, but not to the more general form of T-SPECs[1]. Their analytical result for FIFO scheduling coincides with Equation 16 on page 24.

In [WJ03], WATSON and JASPERNEITE apply the Network Calculus of LE BOUDEC [BT01] to Switched Ethernet and discuss multiple switches in a line topology. WATSON provides analytical and simulation results of a 100 MBit/s switched Ethernet. With 50 nodes connected in a line topology and producing bursts of 15 KByte with a high ($\geq$90%) network utilization, the delay bounds for message transfer are in the order of 400 ms. In contrast to the traffic model I use, WATSON and JASPERNEITE use the simpler model of single leaky-bucket shaped streams, and further neglect the processing time in the switch. For one switch, their results coincide with the estimations given in Equations 14 and 16 on page 23f. Also, their work is done on a theoretical basis only. The main aspect of my dissertation are the operating system requirements of real-time systems. Assuming that the operating system issues have become clear and well understood once traffic shaping is applied to Ethernet networks with one switch, an extension to networks with multiple switches solely requires extending the mathematical model. The principal operating system requirements however do not change.

In [JE04] JASPERNEITE and others discuss the emerging **PROFINET IRT** standard. Multiple nodes are connected by switches. The approach is time-triggered with a real-time phase and a non–real-time phase. In the real-time phase, the switches forward packets according to a time-triggered database resident in each switch and the arrival time and input port of the packets. In the non–real-time phase, normal MAC-based packet forwarding is used. According to [JE04], this separation of time in real-time phases and non–real-time phases decreases the latency of the real-time traffic considerably, especially when multiple switches are connected in a row. However, to achieve these low latencies, PROFINET RT requires specific hardware, both for the switches and for the end nodes.

YOSHIGOE and CHRISTENSEN propose in [YC01] a **rate control** mechanism for bandwidth allocation on an Ethernet in the first mile (EFM) subscriber service. Their emphasis is on bandwidth allocation, not on end-to-end delay guarantees. They propose extending the 802.3 standard, using a leaky-bucket controller at the transmitter of each Ethernet connector. The rate control applies uniformly to all traffic coming from one node. Therefore, it cannot help in policing different real-time connections. Also, this proposal requires specialized hardware. The software implementation proposed in this dissertation gives delay guarantees and uses available standard hardware.

GUÉRIN and others present in [GKPR98] a mechanism for providing quality of service (QoS) through **buffer management**. The idea is to limit explicitly the amount of buffer space the switch

[1] In the literature, the terms *token-bucket* and *leaky-bucket* are often used interchangeably. Section 4.2.2 on page 20 clarifies this and defines the terms as used throughout this dissertation.

provides for a specific connection. With their approach bandwidth allocations can easily be managed, and free bandwidth is distributed fairly among best-effort traffic. GUÉRIN reports a simulation of a 48 MBit/s switched network. With a switch buffer size of 500 KByte and FIFO scheduling, he achieves a network utilization of 85% without loosing data. For a 99% utilization, 5 MByte switch buffer are needed. In contrast to the approach of this dissertation, GUÉRINs proposal requires installation of a filtering instance inside the switch, thus it does not work with off-the-shelf hardware.

Several projects aim at building QoS-enabled switches or routers from scratch [GLN+99, CL01]. Notable work is **EtheReal** by CHIUEH and VARADARAJAN [VC98]. They built a switch that allows connection establishment with policing. The main feature is the integration in *any* OS: They use special IP-addresses and MAC-addresses to place the communication-ID therein. This way, they do not change the host operating system, but require elaborate switches executing their applications.

Ethernet flow control as defined in the IEEE 802.3 extension 802.3x supports a MAC-level flow control with special *pause frames*. Flow control allows a receiver to ask the sender to stop generating data. Flow control can help preventing retransmissions due to packet loss. However, the current specification of flow control does not differentiate between multiple connections and pause frames are sent uniformly to all ports of a switch. Further, pause frames are interpreted by the network interface card at a node (NIC). The NIC has no notion of real-time and best-effort traffic and suspends the transmission of all traffic upon the reception of pause frames. Hence, rate control in the sending nodes is still necessary to isolate different connections and to prevent flooding the switch.

In his PhD dissertation, BORRISS implemented a real-time ATM network stack for the DROPS operating system [Bor99]. He solved the resource problem of concurrent real-time and non–real-time communications by explicitly assigning resources to connections, and by separating connection handling into individual threads. His work revealed that a fast demultiplexing of data received from the network to the appropriate connection is crucial for the overall performance. Based on this observation, DANNOWSKI implemented an ATM firmware that did the demultiplexing at the network interface card [Dan99], significantly improving the overall performance. The firmware offloading approach could also be used in the scope of this dissertation, as discussed in Section 6.10 on page 83.

4. QoS at the hardware level

In this section I review the theory on delay and buffer calculation on network elements and apply it to the projected network configuration – a Switched Ethernet network with one switch and uniform medium bandwidth. I start with an introduction of the general network calculus of LE BOUDEC as the basic theoretical model to calculate network behavior. Using this calculus I (1) characterize and quantitatively analyze traffic flows that are sent from end nodes to a switch, (2) analyze the delays and buffer needs of an Ethernet switch by deriving formulae for safe bounds depending on the input traffic characteristic to the switch, and (3) give bounds for the traffic flow characteristics as they leave the switch to the destination nodes. Finally, I detail the traffic reservation scheme needed for proper operation and outline the aspects for networks with multiple switches.

4.1. Network calculus

Obtaining the maximum queue lengths (backlog) and the maximum queueing delay in network switches has been intensively researched in the past, especially in the context of ATM networks. CRUZ [Cru91a] was the first who published a calculus on networking delays, and LE BOUDEC [BT01] later developed a more elegant calculus. Based on this, numerous work was done to calculate delays, buffer requirements and loss probabilities for statistical real-time systems [BH00, Wat02, KS00a, LWD96, PL01, WXBZ01, WJ03].

4.1.1. Definitions and theorems

In [BT01] LE BOUDEC introduces the following terms and theorems:

Data flows $R(t)$: Data flows are described by means of the cumulative function $R(t)$, defined as the number of bits seen on a flow in time interval [0,t]. By convention, $R(0) = 0$.

Set F of wide-sense increasing functions A function f is wide-sense increasing if and only if $f(s) \leq f(t)$ for all $s \leq t$. F denotes the set of wide-sense increasing sequences or functions such that $f(t) = 0$ for $t < 0$.

Arrival Curve: Given a wide-sense increasing function α defined for $t \geq 0$ (namely, $\alpha \in F$), we say that a flow R is constrained by the arrival curve α if and only if for all $s \leq t$:

$$R(t) - R(s) \leq \alpha(t-s)$$

min-plus convolution $f \otimes g$: Let f and g be two functions or sequences of F. The *min-plus convolution* of f and g is the function

$$(f \otimes g)(t) = \inf_{0 \leq s \leq t} (f(s) + g(t-s))$$

min-plus deconvolution $f \oslash g$: Let f and g be two functions or sequences of F. The *min-plus deconvolution* of f by g is the function

$$(f \oslash g)(t) = \sup_{u \geq 0}\{f(t+u) - g(u)\} \tag{1}$$

Service Curve: Consider a system S and a flow through S with input and output function R and R^*. We say that S offers to the flow a **service curve** β if and only if $\beta \in F$ and $R^* \geq R \otimes \beta$.

Backlog Bound Assume a flow, constrained by arrival curve α, traverses a system that offers a service curve β. The backlog $R(t) - R^*(t)$ for all t satisfies:

$$R(t) - R^*(t) \leq \sup_{s \geq 0}[\alpha(s) - \beta(s)] \tag{2}$$

Delay Bound Assume a flow, constrained by arrival curve α, traverses a system that offers a service curve of β. The virtual delay $d(t)$ for all t satisfies:

$$d(t) \leq h(\alpha, \beta) \tag{3}$$

with $h(\alpha, \beta)$ being defined as the horizontal derivation, $h = \sup_s \delta(s)$ with $\delta(s) = \inf\{\tau \geq 0 : \alpha(s) \leq \beta(\tau + s)\}$.

Output Bound Assume a flow, constrained by arrival curve α, traverses a system that offers a service curve of β. The output flow is constrained by the arrival curve

$$\alpha^* = \alpha \oslash \beta \tag{4}$$

Minimum arrival curve Consider a flow $R(t)_{t \geq 0}$. Then

- Function $R \oslash R$ is an arrival curve for the flow.
- For any arrival curve α that constraints the flow, we have: $R \oslash R \leq \alpha$.

4.1.2. Application to Switched Ethernet

The system S in the previous section is a network element that delays data flows. In the context of Switched Ethernet, this corresponds (1) to the NICs' hardware FIFO queues at the sending nodes, and, (2) to the demultiplexing logic and an output port of a switch. Both schedule aggregates of data flows in a FIFO manner. Each such aggregate consists of a number N of flows $\alpha_{1 \leq i \leq N}$, and is constrained as a whole by the sum of the flows, described by $\sum_{1 \leq i \leq N} \alpha_i$. Note that the data flows, and as such the aggregate, changes its characteristics when it traverses a network element.

If data is available to a NIC's hardware FIFO queue, the NIC transmits this data the rate C of the network medium. Thus, the service curve of a NIC is given by the *rate function*:

$$\beta_{NIC}(t) = C \cdot t$$

If data is available to a switch output port, the switch transmits the data with the rate C of the network medium. The multiplexing of packets and other internal management adds a bounded delay to the packets of a flow, which is denoted by t_{mux}. Thus, the service curve of a switch output port is given by the *rate-latency function*:

$$\beta_{switch}(t) = C \cdot (t - t_{mux})^{+} \tag{5}$$

with $(t - t_{mux})^{+}$ defined to be 0 for $t < t_{mux}$.

4.2. Bounding network delays

In this section, I give exact delay and buffer bounds formulae for networks with one switch. For quick delay and buffer estimations, the section also contains the derivation of simple formulae for bounds that are safe but not necessarily tight.

4.2.1. Definitions

Throughout this dissertation, the following terms refer to times related to frame and packet transmission.

frame-transmission delay $(\mathbf{t_{frame}})$ is the time needed to transmit a frame over the Ethernet medium. For maximum-sized frames (1514 bytes) t_{frame} is 121 µs for Fast Ethernet and 12 µs for Gigabit Ethernet [2].

NIC-queueing delay $(\mathbf{t_{NIC}})$ is the time a queued frame sits in the hardware FIFO queue of a NIC plus the time needed to transmit it finally. The NIC-queueing delays solely depend on the queue length. Bounding this length results in bounded NIC-queueing delays.

switch-multiplexing delay $(\mathbf{t_{mux}})$ is a switch-specific parameter describing the maximum delay (without queueing effects) after which the switch starts to transmit a frame once it is received.

queueing delay $(\mathbf{t_{queue}})$ is the time a queued frame sits in the output queue of a switch plus the time needed to transmit it finally. With first-in–first-out queues (FIFOs), queueing delays solely depend on the queue length. Bounding this length results in bounded switch-queueing delays.

switch delay $(\mathbf{t_{switch}})$ is the time a frame is delayed at a switch. $t_{switch} = t_{mux} + t_{queue}$.

[2] 1514 bytes at 100 MBit/s or 1000 MBit/s considering the framing overhead inter-packet gaps

operating-system delay ($\mathbf{t_{os}}$) covers the delays at the nodes due to interrupt handling and scheduling. It is the sum of the maximum delay at the sender and the maximum delay at the receiver.

packet-transmission delay ($\mathbf{t_{trans}}$) is the application-to-application delay of a packet sent over the network. For two nodes connected by a switch, $t_{trans} = t_{switch} + t_{os} + t_{NIC}$.

transmission-delay bound ($\mathbf{t_{max}}$) is the upper bound of the packet-transmission delay. This especially requires knowledge about the maximum queueing delay.

observed transmission delay ($\mathbf{t_{obs}}$) is the measured application-to-application delay of a packet sent over the network.

4.2.2. Modeling network traffic

For later use, I introduce the definition of a *leaky-bucket controller*, *token-bucket controller* and derived terms as they can for instance be found in [BT01]:

Definition (Leaky-Bucket Controller): A *leaky-bucket controller* is a device that analyzes the data on a flow $R(t)$ as follows. There is a pool (bucket) of fluid of size b. The bucket is initially empty. The bucket has a hole and leaks at a rate of r units of fluid per second when it is not empty. Data from the flow $R(t)$ has to pour into the bucket an amount of fluid equal to the amount of data. Data that would cause the bucket to overflow is declared *nonconformant*, otherwise the data is declared *conformant*.

Corollary (Conformance to a leaky-bucket): As an immediate consequence, a flow $R(t)$ is *conformant to a leaky-bucket with parameters* (r,b) if $R(t)$ is constrained by the arrival curve $\alpha(t) = rt + b$.

It follows from the definition that for a given data flow $R(t)$ with average rate r, Equation 6 gives the *burstiness* parameter b, so that $R(t)$ is conforming to a leaky-bucket with parameters (r,b): The term in the braces is the difference between the amount of data being sent in the interval from s to t and the bucket replenishment in this interval.

$$b_i = sup_{t>s}\{R(t) - R(s) - r_i \cdot (t-s)\} \tag{6}$$

Definition (Dual leaky-bucket): When a flow traverses two leaky-bucket controllers with parameters (r_1,b_1) and (r_2,b_2), $b_1 > b_2$ and $r_2 > r_1$ in a sequence, the flow becomes a *dual leaky-bucket* constrained flow with parameters (r_1,b_1,r_2,b_2).

Corollary (Arrival curve of a dual leaky-bucket constrained flow): A dual leaky-bucket constrained flow with parameters (r_1,b_1,r_2,b_2) has the arrival curve $\alpha(t) = \min(r_1t + b_1, r_2t + b_2)$. The same holds for a leaky-bucket constrained flow with parameters (r_2,b_2,r_1,b_1). The proofs can be found in [BT01].

Definition (Token-bucket Controller): A *token-bucket controller* is a device that analyzes the data on a flow $R(t)$ as follows. There is a pool (bucket) of fluid of size b. The bucket is initially

full. Fluid drips into the bucket with a rate of r units of fluid per second when it is not full.
Data from the flow $R(t)$ has to remove from the bucket an amount of fluid equal to the amount of data. Data that finds not enough fluid in the bucket is declared *nonconformant*, otherwise the data is declared conformant.

Remark: The definition of a *token-bucket controller* is equivalent to the definition of a *leaky-bucket controller* given before. Both terms can be found in the literature describing the same behavior. This section regularly references the work of LE BOUDEC, FIEDLER and others preferring the term *leaky-bucket*, and as such uses this term. The model of a *token-bucket* controller, however, is more intuitive when describing the implementation and hence will be used in the later implementation sections. The term **conformance to a token-bucket controller** shall be defined accordingly.

The concept of dual leaky-bucket constrained flows is commonly used. In the context of ATM and the Integrated Services framework of the Internet ATM, the parameter set is called a T-SPEC (traffic specification). T-SPECs (C, M, r_k, b_k) describe data flows with a long-term average rate r_k, which can be exceeded for a bounded number of bits b_k to tolerate jitter. This (r_k, b_k) shaping is typically done actively by a single leaky-bucket shaper. The additional (C, M) constraint is the result of sending the (r_k, b_k) shaped flow over a network medium that transmits data with rate C and has a maximum packet size M. The network medium resembles a (C, M) leaky-bucket shaper.

Note that by definition dual leaky-bucket constrained flows with parameters (r_1, b_1, r_2, b_2) are always single leaky-bucket constrained by both (r_1, b_1) and (r_2, b_2). The constraints are upper bounds, and thus a T-SPEC (C, M, r_k, b_k) describes a flow more precisely than the single leaky-bucket constraint (r_k, b_k). If the T-SPECs describing flows through an Ethernet switch are tight, tight bounds for delay and buffer usage at the switch can be derived[3]. Nonetheless, valid network calculation can also be done by using the simpler form of single leaky-bucket constraints. The use of single leaky-bucket constraints is common in the literature [LWD96, WJ03], as it easily allows an analysis of larger networks containing multiple switches. However, omitting the restriction of the medium results in pessimistic assumptions of the arriving traffic at the network switches, not giving tight delay and buffer bounds therein.

Consequently, for network calculations in this dissertation I model the data flows arriving at the switch by T-SPECs. Note that before the end nodes pass their data flows to the network medium, they are constrained by single leaky-buckets only.

4.2.3. Delay and burstiness increase at NICs

When multiple, independently produced data flows are sent to the NIC of the producing node, the data flows are delayed and their burstiness increases. Quantitative analyses based on the formal description of a traffic flow must be aware of this, such as the calculation done at the bandwidth manager described in Section 6.1 on page 41.

[3] For a proof see [BT01].

In [CEB02], CHOLVI, ECHAGÜE and LE BOUDEC applied the network calculus to the multiplexing of multiple leaky-bucket shaped data flows to one physical channel. Equation 7 gives the resulting burstiness parameters for each flow, if the original flows conform to (r_k, b_k^0) leaky buckets. Once these flows are transmitted at the network medium, they are constrained by the T-SPECs (C, M, r_k, b_k).

$$b_k \leq b_k^0 + r_k \cdot \frac{\sum_{j \neq k} b_j^0}{C} \tag{7}$$

The delay bound for delaying these flows is given by Equation 8, and can be found in [BT01].

$$d_k \leq \frac{\sum_i b_i}{C} \tag{8}$$

4.2.4. Delay and buffer calculation of switches

Recapitulating Sections 4.1 and 4.2.2, the delay and buffer bounds of an Ethernet switch transmit port depend

1. on the traffic arriving at the switch for that transmit port, described by its *arrival curve* α
2. on the availability of the switch to send that data, described by the *service curve* β.

The arrival curve α is the sum of the arrival curves of the traffic at the receive ports α_k, with k denoting the receive port. α_k is described by T-SPEC k with $\alpha_k(t) = \min(Ct + M, r_k t + b_k)$.

$$\alpha(t) = \sum_{k=1}^{N} \min\{Ct + M, r_k t + b_k\}. \tag{9}$$

The service curve of the Ethernet switch is described by the *rate-latency function*:

$$\beta(t) = C \cdot (t - t_{mux})^+$$

Figure 8 shows the arrival curve of a switch with two receive ports and its service curve.

Obviously, the sum of the long term average input rates r_k of traffic for one switch transmit port must not exceed the maximum rate of the network medium, thus the following must must:

$$\sum_{k=1}^{N} r_k \leq C \tag{10}$$

According to Section 4.1, the maximum backlog B is the maximum vertical distance between the arrival curve α and the service curve β. Let g_k denote the time of the inflexion point of arrival curve α_k:

$$g_k = \frac{b_k - M}{C - r_k} \tag{11}$$

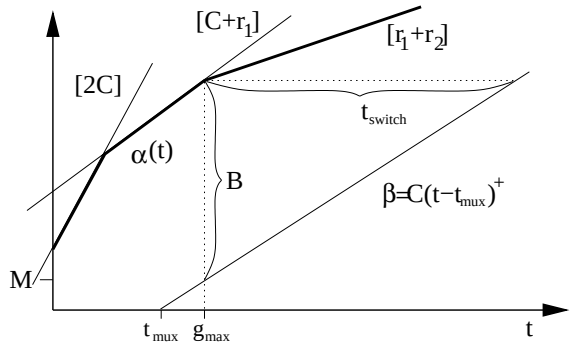

Figure 8: Illustration of the arrival curve $\alpha(t)$ (thick line) as the sum of two flows (C, M, r_1, b) and (C, M, r_2, b). The slopes of the 3 parts of $\alpha(t)$ are $2C$, $C + r_1$ and $r_1 + r_2$.

and let further g_{max} denote the maximum of all g_k:

$$g_{max} = max_{k=1}^{N}(g_k) \tag{12}$$

As argued in [Loe03a], $t_{mux} \leq g_{max}$ in practice. Hence, the maximum vertical distance between α and β is at g_{max} and the buffer bound B is

$$\begin{aligned} B &= \sum_{k=1}^{N} b_k + \sum_{k=1}^{N} r_k \cdot g_{max} - C \cdot (g_{max} - t_{mux}) \\ B &= \sum_{k=1}^{N} b_k - g_{max} \cdot (C - \sum_{k=1}^{N} r_k) + C \cdot t_{mux} \end{aligned} \tag{13}$$

If B exceeds the amount of memory the switch can use for buffering, frame loss may occur. For reliable hard real-time communication this must be prevented.

According to Equation 10, the second addend in Equation 13 is negative or zero, and hence an upper bound for the backlog formula is

$$B_{est} = \sum_{k=1}^{N} b_k + C \cdot t_{mux}. \tag{14}$$

This means, the memory required in the switch can be estimated by the sum of the bursts in the T-SPECs plus a small fixed amount ($C \cdot t_{mux}$).

According to Section 4.1, the maximum delay d of a system that offers a service curve β to a flow that is constrained by an arrival curve α and serviced in FIFO order, is given by the maximum horizontal deviation between α and β. It is the distance between α and $C \cdot (t - t_{mux})$ at g_{max}, divided by the slope of β, which is C.

Hence, the delay bound is

$$t_{switch} = \sum_{k=1}^{N} \frac{b_k}{C} - g_{max} \cdot (1 - \sum_{k=1}^{N} \frac{r_k}{C}) + t_{mux} \tag{15}$$

In analogy to Equation 14, the switch delay is also bounded by:

$$t_{est} = \sum_{k=1}^{N} \frac{b_k}{C} + t_{mux}. \tag{16}$$

This means, an estimation for the maximum delay of the switch is given by the time needed to transmit the bursts of the T-SPECs with the ports maximum bandwidth plus the delay imposed by the electronics of the switch.

LE BOUNDEC has shown in [BT01] that the buffer and delay bounds in Equation 2 and Equation 3 are tight. As I used tight arrival and service curves, Equations 13 and 15 give tight bounds. If, however, the traffic is described by a single leaky-bucket (r,b), the results are pessimistic. Equations 14 and 16 coincide with the results obtained in [LWD96, WJ03].

4.3. Burstiness increase at the switch

When data flows with a known T-SPEC characteristic are merged on network elements by aggregate FIFO scheduling, as if they were traversing a switch output port, their burstiness increases. In contrast to the burstiness increase at the NIC, where the data flows are single leaky-bucket constrained, the flows at a switch are modeled with dual leaky-buckets constraints. This allows to give tighter bounds on their scheduling delay, buffer need and burstiness increase, as shown in the previous section. However, the calculation becomes more complex. In this section, I apply the work of FIDLER et al [FSK05] to calculate the burstiness increase of data flows constrained by two leaky-buckets (T-SPECs) when they traverse a switch. Section 6.4.1 uses these results for predicting the amount of CPU needed to handle arriving data flows at a node.

4.3.1. Theoretical background

In [FSK05], Fidler proves the following theorem and concludes the corollary:

Theorem (Output Bound, Rate-Latency Case) Consider two flows 1 and 2 that are α_1^j and α_2^j upper constrained. Assume these flows are served in FIFO order and in an aggregate manner by a node j that is characterized by a minimum service curve of the rate-latency type $\beta^j(t) = R^j \cdot [t-T]^+$. Then, the output of flow 1 is α_1^{j+1} upper constrained according to Equation 17, where θ is a function of t and itself and has to comply with Equation 18.

$$\alpha_1^{j+1}(t) = \alpha_1^j(t+\theta(t)) \tag{17}$$

$$\theta(t) = \frac{\sup_{v>0}[\alpha_1^j(v+t+\theta(t)) - \alpha_1^j(t+\theta(t)) + \alpha_2^j(v) - R^j \cdot v]}{R^j} + T^j \tag{18}$$

Corollary (Output Bound, Single Leaky Bucket Case) In case of a single leaky bucket constrained flow or traffic trunk 1, with rate r_1 and burst size b_1^j, Equation 18 can be simplified applying $\alpha_1^j(v+t+\theta(t)) - \alpha_1^j(t+\theta(t)) = r_1 \cdot v$. As an immediate consequence, θ becomes independent

of t. With Equation 17 we find that the output flow 1 is leaky bucket constrained with r_1 and b_1^{j+1} according to Equation 19.

$$b_1^{j+1} = \alpha_1^j(\theta(0)) = b_1^j + r_1 \cdot \theta(0) \tag{19}$$

Equation 18 becomes Equation 20

$$\theta(0) = \frac{\sup_{v>0}[r_1 \cdot v + \alpha_2^j(v) - R^j \cdot v]}{R^j} + T^j \tag{20}$$

4.3.2. Application to Switched Ethernet

Equation 19 can be applied immediately to data flows described by T-SPECs that are multiplexed to an output port of a switch. The node j corresponds to the switch output port, its output rate R^j to the medium bandwidth C, and its service curve is given by Equation 5.

When calculating the burstiness of one flow i out of N flows $R_{1\leq i\leq N}$ that are constrained by arrival curves $\alpha_k(t)$ and that are multiplexed together to the switch output port, flow 1 in Fidlers theorem corresponds to R_i and flow 2 corresponds to the aggregate of all flows R_k with $k \neq i$: $\alpha_1^j(t) = \alpha_i(t)$ and $\alpha_2^j(t) = \sum_{k\neq i} \alpha_k(t)$.

Consequently, the flow R_i that traverses the switch becomes $\tilde{R}_i$ constrained by $\tilde{\alpha}_i(t) = r_i \cdot t + \tilde{b}_i$ with $\tilde{b}_i$ given by Equation 21

$$\tilde{b}_i = \alpha_i(\theta_i(0)) \tag{21}$$

$$\theta_i(0) = \frac{\sup_{v>0}[r_i \cdot v + \sum_{k\neq i} \alpha_k(v) - C \cdot v]}{C} + t_{mux} \tag{22}$$

Using Equation 10, the $sup[...]$ of Equation 22 is found to be at v_i with v_i given by the following equation:

$$v_i = \max_{k\neq i} g_k \tag{23}$$

4.3.3. Remark

The output flow $\tilde{R}_i$ is described by a single leaky-bucket now. As arrival curves are upper bounds, this constraint is not necessarily tight.

$\tilde{R}_i$ traverses the network medium after leaving the switch. The shaping effect of this has already been considered in the calculation of the switch delay and the calculation of $\tilde{b}_i$. As such, $\tilde{R}_i$ is constrained by a (C,M) single leaky bucket as well, and the flow arriving at the destination node conforms to a T-SPEC with parameters $(C,M,r_i,\tilde{b}_i)$. If multiple flows arrive at a destination node, their aggregate A is bounded by a (C,M) single leaky bucket as well. Consequently, their aggregate conforms to a T-SPEC with parameters $(C,M,\sum_{j\in A} r_j, \sum_{j\in A} \tilde{b}_j)$.

4.4. Traffic reservation

To prevent overloading the switch, a centralized traffic management is needed. In this section, I review established traffic management techniques and briefly present the technique used in this dissertation. Section 6.1 describes the implementation of the traffic management in detail.

4.4.1. Established traffic reservation techniques

To provide QoS guarantees in the context of the Internet, several technologies have been developed and applied with different success. The integrated services approach *IntServ* [BCS94] provides QoS guarantees to data flows based on individual reservations at every traversed router in a larger network. Networks using the IntServ scheme require data flows to use a fixed route through the network, once they are established. For reservation management the resource reservation protocol *RSVP* [BZB+97] has been defined. The strength of RSVP is to control the admission and management of data flows in a large network. It is based on a receiver-initiated reservation process, which forwards reservation requests from the receiver to the direction to the sender, and the way back. This especially means that all nodes on the path keep status information for every individual flow. The traffic flows in RSVP are described with T-SPECs.

This dissertation targets hard real-time application-to-application guarantees on a Switched Ethernet network, solving the problems at the operating system level in the end-nodes specific to Switched Ethernet. For this, I concentrate on networks with one switch. Although the traffic flow characteristics introduced in Section 4.2.2 fit the RSVP model, RSVP solves a problem orthogonal to my target. Therefore, I do not consider using RSVP for traffic flow management in this dissertation.

Another technology developed for providing QoS guarantees in the Internet is the differentiated services approach *DiffServ* [BBC+98]. Its purpose is to overcome the scalability problems related to the per-flow state in every router in IntServ networks. In contrast to IntServ, DiffServ combines traffic flows with similar QoS requirements to behavior aggregates, which are managed as a whole within the network. Individual traffic flows are verified for conformance to a traffic specification only at the entrance to a DiffServ network and assigned to a behavior aggregate then. Inside the network, no individual flow information is kept. Thus, in its most general form, DiffServ allows the routes for the individual packets of a flow to vary. The coarse-grained traffic description of DiffServ and the missing per-flow state within the network result in pessimistic worst-case assumptions for the behavior within the network [BT01]. As a consequence, DiffServ results in lower overall QoS guarantees in comparison to IntServ networks.

As with IntServ, DiffServ solves the problem of managing large networks. DiffServ especially targets the scalability problems related to the handling of individual flows, and as such solves problems orthogonal to this dissertation.

4.4.2. Traffic reservation technique for a Switched Ethernet network

The networks I consider in this dissertation consist of one switch and a limited number of nodes connected to it. As such, the management can be much simpler than the established reservation

schemes for large distributed systems. Especially, scalability problems do not arise.

The nodes communicate with each other using traffic flows described by T-SPECs and (source-node, target-node) pairs. Both the number of nodes and the number of traffic flows can change dynamically. The QoS guarantee provided to real-time traffic flows consists of: the assurance of data delivery, a delay bound for the delivery of data, and a guaranteed communication bandwidth. Node communication without QoS needs uses non–real-time data flows. Non–real-time data flows have traffic characteristics assigned as well, but are not guaranteed a quality of service.

To provide QoS guarantees to real-time traffic flows, the traffic on the network must be managed by an instance that keeps track of all established flows. Before admitting a new data flow, that instance must validate that the QoS guarantees of the established traffic flows hold also with the addition of the new flow, and the potential QoS requirements of the new flow will be met. I will refer this traffic management instance as *bandwidth manager* in the following.

The bandwidth manager runs as an application program on one of the nodes at the network. All nodes that want to establish a new data flow so as to transmit data, connect to the bandwidth manager and ask it to admit the new connection with its particular parameters. Only if the bandwidth manager successfully verified all QoS guarantees, the node is allowed to communicate using the new data flow. Section 6.1 describes the bandwidth manager in detail.

4.5. Networks with multiple switches

Although extending the work to networks with multiple switches is not the scope of my dissertation, I will outline the main aspects here.

It is known that the burstiness of flows traversing multiple network elements (switches) increases with the number of traversed hops and the number of flows joining along path [Cru91b, BT01]. This results in high worst-case packet delays for a moderate number of switches. For a setup of 50 switches connected in a row, WATSON found in [WJ03] an upper delay bound of 400 ms on Fast Ethernet when a flow with 15 KByte burst size joins the network at every switch. A countermeasure to the burstiness increase is to reshape the streams at the network elements. Although there are Ethernet switches providing different forms of traffic shaping, it is not applicable in general. As such, the scalability of networks solely consisting of Ethernet switches is limited.

A practical solution for larger networks is to use medium-sized Ethernet networks and connect them by routers that reshape the streams. Reshaping at intermediate nodes has been researched in detail in the context of ATM and IntServ already. Each medium-sized sub-network would be managed by its own bandwidth manager then. RSVP could be used here to pass the reservation information between the bandwidth managers.

For managing medium-sized networks with multiple switches, the formulae of Section 4.2.4 need to be adapted. The burstiness increase of a traffic flow that is merged with other flows in a switch must be quantized. For simpler forms of traffic description the theory has been developed many years ago, for example by CRUZ [Cru91b] and WATSON [Wat02]. For the more complex T-SPEC traffic description results were published recently by FIDLER [FSK05]. However, the burstiness

increase is believed by the community to be too complex to be handled in practice. Therefore, in multiple-router networks the traffic flows are described by single leaky-buckets typically.

With multiple switches the network management must be extended by topology information: Knowing the exact topology, and hence the route of frames through the network, is crucial for guaranteeing safe and tight delay bounds.

If network segments with different bandwidths are connected by switches, the formulae of Section 4.2.4 become more complicated. The arrival and service curves do not use the same maximum bandwidth C any more, and the delay and burstiness bounds need to be reevaluated. Recent research analyzed such scenarios carefully [LMS05], although obtaining optimal results becomes quite complex.

5. Traffic shaping in the nodes

As described in Section 2.1, Ethernet switches have no notion of connections and do no traffic policing on their own. Hence, nodes connected to a switch must cooperate to ensure that traffic leaving a node conforms to previously defined T-SPECs. Therefore, all *sending nodes* apply traffic shaping to convert potentially highly bursty flows into flows satisfying a desired average data rate and maximum burstiness the switch can handle (Figure 9).

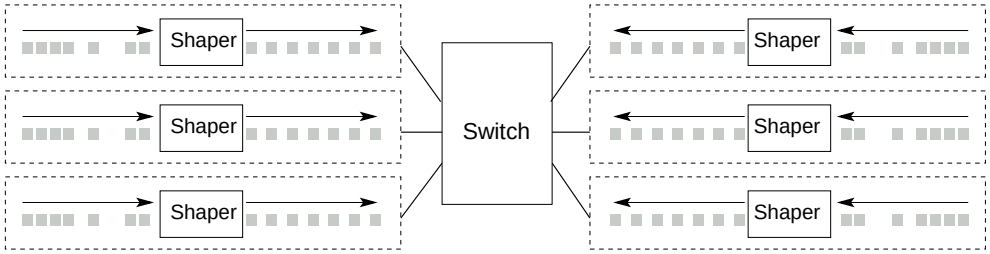

Figure 9: Shaping the traffic at the sending nodes.

Traffic shaping is an established technique in networks providing quality of service. It is a common feature in Internet routers to provide first-class services, mainly for participating in the emerging voice-over-IP market. Linux provides several traffic shaping algorithms as part of its network QoS framework [LxQ]. Surprisingly, a thorough analysis of deployed routers reveals that their performance-optimized implementations often fail to produce flows with specified burstiness bounds [FSK05, LH04a]. This might be caused by the principal trade-off between execution performance and achievable smoothness of the generated flows. In large systems, where first-class traffic only presents a small fraction of the overall traffic and where many flows are multiplexed together, rare and short traffic violations are likely to average out. Thus, they are tolerated to achieve a high overall performance. Furthermore, designing traffic shapers on systems with coarse-grained timers is a challenging task, and scheduling-related problems are easily overlooked [CKPW99].

This section details the steps necessary to shape traffic on a priority-based, real-time operating system so as to give hard and tight guarantees on the achievable delays at the network. This section argues about possible implementations and their integration in the scheduling model of DROPS. Special care is taken on the quantitative analysis of the generated flows, taking the granularity of timers and possible scheduling jitters into account. As a result, different traffic shapers are compared regarding their scheduling needs, the delay induced at the send node and the delay at the network. Section 6 derives the delay at the destination node.

5.1. Application network model

Applications in a real-time networked environment use the abstraction of a *leased line*, as this approach is called in the network community, to describe their communication needs: A leased line i is a channel with a guaranteed bandwidth r_i and a guaranteed maximum delay d_i, capable of

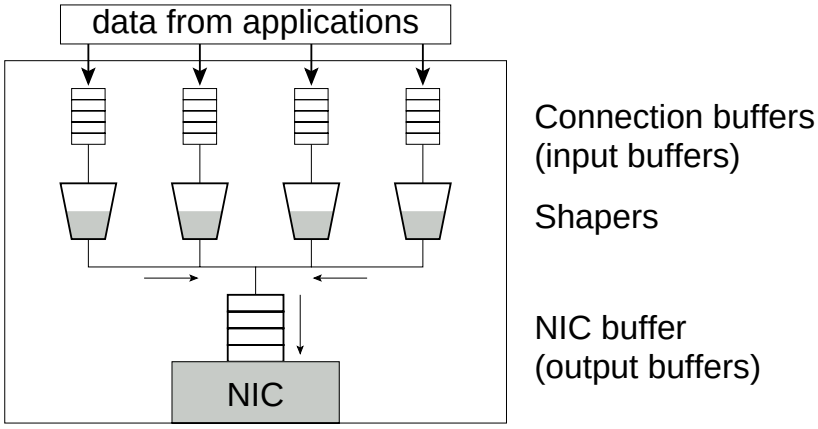

Figure 10: Multiplexing of multiple shaped connections to one NIC.

transporting messages up to a certain size M. The real-time environment must offer mechanisms for creating (r_i, d_i, M)-leased lines, and guarantee their properties until they are closed down. As a consequence, the real-time environment may throttle the data flows of an application: Packets exceeding the bandwidth of a leased line might be queued and being sent later instead of being sent immediately.

In the following, I will use the term *connection* to describe the instance of a leased line inside an operating system. A connection relates to the operating system internal management data, end-to-end reservation state and operating system internal resources such as threads handling the connection.

5.2. Traffic shaping implementation aspects

Figure 10 gives a general idea of traffic shaping at a node: Multiple applications with potentially different bandwidth requirements and traffic destination nodes put packets into connection buffers and signal the availability of the packets to connection-specific traffic shapers. After shaping the data of the individual connections, they multiplex the resulting flows to the network interface card. Depending on the scheduling model the shapers either periodically look into their connection buffers or wake up after the client submitted a packet. If packets are available a shaper takes some of them and moves them to the output buffer of the network card.

Definition: A packet of size $\leq M$ that is sent by the client of connection i with bandwidth b_i is called *conforming* if its temporal distance to the previous sent packet is at least M/b_i.

Applications in a real-time environment typically experience a scheduling jitter, denoted by J_i. If a periodic client produces a packet per period, it may produce a packet J_i time units late in one period, and may produce a packet on time in the next period. It may produce nonconforming packets accidentally. These nonconforming packets will experience an additional delay of up to J_i to fit to the leased line model. Thus, a client experiencing a certain scheduling jitter must expect

this scheduling jitter to be added to the delay guaranteed by the network environment. In the following, I will not consider this client-induced delay any longer, but assume that packets are conform.

5.2.1. Performance of shapers

There are multiple ways to implement traffic shapers that generate a leaky-bucket–shaped flow of rate r_i. However, the flows may differ in their *burstiness* b_i as a result of the particular shaper implementation. As outlined in Section 4.2.3f, the *burstiness parameter* influences the delays of the other flows at the NIC and at the switch, and hence it is an important performance measure. Another performance measure is the *maximum delay* of conforming packets at the shaper.

5.2.2. CPU utilization

Another important parameter of the shaper implementation is its CPU usage and scheduling requirements. As the shaper is run in its own operating-system context (thread), frequent changes between the applications and the shaper severely influence the CPU usage. As shown in [LH04a], the CPU utilization is dominated by the number of shaper invocations: A node sending with a bandwidth of 32 MBit/s used its CPU to 9% with a shaper invoked every 1 ms, but only to 2.9% when the shaper was called every 10 ms to send larger chunks.

The scheduling priority of a shaper thread derived from its maximum deadline is of importance, too. When the shaper thread is assigned a high priority to achieve low scheduling delays, other threads in the system may suffer high scheduling delays.

5.2.3. Shaper versions

In the following sections I will analyze five traffic shaper implementations:

Strictly periodic shaper This most basic form of a traffic shaper runs strictly periodically and allows one packet to pass in each period.

Periodic shaper with data dependency Similar to the strictly periodic shaper this shaper sends one packet per invocation. But, instead of a strict period it has a minimal inter-release time. Whenever a packet is ready in the connection buffer, and the minimal inter-release time has elapsed since the last invocation, the shaper is started and sends a packet.

Periodic shaper with long periods A straight-forward extension of the periodic shaper that sends multiple packets per period.

Token-bucket shaper This shaper runs strictly periodically while managing a bucket containing tokens for sending packets. The bucket is replenished on each invocation by some amount and the shaper sends up to as much data as there are tokens in the bucket.

Token-bucket shaper for best-effort connections This shaper is similar to the token-bucket shaper, but waits for packets to decrease the delay the shaper adds to conforming packets.

For each shaper, I will analyze (1) the maximum delay it adds to a conforming packet, and (2) the burstiness parameter b_i of the generated flow.

5.3. Strictly periodic shaper

The straight-forward shaper implementation is a strictly periodic thread that sends up to one packet per invocation. Listing 2 shows its pseudo-code. T_i denotes the period of the thread and D_i denotes its deadline ($D_i \leq T_i$).

```
strictly_periodic_shaper (Ti, Di, M) {
    set_periodic (Ti)
    while(1) {
        p = next_packet()    /* nonblocking, returns 0 if no packet available */
        if (p!=0) send_packet(p)
        next_period ();
    }
}
```

Listing 2: Strictly periodic shaper, one packet per period

To generate a flow with rate r_i and packet size M, T_i must be set to

$$T_i = M/r_i \tag{24}$$

Figure 11 illustrates the maximum delay that is induced by the traffic shaper to a conforming packet: The traffic shaper is activated early in the first period, and the packet is sent just after this. In the next period, the traffic shaper is activated as late as possible to just meet the deadline. Thus, the maximum delay is:

$$d_i = T_i + D_i \tag{25}$$

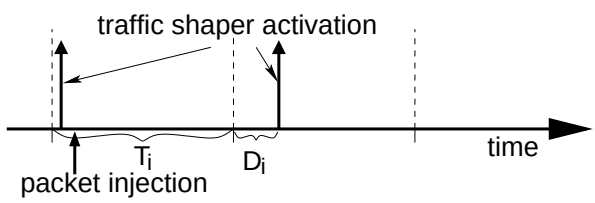

Figure 11: Maximum delay of a conforming packet at the strictly periodic traffic shaper.

The maximum burst of the generated flow corresponds to two consecutive packets sent in their minimum distance. The thread can send a packet up to D_i time units after the begin of one period. The following packet can be send at the beginning of the next period. The minimum distance of two packets is thus $T_i - D_i$, as illustrated in Figure 12.

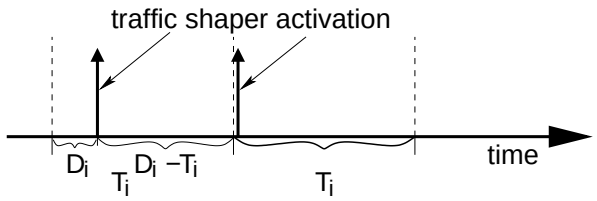

Figure 12: Obtaining the burstiness parameter of the flow generated by the strictly periodic shaper.

According to Equation 6 on page 20, the burstiness parameter of the generated flow can be calculated as

$$\begin{aligned} b_i &= 2 \cdot M - (T_i - D_i) \cdot r_i = 2 \cdot M - T_i \cdot ri + D_i \cdot r_i \\ b_i &= M + D_i \cdot r_i \end{aligned} \tag{26}$$

Thus, the generated flow conforms to a $(\mathbf{r_i}, \mathbf{M} + \mathbf{D_i} \cdot \mathbf{r_i})$ token-bucket shaper (leaky-bucket shaper).

5.4. Periodic shaper with data dependency

If an application is not executed in-phase with the shaper and thus cannot guarantee that a data packet is generated immediately before the shaper is activated, the worst-case delay of the strictly periodic shaper is more than one period. Modifying the shaper to wait until data is available avoids an out-of-phase client to miss the send operation of the current period (Figure 13). T_i now denotes the minimal inter-release time of the thread.

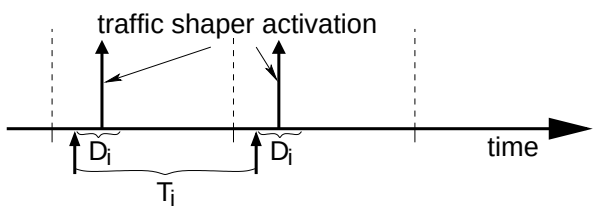

Figure 13: Maximum delay induced by the periodic shaper with data dependency to a conforming packet.

To implement the shaping with data dependency, Fiasco's minimal inter-release time scheduling as described at page 11 is used: The shaper waits for a data packet using a next-period IPC. Whenever a client application sends a packet, and thus answers the next-period IPC of the shaper, Fiasco releases the shaper thread as soon as possible, but not earlier than T_i time units after the previous packet transmission to the shaper.

As the traffic shaper thread is guaranteed to be finished within D_i time units after getting a conforming packet from its client, the maximum delay d_i that is induced by the traffic shaper is

$$d_i = D_i \tag{27}$$

The burstiness parameter of the generated flow is calculated the same way as for the strictly periodic shaper.

5.5. Strictly periodic shaper with long periods

To lower the CPU load caused by context switches, the period of the traffic-shaper thread can be increased at the cost of larger bursts and thus larger delays. This modification also favors systems were the thread period lengths are fixed or harmonic, such as that implementing the current QAS scheduling model of DROPS.

The naive approach is to modify the strictly periodic shaper to send not just one but up to a certain number of packets on each invocation. The input parameters to this shaper are the application-requested rate r_i, the desired period T_i and the relative deadline D_i. The number of packets that are allowed to be sent per period is thus

$$n_i = \lceil r_i \cdot T_i / M) \rceil \tag{28}$$

It is easy to see that this shaper results in a coarse granularity of possible bandwidth reservations: The bandwidth granularity is given by one packet that is sent every period:

$$g_i = M / T_i \tag{29}$$

For Fast Ethernet and a period of T_i=1 ms, the difference between sending n packets of 1514 bytes per 1ms period and sending $n+1$ packets per period accumulates to 12.1 MBit/s or 1/8 of the overall bandwidth. One way to decrease the coarse bandwidth granularity is to extend the period lengths. However, this increases the delay and burstiness bounds obversely. Given these properties, I do not consider this type of traffic shaper to be used in a dynamic, multi-application environment.

5.6. Token-bucket shaper

A token-bucket shaper avoids the coarse bandwidth granularity problem identified in the previous section.

The input parameters to the algorithm are the rate r_i, the packet size M, the bucket size B_i, the deadline D_i and the period T_i. The T_i selected for the token-bucket shaper is typically larger than it is for the shapers from Sections 5.3 and 5.4.

Obtaining the worst-case delay of a conforming packet is done the same way as in Section 5.3, and thus it is calculated as:

$$d_i = T_i + D_i \tag{30}$$

```
token_bucket_shaper(B_i, r_i, T_i, D_i, M) {
    set_periodic (T_i)
    level = B_i
    while(1) {
        p = next_packet()      /* nonblocking, returns 0 if no packet available */
        if ( level < M || p==0) do
            { next_period (); level = min(level + r_i*T_i, B_i); }
            while ( level <M)
        if (p)
            { send_packet(p) ; level -= M; }
    }
}
```

Listing 3: Token-bucket shaper

Figure 14 illustrates the burstiness bound of the generated flow: In the first period the maximum amount of data is sent as late as possible, thus B_i bytes are sent at offset D_i within the period. The following transmission occurs as early as possible in the period, thus at offset 0 $T_i \cdot r_i$ bytes are sent. Consequently, the burstiness parameter is calculated as

$$\begin{aligned} b_i &= B_i + T_i \cdot r_i - r_i \cdot (T_i - D_i) \\ b_i &= B_i + r_i \cdot D_i \end{aligned} \tag{31}$$

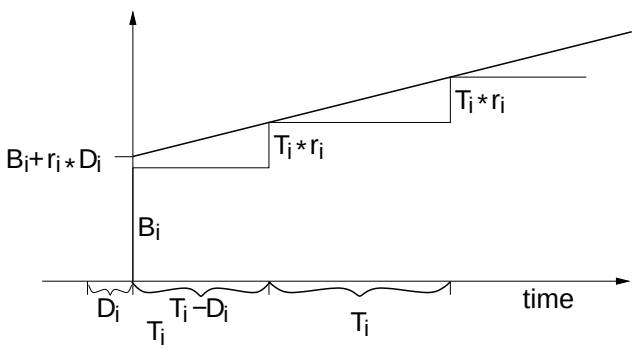

Figure 14: Obtaining the burstiness parameter of the flow generated by the token-bucket shaper.

Minimum bucket size for the Token-bucket shaper

Apparently, the token bucket must have a minimum size to work properly. If the bucket is too small, tokens will be thrown away in the replenishment process, although packets for sending are

available. Lost tokens correspond to lost bandwidth, but as long as the input buffer has enough packets, no tokens must be thrown away.

The bucket must have at least a size that all tokens that may arrive between the blocking of the shaper (due to missing tokens) and its next activation fit into it. When the shaper blocks, at most M tokens are in the bucket. At the next period the bucket is replenished by $r_i \cdot T_i$ and this must fit into it. Thus, the minimum bucket size is

$$B_i = r_i \cdot T_i + M \tag{32}$$

5.7. Token-bucket shaper for best-effort connections

The token-bucket shaper discussed in the previous section guarantees an upper bound for packet delays. This property is sufficient for real-time applications that are more concerned about the worst case than about the average case. Best-effort applications, however, prefer a good average case behavior: Considering synchronous protocols such as TFTP, a latency in the order of a millisecond results in the transfer of about 500 packets per second, limiting the effective bandwidth to 750 KByte/second independently of the medium capacity. Thus, best-effort applications have a need for low delays in the average.

A shaper combining the data dependency approach with a token-bucket attains both low delays in the average and a low CPU consumption for peak load. The shaper waits until client data is available, replenishes the bucket and transmits this data as long as the bucket contains enough tokens. If the bucket becomes empty, the shaper waits for a specific amount of time and then refreshes the bucket.

A shaper for best-effort connections should not rely on CPU reservations, and as such must be designed to be preemptible. This especially means that the shaper cannot make any assumption about the actual transmission start times of consecutively send packets, unless it takes timestamps between the send operations. As a consequence, the best-effort shaper needs to refresh its bucket after each send packet, in contrast to the real-time token-bucket shaper that can send short packet bursts and refresh the bucket at the end. The overhead of the best-effort shaper is moderate as long as the clock for bucket replenishment can efficiently be read.

Listing 4 gives the pseudo-code of the resulting best-effort shaper. The algorithm works periodically too, but in contrast to the real-time shaper no CPU reservation is done at the kernel. Instead, the period is used to update the bucket state.

The worst-case delay of a conforming packet is only influenced by the scheduling delay. Thus on an unloaded system, the delay is small. However, if higher-prioritized threads are ready, they will be executed by the scheduler, and consequently no upper delay bound can be guaranteed.

Figure 15 illustrates the burstiness bound of the generated flow: Due to scheduling issues, one packet may be delayed from a previous send operation to point ‘X’. At point ‘X’, the shaper is scheduled again and realizes that enough time has passed to refill the bucket completely. An amount of B_i can be sent immediately by the shaper. The following transmissions occur as early as possible. Thus, the burstiness parameter is calculated as

$$b_i = B_i + M \tag{33}$$

```
replenish () {
    time = now()
    level = min ( level + r_i * (time − replenish_time ), B_i)
    replenish_time = time
}
best_effort_shaper (B_i, r_i, T_i, M) {
    level = B_i
    replenish_time = now()
    wakeup = replenish_time + T_i
    while(1) {
        p = next_packet_blocking ()
        replenish ()
        while ( level <M) do
            { wait_until (wakeup); wakeup += T_i; replenish(); }
        send_packet(p); level −= M;
    }
}
```

Listing 4: Best-effort shaper

Minimum bucket size for the best-effort shaper

In contrast to the token-bucket shaper for real-time connections, the best-effort shaper is not guaranteed to be scheduled within finite time. As such, there is no minimum bucket size ensuring proper operation, instead there is a tradeoff between the bucket size and the probability of lost bandwidth. On an unloaded system, the bucket size can be set as with the token-bucket shaper: $B_i = r_i \cdot T_i + M$. As the system load increases, the scheduling jitter for the best-effort shaper becomes larger, and thus the probability for lost bandwidth. As a countermeasure the bucket can be enlarged. Unfortunately, a theoretical analysis of the minimum bucket size is near to impossible, as it heavily depends on the actual behavior of all threads in the system. In practice, however, this is hardly a problem, as indicated by the evaluation of the L^{4}Linux network-driver stub in Section 7.4.7.

5.8. Comparison

Table 1 compares the different traffic shapers with respect to scheduling needs—that is, period and delay—and their resulting burstiness and maximum delay added to conforming packets. An experimental analysis including the resulting CPU usage is given in Section 7 on page 84ff.

The numerical analysis of example setups using different traffic shapers gives an idea of the influence of the shapers to the application-to-application delay. Figure 16 shows the assumed network. All setups consists of five identical nodes sending traffic with 16 MBit/s to a sixth node over Fast

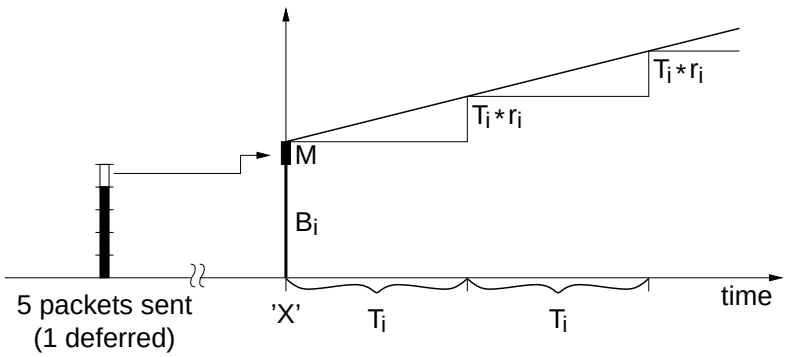

Figure 15: Obtaining the burstiness parameter of the flow generated by the best-effort shaper.

Shaper	Period T_i	Deadline D_i	TBF: bucket size B_i	burstiness parameter b_i	delay d_i
strictly periodic	M/r_i	$< T_i$	–	$M + D_i \cdot r_i$	$T_i + D_i$
strictly periodic with data dependency	$\geq M/r_i$	$< T_i$	–	$M + D_i \cdot r_i$	D_i
token-bucket	arbitrary	T_i or less	$r_i \cdot T_i + M$	$(T_i + D_i) \cdot r_i + M$	$T_i + D_i$
best-effort token-bucket	arbitrary	–	$\geq r_i \cdot T_i + M$	$\geq T_i \cdot r_i + 2M$	(small)

Table 1: Comparison of the traffic-shaper implementations.

Ethernet. t_{mux} is 45 µs as measured in Section 7 for the Fast Ethernet switch. C is 12325 bytes/ms considering the framing overhead with 1514 byte frames.

Table 2 displays the maximum time needed for a conforming packet to be processed by the specified traffic shaper, to be sent to the switch, and to be processed by the switch and forwarded to the destination node. The token-bucket shaper is evaluated with periods of T_i=1 ms and T_i=10 ms for comparison. The periods of the other shapers are calculated by Equation 24. To compare the shapers with respect their scheduling demand, they are evaluated with deadlines of D_i=200 µs and D_i=T_i.

The resulting numbers illustrate that low scheduling delays and high invocation frequencies result in moderate networking delays. However, if the scheduling delays increase, for instance due to other high-priority tasks in the system, or if the invocation frequency of the shapers is decreased to lower the CPU consumption, the application-to-application delay increases significantly.

Moreover, the traffic shaping on one node also influences the network delay of traffic originating from other nodes. The t_{switch} column of Table 2 shows the significant share of the switch delay on the overall delay. Even if a connection on one node would be shaped with the best possible shaper, its traffic would still suffer the switch delay induced by the other nodes.

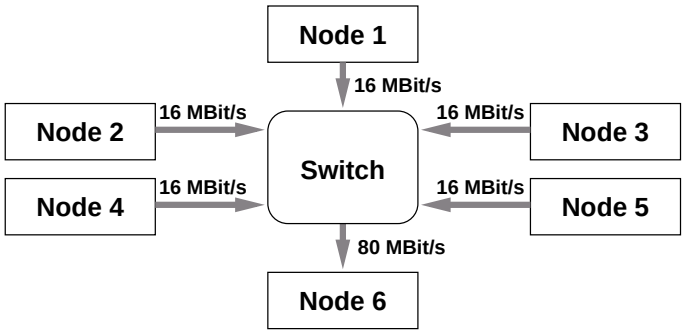

Figure 16: Setup for comparing the effect of the different traffic shapers.

Shaper	T_i	D_i	b_i	t_{switch}	$d_i + t_{frame} + t_{switch}$
Strictly periodic shaper	0.76 ms	0.2 ms	1914	0.81 ms	1.89 ms
		0.76 ms	3034	1.25 ms	2.89 ms
Periodic with data dependency	0.76 ms	0.2 ms	1914	0.81 ms	1.13 ms
		0.76 ms	3034	1.25 ms	2.12 ms
Token-bucket shaper	1.00 ms	0.2 ms	3914	1.59 ms	2.91 ms
		1.0 ms	5514	2.21 ms	4.33 ms
	10.00 ms	0.2 ms	21914	8.56 ms	18.88 ms
		10.0 ms	41514	16.16 ms	36.28 ms

Table 2: Comparison of the application-to-application delays on Fast Ethernet depending on the traffic shaper used. C=12325 bytes/ms, M=1514, t_{mux}=45 µs, t_{frame}=121 µs.

5.9. Summary

The DROPS scheduling models restrict the period lengths of real-time applications to certain values. The restriction depends on the actual scheduling model used, but it is discrete in all currently implemented models. This section presented different traffic shaper implementations suited for DROPS and carefully analyzed the properties of the generated flows. This section also discussed and compared the scheduling requirements and performance of the shaper implementations. As a general result, there is a trade-off between a low CPU usage by to the traffic shaping process on the one hand, and low delays at the Ethernet switch due to smoothly shaped traffic flows on the other hand.

The token-bucket shaper presented in Section 5.6 was constructed to be used for real-time connections. The periodic shaper with data dependency, presented in Section 5.4, achieves lower delays at the cost of increased CPU utilization. It will be used for demanding real-time connections that require a low delay. Best-effort connections will use the modified version presented in Section 5.7 to meet their need for low delays in the average case.

6. Implementation

This section describes the implementation of Switched-Ethernet–based real-time networking on DROPS. After deriving the network node architecture I detail the task and thread model, the real-time and non–real-time traffic classes, the client communication, best-effort send traffic, and the admission process.

Figure 17 shows the classic network stack architecture. A network device driver is responsible for the communication with the hardware device. Its API allows to exchange packets of the data link layer with the IP stack or another network layer protocol. The IP stack is used by applications, for instance by web servers for TCP/IP communication. In monolithic operating systems both the device driver and the network stack reside within the kernel. Only one network stack exists in the whole system.

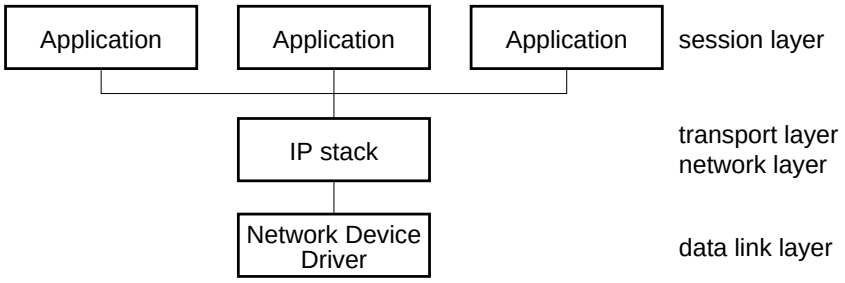

Figure 17: Classical network node architecture.

In contrast to this, multi-server operating systems such as DROPS split the network driver and the network stack into separate entities. L^{4}Linux as one of the DROPS applications uses its own IP stack but should be able to share the network device with real-time and best-effort DROPS applications that do not use Linux. As a consequence, there must be a support for multiple IP stacks in the system, all of them using the same network device.

Further, the real-time guarantees at the network require that all transmitted data conforms to specific traffic characteristics. To ensure these characteristics, a trustworthy instances at each node must shape the outgoing traffic of that node.

Figure 18 shows the resulting real-time–capable network-stack architecture.

The **RT-Net server** is an extended network driver that demultiplexes the network device to multiple network layer clients. The RT-Net server directly interacts with the network interface card (NIC). It is responsible for shaping the outgoing traffic accordingly and for policing incoming traffic to avoid overload situations. It offers connection-oriented packet-based interfaces to its clients. This allows accounting of transmit traffic and early demultiplexing of received traffic, for real-time traffic as well as for non–real-time traffic. In detail, the RT-Net Server performs the following tasks:

- Manage multiple clients and their traffic reservations

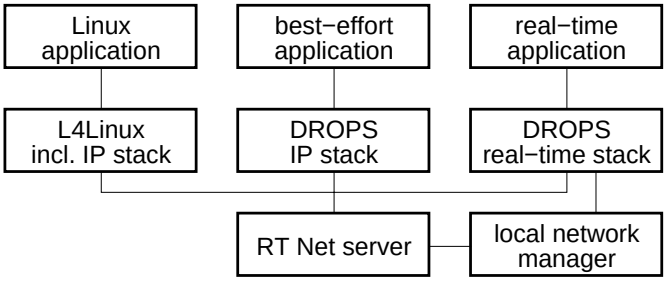

Figure 18: DROPS real-time capable network node architecture providing multiple clients of the device driver. The device driver is extended in its functionality and called a server in DROPS.

- Dispatch received packets to the correct client according to filter criteria
- Shape outgoing traffic according to a prior reservation

The **local network manager** is a proxy that handles reservation requests of the RT-Net server. The reservation for the switch must be coordinated among the nodes on the network. This coordination uses higher-level communication protocols and is therefore separated from the network driver.

The **real-time stack** is a minimal IP stack providing the UDP/IP transport layer protocol to real-time applications.

The remainder of this section is organized as follows: After a description of the local network manager, Section 6.2 on page 42 highlights important design issues for a real-time network stack for mikrokernel-based systems. Section 6.3 on page 44 describes the send process of the client and the RT-Net server. The section contains a quantitative CPU analysis of the send path needed for a CPU reservation. Section 6.4 on page 47 describes how the RT-Net server receives data and sent packet notifications from the NIC and communicates them to client applications. A quantitative analysis of CPU costs, memory requirements and delay guarantees accompanies the section. Section 6.5 on page 73 summarizes the quantitative analyses and Table 3 on page 71 recapitulates the relevant obtained results and their assigned symbols. Section 6.6 on page 74 describes the process of establishing a new real-time network connection and Section 6.7 on page 76 presents the resulting real-time client API. Section 6.8 describes the implementation of the non–real-time paths, that do not give any QoS guarantees, but nonetheless must not block the parallel running real-time paths.

6.1. Local network manager

The local network manager serves multiple purposes:

It acts as the *bandwidth manager* described in Section 4.4.2 on page 26 to manage the traffic flows on the network. Therefore, it keeps track of all established flows specifications and admits new flows. Each traffic flow is described by a T-SPEC, a (source-node, target-node) pair and a maximum acceptable network delay. As motivated in Section 6.5.2 on page 73, each traffic flow is also assigned a maximum burstiness at the switch output. Before admitting a new flow, the

bandwidth manager verifies that the delay and burstiness requirements of the established traffic flows and the new traffic flow can be met, and that the buffer needs do not exceed the capacity of the switch. Therefore, the bandwidth manager applies the delay and buffer calculations described in Section 4. If multiple flows originate from one node, the burstiness parameters in their T-SPECs are adapted as described in Section 4.2.3 on page 21 before the delay and buffer calculations. If multiple flows target to the same destination node, their burstiness parameters at the switch output used for comparing with the maximum tolerated burstiness are increased as described in Section 4.3.3 on page 25.

The bandwidth management is only activated on one node in the network. I refer to this node as the *master node*. At the master node, the local network manager also manages a network-wide *MAC address pool* for non–real-time network stacks. As described earlier in this section, multiple network stacks share the same network device. To demultiplex arriving network packets to a network stack, each network stack has its own MAC address assigned. For details on multiple network stacks sharing one NIC, see Section 6.8 on page 77. For details on demultiplexing, see Section 6.4.1 on page 49.

On all non–master nodes, the local network manager acts as a *proxy* to the master node. The proxy forwards bandwidth reservation requests and MAC address requests. For client applications, this proxy functionality is fully transparent.

Further, the local network manager implements a *part of the DROPS real-time stack* by managing the UDP ports used for real-time communication (Section 6.2). Therefore, the local network manager obtains its own IP address, either by using the DHCP protocol or by explicit configuration. This IP address is used for the proxy communication as well. As part of its real-time stack functionality, the local network manager also provides an ARP translation of IP addresses to MAC addresses. This ARP translation is used by the RT-Net server on real-time connection setup. For details on connection setup, see Section 6.6 on page 74.

A general problem of distributed reservation techniques is the bootstrap problem: For their first reservation, the non–master nodes need to communicate with the master node. However, this communication requires an appropriate reservation as well. The bandwidth manager solves this problem pragmatically by initially reserving a small amount of bandwidth for this first contact.

6.2. Design issues

Before describing the transmission and reception of data in detail, this section discusses main design issues for implementing a reservation-based, high-performance real-time network stack on a mikrokernel-based system.

UDP as real-time communication protocol Real-time traffic requires bounded transfer times. In Section 4.2 I described how this can be achieved at the hardware level using Switched Ethernet technology. The result was reliable packet transmission with bounded transfer times. The operating system has to perform two tasks: Ensuring that a node does not send more data than its quantum, and ensuring timeliness in the execution of the applications and of the transfer of data to the NIC. The higher-level network protocol of real-time applications has to be efficient

and predictable in its execution time. Retransmission because of memory shortage in one of the network elements or a dynamic bandwidth adaption is not needed in the protocol to guarantee reliable packet transmission. As such, IP-based UDP is an appropriate communication protocol upon which real-time applications can build their own application protocol.

Fixed addresses on connection establishment For a proper reservation of switch resources as described in Section 4, the target node of a connection must be known. For IP-based communication, the target node is determined by the target IP address. Thus, IP address binding of a real-time connection must be done at connection establishment. To uniquely identify a connection at a node, the real-time stack uses its local UDP port. Hence, for reservation purposes at the network stack, the *local* UDP port is bound at connection establishment as well. As with the *BSD socket* interface, the UDP *remote* port is not specified at connection establishment.

UDP protocol handling at the network driver Encapsulating application data into UDP packets is a simple and fast operation. It merely requires adding a header and performing a checksum. Encapsulation is a fast operation too, as the header information is mostly static for packets of one connection. Extracting the payload of a UDP packet also is fast: verifying the checksum and stripping the header. Therefore, both encapsulating and extracting of application data can easily be done within the network driver. No additional IP stack server is needed for client-network communication. As the consequence, real-time applications communicate directly with the RT-Net server. For data transmission and reception, they exchange application payload data with the server, not containing any UDP, IP or MAC headers. As with the *BSD socket* interface, the UDP remote *port* is optionally passed as an additional argument.

Multi-threaded traffic processing For each established real-time connection, the RT-Net server assures a timely delivery of data within the traffic specifications of the connection. This especially holds if misbehaving clients try to overuse their respective network connection or try to hamper the server in other ways. To isolate the network connections with respect to their CPU usage, they are processed in separate threads at the RT-Net server. By reserving appropriate CPU quanta for each thread at the kernel, CPU-related interferences between different connections are avoided as far as possible. Further, the multi-threaded approach permits threads to block independently of each other, for instance to wait for client-requests or to apply traffic shaping.

Thread-safe memory management Due to the multi-threaded processing, care must be taken on the memory management. The parallel threads allocate and deallocate memory for network packets, and hence the memory management must be thread-safe. To avoid mutual blocking, the threads use specific memory pools with nonblocking, thread-safe allocator and deallocator functions, similar to those presented in [Ber93].

Copy avoidance It is widely accepted and shown by multiple publications on inter-address-space communication interfaces that copy-avoidance is essential for a reasonable throughput between components [DP93, PDZ00, MKT98]. Using shared memory for communication between the components is an established technique here. As shown by Pai and Druschel in [PDZ00], mapping and un-mapping of memory pages are costly operations and zero-copy implementations using dynamic mappings suffer from performance penalties. On x86-based L4 implementations, dynamic mapping and un-mapping outperforms copying of data only for block-sizes larger than

2–4 KByte, depending on the actual hardware used. In either case, dynamic mapping is no adequate solution for transferring single Ethernet packets with a maximum size of 1.5 KByte between address spaces.

Event coalescing The costs for transmitting information between address spaces are typically dominated by the address-space switches, the necessary crossing of kernel/user-level boundaries and the resulting flushing of the various memory caches.[4] By coalescing multiple events, the overall performance can be increased, although sometimes at the expense of increased delays in the processing of individual data.

Figure 19 shows the resulting thread structure of the RT-Net server and its communication relations to relevant real-time components.

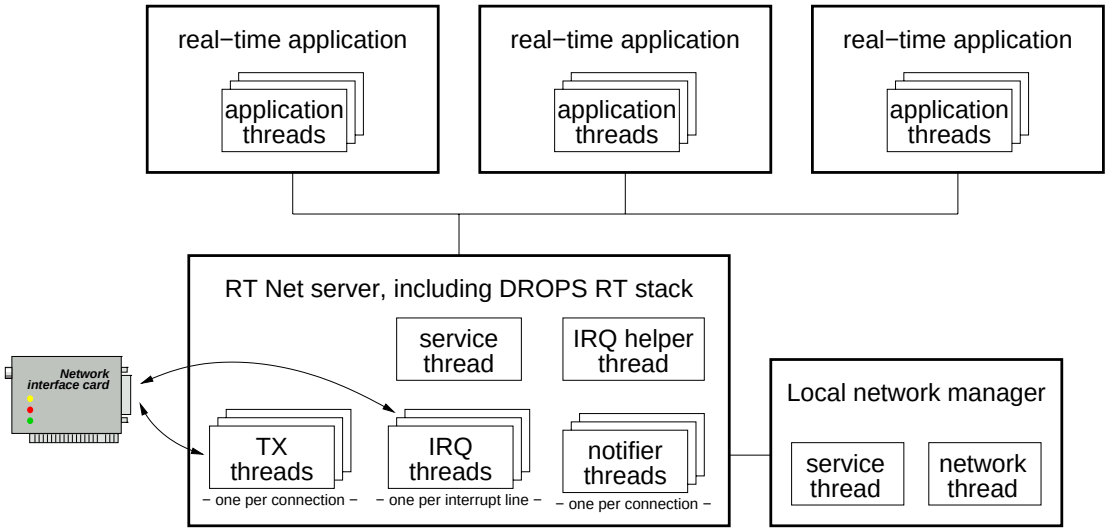

Figure 19: Thread structure of the network driver and real-time applications. The IRQ threads are device (IRQ-line) specific. The *TX-threads* perform the per-connection traffic shaping. The per-connection notifier threads signal their particular clients any received packets and successfully sent packets.

6.3. Real-time send path

The send path performs the following tasks:

- Reservation of send traffic and CPU resources
- Shaping of the send traffic according to the reservation
- Enqueueing of transmit packets at the NIC
- Isolating connections with respect to their CPU usage

[4] L1-, L2-, trace-cache, TLB

The reservation of send traffic uses the local network manager described in Section 6.1 on connection setup. In this section, I describe the data transmission on an established connection, and come back to the traffic reservation when detailing the connection setup in Section 6.6.

To isolate connections with respect to their CPU usage, traffic shaping and management is performed in separate threads at the RT-Net server. The send path involves an application thread within the client application and a per-connection *TX thread* within the RT-Net server. Each *TX thread* shapes the traffic of its connection and enqueues the corresponding network packets at the NIC.

For transporting the data from the application's address space to the address space of the RT-Net server, the application and the server use a connection-specific shared memory region, called the data area. The data area is managed by a connection-specific ring-buffer shared between the two instances as well (Figure 20). As the RT-Net server can transmit packets directly out of the data area, zero-copying is provided for the send path.

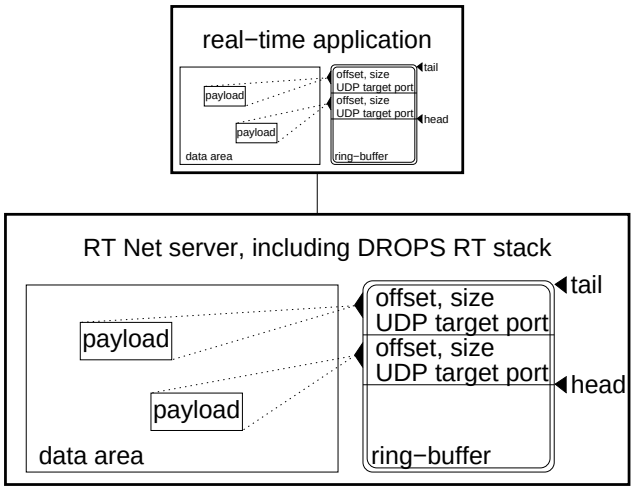

Figure 20: Data sharing between the client and the RT-Net server. Each send connection has a shared data area and a shared ring-buffer attached. The ring-buffer addresses the payload chunks in the data area.

The following sections describe the client interface and the *TX thread* in detail. Section 6.3.3 analyzes the *TX thread* quantitatively to obtain bounds on its resource usage.

6.3.1. Client interface abstraction

The *RT-Net library* provides multiple functions for the communication between the client and the RT-Net server. To send a data, the client calls the *rt_txdesc(int size)* function.[5] This function

i) allocates an entry in the shared ring-buffer

[5]The actual library names are different and prefixed appropriately. To focus on their functionality, I use abbreviated names throughout this dissertation.

ii) allocates a piece of memory of the appropriate size in the data area
iii) writes a descriptor to the piece of memory into the ring-buffer entry
iv) returns a pointer to the descriptor

The client generates its data into the piece of memory and calls the *rt_send(desc_t d, int16 port)* function. *rt_send()* writes the destination UDP port number in the descriptor and marks the descriptor as ready for transmission. Optionally, it notifies the *TX thread* at the RT-Net server with an IPC.

rt_txdesc() fails if the client produces its data too fast and all entries of the shared ring-buffer are occupied. If this happens, the client must wait until some data has been sent by the RT-Net server. Therefore, the client can use the *rt_wait()* function, which is described in detail in Section 6.4.3 on page 57. An alternative is to wait for some time, which is appropriate for specific time-driven client implementations.

If the clients needs to use a *write()*-like interface, which allows to specify an arbitrary address for the data to be sent, it can use the *rt_write()* function. This function combines *rt_txdesc()* and *rt_send()* at the expense of an additional data copy.

6.3.2. Connection-specific TX thread

The actual data transmission is done at the *TX thread* of the RT-Net server. It examines the ring-buffer, encapsulates data by prepending the necessary UDP, IP, and MAC headers, performs the traffic shaping and enqueues the network packets at the NIC. The traffic shaping uses one of the two algorithms for real-time connections presented in Section 5.9. Which one, will be determined as part of the connection setup, described in Section 6.6.

Token-bucket shaper When the connection is shaped by a token-bucket shaper, the *TX thread* at the RT-Net server periodically checks its ring-buffer for descriptors provided by the client. If one is found, the data is encapsulated and the token-bucket traffic-shaping is applied: If the connection's traffic pattern would exceed the negotiated traffic characteristics, the *TX thread* waits for the next period. Immediately after this, the packet is enqueued at the NIC. Note that the client application can throttle its send traffic by examining the ring-buffer and adapting to the shaping of the *TX thread*.

Periodic shaper with data dependency When the connection is shaped with the periodic shaper with data dependency, the *TX thread* waits for a wakeup notification from the client. The waiting is implemented by a next-period receive IPC (Section 2.2.3). The according send IPC is submitted by the client as part of its *rt_send()* function. If this IPC is sent too early, the mikrokernel delays its delivery according to the scheduling parameters of the *TX thread*. Once the IPC is seen by the *TX thread*, it immediately encapsulates the payload data found using the ring-buffer and enqueues it at the NIC.

To enqueue data at the NIC, the *TX thread* calls native Linux driver code that is included in the RT-Net server. This driver code accesses the NICs hardware registers and must be protected against concurrent accesses from other threads. Typically, mutual exclusion is achieved by a user-level semaphore or mutex implementation using L4 IPC between threads. However, as the L4

IPC mechanism is specified and implemented today, it sacrifices the running real-time reservation of a thread: Under specific priority constellations, IPC *send*- and *call*-operations hand the active timeslice of a thread to the receiver of that IPC. Finally, it is not guaranteed that the receiver has a chance to hand the timeslice back to the original sender. Therefore, the RT-Net server uses another approach to avoid concurrent access to the NIC – it disables the processor's hardware interrupts during calls to Linux driver code. To ensure a correct response time analysis despite the interrupt locking, the worst-case execution time of the locked code section is reserved as a uninterruptible time interval at the user-level CPU admission controller when the RT-Net server is started.

6.3.3. CPU usage of the TX thread

The worst-case per-period CPU usage of the *TX thread* mainly depends on the number of packets that can be sent in a period. For the token-bucket shaper, this number is bounded by the bucket size B_i divided by the packet size M_i. To calculate the actual CPU reservation, a parameter set *cpu* is used, which describes the amount of CPU needed to various network-related operations. This parameter set is obtained by measurements. In detail, the components of *cpu* relevant for transmitting a number of packets of a specific size to a given network interface are $cpu.tx_base$ and $cpu.tx_packet$. $cpu.tx_base$ quantifies the base amount of CPU needed to schedule the *TX thread* and to start it processing the packets. $cpu.tx_packet$ quantifies the per packet costs. As the *TX thread* does not access the data, the CPU costs are independent of the size of the packets. Equation 34 quantifies the amount of CPU for the token-bucket shaper.

$$CPU_{i,tx}^{tbf} = cpu.tx_base + cpu.tx_packet \cdot \left\lceil \frac{B_i}{M_i} \right\rceil \tag{34}$$

The strictly periodic shaper with data dependency sends no more than one packet per period. As such, the equation simplifies to:

$$CPU_{i,tx}^{sp} = cpu.tx_base + cpu.tx_packet \tag{35}$$

6.4. Real-time notification path

The notification path performs the following tasks:

1. Demultiplexing of sent packet-events from the NIC to the according connection
2. Notification of successfully sent packets to the client applications
3. Demultiplexing of received packets from the NIC to the according connection
4. Transmission of received payload to the client applications

To isolate connections with respect to their CPU usage, the notification path uses multiple threads as well. At the RT-Net server the execution of the notification path is distributed to the following threads:

- A NIC-specific *IRQ thread*
- A connection-specific *notifier thread*

The demultiplexing (1 and 3) is done within the NIC-specific *IRQ thread.* The other tasks (2 and 4) use the connection-specific notifier thread that directly communicates with the corresponding client applications. Figure 21 shows the architecture of the notification path and the communication structures between the involved threads.

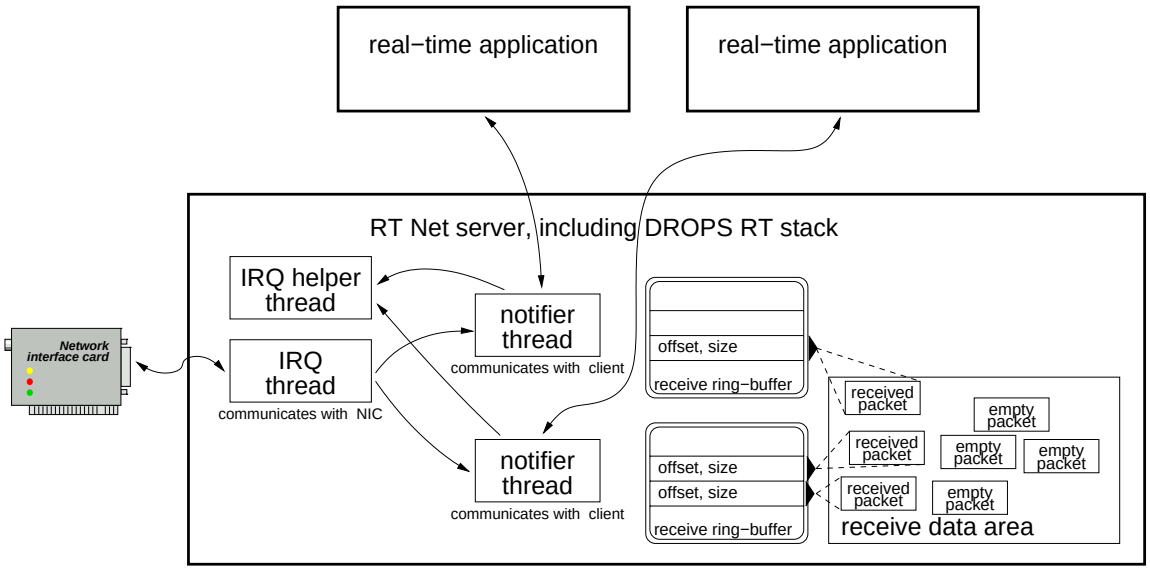

Figure 21: Architecture of the notification path at the RT-Net server. The receive ring buffers hold the packets received from the *IRQ thread* until their data is transmitted to the clients by the notifier threads.

In contrast to the send path, the receive path cannot be implemented without copying the network data at least once. As described at the begin of Section 6, each NIC in the system is connected to multiple network stacks, and consequently receives data for multiple stacks. As todays NICs do not provide a separation of received data according to higher-level filtering criteria (IP addresses or UDP/TCP ports), this filtering must be done in software. At the time the host CPU is able to inspect the data, it has already been copied to the hosts main memory by the NIC. As the client network stacks use different address spaces, the data must be copied to them.

With the RT-Net server architecture, the NIC places received packets into a data area that is shared by all connections. The *IRQ thread* de-multiplexes the packets to the connections and puts references to the packets into connection-specific receive rings. The notifier thread finally transmits the data to the clients and frees the data memory for later reuse.

The following sections describe the *IRQ thread* and the notifier threads in detail. They analyze the work of the threads quantitatively to obtain bounds on their resource usage and delay guarantees.

6.4.1. Device-specific IRQ thread

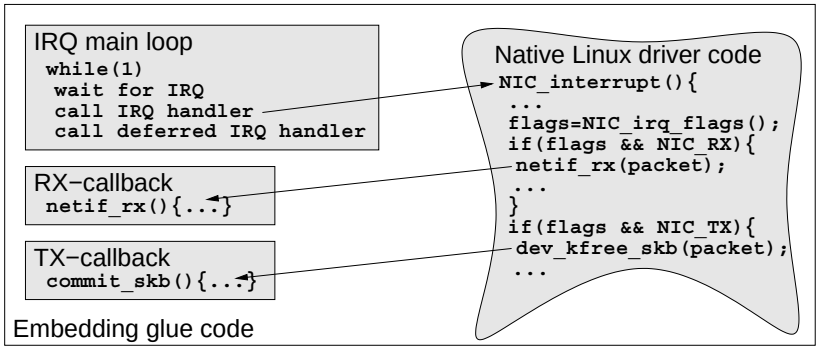

Figure 22: Interaction between the RT-Net server code and the native network Linux driver code within an *IRQ thread.*

In general, interrupt requests (IRQs) are issued by a network device if either a network packet was received or a network packet scheduled for transmission was successfully sent. Some of the recent network devices coalesce interrupts, so that an interrupt is raised only if a specific threshold of sent or received packets is reached or a certain timeout is over.

The *IRQ thread* handles interrupt requests of its associated network device. Each interrupt request is passed to native Linux driver code that controls the NIC. This native code determines the causing event and performs the necessary device-specific operations. Using callbacks to the embedding glue code, the appropriate RT-Net server code is activated to dispatch the event to the corresponding connection (Figure 22). After returning from the native Linux driver code, the *IRQ thread* calls a deferred interrupt handler. Its purpose is to signal the notifier thread, described in detail in Section 6.4.3 on page 57.

As with the *TX thread*, the access to the NIC needs to be synchronized between the threads. The interrupt locking mechanism is used here as well.

Packets received from the network

In the case of a receive-event the Linux driver code calls a function named `netif_rx()` with a pointer to the received packet as argument. The `netif_rx()` function, still executed in the *IRQ thread* of the device, tries to demultiplex the received packet to one of the real-time or best-effort connections. Listing 5 shows its pseudo-code.

For finding the corresponding real-time connection of a packet, the MAC address of the received packet is verified, the level 3 protocol type in the MAC header is checked against the IP protocol, the level 4 protocol type in the IP header is checked against the UDP protocol and then the

```
void netif_rx (struct sk_buff *skb){
  ...
  if (memcmp(skb->data, nic->mac, ETH_ALEN)!=0 &&      /* check for MAC address */
      memcmp(skb->data, MAC_BROADCAST, ETH_ALEN)!=0) return;
  if (skb->protocol != htons(ETH_P_IP)) return;        /* check for IP protocol */
  if (skb->nh.iph->protocol != IPPROTO_UDP) return;
/* check for UDP protocol */

  for (conn=rx_conns.first(); conn; conn=conn->next()){   /* lookup connection */
    if (skb->nh.iph->daddr != conn->local_ip ||           /* check for IP address */
        skb->h.uh->dest !=conn ->local_udp_port){         /* check for UDP address */
          continue;
    }
    conn->receive(skb);
}
```

Listing 5: Receive path demultiplexing pseudo code

connection is looked up using the UDP port number. When a connection is found, the packet is enqueued in the driver-internal connection-specific RX ring by `conn->receive()`, depicted in Listing 6.

```
void Conn::receive (struct sk_buff *skb){
    if (rx_ring_len == rx_ring_size){        /* RX ring full, drop the packet */
        dropped++; return;
    }
    atomic_inc(&skb->users);                 /* increase reference-count of skb */
    ... put skb into the rx-ring ...
    flush_rx_wake_queued(this);              /* don't forget to wakeup the notifier thread */
}
```

Listing 6: Connection-specific receive function that enqueues a packet into the receive ring

`Conn::receive()` tries to put the received packet into the connection-specific receive ring. If this succeeds, the connection calls `flush_rx_wake_queued()` to add itself to a list of connections whose notifier thread needs to be signalled. `flush_rx_wake_queued()` takes care no to add a connection twice, and it does not add it at all if no client is waiting for received packets on that connection.

Packets sent to the network

In case of a sent-event, the Linux driver code deletes the structure holding the network packet, called an *sk_buff*. Using a delete-callback of that structure, the RT-Net server is notified of the sent packet. The callback function obtains the connection of the packet from the *sk_buff*, and adds it to the list of connections to be signalled, similarly to the receive path.

Signalling the notifier thread

When the native Linux driver code returns from the IRQ handler, a deferred interrupt handler traverses the list of connections to be signalled. For each connection in the list, it calls `flush_rxtx_wake()`, whose pseudo-code is given in Listing 7. At the end, the connection list is cleared.

```
1 void Conn::flush_rxtx_wake(void){
2     if( flush_waiting && /* avoid other tests if notifier is not waiting */
3         (!timeout || thresholds_over() ) &&
4          test_and_clear_bit (0, &flush_waiting)){
5           l4_ipc_send( flush_thread_id , TIMEOUT_NONE);
6 }   }
```

Listing 7: Waking the notifier threads

Once per IRQ `flush_rxtx_wake()` wakes the connection-specific notifier thread, if the latter is waiting for notification (`flush_waiting` tests at lines 2 and 4). Often, multiple events are processed in one IRQ handler call, especially if the CPU scheduler defers the IRQ handler execution. The time needed for managing the list of connections to be notified is significantly less expensive than a wakeup that includes IPC operations. As such, the reduction to one wakeup per IRQ reduces the execution time compared to one wakeup per received and sent packet. Setting `flush_waiting` is triggered by the notifier thread of a connection, described in Section 6.4.3.

The condition in line 3 of Listing 7 takes care of a potential event coalescing requested by best-effort clients (described in Section 6.8.2 on page 80). If the client requested a coalescing, the wakeup is only done if the thresholds of received or sent packets have been crossed.

As the final condition, the `flush_waiting` flag is checked again and atomically reset. The test is necessary, as the notifier thread might reset the flag in between, for instance due to an expired coalescing timeout. In that case, the notifier thread takes care of the necessary tests before waiting for an update again.

Only if the `flush_waiting` flag was set, an IPC to the notifier thread is sent that wakes it up. The timeout of the IPC is 0, thus the *IRQ thread* cannot block in the case that the notifier thread just fall out of its IPC but did not reset the flag yet.

CPU usage of the IRQ thread

In contrast to the *TX thread*, a worst-case CPU bound cannot that easily be derived for the *IRQ thread*. For scheduling reasons, the *IRQ thread* has a period assigned, denoted by T_{IRQ}, and a relative scheduling deadline, denoted by D_{IRQ}. As with the other real-time threads, the kernel refreshes the time quantum of the *IRQ thread* once every T_{IRQ}. To predict the amount of CPU time needed by the *IRQ thread* within T_{IRQ}, the number of interrupts within T_{IRQ} must be known in advance. With no further assumptions to the NIC, one interrupt per received and sent packet must be assumed.

The maximum number of packets sent within T_{IRQ} results from the transmit reservations. Due to the scheduling jitter of the *TX threads* and the a priori unknown ordering of the send packets in the NIC's output FIFO, the sent IRQs are not generated in equidistant time intervals but in bursts. The burstiness of the data flows, and thus the burstiness of the interrupts, has been derived in Section 5 and in Equation 7 at page 22. Equation 36 gives the bound for the number of transmitted *bytes* as they are seen by the *IRQ thread* within T_{IRQ}. The $\min(C \cdot (T_{IRQ} + D_{IRQ}) + M, \ldots)$ reflects the bandwidth limitation by the network medium. Due to scheduling jitter, the IRQ handler may be deferred by D_{IRQ}, and thus the relevant interval is $T_{IRQ} + D_{IRQ}$. b_i^{tx} is the burstiness of the data flow i after it is multiplexed to the NIC's FIFO, as described by Equation 7 at page 22. $conns_{tx}$ denotes the number of transmit connections.

$$tx_{IRQ} = \min\left[C \cdot (T_{IRQ} + D_{IRQ}) + M, \sum_{i=0}^{conns_{tx}} r_i^{tx} \cdot (T_{IRQ} + D_{IRQ}) + b_i^{tx}\right] \tag{36}$$

Assuming that all transmitted packets of connection i have the maximum packet size M_i, the maximum number of sent notifications the *IRQ thread* sees in each period is the maximum of all combinations of sent packets that fit into their connection-specific bound and in the $\min[C \cdot \ldots]$ bound. The maximum can be calculated as following: Let the connections be ordered, so that $M_i \leq M_j$ for $i < j$. Let further $t = T_{IRQ} + D_{IRQ}$. Connection 0, having the smallest packets, can cause sent notifications for up to $\min[C \cdot t + M, r_0^{tx} \cdot t + b_0^{tx}]$ bytes within t, thus the *IRQ thread* sees no more than

$$n_0 = \left\lceil \frac{\min[C \cdot t + M, r_0^{tx} \cdot t + b_0^{tx}]}{M_0} \right\rceil$$

notifications for connection 0 per period. If connection 0 actually causes that many notifications, connection 1 can cause notifications for up to $\min[(C \cdot t + M - n_0 \cdot M_0)^+, r_1^{tx} \cdot t + b_1^{tx}]$ bytes, corresponding to n_1 packets:

$$n_1 = \left\lceil \frac{\min[(C \cdot t + M - n_0 \cdot M_0)^+, r_1^{tx} \cdot t + b_1^{tx}]}{M_1} \right\rceil$$

Generalized, the number of sent notifications the *IRQ thread* sees in each period is bound by $\overline{tx}_{IRQ}$, given by Equation 37.

$$\overline{tx}_{IRQ} = \sum_{i=0}^{conns_{tx}} n_i^{tx} \qquad \text{with } n_i^{tx} \text{ given by the following equation} \tag{37}$$

$$n_i^{tx} = \left\lceil \frac{\min[(C \cdot t + M - \sum_{j=0}^{j<i} n_j^{tx} \cdot M_j)^+, r_i^{tx} \cdot t + b_i^{tx}]}{M_i} \right\rceil \quad \text{and} \quad t = T_{IRQ} + D_{IRQ}$$

If the $C \cdot T_{IRQ} + D_{IRQ}$ term in Equation 36 is neglected, the arrival curve is still valid, though not tight. The computation however simplifies to

$$\overline{tx}_{IRQ} = \sum_{i=0}^{conns_{tx}} \left\lfloor \frac{r_i^{tx} \cdot (T_{IRQ} + D_{IRQ}) + b_i^{tx}}{M_i} \right\rfloor + 1 \tag{38}$$

If the min in Equation 36 is determined by the $\sum_{i=0}^{conns_{tx}} r_i^{tx} \cdot (T_{IRQ} + D_{IRQ}) + b_i^{tx}$ term, the bound in Equation 38 is tight.

Note that in either case a useful upper bound for the number of transmitted packets cannot be given, if the transmitted packets are arbitrarily small.

For receive interrupts, the situation is similar: The jitter introduced at the switch allows no reliable prediction of the arrival times of packets at the NIC. It cannot be assumed, that the packets arrive in equidistant time intervals, but instead they arrive in bursts. Let α_{NIC} denote the arrival curve of the aggregate A of the flows arriving at the NIC. Section 4.3 at page 24f derived α_{NIC} already – a T-SPEC with parameters $(C, M, \sum_{i \in A} r_i^{rx}, \sum_{i \in A} \tilde{b}_i^{rx})$. r_i^{rx} denotes the average bandwidth of connection i, and $\tilde{b}_i^{rx}$ denotes its burstiness parameter. It was derived by Equation 21 at page 25.

The *IRQ thread* puts a received packet into the receive ring of the according connection no later than D_{IRQ}. This means, it adds a delay of up to D_{IRQ} to received packets. Let α_{IRQ} denote the arrival curve of data as it is seen by the *IRQ thread*: It is the result of a flow with arrival curve α_{NIC} that is delayed by up to D_{IRQ}. According to [BT01], α_{IRQ} is thus given by Equation 39.

$$\alpha_{IRQ}(t) = \alpha_{NIC}(t + D_{IRQ}) \tag{39}$$

The resulting bound for the number of received *bytes* that the *IRQ thread* gets from the NIC within a time interval of length T_{IRQ} is given by Equation 40.

$$rx_{IRQ} = \min \left[C \cdot (T_{IRQ} + D_{IRQ}) + M, \sum_{i=0}^{conns_{rx}} r_i^{rx} \cdot (T_{IRQ} + D_{IRQ}) + \tilde{b}_i^{rx} \right] \tag{40}$$

Analogously to sent notifications, the number of receive notifications the *IRQ thread* sees in each period is bound by $\overline{rx}_{IRQ}$, given by Equation 41.

$$\overline{rx}_{IRQ} = \sum_{i=0}^{conns_{rx}} n_i^{rx} \quad \text{with } n_i^{rx} \text{ given by the following equation} \tag{41}$$

$$n_i^{rx} = \left\lceil \frac{\min[(C \cdot t + M - \sum_{j=0}^{j<i} n_j^{rx} \cdot M_j)^+, r_i^{rx} \cdot t + \tilde{b}_i^{rx}]}{M_i} \right\rceil \quad \text{and} \quad t = T_{IRQ} + D_{IRQ}$$

As with the transmit notifications, a simpler bound for the number of received packets by the IRQ can be derived (42). If the min in Equation 40 is determined by the $\sum_{i=0}^{conns_{rx}} r_i^{rx} \cdot (T_{IRQ} + D_{IRQ}) + \tilde{b}_i^{rx}$, the bound is tight.

$$\overline{rx}_{IRQ} = \sum_{i=0}^{conns_{rx}} \left\lfloor \frac{\tilde{b}_i^{rx} + r_i^{rx} \cdot (T_{IRQ} + D_{IRQ})}{M_i^{rx}} \right\rfloor \tag{42}$$

To calculate the CPU usage bound, *cpu* contains components describing the amount of CPU needed to handle a number of packets by the *IRQ thread*. In detail, *cpu.irq_base* quantifies the base CPU costs needed to schedule the *IRQ thread* and to start it processing the packets. *cpu.irq_tx* quantifies the costs to handle one sent-notification in the *IRQ thread*. *cpu.irq_rx* quantifies the costs to demultiplex one received packet in the *IRQ thread*. Equation 43 gives the CPU usage bound of the *IRQ thread* per period of length T_{IRQ}, denoted by CPU_{IRQ}.

$$\begin{aligned} CPU_{IRQ} &= (cpu.irq_base + cpu.irq_tx) \cdot \overline{tx}_{IRQ} + \\ &\quad (cpu.irq_base + cpu.irq_rx) \cdot \overline{rx}_{IRQ} \end{aligned} \tag{43}$$

However, this CPU bound has two weaknesses:

(1) It is pessimistic, as it assumes the worst-base burstiness of both received and sent traffic. In practice, these worst cases are extremely unlikely to happen. As shown in the experimental evaluation section, the IRQ related base CPU costs are significantly higher than the costs related to packet handling. If the number of IRQs could be reduced by IRQ coalescing, the CPU reservation could be reduced significantly. Section 6.4.2 describes the according approach and its costs.

(2) It is optimistic in the sense that all packets send on a connection have the maximum size. Especially for best-effort connections this does not always hold. Relaxing this assumption, however, results in an practically unbound number of sent and received packets. The only solution to this problem is an engineering approach: Reducing the number of IRQs by IRQ coalescing and obtaining CPU usage bounds by measurements.

Quantitative analysis of the IRQ thread

For a later calculation of the CPU usage and delay at the notifier threads, this section derives the amount of data that the *IRQ thread* receives for a specific connection. Let $\alpha_{i,NIC}$ denote the arrival curve of traffic received for connection i at the NIC. Section 4.3 at page 24f derived $\alpha_{i,NIC}$ already – a T-SPEC with parameters $(C, M, r_i^{rx}, \tilde{b}_i^{rx})$. Let further $\alpha_{i,rxring}^{dual}$ denote the arrival curve of the flow of data that is put into the receive ring of connection i by the *IRQ thread*: It is the result of a flow with arrival curve $\alpha_{i,NIC}$ that is delayed by up to D_{IRQ}. Equation 44 gives the result.

$$\alpha_{i,rxring}^{dual}(t) = \alpha_{i,NIC}(t + D_{IRQ}) \tag{44}$$

$$= \begin{cases} \min\left[M + C \cdot (t + D_{IRQ}), \tilde{b}_i^{rx} + r_i^{rx} \cdot (t + D_{IRQ})\right] & \text{for } t > 0 \\ 0 & \text{for } t = 0 \end{cases} \tag{45}$$

In practice, Equation 45 can be simplified by using a single leaky-bucket arrival curve instead of $\alpha_{i,rxring}^{dual}(t)$. This means to neglect the $M + C \cdot (...)$ term in $\alpha_{i,NIC}(t)$. The resulting bound is still valid, even though it is not tight. Typically however, $\tilde{b}_i^{rx}$ is small enough, and the relevant interval t of the notifier thread is large enough, so that the bounds achieved with the second term of the $\min(...)$ only are sufficiently tight. Equation 46 gives the simplified arrival curve, denoted by $\alpha_{i,rxring}(t)$, and Figure 23 illustrates it.

$$\begin{aligned}\alpha_{i,rxring}(t) &\geq \alpha_{i,NIC}(t+D_{IRQ}) \\ &= \begin{cases} \tilde{b}_i^{rx} + r_i^{rx} \cdot (t+D_{IRQ}) & \text{for } t>0 \\ 0 & \text{for } t=0 \end{cases}\end{aligned} \tag{46}$$

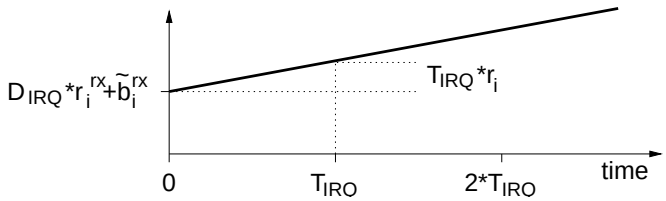

Figure 23: Arrival curve $\alpha_{i,rxring}$ of traffic received by the *IRQ thread* for connection *i*.

6.4.2. Software-based interrupt coalescing

The IRQ-related context switching costs dominate the CPU costs related to packet reception and sent packet handing. Reducing the number of interrupts per transmitted and received packets significantly reduces the worst-case CPU costs related to packet handling.

Although some modern NICs support a interrupt coalescing in hardware, it is not reliable enough for obtaining tight CPU execution time boundaries: Hardware-based interrupt coalescing has been implemented to lower the average-case CPU usage, and as thus most implementations do not guarantee a minimal inter-arrival time of interrupts or guarantee a maximum interrupt load in any other way. Further, the configuration of the coalescing features differs to a high degree between the NICs of different vendors, and there is no common interface to control hardware-based interrupt coalescing.

Instead, I propose a software-based interrupt coalescing by using the minimal inter-release time scheduling of Fiasco. With this, interrupts are only received and handled at certain minimal timely distances, the "period" of the *IRQ thread* T_{IRQ}. Figure 24 shows the relevant changes in the IRQ main loop. Note that common NICs do not need the IRQ handler to be executed for every packet. Arriving packets are enqueued into the hardware receive rings further, and packets from the hardware send ring are send further. As long as these rings do not overflow, the NIC works as expected. At most NICs these rings are software-controlled, so they can be resized as needed.

The CPU usage bound of the *IRQ thread* with interrupt coalescing, denoted by $CPU_{IRQ}^{\wr IRQ \wr}$, is given by Equation 47.

$$CPU_{IRQ}^{\wr IRQ \wr} = cpu.irq_base + cpu.irq_tx \cdot \overline{tx}_{IRQ} + cpu.irq_rx \cdot \overline{rx}_{IRQ} \tag{47}$$

The most costly part, the $cpu.irq_base$ only counts once now, as the activation of the *IRQ thread* only happens once per period of length T_{IRQ}. A disadvantage of coalescing interrupts however is that notifications of received and sent packets are deferred additionally by up to one period, T_{IRQ}.

```
void IRQ_main_loop(int nr){
    l4_rt_begin_strictly_periodic    (...);
  while(1){
    l4_ipc_receive ( irq_kernel_id (nr ),
                     TIMEOUT_INF);
                                          ⇒
    irq_handler (nr );
    irq_def_handler (nr );
} }
```

```
void IRQ_main_loop_coal(int nr){
  l4_rt_begin_minimal_periodic   (...);
  while(1){
    l4_ipc_receive ( l4_nirq_kernel_id (nr ),
                     TIMEOUT_INF,
                     NEXT_PERIOD);
    irq_handler (nr );
    irq_def_handler (nr );
} }
```

Figure 24: Using minimal inter-release time scheduling for the *IRQ thread* to lower its invocation frequency. The *NEXT_PERIOD* flag given to the IPC operation by *IRQ_main_loop_coal* makes the IPC a next-period IPC. For the concept of next-period IPCs, see Section 2.2.3.

Quantitative analysis of the IRQ thread with software-based interrupt coalescing

With the software-based interrupt coalescing, the *IRQ thread* signals the notifier up to once per T_{IRQ}. Figure 25 shows the worst-case, with respect to burstiness, of the amount of data that is put into the receive ring of the connection by the *IRQ thread.*

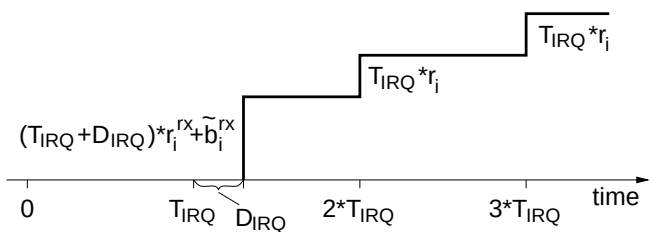

Figure 25: Bound $R_i(t)$ of traffic received by the *IRQ thread* for connection i: At the first time, the *IRQ thread* is activated as early as possible, at time 0. Due to scheduling jitter, its next invocation is at time $T_{IRQ}+D_{IRQ}$. At time $T_{IRQ}+D_{IRQ}$, it receives up to $(T_{IRQ}+D_{IRQ}\cdot r_i)+\tilde{b}_i)$ bytes from the NIC. At later invocations, not earlier than $(n+2)\cdot T_{IRQ}$, the amount of data received is bound by $T_{IRQ}\cdot r$.

Let $\alpha^{\wr IRQ \wr}_{i,rxring}(t)$ denote the arrival curve of the flow $R_i(t)$ of data that is put into the receive ring of connection i by the *IRQ thread* using interrupt coalescing. By applying self-deconvolution as described in Section 4.1 on page 17 to $R_i(t)$, $\alpha^{\wr IRQ \wr}_{i,rxring}(t)$ is obtained as $\alpha^{\wr IRQ \wr}_{i,rxring}(t) = R(t) \oslash R(t)$. Equation 48 gives the bound for the amount of data that can be put into the receive ring of connection i by the *IRQ thread* within a time interval of length t. Figure 26 shows the resulting arrival curve.

$$\alpha^{\wr IRQ \wr}_{i,rxring}(t) = \begin{cases} rx^{\wr IRQ \wr}_{i,IRQ} + \left\lfloor \frac{t+D_{IRQ}}{T_{IRQ}} \right\rfloor \cdot r^{rx}_i \cdot T_{IRQ} & t > 0 \\ 0 & t = 0 \end{cases} \tag{48}$$

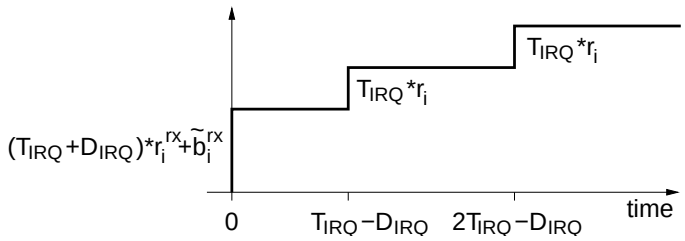

Figure 26: Arrival curve $\alpha_{i,rxring}$ of traffic received by the *IRQ thread* with interrupt coalescing for the receive ring of connection *i*.

with $rx_{i,IRQ}^{\wr IRQ \wr}$ given by:

$$rx_{i,IRQ}^{\wr IRQ \wr} = (T_{IRQ} + D_{IRQ}) \cdot r_i^{rx} + \tilde{b}_i^{rx} \tag{49}$$

Summarizing, the minimal inter-release time scheduling provides an effective method to bound the interrupt-related CPU load, without the need of specific hardware support. The CPU cost is dominated by the IRQ handler activation, whereas the actual number of transmitted or received packets is less important to the overall CPU time. Measurements quantifying the CPU usage of both IRQ scheduling methods depending on the offered load are presented in Section 7.4.3 on page 95ff and Section 7.4.4 on page 100ff.

6.4.3. Connection-specific notifier thread

Each connection has its own notifier thread that performs the operations that do not need to be done in the *IRQ thread*. This mainly includes the client interaction and connection-related memory management. As with the send threads, the multi-threaded architecture avoids a CPU-related interference of different connections as far as possible.

Client interface

The notification path, from a client side view, offers the functionality to receive data on a connection and to get notified about any packets that were successfully sent to the network. As argued in Section 6.4 on page 47, data received from the network must be copied to the client. The most flexible copying solution for the client is an interface that allows it to specify memory locations in its own address space for the placement of received network data. With indirect string IPCs the L4 IPC mechanisms provide an appropriate solution: Indirect string IPCs copy data from one address space into another to locations specified by the receiver of the IPC. This allows a *read()*-like interface that offers the biggest flexibility for client-program construction. As such, the notifier thread uses indirect string IPCs to transmit received network data from the RT-Net server to a client.

In the following, I use the term *event* to describe received data packets (receive event) and sent packets (sent event) to be signalled to the client by the notifier thread. To obtain events, the client calls the *rt_wait()* function. This function sends an request IPC to the notifier thread and waits for a reply IPC.

The request contains the following information:

- if the client is interested in sent events
- the maximum number of received packets the client is going to accept in the requests answer. If the does not want to receive data, for example because it is not interested in receive events, it specifies 0 here.
- if the client is willing to wait (block) for any of the requested events if none of them are available at the time of the request

If the client specified not to block on events, or if any of the requested events are available, the notifier thread replies immediately with an IPC. Otherwise, it delays the reply until at least one of the requested events becomes available.

The number of received network packets in each reply IPC is limited to 31 by the current L4 API. The client may limit this number further, for instance to keep its own request processing simple. Note that a reply may contain no received packets at all, albeit the client requested receive events in blocking mode, if sent events are to be signalled.

Sent events are coalesced by the notifier thread in the sense that a client only receives information whether any packets have been sent since its previous request or not. The client can obtain detailed information about which packets have been sent by inspecting the send ring of the connection.

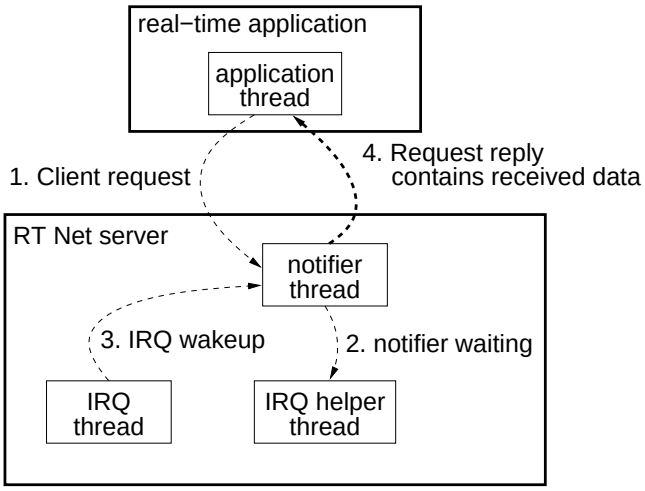

Figure 27: IPC communication structure of the connection-specific notifier thread.

Outer server loop of the notifier thread

Listing 8 shows the pseudo-code of the notifier threads outer client request processing loop. It is a classical server-loop that tries to combine replies to requests and the reception of new requests into one system call (line 6). If the current request requires to wait for the arrival of new events, thus it is a blocking operation, new events are obtained by calling `flush_wait_data()` in line 5.

For nonblocking requests, the reply in line 6 may contain neither received data packets nor sent notifications.

```
1 void Conn:: notifier (void){
2   while(1){
3       flush_wait_client (&client , &request);          /* await client request */
4     do{
5         if ( request . wait )  flush_wait_data ();      /* obtain data */
6     }while( flush_wake_client (&client , &request)==0);  /* answer & get new request */
7 } }
```

Listing 8: Connection-specific notifier thread

Waiting for events from the IRQ thread

`flush_wait_data()` waits for new events from the *IRQ thread*, or, if events are present already, returns immediately. Its pseudo-code is given in Listing 9.

Line 4 waits for a notification from the *IRQ thread*. Therefore, `flush_wait_data()` atomically sends an IPC to the IRQ helper thread and waits for a reply. The helper thread isolates the notifier thread and the *IRQ thread* with respect to real-time reservations and allows an asynchronous signalling between the two. As its need is very specific to the L4 mikrokernel and the used Fiasco implementation, a discussion would be out of scope here. The main arguments are described in [LRH01].

Line 9 of `flush_wait_data()` tests whether the received IPC is a wakeup-IPC from either the *IRQ thread* or the IRQ helper thread. Line 10 checks for a new client request that might have been received instead of an IPC from the *IRQ thread*. Clients may abort and resend their requests anytime, and this must be taken care of by notifier thread whenever it blocks in an IPC.

Notifying the client

When data has to be delivered to the client, or the client has to be informed that no data is available, the notifier thread calls `flush_wake_client()`. Its pseudo-code is given in Listing 10. `flush_wake_client)()` replies to the current client request and, within the same system call, waits for the next request.

Line 3 resets the counter holding the number of successfully transmitted packets to be signalled to the client. This counter is increased by the *IRQ thread* and used by the `data_avail()` and `thresholds_over()` functions. The `fill_client_msgbuffer()` function called at line 4 fills a message buffer structure. At line 5, this message buffer is used to transfer up to 31 received network packets from the address space of the RT-Net server into the address space of the client. If no messages are to be delivered, the IPC at line 5 becomes a *short IPC*, which is

```
1  void Conn::flush_wait_data (void){
2    while(1){
3      if ( ( avail = data_avail ()) ) { return; }
4      err = ipc_reply_and_wait ( irq_signal_thread ,        /* IRQ helper thread */
5                                 this ,                     /* pass connection */
6                                 &sender, &dw0,             /* sender & param of new request */
7                                 TIMEOUT_SEND_RECV); /* blocking IPC */
8      if ( is_no_error ( err ) &&
9          l4_task_equal (sender , irq_thread )) continue;
10     if ( is_no_error ( err ) && l4_task_equal(sender , client )){
11         client_req .v=dw0;                                 /* new client-request. */
12 } } }
```

Listing 9: Waiting for data from the *IRQ thread*

processed faster than a *long IPC* containing messages by the Fiasco mikrokernel. The reply is sent nonblocking to prevent a starvation of the notifier thread due to an aborted client request.

The IPC system call also waits for incoming new requests. Line 9 verifies that the new request was sent from the task of the client to avoid misusing the connection by other applications.

Line 13 removes the packets that have been successfully transmitted to the client from the receive ring shared between the *IRQ thread* and the notifier thread. This involves basic FIFO operations and thread-safe, nonblocking memory de-allocation. I implemented both with lock-free linked lists as described in [Val95].

Client models

The following sections quantitatively analyze the resource usage of the notifier thread. Therefore, they discriminate two client models: (1) the blocking client model, where clients are assumed to issue blocking requests, and (2) the polling client model, where clients regularly poll for events using nonblocking requests.

6.4.4. Blocking client model

The blocking client model assumes a client that receives data from the notifier thread in the following manner: The client receives up to $n_{i,client}$ bytes from the notifier thread no later than $D_{i,client}$ time units after the *IRQ thread* has put a packet into the receive ring. As long as the receive ring of the connection is not empty, the client sends further requests and receives up to $n_{i,client}$ bytes from the notifier thread at least every $D_{i,client}$ time units. In addition to this, the client may request for sent events.

```
int Conn::flush_wake_client (l4_threadid_t *client, request_t *request){
  ...
  tx_committed = 0;                          /* reset committed tx counter */
  len = fill_client_msgbuffer (&msg);        /* create msg-buffer to be sent to client */
  err = l4_ipc_reply_and_wait (*client, msg,          /* send message to client */
                               &thread, request,      /* receive new request */
                               TIMEOUT_RECV);         /* do not block in reply */
  if (is_send_error (err)) return err;
  if (is_no_error (err) && l4_task_equal(thread, *client)){
     *client = thread;                       /* store client thread for later response */
   } else err = 1;

    ... remove len packets from the rx ring-buffer ...
   return err;                                /* 0 if no error */
}
```

Listing 10: Transmitting data to the client

To implement this model, a real-time thread with a relative scheduling deadline of $D_{i,client}$ (the client thread) and a priority lower than that of the RT-Net server periodically sends blocking receive requests to the notifier thread. Optionally, it combines these requests with sent requests. The client threads scheduling period $T_{i,client}$ is somewhat arbitrary, although it determines the CPU reservation $C_{i,client}$, and therefore influences $D_{i,client}$ as well: $C_{i,client}$ results from the amount of traffic the client thread needs to request and process within $T_{i,client}$: $r_i^{tx} \cdot T_{i,client}$ for transmit data and $r_i^{rx} \cdot T_{i,client}$ for receive data. Due to response-time analysis, it is ensured that within $D_{i,client}$ the client (1) is scheduled, and (2) can request and process all traffic it made its CPU reservation for, given that enough data is available at the RT-Net server. If less data is available, the client thread has enough CPU time left to start a next iteration to wait for and to process the further data that may be signalled in the current period. Again, it will execute its request processing within $D_{i,client}$. A lower bound for $n_{i,client}$ is given by Equation 50.

$$n_{i,client} \geq \left\lceil \frac{r_i^{rx} \cdot D_{i,client}}{M_i} \right\rceil \cdot M_i \tag{50}$$

As long as the receive ring is not empty, the client receives $n_{i,client}$ bytes at least every $D_{i,client}$ time units, corresponding to a rate of at least r_i^{rx}.

CPU usage of the notifier thread with blocking client model

The CPU usage bound of the notifier thread depends

i) on the number of signalled events and the amount of data to be transmitted to the client per scheduling period of the notifier thread, and
ii) on the number of invocations due to the client application.

The worst-case scenario with respect to CPU usage of the notifier thread is given by a client issuing one request per sent packet and one request per received packet. For scheduling reasons, the notifier thread has a period assigned, denoted by $T_{i,notify}$, and a relative scheduling deadline, denoted by $D_{i,notify}$.

The amount of sent and received traffic signalled by the notifier thread to a client within $T_{i,notify}$ is calculated analogously to Section 6.4.1 (non–IRQ coalescing) and Section 6.4.2 (IRQ coalescing).

For the **noncoalescing IRQ thread**, the signalled transmit traffic is derived from Equation 36 on page 52. The time interval however changes from T_{IRQ} to $T_{i,notify}$. Further, the "history" from a previous period, expressed by $D_{IRQ} \cdot r_i^{tx}$ in Equation 36 on page 52, is additionally bounded by M_i^{tx}: Multiple packets that were sent in the past are only signalled once as a whole by the notifier thread. Equation 51 gives the bound of the amount of sent traffic signalled by the notifier thread to a client within $T_{i,notify}$. $tx_{i,notify}$ denotes the bound.

$$tx_{i,notify} = T_{i,notify} \cdot r_i^{tx} + \min(M_i^{tx}, D_{IRQ} \cdot r_i^{tx}) + \tilde{b}_i^{tx} \tag{51}$$

The corresponding number of packets, denoted by $\overline{tx}_{i,notify}$, is given by Equation 52. Note that as with the *IRQ thread*, a useful bound cannot be given, if the packets of the connection are arbitrarily small.

$$\overline{tx}_{i,notify} = \left\lfloor \frac{tx_{i,notify}}{M_i^{tx}} \right\rfloor \tag{52}$$

The amount of received traffic signalled by the notifier thread to a client within $T_{i,notify}$ depends on the data in the receive ring at the begin of the time interval and the amount of traffic the *IRQ thread* puts into the receive ring within $T_{i,notify}$. The data in the receive ring at the begin is bound by the length of the receive ring $rxring_i$ itself. The data additionally put in is bounded by $\alpha_{i,rxring}(T_{i,notify})$, given by Equation 46 on page 55. Equation 53 gives the resulting bound, denoted by $rx_{i,notify}$.

$$\begin{aligned} rx_{i,notify} &= rxring_i + \alpha_{i,rxring}(T_{i,notify}) \\ &= rxring_i + \tilde{b}_i^{rx} + r_i^{rx} \cdot (T_{i,notify} + D_{IRQ}) \end{aligned} \tag{53}$$

The corresponding number of packets, denoted by $\overline{rx}_{i,notify}$ is given by Equation 54.

$$\overline{rx}_{i,notify} = \left\lfloor \frac{rx_{i,notify}}{M_i^{rx}} \right\rfloor \tag{54}$$

With $cpu.notify_base$ denoting the base CPU costs of a client request, $cpu.notify_rx$ denoting the base CPU costs per received packet, $cpu.notify_rx_byte$ denoting the CPU costs per byte to copy a received packet to the client and $cpu.notify_tx$ denoting the CPU costs per sent packet, the bound for CPU reservation at the kernel is given by Equation 55.

$$\begin{aligned} CPU_{i,notify} = \; & cpu.notify_base \cdot (\overline{tx}_{i,notify} + \overline{rx}_{i,notify}) + \\ & cpu.notify_tx \cdot \overline{tx}_{i,notify} + \\ & (cpu.notify_rx + cpu.notify_rx_bytes \cdot M_i^{rx}) \cdot \overline{rx}_{i,notify} \end{aligned} \tag{55}$$

Although the client could cause a higher CPU consumption at the notifier thread through endless polling, it is of no practical use: The notifications would not be received any faster. Nonetheless, the CPU usage of the notifier thread is bounded by the kernel according to the CPU reservation at connection setup. As such, a demanding client only effects its own connection.

The bounds for the **coalescing IRQ thread** reflect the larger bursts introduced by the *IRQ thread* due to its changed scheduling policy. For signalled transmit traffic, there is no real difference in practice: Multiple packets sent in the past are only signalled once as a whole. Equation 56 gives the bound, denoted by $tx_{i,notify}^{\wr IRQ\wr}$, of the amount of sent traffic signalled by the notifier thread to a client within $T_{i,notify}$. The corresponding bound for the number of packets, denoted by $\overline{tx}_{i,notify}^{\wr IRQ\wr}$, is given by Equation 57.

$$tx_{i,notify}^{\wr IRQ\wr} = T_{i,notify} \cdot r_i^{tx} + \min(M_i^{tx}, (T_{IRQ} + D_{IRQ}) \cdot r_i^{tx}) + \tilde{b}_i^{tx} \tag{56}$$

$$\overline{tx}_{i,notify}^{\wr IRQ\wr} = \left\lfloor \frac{tx_{i,notify}^{\wr IRQ\wr}}{M_i^{tx}} \right\rfloor \tag{57}$$

The calculation of the bound for the amount of signalled received traffic uses $\alpha_{i,rxring}^{\wr IRQ\wr}$ instead of $\alpha_{i,rxring}$. The calculation also takes a changed receive ring size into account, which is calculated later in this section. The traffic bound, denoted by $rx_{i,notify}^{\wr IRQ\wr}$, is given by Equation 58. The corresponding number of packets, denoted by $\overline{rx}_{i,notify}^{\wr IRQ\wr}$ if given by Equation 59.

$$\begin{aligned} rx_{i,notify}^{\wr IRQ\wr} &= rxring_i^{\wr IRQ\wr} + \alpha_{i,rxring}^{\wr IRQ\wr}(T_{i,notify}) \\ &= rxring_i^{\wr IRQ\wr} + \tilde{b}_i^{rx} + r_i^{rx} \cdot (T_{IRQ} + D_{IRQ}) + \left\lfloor \frac{T_{i,notify} + D_{IRQ}}{T_{IRQ}} \right\rfloor \cdot r_i^{rx} \cdot T_{IRQ} \\ &= rxring_i^{\wr IRQ\wr} + \tilde{b}_i^{rx} + r_i^{rx} \cdot \left(\left\lfloor \frac{T_{i,notify} + T_{IRQ} + D_{IRQ}}{T_{IRQ}} \right\rfloor \cdot T_{IRQ} + D_{IRQ} \right) \end{aligned} \tag{58}$$

$$\overline{rx}_{i,notify}^{\wr IRQ\wr} = \left\lfloor \frac{rx_{i,notify}^{\wr IRQ\wr}}{M_i^{rx}} \right\rfloor \tag{59}$$

The bound for CPU reservation at the kernel, denoted by $CPU_{i,notify}^{\wr IRQ\wr}$ is given by Equation 60.

$$\begin{aligned} CPU_{i,notify}^{\wr IRQ\wr} = \; & cpu.notify_base \cdot (\overline{tx}_{i,notify}^{\wr IRQ\wr} + \overline{rx}_{i,notify}^{\wr IRQ\wr}) + \\ & cpu.notify_tx \cdot \overline{tx}_{i,notify}^{\wr IRQ\wr} + \\ & (cpu.notify_rx + cpu.notify_rx_bytes \cdot M_i^{rx}) \cdot \overline{rx}_{i,notify}^{\wr IRQ\wr} \end{aligned} \tag{60}$$

Receive ring size and notifier delay with the blocking client model

The receive ring of a connection i must be large enough to compensate the jitter of the received traffic and the scheduling jitter of the threads of the receive path – the *IRQ thread*, the notifier thread and the client thread. With the exception of the client thread, all relevant behavior has been defined so far.

At the begin of this section on page 60 the client behavior has been defined. To formally describe the client, I apply the concept of a service curve and model the data received by the client with a service curve, denoted by $\beta^{block}_{i,client}$. Its derivation is done analogously to [BT01]. Equation 61 gives the service curve.

$$\beta^{block}_{i,client}(t) = \left\lfloor \frac{t}{D_{i,client}} \right\rfloor \cdot n_{i,client} \tag{61}$$

Using the arrival and the service curves of the receive ring, its worst-case size bound can be calculated. The delay due to buffering and processing by the notifier thread and the client thread can be calculated as well. The arrival curve is given by $\alpha_{i,rxring}(t)$ and $\alpha^{coal}_{i,rxring}(t)$, respectively. The service curve is $\beta^{block}_{i,client}(t)$.

For the **noncoalescing IRQ thread**, the arrival curve is given by $\alpha_{i,rxring}(t)$, defined by Equation 46 on page 55. This curve conforms to a leaky bucket with bucket size $\tilde{b}^{rx}_i + r^{rx}_i \cdot D_{IRQ}$ and rate r^{rx}_i. Figure 28 shows the arrival and service curves of a connection i.

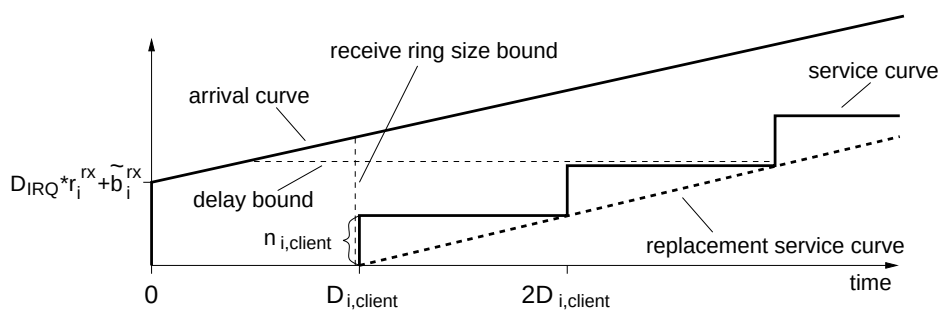

Figure 28: Arrival curve of *IRQ thread* with the clients service curve. To obtain the delay and receive ring size bound, the service curve is replaced by a rate-latency function.

According to Equation 2 on page 18, the buffer bound of a system with service curve $\beta^{block}_{i,client}(t)$ being traversed by a flow with arrival curve $\alpha_{i,rxring}(t)$ is given by $\sup_{s\geq 0}[\alpha_{i,rxring}(s) - \beta^{block}_{i,client}(s)]$. Thus the receive ring size is given by Equation 62.

$$rxring_i = \tilde{b}^{rx}_i + r^{rx}_i \cdot (D_{IRQ} + D_{i,client}) \tag{62}$$

According to Equation 3 on page 18, the delay bound of a system with service curve $\beta^{block}_{i,client}(t)$ being traversed by a flow with arrival curve $\alpha_{i,rxring}(t)$ is given by $\sup_s[\inf\{\tau \geq 0 : \alpha_{i,rxring}(\tau) \leq$

$\beta^{block}_{i,client}(\tau+s)\}]$. Let $d^{rx,n>}_{i,client}$ denote the delay bound for the case that $n_{i,client} > \alpha_{i,rxring}(D_{i,client})$. It is given by Equation 63 [6].

$$d^{rx,n>}_{i,client} = D_{i,client} \tag{63}$$

If $n_{i,client} \leq \alpha_{i,rxring}(D_{i,client})$, finding the bound is more complicated. To ease the calculation, I replace the service curve $\beta^{block}_{i,client}(t)$ by its lower-bound rate-latency curve $\beta_{R,T}$ with rate $R = n_{i,client}/D_{i,client}$ and latency $T = D_{i,client}$. This replacement is valid, as a system that offers a service curve γ offers a service curve $\gamma' \leq \gamma$ also. If, however, $\beta^{block}_{i,client}(t)$ is larger than $\alpha_{i,rxring}(t)$ for some t, the replacement increases the delay bound. To obtain a result that can be expressed by a simple formula, I use the rate-latency service curve to obtain a result that is a safe but not necessarily tight bound.

The delay bound d of a system with service curve $\beta(t) = R \cdot (t-T)^+$ traversed by a flow with a leaky-bucket arrival curve $\alpha(t) = b + r \cdot t$ is given in [BT01] as $d = T + b/R$. Let $d^{rx,n<}_{i,client}$ denote the delay bound for packets until they are processed by the client after being put into the receive ring if $n_{i,client} \leq \alpha_{i,rxring}(D_{i,client})$. The bound is given by Equation 64.

$$d^{rx,n<}_{i,client} = D_{i,client} + \left(\tilde{b}^{rx}_i + r^{rx}_i \cdot D_{IRQ}\right) \frac{D_{i,client}}{n_{i,client}} \tag{64}$$

In the pathological case, the delay bound as given by Equation 64 is too pessimistic by $D_{i,client}$ due to the service curve replacement. In the typical case, where r^{rx}_i is in the same order as $n_{i,client}/D_{i,client}$, the difference is much smaller. If $r^{rx}_i = n_{i,client}/D_{i,client}$, the bound given by Equation 64 is even tight.

With the **coalescing IRQ thread**, the arrival curve is given by $\alpha^{\wr IRQ \wr}_{i,rxring}(t)$, defined by Equation 48 on page 56. In contrast to $\alpha_{i,rxring}(t)$, $\alpha^{\wr IRQ \wr}_{i,rxring}(t)$ is not a leaky-bucket constraint, which aggravates the calculation of tight bounds. If $n_{i,client} \geq \alpha^{\wr IRQ \wr}_{i,rxring}(D_{i,client})$, the derivation in Section A.1 on page A-1 can be used and the following buffer and delay bounds, denoted by $rxring^{\wr IRQ \wr,n>}_i$ and $d^{rx,\wr IRQ \wr,n>}_{i,client}$, are found:

$$\begin{aligned} rxring^{\wr IRQ \wr,n>}_i &= \tilde{b}^{rx}_i + r^{rx}_i \cdot \left(T_{IRQ} + D_{IRQ} + \left\lfloor \frac{D_{i,client} + D_{IRQ}}{T_{IRQ}} \right\rfloor \cdot T_{IRQ}\right) && (65) \\ &\leq \tilde{b}^{rx}_i + r^{rx}_i \cdot (T_{IRQ} + 2 \cdot D_{IRQ} + D_{i,client}) \\ d^{rx,\wr IRQ \wr,n>}_{i,client} &= D_{i,client} && (66) \end{aligned}$$

To obtain results that can be expressed by simple formulas for the case that $n_{i,client} < \alpha^{\wr IRQ \wr}_{i,rxring}(D_{i,client})$, I replace the arrival curve $\alpha^{\wr IRQ \wr}_{i,rxring}(t)$ by its upper-bound leaky-bucket curve α_{b,r^{rx}_i} with the same long-term average rate r^{rx}_i. This replacement is valid, as a flow that has an arrival curve $\alpha(t)$ has also an arrival curve $\alpha'(t) \geq \alpha(t)$. To obtain b, it must especially hold:

$$\begin{aligned} \alpha_{b,r^{rx}_i}(T_{IRQ} - D_{IRQ}) &= \alpha^{\wr IRQ \wr}_{i,rxring}(T_{IRQ} - D_{IRQ}) \\ b + r^{rx}_i \cdot (T_{IRQ} - D_{IRQ}) &= (T_{IRQ} + D_{IRQ}) \cdot r^{rx}_i + \tilde{b}^{rx}_i + r^{rx}_i \cdot T_{IRQ} \\ b &= (T_{IRQ} + 2 \cdot D_{IRQ}) \cdot r^{rx}_i + \tilde{b}^{rx}_i \end{aligned}$$

[6] The proof uses the fact that $\forall t > 0 : \alpha_{i,rxring}(t + D_{i,client}) \leq \alpha_{i,rxring}(t) + n_{i,client}$ with $n_{i,client} \geq r^{rx}_i \cdot D_{i,client}$.

As with the noncoalescing *IRQ thread*, I replace the service curve $\beta^{block}_{i,client}(t)$ by its lower-bound rate-latency curve $\beta_{R,T}$ with rate $R = n_{i,client}/D_{i,client}$ and latency $T = D_{i,client}$. Figure 29 shows the arrival curve, the service curve and their replacements for a connection i if the *IRQ thread* coalesces interrupts.

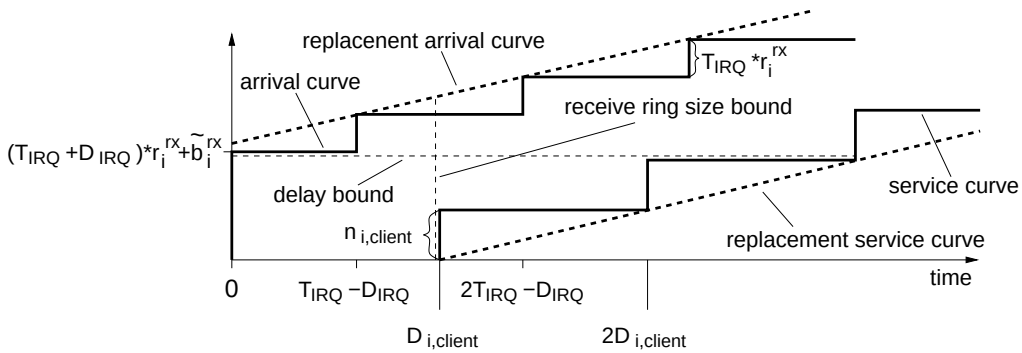

Figure 29: Arrival curve of the coalescing *IRQ thread* with the blocking clients service curve and their replacements.

According to [BT01], the buffer bound for a system with service curve $\beta(t) = R \cdot (r - T)^+$ traversed by a flow with a leaky-bucket arrival curve $\alpha(t) = b + r \cdot t$ is $B = b + r \cdot T$. Hence, the buffer bound for the receive ring, denoted by $rxring_i^{\wr IRQ\wr,n<}$ is given by Equation 67.

$$rxring_i^{\wr IRQ\wr,n<} = \tilde{b}_i^{rx} + r_i^{rx} \cdot (T_{IRQ} + 2 \cdot D_{IRQ} + D_{i,client}) \tag{67}$$

The derivation of the delay bound for the coalescing *IRQ thread*, denoted by $d_{i,client}^{rx,\wr IRQ\wr,n<}$, is similar to that of the noncoalescing *IRQ thread.* Equation 68 gives the result.

$$d_{i,client}^{rx,\wr IRQ\wr,n<} = D_{i,client} + \left(\tilde{b}_i^{rx} + r_i^{rx} \cdot (T_{IRQ} + 2 \cdot D_{IRQ})\right) \frac{D_{i,client}}{n_{i,client}} \tag{68}$$

6.4.5. Polling client model

The application model of a polling client is a strictly periodic client that receives packets from the notifier thread but does not block if no data is available. Let $T_{i,client}$ denote its period length and $D_{i,client}$ its relative scheduling deadline, $D_{i,client} < T_{i,client}$. Figure 30 illustrates the application model: Once in each period, the client issues a burst of requests to receive packets from the notifier thread.

In contrast to receive notifications, sent notifications are not needed with a periodic client: If the client produces its data according to the negotiated rate, its transmit buffer is guaranteed to be emptied in time. If, however, the client tries to transmit more data, it will notify a full transmit ring, and has to poll for an empty transmit ring element in the next period.

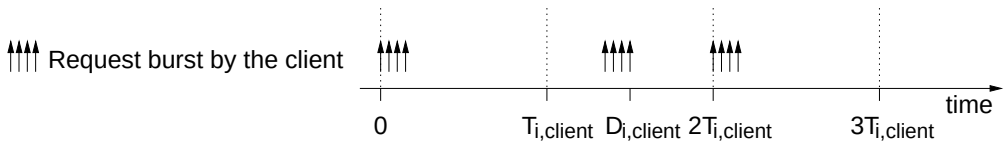

Figure 30: Application model of the client with coalescing notifier thread.

The advantage of the polling client in comparison to the blocking client model as described in Section 6.4.4 is the reduced upper bound on the number of requests a client issues in a given time interval. The sent events are not requested at all, and the receive events are reduced: The nonblocking requests allow a client to wait for a small time before it issues its next requests. As the result, the CPU reservation needed is reduced by the polling client model.

The remainder of this section quantitatively analyzes the notifier thread, if the *IRQ thread* does not coalesce interrupts. Section 6.4.6 on page 69 does the analysis for the coalescing *IRQ thread.*

The number of bytes the noncoalescing *IRQ thread* puts into the receive ring of connection within a time interval of length $T_{i,client}$ is bounded by $\alpha_{i,rxring}(T_{i,client})$, with $\alpha_{i,rxring}(t)$ given by Equation 46 on page 55.

Considering the receive ring of connection i as a network element, $\alpha_{i,rxring}(t)$ corresponds to its arrival curve. The client that receives the packets defines the service curve, that is the minimum amount of traffic the client removes from the receive ring in any time interval of length t. If the client receives up to $\alpha_{i,IRQ}(T_{i,client})$ bytes in each of its periods, Equation 69 gives its service curve. Figure 31 illustrates the scenario.

$$\beta^{poll}_{i,client}(t) = \alpha_{i,IRQ}(T_{i,client}) \cdot \left\lfloor \frac{(t - D_{i,client})^+}{T_i, client} \right\rfloor \tag{69}$$

Figure 31 shows also the maximum required receive ring size $rxring_i^{poll}$ of connection i: it is the maximum vertical distance between the arrival and the service curve:

$$rxring_i^{poll} = \alpha_{i,IRQ}(T_{i,client} + D_{i,client}) \tag{70}$$

The time a packet is stored in the receive ring of connection i is its delay due to the notifier thread and the client application. Its bound $d_{i,client}$ is given by the maximum horizontal deviation of the arrival and the service curve, defined by Equation 3 at page 18. It is given by Equation 71.

$$d^{poll}_{i,client} = T_{i,client} + D_{i,client} \tag{71}$$

CPU usage of the notifier thread with polling client model

The amount of CPU needed by the notifier thread per period results (1) from the amount of data it has to transmit to the client, and (2) from the number of requests it has to answer. A discussion of the bound of the amount of data needs to consider the following aspects:

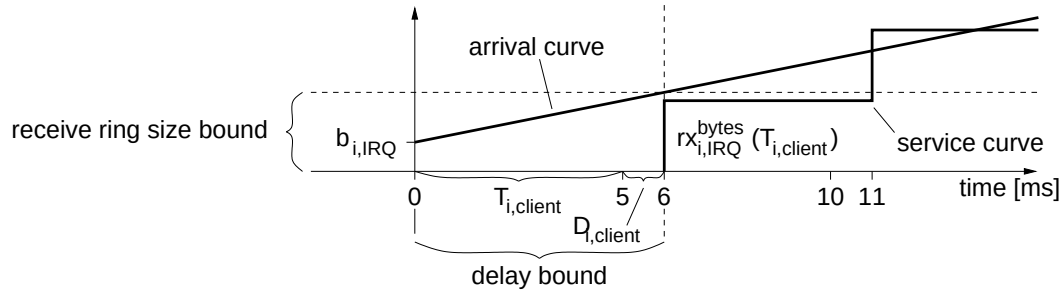

Figure 31: Receive ring of connection i: Arrival and service curves $\alpha_{i,rxring}$ and $\beta^{poll}_{i,client}$. The client has a period of $T_{i,client} = 5$ and a relative scheduling deadline of $D_{i,client} = 1$. The delay bound for packets in the receive ring is 6 and can be found twice in the graph: Once at $\alpha_{i,rxring}(0)$, and another time at $\alpha_{i,rxring}(5)$.

1. The notifier thread does not implement any form of traffic shaping. Doing so would require either to read fine-granular timers or to apply strictly periodic scheduling. The periodic scheduling would result in scheduling delays, and the timers would add additional CPU costs.
2. The notifier thread must not be preempted when servicing a clients request. The clients operate in periodic mode, and expect the notifier thread to answer their requests nonblocking.
3. A safe bound for the amount of data that can arrive within a period of the notifier thread is given by $\alpha_{i,IRQ}(T_{i,notify})$. However, a client might want to read all data queued in the receive ring first, and then $\alpha_{i,IRQ}(T_{i,notify})$. If the CPU bound would be based on $\alpha_{i,IRQ}(T_{i,notify})$, the notifier thread might be preempted due to CPU shortage while servicing a client request.
4. Using the network calculus' output flow concept, a bound for the amount of data that can be transmitted to the client within $T_{i,notify}$ is given by the *output flow* $\alpha^*(T_i, notify)$, with $\alpha^* = \alpha_{i,IRQ} \oslash \beta^{poll}_{i,client}$, as described in Section 4.1 on page 17. This bound corresponds to reading all data queued at the receive ring first, and $\alpha_{i,IRQ}(T_{i,notify})$ bytes then. However, this amount of traffic is more than what the client interface specifies. As such, this bound is a safe bound for the notifier thread, but it is not tight.
5. The client model assumes a periodic client witch receives up to $\alpha_{i,rxring}(T_{i,client})$ bytes per period of length $T_{i,client}$. Thus, within $T_{i,notify}$, a client can receive this amount up to $1 + \lfloor T_{notify}/T_{client} \rfloor$ times, which gives a bound for the data that needs to be transferred to the client within $T_{i,notify}$. However, it is not a tight bound, as it counts the bursts implicitly in $\alpha_{i,rxring}$ multiple times.

Which of the bounds given in 4. and 5. is better depends on the length of the periods, and of the characteristics of the arriving data flows. It cannot be determined in general which one is the better bound. Thus, using the minimum of both is reasonable.

With $\alpha_{i,IRQ}$ and $\beta^{poll}_{i,client}$ given as previously defined, the output flow $\alpha^*(t)$ is given by $\alpha^*(t) = \alpha_{i,IRQ}(t) + rxring^{poll}_i$. The resulting bound for the amount of data the notifier thread has to transmit

to the client per period $T_{i,notify}$ is given by Equation 72. The corresponding number of packets, denoted by $\overline{rx}^{poll}_{i,notify}$, is given in Equation 72.

$$rx^{poll}_{i,notify} = \min\left[\alpha_{i,rxring}(T_{i,client})\cdot\left(1+\left\lfloor\frac{T_{i,notify}}{T_{i,client}}\right\rfloor\right),\quad \alpha_{i,rxring}(T_{i,notify})+rxring^{poll}_{i}\right] \tag{72}$$

$$\overline{rx}^{poll}_{i,notify} = \left\lfloor\frac{rx^{poll}_{i,notify}}{M^{rx}_{i}}\right\rfloor \tag{73}$$

At connection setup, the client specifies how many packets it is going to receive per request. nr^{rx} denotes this number. Due to kernel interface-limitations, no more than 31 packets can be transmitted per request. The resulting number of requests the notifier thread needs to handle per period is given by Equation 74.

$$\check{rx}^{poll}_{i,notify} = \left\lfloor\frac{rx^{poll}_{i,notify}}{\min(nr^{rx},31)\cdot M^{rx}_{i}}\right\rfloor \tag{74}$$

As a result, the CPU bound of the coalescing notifier thread to handle client requests are reduced in comparison to the noncoalescing notifier: First, the number of client requests for receiving data is reduced, and second, the sent-notifications are not sent at all. Equation 75 gives the resulting CPU bound, as needed for the CPU reservation.

$$\begin{aligned} CPU^{poll}_{i,notify} = {} & cpu.notify_base\cdot\check{rx}^{poll}_{i,notify}+cpu.notify_rx\cdot\overline{rx}^{poll}_{i,notify}+ \\ & cpu.notify_rx_bytes\cdot rx^{poll}_{i,notify} \end{aligned} \tag{75}$$

6.4.6. Interrupt coalescing with the polling client model

If, in addition to the scenario analyzed in Section 6.4.5, the *IRQ thread* applies interrupt coalescing, the bounds derived in Section 6.4.5 change due to the difference of the arrival curve $\alpha^{\wr IRQ\wr}_{i,rxring}$ compared to $\alpha_{i,rxring}$.

Using the same argumentation as in Section 6.4.5 on page 66, the client needs to receive at least $\alpha^{\wr IRQ\wr}_{i,rxring}$ bytes in each of its period. Equation 76 gives the corresponding service curve.

$$\beta^{\wr IRQ\wr,poll}_{i,client}(t) = \alpha^{\wr IRQ\wr}_{i,rxring}(T_{i,client})\cdot\left\lfloor\frac{(t-D_{i,client})^{+}}{T_i,client}\right\rfloor \tag{76}$$

Figure 32 shows the arrival curve $\alpha^{\wr IRQ\wr}_{i,rxring}$ due to the *IRQ thread* and the clients service curve $\beta^{\wr IRQ\wr,poll}_{i,client}$.

Figure 32 shows also the maximum required receive ring size $rxring^{\wr IRQ\wr,poll}_{i}$ of connection i: it is the maximum vertical distance between the arrival and the service curve, which is at $T_{i,client}+D_{i,client}$. The receive ring size is given by Equation 77.

$$rxring^{\wr IRQ\wr,poll}_{i} = \alpha^{\wr IRQ\wr}_{i,rxring}(T_{i,client}+D_{i,client}) \tag{77}$$

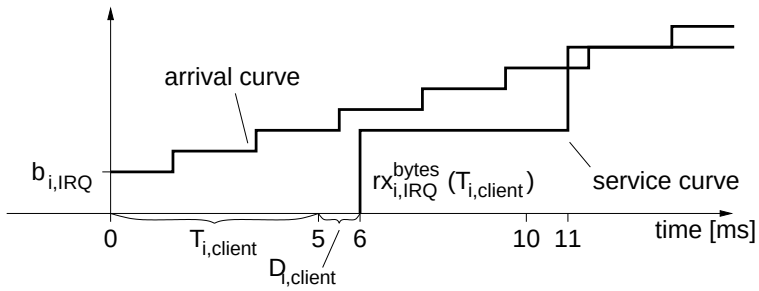

Figure 32: Arrival curve of the coalescing *IRQ thread* with the clients service curve. Both are step curves.

The time a packet is stored in the receive ring of connection i is its delay due to the notifier thread and the client application. Its bound $d_{i,client}^{\wr IRQ\wr,poll}$ is given by the maximum horizontal deviation of the arrival and the service curve, defined by Equation 3 at page 18. It is given by Equation 78.

$$d_{i,client}^{\wr IRQ\wr,poll} = T_{i,client} + D_{i,client} \tag{78}$$

The prove for both bounds is given in Section A.1 on page A-1.

With $\alpha_{i,rxring}^{\wr IRQ\wr}$ and $\beta_{i,client}^{\wr IRQ\wr,poll}$ given as previously defined, the output flow $\alpha_i^{poll,*}(t)$ is given by $\alpha_i^{poll,*}(t) = \alpha_{i,rxring}^{\wr IRQ\wr}(t + rxring_i^{\wr IRQ\wr,poll})$. The resulting bound for the amount of data the notifier thread has to transmit to the client per period $T_{i,notify}$, denoted by $rx_{i,notify}^{\wr IRQ\wr,poll}$, is given by Equation 79. The corresponding number of packets, denoted by $\overline{rx}_{i,notify}^{\wr IRQ\wr,poll}$, is given in Equation 80.

$$\begin{aligned} rx_{i,notify}^{\wr IRQ\wr,poll} &= \min\left[\alpha_{i,rxring}^{\wr IRQ\wr}(T_{i,client}) \cdot \left(1 + \left\lfloor \frac{T_{i,notify}}{T_{i,client}} \right\rfloor\right), \right. \\ &\qquad \left. \alpha_{i,rxring}^{\wr IRQ\wr}(T_{i,notify}) + rxring_i^{\wr IRQ\wr,poll}\right] \end{aligned} \tag{79}$$

$$\overline{rx}_{i,notify}^{\wr IRQ\wr,poll} = \left\lfloor \frac{rx_{i,notify}^{\wr IRQ\wr,poll}}{M_i^{rx}} \right\rfloor \tag{80}$$

The resulting number of requests the notifier thread needs to handle per period, denoted by $\check{rx}_{i,notify}^{\wr IRQ\wr,poll}$, is given by Equation 81.

$$\check{rx}_{i,notify}^{\wr IRQ\wr,poll} = \left\lfloor \frac{rx_{i,notify}^{\wr IRQ\wr,poll}}{\min(nr^{rx}, 31) \cdot M_i^{rx}} \right\rfloor \tag{81}$$

Equation 82 gives the resulting CPU bound, as needed for the CPU reservation.

$$\begin{aligned} CPU_{i,notify}^{\wr IRQ\wr,poll} &= cpu.notify_base \cdot \check{rx}_{i,notify}^{\wr IRQ\wr,poll} + cpu.notify_rx \cdot \overline{rx}_{i,notify}^{\wr IRQ\wr,poll} + \\ &\quad cpu.notify_rx_bytes \cdot rx_{i,notify}^{\wr IRQ\wr,poll} \end{aligned} \tag{82}$$

Table 3: Symbols used in Sections 6.3 and 6.4.3 to denote derived results.

Symbol	Meaning	Reference
Connection-specific transmit thread *TX thread*		
$CPU_{i,tx}^{tbf}$	CPU bound for the *TX thread* using the token-bucket shaper	Equation 34 on page 47
$CPU_{i,sp}^{tbf}$	CPU bound for the *TX thread* using the strictly periodic shaper with data dependency	Equation 35 on page 47
Device-specific Interrupt thread *IRQ thread*		
T_{IRQ}	scheduling period of the *IRQ thread*	Section 6.4.1 on page 52
D_{IRQ}	scheduling deadline of the *IRQ thread*	Section 6.4.1 on page 52
$\alpha_{i,rxring}(t)$	arrival curve of received data for connection i in its receive ring	Equation 46 on page 55
$\alpha_{i,rxring}^{\wr IRQ \wr}(t)$	arrival curve of received data for connection i in its receive ring with software-based interrupt coalescing of the *IRQ thread*	Equation 48 on page 56
CPU_{IRQ}	CPU bound for the *IRQ thread*	Equation 43 on page 54
$CPU_{IRQ}^{\wr IRQ \wr}$	CPU bound for the *IRQ thread* with software-based interrupt coalescing	Equation 47 on page 55
Connection-specific notifier thread, blocking client model		
$T_{i,notify}$	Scheduling period of the notifier thread for connection i	Section 6.4.4 on page 61
$D_{i,client}$	Scheduling deadline of a blocking client.	Section 6.4.4 on page 60
$n_{i,client}$	Lower bout for the amount of data a blocking client receives every $T_{i,client}$ time units	Equation 50 on page 61
$CPU_{i,notify}$	CPU bound for the notifier thread with a noncoalescing *IRQ thread*	Equation 55 on page 62
$CPU_{i,notify}^{\wr IRQ \wr}$	CPU bound for the notifier thread with an interrupt-coalescing *IRQ thread*	Equation 60 on page 63
$\beta_{i,client}^{block}(t)$	Service curve of a blocking client	Equation 61 on page 64
$rxring_i$	Receive ring size bound with a noncoalescing *IRQ thread*	Equation 62 on page 64
$d_{i,client}^{rx,n>}$	Delay bound for packets in the receive ring for a greedy client and a noncoalescing *IRQ thread*	Equation 63 on page 65
$d_{i,client}^{rx,n<}$	Delay bound for packets in the receive ring for a slow client and a noncoalescing *IRQ thread*	Equation 64 on page 65
$rxring_i^{\wr IRQ \wr,n>}$	Receive ring size bound for a greedy client and an interrupt-coalescing *IRQ thread*	Equation 65 on page 65
$d_{i,client}^{rx,\wr IRQ \wr,n>}$	Delay bound for packets in the receive ring for a greedy client and a coalescing *IRQ thread*	Equation 66 on page 65
$rxring_i^{\wr IRQ \wr,n<}$	Receive ring size bound for a slow client and an interrupt-coalescing *IRQ thread*	Equation 67 on page 66
$d_{i,client}^{rx,\wr IRQ \wr,n<}$	Delay bound for packets in the receive ring for a slow client and a coalescing *IRQ thread*	Equation 68 on page 66

Table 4: Continued: Symbols used in Sections 6.3 and 6.4.3 to denote derived results.

Connection-specific notifier thread, polling client model		
Symbol	Meaning	Reference
$T_{i,notify}$	Scheduling period of the notifier thread for connection i	Section 6.4.4 on page 61
$T_{i,client}$	Scheduling period of a polling client.	Section 6.4.5 on page 66
$D_{i,client}$	Scheduling deadline of a polling client.	Section 6.4.5 on page 66
$\beta_{i,client}^{poll}(t)$	Service curve of a polling client	Equation 69 on page 67
$rxring_i^{poll}$	Receive ring size bound with a noncoalescing *IRQ thread*	Equation 70 on page 67
$d_{i,client}^{poll}$	Delay bound for packets in the receive ring for a a noncoalescing *IRQ thread*	Equation 71 on page 67
$\beta_{i,client}^{IRQ\wr,poll}(t)$	Service curve of a polling client with an interrupt-coalescing *IRQ thread*	Equation 76 on page 69
$rxring_i^{IRQ\wr,poll}$	Receive ring size bound for a polling client and an interrupt-coalescing *IRQ thread*	Equation 77 on page 69
$d_{i,client}^{IRQ\wr,poll}$	Delay bound for packets in the receive ring for a polling client and a coalescing *IRQ thread*	Equation 78 on page 70
$CPU_{i,notify}^{poll}$	CPU bound for the notifier thread of a polling client with a noncoalescing *IRQ thread*	Equation 75 on page 69
$CPU_{i,notify}^{IRQ\wr,poll}$	CPU bound for the notifier thread of a polling client with an interrupt-coalescing *IRQ thread*	Equation 82 on page 70

6.5. Summary

Sections 6.3 and 6.4 described the implementation to send and receive packets with the RT-Net server. The sections also quantitatively analyzed the resource needs to handle traffic flows of given characteristics, and calculated resulting delays at the involved components.

6.5.1. Analyzed execution models

Section 6.3 analyzed two different models to send data. One model uses Fiasco's minimal inter–release-time scheduling to achieve low delays at sending and to generate flows with a small burstiness. As this model has a high CPU demand, it can only be used for connections with a small bandwidth. The other model uses a token-bucket traffic shaper and is suitable for higher bandwidths at the expense of an increased worst-case delay at the sender and an increased burstiness of the generated flow.

Sections 6.4.1 and 6.4.2 analyzed two different models to process interrupt requests from the NIC. The straight-forward approach executes the interrupt handler whenever the NIC signals an event. This results in a low event processing delay but a high CPU demand. In contrast to this, the software coalescing approach executes the interrupt handler not more often than a certain minimal time interval. This reduces the CPU utilization, but increases the event processing delays and the memory needed to buffer the received data flows. While the first approach is appropriate for nodes with stringent needs for low delays, the latter approach can cope with high bandwidths.

Finally, Sections 6.4.3 to 6.4.6 analyzed the connection-specific event processing and the interaction with client applications. Two client models with different delay and CPU usage characteristics were defined: The blocking client model achieves low delays for signalling received data to client applications, but requires clients to wait, that is block, for newly arrived data. The polling client model allows clients to synchronize to other, periodic, events, and to poll for data regularly. In contrast to the blocking model, the polling model guarantees a minimal timely distance between client requests. Consequently, it achieves a lower CPU utilization bound at the RT-Net server at the expense of (1) higher delays for signalling received data to client applications, and (2) an increased memory requirement to buffer the received data flows. The analysis revealed that the resource consumption of the connection-specific event processing does not only depend on the client model used but depends also on which interrupt processing model is used, and how the client is going receive data from the RT-Net server.

6.5.2. Resource usage dependencies

As another result of the previous sections, it turned out that the resource usage for handling a specific flow at the receiving node not only depends on the flow characteristic at the send node. It also is influenced by other flows sharing resources along the path of the specific flow: Other flows transmitted from the same send node have an influence as well as flows originating from other nodes but targeting to the considered receiving node. Moreover, even flows from other nodes to other nodes might influence the resource usage at the receiving node, as they increase the

burstiness of flows originating at those other nodes but targeting to the considered receiving node, which in turn increases the burstiness of all flows to the receiving node.

This implies that each newly established traffic flow potentially influences the CPU requirements, buffer needs and delays at all attached nodes. In a dynamic environment this dependency is not desirable: Once resources are allocated for a connection and the connection is established, these resources should not need to be changed. Especially, the behavior at the client interface should not change.

To achieve this, a maximum allowed burstiness is specified for each flow on its creation. This maximum allowed burstiness corresponds to the $\tilde{b}_i$ in the previous sections, and is used for resource allocation at the receiving nodes. It is the responsibility of the bandwidth manager to only admit new flows if the actual burstiness bounds of established flows do not exceed their maximum allowed burstiness. Although this may result in a resource over-reservation, it is the concession to a network architecture that does not reshape its traffic along the network path. If, however, the application scenario is known in advance, and as such all network flows, no over-reservation is needed.

6.5.3. Alternative scheduling schemes

According to the traffic shaper analysis in Section 5 on page 29, the burstiness of the flows generated at the sending nodes, and therefore the delays at the network, heavily depends on the scheduling jitter due to CPU scheduling. The scheduling jitter at the receive nodes increase this burstiness further, resulting in increased resource reservation needs (Sections 6.4.3 to 6.4.6). However, achieving a low scheduling jitter is costly on priority-based systems: As a prize for their flexibility, priority-driven systems can guarantee low scheduling jitters only to their highest-priority threads. The lower the priority of a thread becomes, the higher its scheduling jitter is.

There are other scheduling schemes that achieve low scheduling jitters for more than a few threads in the system. Time-triggered systems for instance have a predictable scheduling scheme, and as such guarantee a lower scheduling jitter to all their scheduled threads. Consequently, time-triggered systems are likely to achieve even lower network delay bounds than priority-based systems. However, time-triggered systems require a careful design of their scheduling tables, which reduces their flexibility regarding the dynamic creation of new connections or scheduling threads with different periods.

As a conclusion, costly high-priority CPU reservations made to achieve low delays at the network are not a problem inherent to Switched Ethernet communication with software-based traffic shaping. Instead, they are an artifact of the implementation on top of a priority-driven system.

6.6. Real-time connection setup

The connection setup is triggered by a real-time client that sends an *rt-open* request to the service thread of the RT-Net server. The request contains the following parameters:

- The address of the remote node, encoded in an IP address
- Optionally the desired local address and UDP port
- The transmit stream description: the transmit mode, the bandwidth, the maximum packet size, the maximum tolerated delay at the local node and at the switch, the maximum tolerated burstiness at the switch and the maximum number of parallel outstanding send-requests
- The receive stream description: the receive mode, the bandwidth, the maximum packet size, the maximum tolerated delay at the local node and the maximum burstiness

After receiving an *rt-open* request, the RT-Net server performs the following actions:

Address translation It asks the local network manager to translate the remote IP address into their corresponding MAC address. Then the local address of the connection is determined with the help of the local network manager. If the client specified a local address, it is verified and marked as in-use. Otherwise, the local network manager returns its own IP address together with a newly allocated UDP port.

Network traffic reservation Next, the RT-Net server asks the local network manager to do a reservation for a send-connection. The bandwidth is specified by the client. The destination node is determined by a MAC lookup using ARP. The burstiness parameter of the generated flow is determined by the RT-Net server according to the traffic shaping algorithm used and the corresponding scheduling parameters. The local network manager contacts the bandwidth manager that knows about all the flows in the network. It calculates and verifies the effective burstiness parameters that apply after merging multiple flows at the send nodes and at the switch.

Thread creation The RT-Net server creates the local threads to handle the connection: the *TX thread* and the notifier thread.

CPU reservation Using the CPU model obtained from earlier measurements, the RT-Net server determines the amount of CPU time it needs to properly handle the desired network traffic and reserves that amount at the CPU reservation server.

Memory allocation The RT-Net server allocates memory for the receive ring from its own memory pool and the shared transmit ring from the memory manager in the system. To allow the client accessing the transmit ring, the memory manager is asked to allow the client to map the corresponding data space.

Filter rules The local receive-filtering rules are setup, connecting the MAC address used for real-time communication, the IP address and the UDP port of the connection with the receive-buffer and the notifier thread of the connection.

Finally, the connection is established. A handle to uniquely identify the connection together with an ID to the data space of the transmit ring are returned to the client that can start transmitting and receiving data on this connection.

If any of the operations fails due to missing resources or exceeded limits, the previous operations are completely unrolled in reverse order.

6.7. Real-time client API

The real-time communication API of the RT-Net server results in the following: Client communication is based on UDP/IP connections. Data exchange between the client and the network driver is restricted to UDP payloads and the target UDP port.

```
rt_open     ([in, out] local_ip , [in, out] local_port , [in] remote_ip,
             [in] tx_bandwidth, [in] tx_packet_size , [in] tx_packets ,
             [in] tx_mode, [in] tx_period , [in] tx_max_burstiness , [in] tx_max_delay,
             [in] rx_bandwidth, [in] rx_packet_size , [in] rx_mode], [in] rx_burstiness ,
             [in] rx_amount, [in] rx_notify_period , [in] rx_client_period ,
             [in] rx_client_delay , [in] rx_max_delay,
             [out] err , [out] connection_handle , [out] tx_ring_id );
rt_tx_desc  ([in] connection_handle , [in] size ,
             [out] err , [out] tx_desc );
rt_send     ([in] connection_handle , [in] tx_desc , [in] tx_udp_dest_port ,
             [out] err );
rt_wait     ([in] connection_handle , [in] max_rx_events, [in] wait_flag ,
             [out] err , [out] rx_count , [out] rx_data []);
rt_close    ([in] connection_handle ,
             [out] err );
```

Figure 33: Client API for communication with the network driver in real-time mode.

rt_open() The *tx_packets* parameter indicates the maximum number of packets in the TX ring and is used to calculate the TX ring size. The *tx_mode* parameter determines which traffic shaper the *TX thread* should use – the token-bucket shaper or the periodic shaper with data dependency. If the client requested the token-bucket shaper, *tx_period* specifies the scheduling period of the *TX thread*. *tx_max_burstiness* is the maximum burstiness of the generated data flow at the switch. It will be verified by the bandwidth manager during the lifetime of the connection. *tx_max_delay* is the upper bound for the delay at the node plus the delay at the switch the client is going to accept. It includes the delays due to actual transmission at the network.

rx_mode specifies the client mode of operation – blocking or polling client model. *rx_burstiness* gives the maximum burstiness of the flow as it enters the node. This parameter should be the same as *tx_max_burstiness* at the sending side of the connection. *rx_amount* specifies the amount of data the client receives at least every *rx_client_period* time units. As argued in Sections 6.4.4 and 6.4.5, in the blocking client mode the number of receive-requests per *rx_notify_period* is upper-bounded approximately by the maximum number of packets received within *rx_notify_period*. In the polling client mode however, the number of receive-requests can be upper-bounded approximately by the amount of traffic received within *rx_notify_period* divided by 31. In the polling model, both the CPU reservation and the ring buffer size are influenced by the relative scheduling deadline *rx_client_delay* of the client. Further, the *rx_max_delay* specifies the upper bound for the delay at the node the client is going to accept. Thus, the end-to-end-delay of packets sent

over an established connection is bounded by the *tx_max_delay* parameter at the sender plus the *rx_max_delay* parameter at the receiver, increased by the potential scheduling jitter of the sending client application.

rt_tx_desc() This function returns a send descriptor containing a pointer to a memory area of at least *size* bytes. The client puts its data to the address in the descriptor and calls *rt_send()* then. If the TX ring is full, the function returns an error. In this case, the client should call *rt_tx_desc()* at a later time, for instance after waiting for sent events using *rt_wait()*.

rt_send() The *tx_desc* parameter contains the descriptor returned by *rt_tx_desc()*. *tx_udp_dest_port* contains the destination UDP port number.

rt_wait() The *max_rx_events* specifies the maximum number of receive events the client is going to accept in the reply to this *rt_wait()* request. After successful return, *rx_data[]* contains *rx_count* received packets. Each packet contains the UDP source port and the UDP payload data.

rt_close() After receiving the *rt_close()* request from its client, the RT-Net server deallocates all resources assigned to the connection specified by *connection_handle*. This includes resources at local network manager, the bandwidth manager, the CPU reservation server, the memory server and local resources.

6.8. Best-effort communication

In contrast to the real-time communication path, the best-effort path guarantees no upper delays on packet transmission or a bandwidth thereof. As both architectures are nonetheless similar, I only highlight the differences between the best-effort path and the real-time path in this section.

The best-effort communication is intended to be used by whole network stacks, such as IP stacks, Appletalk stacks or other OSI layer 3 protocol stacks. These different stacks share a NIC together with the DROPS real-time stack. The RT-Net server, the only instance directly accessing the NICs, does not know all the possible layer 3 protocol stacks. Consequently, the best-effort clients and the RT-Net server exchange complete Ethernet frames including layer 2 MAC headers and layer 3 headers according to the protocols used by the clients. To these clients, the RT-Net driver resembles a virtual NIC. This virtual NIC requires its own client driver, but behaves like a normal NIC otherwise.

To allow a demultiplexing of received network data to the different clients, each best-effort client uses its own MAC address. A client obtains its MAC address as part of the connection setup process with the RT-Net server. The RT-Net server in turn contacts the local network manager (Section 6.1 on page 41) for MAC address management.

To receive network packets for arbitrary MAC addresses, the RT-Net server enables the promiscuous mode of its NICs. Note that the promiscuous mode adds no computational overhead, as all nodes are connected to a switch. As soon as the switch learns about a MAC address, it routes packets targeted to that MAC address only to the corresponding node.

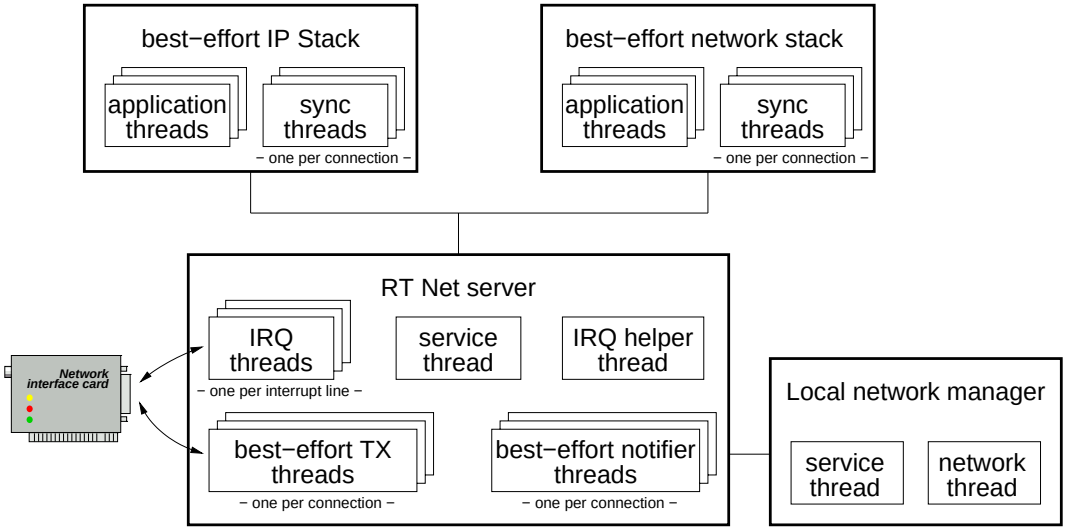

Figure 34: Thread structure of the network driver and best-effort applications. Compared to real-time clients, the best-effort clients have additional synchronization threads.

6.8.1. Best-effort send path

The *TX thread* for a best-effort connection uses the token-bucket traffic shaper for best-effort connections presented in Section 5.7 on page 36. In contrast to the token-bucket shaper for real-time connections, the best-effort shaper does not poll for packets to be transmitted. Instead, it waits for packets to achieve low delays in the average case. This synchronization process uses the DROPS streaming interface (DSI), described in [LRH01]. It establishes a shared data area and a shared ring-buffer similar to the real-time send path. But in addition, the *TX thread* blocks if it finds no packets in the ring-buffer. As described in [LRH01], the blocking uses an additional *synchronization thread* at the clients address space. If the client inserts a packet into the ring-buffer, it notifies the synchronization thread that in turn de-blocks the *TX thread* by sending it an IPC.

As the send path does not give any delay guarantees, the *TX thread* does no CPU reservation at the kernel. It runs at a lower priority than all real-time threads and is scheduled if CPU time is available.

The best-effort token-bucket shapers share a common bucket for all best-effort connections. This increases a nodes transmit capacity in comparison to separated buckets if any of the best-effort connections do not transmit data or transmit data only at a low rate.

The use of a shared bucket also means that the best-effort data flows are shaped as an aggregated flow. As a result, the accumulated burstiness of the generated flows is reduced: Instead of multiple independently shaped flows that can all transmit their burst at the same time, the burst can only be sent once. However, due to scheduling jitter, each *TX thread* can delay one packet arbitrarily long. As such, the analysis of Section 5.7 must be adapted. The length of each delayed packet is

bounded by the maximum Ethernet frame length, 1415 bytes. The burstiness b_i of the aggregated flow is thus

$$b_i = B_i + n_{be} \cdot M$$

if B_i is the bucket size of the common token-bucket, n_{be} is the number of established best-effort connections and M is 1514.

To illustrate this, assume two best-effort connections with a rate of $r_i = 8MBit/s$ each and a *TX thread* scheduling period of $T_i = 1ms$. If they use a common bucket, its minimum size is $T_i \cdot 2 \cdot r_i + M = 3514$ according to Section 5.7. The burstiness parameter of the generated leaky-bucket constrained flow is $3514 + 2 * M = 6542$. If the best-effort connections would use separated buckets, each bucket had a minimum size of $T_i \cdot r + M = 2514$. The burstiness parameter of each generated leaky-bucket constrained flow would be $2514 + 1514 = 4028$. When the two flows are multiplexed at the NIC, their burstiness increases. Neglecting the Ethernet framing overhead, this individual resulting burstiness parameter on Fast Ethernet would be $4028 + 8MBit/s \cdot 4028/100MBit/s = 4350$ according to Equation 7 on page 22. Their accumulated burstiness hence would be $2 \cdot 4350 = 8700$, which is a 33% increase compared to the shared leaky-bucket approach.

Traffic reservation

Like real-time connections, best-effort connections need a traffic reservation as well. For a transparent network connectivity, the RT-Net server takes care of the traffic reservation for best-effort connections. As such, it contacts the local network manager on the first best-effort connection establishment to reserve a low bandwidth (200 KByte/s) with the corresponding burstiness parameter. When establishing further connections, or tearing connections down, the RT-Net server adapts the burstiness parameter. On the tear-down of the last best-effort connection, the RT-Net server revokes the reservation.

As best-effort clients can use their connection to transmit data to any node of the network, best-effort traffic reservations do not relate to a specific target node. Instead, the bandwidth manager treats their reservations as reservations of broadcast traffic and applies the traffic specifications for all target nodes on the network. As a consequence, the accumulated bandwidth reservation of all best-effort clients on a network cannot exceed the medium capacity of a single link.

If, during the lifetime of a best-effort connection, the RT-Net server notices that the connection has a demand for a higher bandwidth, it transparently tries to boost the reservation. Therefore, it contacts the local network manager to increase the bandwidth/burstiness reservation. After a configurable amount of time (typically 100 ms–500 ms), the bandwidth need is reevaluated and adapted at the local network manager if necessary.

Section 7.4.7 on page 107 analyzes the performance that best-effort connections can achieve using this adaptive approach.

6.8.2. Best-effort notification path

To demultiplex received network packets to their corresponding client connections, the *IRQ thread* applies additional MAC-based filtering. If no real-time connection is found for a received packet, the *IRQ thread* tries to find a best-effort connection that has the target MAC address of the received packet assigned. Broadcast packets are enqueued at each best-effort connections receive ring.

The CPU analysis for the *IRQ thread* in Section 6.4.1 on page 49ff requires that all traffic arriving at a node is known in advance. While this assumption is reasonable for real-time traffic, it is questionable for best-effort traffic: First, the bandwidth of best-effort traffic is likely to vary to a high degree over time, especially when the one-shot reservations are used. Second, the packet sizes of best-effort traffic can hardly be predicted. Consequently, there is a huge gap between the predictable worst-case CPU consumption of the *IRQ thread* and the average case. Although the one-shot reservations could be combined to CPU reservations at all potential target nodes on the network, it would hardly reflect the actual load situation.

Therefore, application scenarios that require frequent and spontaneous high-bandwidth best-effort communication use the software-based interrupt coalescing approach presented in Section 6.4.2. This reduces the influence of network traffic to the CPU consumption of the *IRQ thread* already. Another, pragmatic, step is to obtain the worst-case CPU consumption by experimentally analyzing the CPU consumption of a node under network pressure, instead of doing a theoretical worst-case analysis based on micro-benchmarks.

Coalescing of receive events

As with the real-time connections, multiple received packets can be transmitted to a best-effort client in the reply to its event-request *be_wait()*, and multiple sent-notifications can be signalled per request. However, best-effort clients normally have no periodic execution model, but post one blocking request after the other as soon as possible. This can end up in a situation where just one event is signalled per request, resulting in a high CPU consumption due to rapid context switches. To coalesce events, client can add a coalescing timeout to their requests. New events are signalled by the RT-Net server only if that timeout passed since the previous client request or the number of outstanding events exceeds a certain threshold.

Otherwise, the notifier threads for best-effort connections do not differ much from the real-time version. The best-effort notifier threads pass received networks packets unmodified to their clients, including all lower-layer headers. Further, the notifier threads have no CPU reservation and run at a lower priority than all real-time threads.

Listing 11 shows the modified version of `flush_wait_data()` implementing the event coalescing. Line 3 looks for present events, taking care of an optional event coalescing a client might have requested. If the client requested event coalescing and the coalescing timer has not been started yet, it is started at line 6. Line 7 waits for a notification from the *IRQ thread.* In contrast to the real-time version, the best-effort version of `flush_wait_data()` uses a timeout value to the IPC to abort the IPC when the coalescing timer expires.

```
void Conn::flush_wait_data (void){
  while(1){
    if ( ( ( avail = data_avail ()) && !client_req.d.coal_timeout ) || thresholds_over ())
    { return; }
    if ( avail && timeout==0)
      { timeout = clock_value_us () + client_req.d.coal_timeout ; }
    err = ipc_reply_and_wait ( irq_signal_thread ,        /* IRQ helper thread */
                               this ,                     /* pass connection */
                               &sender, &dw0,             /* sender & param of new request */
                               timeout );                 /* timeout for event coalescing */
    if ( is_no_error ( err ) &&
        l4_task_equal (sender , irq_thread )) continue;
    if ( is_timeout_error ( err )) return;                /* timer hit , notify client now. */
    if ( is_no_error ( err ) && l4_task_equal(sender , client )){
        client_req .v=dw0;                                /* new client-request. */
} } } }
```

Listing 11: Waiting for data from the *IRQ thread*

The client notification, implemented in *flush_wake_client()* is also modified to additionally resets a potentially running event coalescing timer. Its pseudo-code is given in Listing 12.

```
int Conn::flush_wake_client (l4_threadid_t *client , request_t *request){
  ...
  timeout = 0;                          /* reset potentially running timer */
  tx_committed = 0;                     /* reset committed tx counter */
  len = fill_client_msgbuffer (&msg);   /* create msg-buffer to be sent to client */

  ... continues as the real-time version ...
```

Listing 12: Transmitting data to the client

6.9. Best-effort client network stacks

The DROPS project contains multiple network stacks that use the RT-Net server: L^{4}Linux 2.2, L^{4}Linux 2.6 and Flips.

6.9.1. L^{4}Linux

L^{4}Linux [HHL+97, HHW98, Lac04] is the reference application of DROPS for non–real-time applications. L^{4}Linux is a modified version of the Linux operating system that runs atop the L4

mikrokernel family. This approach allows to execute unmodified Linux programs in parallel to L4 programs, such as real-time programs or a real-time network stack. Linux has been ported to L4 beginning with the Linux 2.0 kernel version. The currently maintained versions of Linux for L4 are L^{4}Linux 2.2 and L^{4}Linux 2.6.

Control flow

To give L^{4}Linux access to the network using the RT-Net server, an appropriate *driver stub* must be integrated into the L^{4}Linux kernel. The implementation in general follows the design rules of network drivers for Linux. Care must be taken on IPC-based communication with the RT-Net server however. When submitting data to a network driver, Linux does not expect to block, and so the L^{4}Linux network driver stub also must not block. As described in Section 6.8.1, the best-effort RT-Net server client interface uses DSI for its send path. Nonblocking communication was one of the main design goals of DSI, and hence it solves the problem of the send path. The notification path in the original Linux kernel is executed in an interrupt context whenever a hardware interrupt occurs. L4-based driver stubs that rely on specific IPC-based driver protocols require another mechanism. Therefore, a *virtual-interrupt* extension has been implemented in L^{4}Linux. The virtual-interrupt mechanism allows a driver stub to actively enter the interrupt context in reaction to an event, instead of being called in an interrupt context.

During the development of RT-Net, I implemented the driver stub for L^{4}Linux 2.2. Based on this, the driver stub for L^{4}Linux 2.6 has been implemented by Adam Lackorzynski.

Data flow

The main focus of L^{4}Linux 2.2 was to provide the Linux API on DROPS. In contrast to recent Linux ports, it makes no use of the data-space concept, a generic abstraction of memory containers [APJ$^+$01, L4E]. The RT-Net server relies on data-spaces for sharing the memory areas on its send path. However, L^{4}Linux 2.2 can be configured to use a pinned, physically contiguous piece of memory as kernel memory. As such, it can easily obtain the physical addresses of data it wants to transmit to the network. As an extension, the RT-Net server allows L^{4}Linux 2.2 to pass these physical addresses to reference its transmit data. The RT-Net server verifies that the referenced memory belongs to L^{4}Linux and passes the physical addresses directly to the NIC. As a result, the send path of the L^{4}Linux 2.2 driver stub requires no copy of the transmitted data. If, however, L^{4}Linux 2.2 is not configured to use a physically contiguous piece of memory as kernel memory, data needs to be copied once on the network send path. The L^{4}Linux 2.6 RT-Net driver stub is an adaption of the version 2.2 stub to the modified L^{4}Linux 2.6 API.

6.9.2. Flips

The *flexible IP stack* (FLIPS) is a server for DROPS that provides an TCP/IP protocol stack. It has been implemented as part of the MikroSina project [HWF05] and is still under development. During the development of RT-Net, I implemented an RT-Net stub for an earlier versions of FLIPS.

It is now maintained by the former MikroSina group members, and an adaption to current FLIPS versions is ongoing work.

6.9.3. Routing between multiple IP-Stacks

A problem related to multiple IP stacks that are executed at one node without knowing each other is the routing of data between them: If one stack sends a packet to the other, it would be up to the NIC to forward the packet to the other stack. However, the real-time architecture of the RT-Net server does not support this routing.

This problem can be addressed by adding an additional network driver stub to each IP stack. This additional stub is connected to a central instance that applies the routing to the other IP stacks.

An alternative solution is to add one additional driver stub to each IP stack per other stack and to connect these stubs pairwise. By adding routing entries for theses additional interfaces and IP-addresses, the routing is entirely based on the IP stacks, and no additional instance is needed to route the traffic thru.

The central router approach has been implemented as part of the MikroSina project [HWF05] at TU Dresden and can be used with the IP stack implementations for DROPS independently of the RT-Net server.

6.10. Outlook: Offloading traffic handling to network cards

The measurements in Section 7 show that the traffic handling at the real-time network stack consumes a substantial amount of CPU cycles. This motivates a technology that has been applied successfully to lower the CPU utilization of network processes – firmware offloading [BBVvE95, DH00, Myr]. The idea is to instruct an intelligent network interface card (NIC) to perform some of the resource critical tasks, disburdening the host CPU.

While the U-Net project [BBVvE95] and the Myrinet GM protocol [Myr] used firmware offloading for performance reasons, Dannowski implemented the policing of incoming network traffic at an ATM NIC [DH00] to bound the CPU utilization in a real-time system. A side-effect of the offloaded policing was an offloaded demultiplexing of received traffic, allowing a real zero-copy receive process. As copying of network data is known to seriously influence the performance of network processing, zero-copy implementations should be favored whenever possible.

Regarding traffic shaping on Switched Ethernet, firmware offloading can be used for early demultiplexing in the receive path and for accelerating the traffic shaping process in the transmit path. Furthermore, an interaction with the NIC driver for normal transmit operations might even be circumvented at all, meaning that no context switch to the driver is needed for sending data.

In [LH04b] Härtig and I analyzed the additional requirements to Ethernet cards allowing to offload the traffic shaping for sending data and to offload the early demultiplexing for receiving data. We found the changes required to currently established Ethernet chips to be moderate, resulting in production costs comparable to those of normal Ethernet cards.

7. Experimental evaluation

This section validates the practical applicability of the theory developed in this dissertation. Three basic measurement subsections and a detailed application measurement subsection analyze different aspects of hard real-time communication on Switched Ethernet:

Section 7.1 first introduces fundamental measurement methods used throughout the experiments of Section 7. Then it provides an analysis to reliably identify characteristics of Ethernet switches to be used for later shaping decisions. The analysis uses a basic version of the RT-Net server that is given the highest priority in the system; thus, it does not suffer from scheduling jitter and effects of other software-components are minimized.

Section 7.2 uses the basic version of the RT-Net driver to find out safe network utilization bounds and to identify bounds for delay guarantees that can be achieved by software-controlled traffic shaping on Switched Ethernet. A fundamental result of this section is a CPU–delay trade-off, where lower delays at the network can be achieved in exchange for a higher CPU usage at the attached nodes.

Section 7.3 analyzes to what extend a Switched Ethernet network can be shared by non–real-time nodes and nodes doing real-time communication.

Section 7.4 provides detailed measurements of a dynamic real-time system, in particular, measurements of the DROPS real-time network stack as described in Section 6 on page 40ff. It aims to guarantee end-to-end delay bounds when the network stacks are scheduled together with other real-time applications. Therefore, Section 7.4 provides the parameter sets for predicting the CPU usage, investigates into performance numbers and CPU usage of different application configurations, and measures the performance of the L^{4}Linux integration of the RT-Net server.

7.1. Network hardware analysis

This section presents basic measurements to find out the delays of switches and their effective queueing buffer capacities. Therefore, it first describes the setup used in the basic measurement subsections and introduces the approach to measure network delays with microsecond resolution.

7.1.1. Measurement setup

Figure 35 depicts the general measurement setup: a single switch was connected to five nodes. Node **A** periodically generated test packets and sent them to node **B**. Nodes **C**, **D** and **E** sent traffic of different characteristics to node **B**. An additional “black cable” connected the nodes **A** and **B** for a precise clock synchronization (detailed in Section 7.1.2). By testing for packet loss and measuring the packet transmission delays from **A** to **B** with different traffic patters and different software configurations at the nodes, results on the hardware and software behavior were obtained.

The measurements analyzed three different 8-port Ethernet switches: a Fast Ethernet Level-One “FSW-2108TX” switch, a Fast Ethernet 3Com “OfficeConnect Dual Speed Switch 8” switch and a Gigabit Ethernet Intel “Netstructure 470F” optical switch.

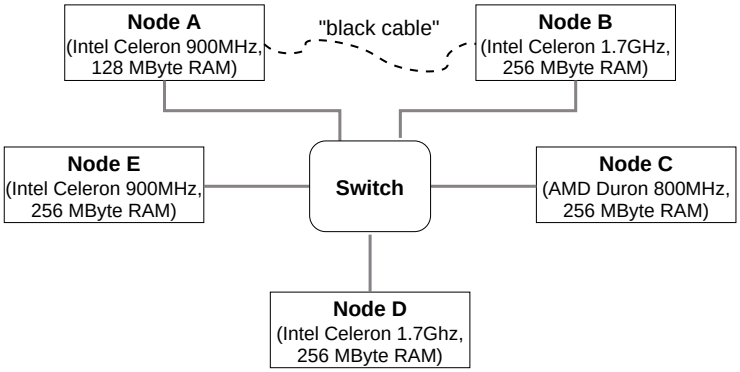

Figure 35: General measurement setup: five nodes are connected to a switch. Nodes **A** and **B** are additionally connected by the "black cable" for precise time synchronization.

Node	CPU type	CPU clock	Memory
A	Intel Celeron	900 MHz	128 MByte
B	Intel Celeron	1.7 GHz	256 MByte
C	AMD Duron	800 MHz	256 MByte
D	Intel Celeron	1.7 GHz	256 MByte
E	Intel Celeron	900 MHz	256 MByte

Table 5: Nodes used for measurements in Section 7.1 to Section 7.3.

Table 5 shows the hardware configuration of the nodes used in Section 7.1 to Section 7.3. For Fast Ethernet measurements, all nodes were equipped with Intel EEPro/100 Fast Ethernet network cards. For Gigabit measurements, all nodes used 3Com 3C985B-SX type optical network cards (AceNIC II).

7.1.2. Measuring inter-node µ-second delays

To measure transmission delays, a send application at node **A** generated test packets carrying timestamps and sequence numbers. At node **B**, a receiving application compared the timestamps with its local clock and calculated the transmission delay (observed transmission delay).

To calculate the delay based on node **A**'s timestamps and node **B**'s time, a clock synchronization mechanism mapped node **A**'s local time to node **B**'s local time. The expected network delays are in the order of microseconds to a few milliseconds, and thus nodes **A** and **B** must have been be synchronized with an accuracy of a few microseconds. Therefore, a parallel cable (the "black cable" in Figure 35) connected **A** and **B**. The synchronization mechanism worked similar to that of NTP as defined in RFC 1305: Using the black cable, node **B** periodically raised an interrupt at **A** to start the synchronization process (Figure 36). In reaction, **A** sent a ready-signal and both nodes

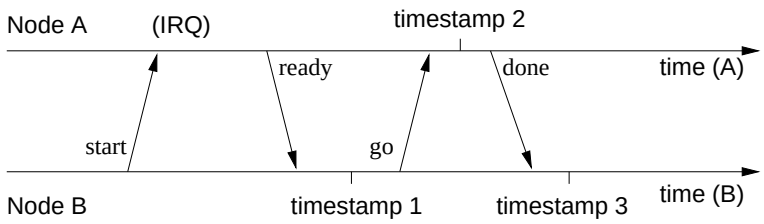

Figure 36: Resynchronization process. The signalling is done using the black cable, time-stamp 2 is sent over the network.

simultaneously took time-stamps.[7] Later, node **A** sent its time-stamp to **B** that used it to calculate the clock difference and the clock drift. After 20 µs the process was aborted, and it started again after some time. In the meantime, old measurement values were used. A detailed description of the synchronization process as well as the derivation of its accuracy can be found in [Loe03b]. In the experiments presented in this dissertation, the achieved clock accuracy between nodes **A** and **B** was better than 10 µs. The resynchronization run not more often than once every second, and less often in most cases. A resynchronization procedure took 10 µs in the average.

In all measurements in Sections 7.1 to 7.3, node **A** generated the test packets the same way. It used the basic version of the RT-Net driver to generate UDP-packets in minimum-sized Ethernet frames (64 bytes including all headers, 22 bytes UDP payload) every millisecond. At node **B** the RT-Net driver dispatched the test packets based on their UDP port and handed them over to the test application. As the test traffic was very regular and had only a small bandwidth, the results of the experiments were mainly influenced by the traffic generated additionally at the other hosts.

7.1.3. Achieving worst-case delays

Oechslin [Oec97] reported the difficulties of reliably reproducing worst-case queueing delays and buffer usage with traffic that is shaped according to a given set of T-SPECs. He found periodic traffic patterns, *symmetric bursts*, that lead to maximum queue lengths with a high probability. Figure 37 shows the general pattern of symmetric bursts. The experiments in Section 7.2 use these symmetric bursts to achieve a worst-case behavior at the network switch.

7.1.4. Switch multiplexing delays

The equations in Section 4.2.4 for calculating switching delay and buffer bounds use the t_{mux} parameter. t_{mux} expresses the time it takes for a switch to start sending a packet after it received it, given the packet is not enqueued. This time was measured by comparing the maximum observed transmission delay of a maximum sized Ethernet frame from nodes **A** to **B**, once directly connected

[7] In detail, **A** signalled *ready* on the black cable; **B** took time-stamp1, signalled; **A** took time-stamp2, signalled; **B** took time-stamp3. The precision of one measurement point is given by (time-stamp3 - time-stamp1).

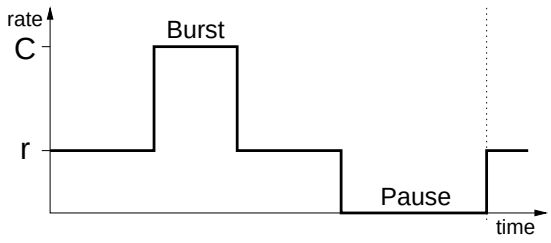

Figure 37: One period of a symmetric burst.

and once connected by a switch. In the latter case, the three other nodes (**C**, **D**, **E**) mutually exchanged traffic to put load on the switching fabric, but prevented queueing in the switch.

Node **B** collected one million samples for each test. As the result, the Fast Ethernet switches add 45 µs to the transmission delay. The Gigabit switch adds 25 µs.

7.1.5. Switch buffer capacities

To use the traffic shaping approach for real-time transfer, the switches must have enough buffer capacity for queueing packets. The experiments in this section analyzed which switches can be used by determining their buffer capacities available for queueing. Based on the general setup of Figure 35, node **A** sent 64-byte test packets to node **C** and nodes **C** and **D** sent two bursty traffic flows to node **B**. Node **E** did not send any data. The flows generated by **C** and **D** had a rate slightly under half of the maximum medium bandwidth (100 MBit/s and 1000 MBit/s) each. They were shaped in an on-off form, thus a burst of an adjustable length b was followed by a pause. The maximum switch backlog required by these two flows is b [Loe03a].

During the experiments, nodes **C** and **D** started to generate flows with small burst lengths and increased the burst lengths until packet loss occurred. Table 6 shows the maximum burst lengths where no packet loss occurred.

Switch	(in 1514 Byte-frames)	(in KByte)
100 MBit, 3Com	14	20.5 KByte
100 MBit, Level-One	87	127.4 KByte
1000 MBit Intel	200	293 KByte

Table 6: Maximum burst lengths without packet loss.

Although the documentation of the 3Com switch states a memory capacity of 256 KByte, it reliably buffers only 20 KByte. Reasons for the low effective buffer capacity are most probably an implemented early-dropping algorithm intended to throttle best-effort connections in high-load situations, fixed buffer memory pools per switch port, a large memory pool reserved for the MAC table, or a combination thereof. Thus, the 3Com switch is ineligible for the traffic-shaping approach and consequently was not used in further experiments.

The documentation of the Intel Gigabit states that it can store up to 2 MByte of data per output port. It turned out that memory bus and PCI bus limitations at node **B** prevented a successful reception of bursts of this size with Gigabit bandwidth.

7.2. Application-to-application effects

This section verifies the theory on the effects of traffic shaping to packet transmission in practice using the basic version of the RT-Net server. The purpose is to obtain fundamental results about delay bounds, utilization bounds and CPU usage that can be achieved with software-based traffic shaping on Switched Ethernet. The basic version of the RT-Net server uses no CPU reservation, but is given the highest priority in the system. The *TX threads* use the token-bucket traffic shaper with data dependency as described in Section 5.7 on page 36. In the following, I will use the term *traffic-shaping interval* to refer to the waiting time of the shaper. The *IRQ threads* and the notifier threads have no strictly periodic scheduling enforced but are granted the CPU whenever they need it and the CPU is available. As such, this setup resembles the best-case from the network point of view: The obtained results represent general bounds for a broader range of scheduling approaches, such as dynamic-priority-based and time-triggered scheduling.

7.2.1. Application-to-application test packet transmission delays

In the first experiment node **B** measured the maximum packet transmission delays of 64-byte test packets sent by node **A** under the condition that the switch had not to queue any packets. As in Section 7.1.4, nodes **C**, **D** and **E** loaded the switching fabric with parallel load. Table 7 shows the maximum observed transmission delays.

Switch	maximum observed transmission delay
100 MBit, Level-One “FSW-2108TX”	80 µs
1000 MBit, AceNIC interrupt coalescing disabled	175 µs
1000 MBit, AceNIC interrupt coalescing enabled	238 µs

Table 7: Maximum application-to-application packet transmission delays with different switches and driver features.

The AceNIC Gigabit Ethernet cards provide a sophisticated interrupt coalescing feature. While this interrupts coalescing reduces the interrupt load on the one hand, it possibly increases the packet reception delay on the other hand. For later comparison, the table contains the values for both configurations.

7.2.2. Fast Ethernet with DROPS

In the next experiment, all nodes were connected to the Fast Ethernet Level-One switch. Nodes **C**, **D** and **E** sent data to node **B**. During the experiment, they varied the period of the *TX thread*, but

kept the bandwidth reservation constant. This resulted in different bucket sizes and, consequently, in different burstiness parameters of the generated flows. The bucket size of each node was calculated as $b = r \cdot T_i + M$ with r being the reserved bandwidth of that node and M the length of a maximum-sized Ethernet frame, which is 1514 Bytes. Each node only sent one flow, and thus the bucket size did not increase due to NIC multiplexing at the send nodes. Table 8 lists the reserved gross bandwidths and bucket sizes.

Node	**C** (40 MBit/s)	**D** (32 MBit/s)	**E** (20 MBit/s)
T_i=10 ms	51514 bytes	41514 bytes	26514 bytes
T_i=1 ms	6515 bytes	5514 bytes	4014 bytes
T_i=100 µs	2014 bytes	1914 bytes	1764 bytes

Table 8: Bucket sizes depending on the traffic shaping interval T_i.

Buffer bounds and worst-case delays Table 9 shows the resulting buffer bounds and delays of the three configurations. The buffer bound is calculated from Equation 13 on page 23. t_{max} is calculated from Equation 15, increased by the 80 µs from Table 7. t_{est} results from the delay estimations given in Equation 16. t_{obs} is the maximum transmission delay node **B** has observed in the experiments. In each experiment **B** collected 350,000 samples. No packets were lost.

	buffer bound	t_{max}	t_{est}	$t_{obs} \leq$
T_i=10 ms	111.8 KByte	9357 µs	9731 µs	8759 µs
T_i=1 ms	15.7 KByte	1380 µs	1345 µs	1300 µs
T_i=100 µs	6.1 KByte	582 µs	506 µs	438 µs

Table 9: Buffer bounds in the switch and delay bounds for packet transmission from node **A** to node **B** depending on the traffic shaping interval T_i.

The observed transmission delays are actually smaller than the theoretical bounds. This can be explained by the observation that, even with symmetric bursts, the maximum queue length is only achieved in extremely rare situations at the switch. These situations just did not happen in the experiments.

CPU usage To measure the CPU requirement of traffic shaping, the experiments were repeated with modified send applications: Instead of generating symmetric bursts, they generate the traffic as fast as possible. The symmetric burst generation requires fine and therefore expensive timers that would have affected the CPU measurements.

The CPU usage is the ratio of how many CPU cycles are consumed by the system to the CPU cycles available during a time interval. It was measured by a low-priority looper that consumed and counted all idle CPU cycles. Table 10 shows the CPU usage at nodes **C**, **D** and **E** with the modified send applications.

Node	C (40MBit/s)	D (32MBit/s)	E (20MBit/s)
T_i=10 ms	4.1%	2.9%	2.3%
T_i=1 ms	11%	9%	7.2%
T_i=100 µs	21.2%	17.2%	11.9%

Table 10: CPU load depending on the traffic shaping interval.

The delay–CPU trade-off is demonstrated in Figure 38. It clearly shows the influence of the decreased shaping intervals to the CPU usage. Thus, there is another trade-off between traffic shaping accuracy, and hence transmission delay bounds, and CPU usage in the nodes connected to the network.

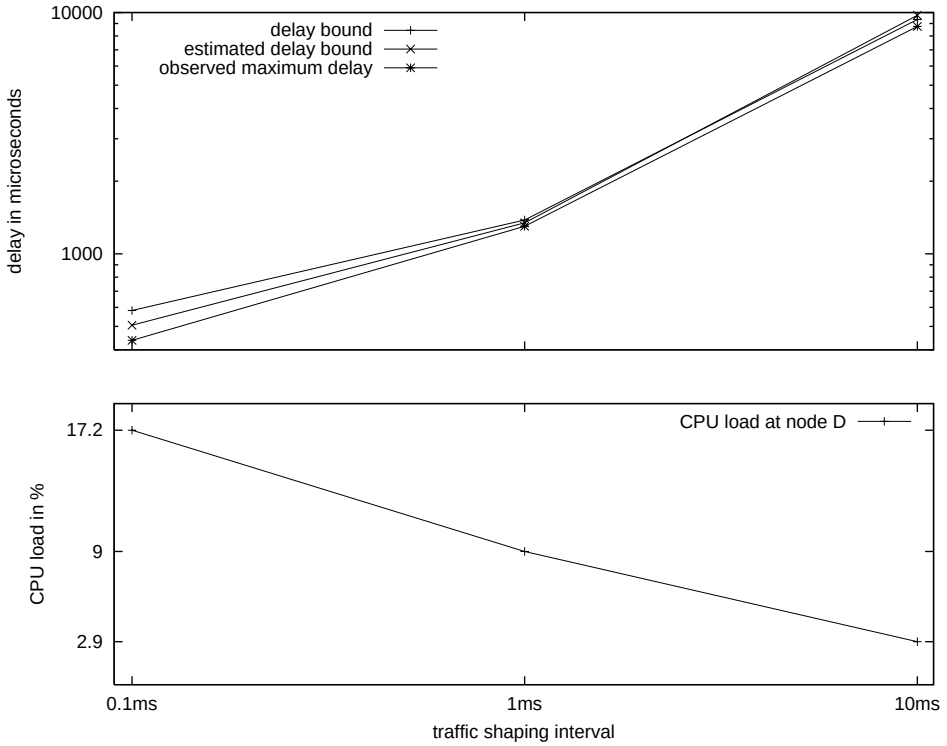

Figure 38: Delay–CPU trade-off with different traffic shaping intervals. The depicted CPU load is obtained from node **D**.

Interpretation of results With maximum sized frames of 1514 Bytes on Fast Ethernet the achievable bandwidth is limited to 98.6 MBit/s due to framing overhead and inter-packet gaps (corresponding to 8 + 12.5 Bytes). Nodes **A**, **C**, **D** and **E** actually sent slightly over 92 MBit/s to node **B**, thus utilized its link to 93%. With this utilization, delay bounds of 9.4 ms, 1.4 ms and 0.582 µs, can be guaranteed, depending on the amount of CPU cycles one is willing to spend.

Equation 16 derived in Section 4.2.4 on page 22 gives an estimation for the switch queueing delay that can be calculated easily. Table 9 shows that the error by this delay estimation was less than 16% in the experiments of this section.

With a traffic shaping interval of 10 ms, nearly all the buffer capacity of the switch is needed for a single output port. In another experiment, nodes **C** and **E** tried to send two additional 30 MBit-flows to node **D**, which immediately resulted in lost packets. With a traffic shaping interval of 1 ms no packet loss occurred.

7.2.3. Gigabit Ethernet with DROPS

Although Gigabit Ethernet is similar to Fast Ethernet at the hardware level, the increased bandwidth is a challenge for the communicating nodes: The bandwidth of Gigabit Ethernet is higher than a standard PCI bus can transfer in practice. As the maximum packet size of Gigabit Ethernet is the same as that of Fast Ethernet, the amount of CPU required to handle the higher bandwidth increases as well. This section presents experiments to analyze to what extend the software-based traffic shaping approach can benefit from Gigabit Ethernet.

The experimental setup was similar to that Section 7.2.2. The network switch was replaced by the Intel Netstructure optical Switch, and the nodes used the AceNIC network cards instead of the Intel EEPro/100.

In the first experiment, each one of the nodes **C**, **D** and **E** sent traffic with 160 MBit/s to node **B**. The nodes used a traffic shaping interval of T_i=1 ms. The interrupt coalescing feature of the AceNIC cards was enabled. Table 11 shows the bucket sizes and the CPU load at the sending nodes. According to Equation 13, 64 KByte of switch buffer were needed. The delay bound of this configuration was expected to be **687 µs**: the switch delay according to Equation 15 plus the 238 µs maximum transmission delay at the non–queued switch from Table 7. Node **B** actually observed a maximum packet transmission delay of **906 µs** with no packet loss.

Node	**C**	**D**	**E**
bandwidth	160 MBit/s	160 MBit/s	160 MBit/s
bucket size	21514 bytes	21514 bytes	21514 bytes
CPU load	48 %	30 %	39 %

Table 11: Bucket sizes and CPU load for the Gigabit Ethernet experiment with a traffic shaping interval T_i=1 ms.

In a second experiment, nodes **C**, **D** and **E** used a traffic shaping interval of 100 µs. To measure the expected small transmission delays with a better accuracy, the nodes disabled the interrupt coalescing features of the AceNIC network cards. This resulted in lower delays on packet transmission and reception, but it increased the interrupt load at all nodes. To prevent packet loss, nodes **C**, **D** and **E** sent with a reduced bandwidth of 80 MBit/s each. Nonetheless, the CPU load increased significantly. The bucket size in all nodes was selected to 2114 Bytes. The resulting theoretical delay bound was **247 µs** (72 µs switch delay and 175 µs according to Table 7). Node **B** observed a maximum packet transmission delay of **341 µs** with no packet loss.

Node	**C** (40M Bit/s)	**D** (32 MBit/s)	**E** (20M Bit/s)	$t_{max} \leq$	$t_{obs} \leq$
Linux 2.4.22, HTB, 10 ms	54022 bytes	43536 bytes	27810 bytes	9807 µs	6928 µs
Linux 2.6.0-test9, TBF, 1 ms	6450 bytes	5495 bytes	4000 bytes	1372 µs	995 µs

Table 12: Bucket sizes (in bytes), delay bound and observed delays for the Linux experiments.

Interpretation of results The observed packet delays are longer than the expected delays. Additional experiments revealed that the receiving node **B** was just overloaded and it deferred the packet delivery to the test application.

This shows a general problem at the receiver: the demultiplexer has to spend CPU cycles for demultiplexing each packet, sometimes just to find out that no application is waiting for them. A solution to this was already given by Dannowski [DH00]. He applied early demultiplexing at the firmware level of an ATM card, and successfully removed load from the CPU.

With a maximum delay of 906 µs the switch output connected to node **B** can be utilized to 49%. The limitation are the attached nodes, not the network switch. The buffer requirement of the one output port would allow for higher bandwidths on more output ports.

This result compares favorable to time-slotted approaches, which are very sensitive to the network jitter and delays inherent to switches. Schwarz reports in [Sch02] about an implementation of the time triggered TTP/C protocol on Gigabit Ethernet. The delay and jitter of the network he analyzed result in a maximum overall utilization of 37 MBit/s, corresponding to a 3.7% utilization.

7.3. Sharing a network with non–real-time nodes

This section presents experiments to find out to what extend the network can be shared between non–real-time nodes and nodes doing real-time communication. Of course, the non–real-time nodes must obey some sort of traffic shaping, otherwise they could easily flood the buffers in the switch. Beginning with kernel version 2.4 Linux includes an QoS subsystem and provides a number of queueing disciplines for this [LxQ]. Among them are a token-bucket traffic shaper (TBF) and a hierarchical token-bucket traffic shaper (HTB).

The experiments were executed on Fast Ethernet hardware with two Linux versions: the Linux-2.4.22 kernel and the Linux-2.6.0-test9 kernel. For traffic shaping both Linux kernel versions use the periodic clock interrupt. Linux-2.4 kernels on the x86 architecture generate the clock interrupt with a frequency of 100 Hz. With Linux-2.6 the frequency is 1kHz. Thus with Linux-2.4.22 the expected shaping interval is 10 ms resulting in switching delays in the order of 10 ms. With Linux-2.6.0-test9 one can expect a 1 ms traffic shaping interval and delays in the order of 1 ms.

Linux-2.4.22 with HTB traffic shaper Instead of DROPS, Nodes **C**, **D** and **E** executed the Linux-2.4.22 kernel and used its hierarchical token-bucket traffic shaper *HTB* [LxH]. The HTB is often used in conjunction with DSL- and cable modems to minimize queueing delays inside the modems. The experimental analysis of the HTB from the standard kernel however revealed that

it sometimes shapes the traffic in intervals of 20 ms, not in intervals of 10 ms. After contacting the HTB author and tuning the kernel[8], it finally shaped the traffic periodically with an interval of 10 ms, for both kernel versions 2.4 and 2.6.

Nodes **C**, **D** and **E** configured their HTB with the same bandwidths as in the previous experiments: 40 MBit/s, 32 MBit/s and 20 MBit/s. HTB determined the token buffer sizes for itself, Table 12 shows the resulting values. The table also contains the transmission delay bound according to Equation 15, increased by the 80 µs from Table 7 for packet transmission delay without queueing.

For reasons I was unable to find out, the generation of symmetric bursts as with the DROPS setup did not work – the HTB traffic shaper sooner or later delayed the traffic for 10 ms preventing any useful results. This should not happen, as the traffic was generated conforming to the reservation. Thus, the nodes randomly generated bursts with lengths according to the bucket size and breaks in between.

During the experiment node **A** sent 3 million test packets to node **B**. No packets were lost. The maximum observed packet transmission delay for the test packets was 7 ms (last column in Table 12). The bigger difference to the theoretical bound compared to the DROPS experiments can be explained by the on-off shape of the bursts used.

Linux 2.6.0-test9 with TBF traffic shaper In the second experiment nodes **C**, **D** and **E** executed the Linux-2.6.0-test9 with its 1kHz timer. The traffic was shaped by the token-bucket traffic shaper TBF. The bandwidth configured at the TBF was the same as in the previous experiments: 40 MBit/s, 32 MBit/s and 20 MBit/s. In contrast to HTB the TBF traffic shaper requires the user to additionally specify the bucket sizes. By experiments the lowest bucket sizes were found that achieve the desired bandwidths. Table 12 shows the obtained numbers. As with the Linux 2.4 experiment, the client applications did not send symmetric bursts but generated bursts randomly.

During the experiment node **A** sent half a million test packets to node **B**. Again, node **B** observed no packet loss, and the maximum packet transmission delay was slightly under 1 ms.

Robustness of a shared network I repeated both experiments and tried to distort the traffic shaping process so that it generate a higher network load than allowed. The idea was to use high interrupt load to force the system into a state where the shaper cannot send packets for a while. As a potential consequence, the shaper might catch up this lag later on and might generate larger bursts. The experiments however showed that the traffic shaper cannot be influenced to generate longer bursts or higher traffic than expected.

Interpretation of results The experiments showed that Linux nodes can share a network with real-time nodes, although the different Linux kernel versions lead to different transmission delays and switch buffer requirements. With the 2.4 series Linux, the 127 KByte buffer need in the switch do not allow for more than one fully utilized switch output port. For the 2.6 series Linux with its 1 kHz timer, there is no such limitation.

[8] changing defines: net/sched/sch_htb.c: HTB_HYSTERESIS=0, and include/net/pkt_sched.h: PSCHED_CLOCK_SOURCE=PSCHED_CPU

7.4. Dynamic real-time system performance

This section provides a detailed analysis of the DROPS real-time network stack as described in Section 6 on page 40ff. Therefore, an approach for detailed CPU usage measurement on DROPS is introduced first. Then, elaborated measurements are used to derive the parameter sets needed for the CPU usage prediction in Section 6. A complete application scenario shows the achievable end-to-end of the complete DROPS real-time network stack under various configurations. Finally, this section provides performance numbers of the L^{4}Linux integration of the RT-Net server.

7.4.1. Measuring CPU costs

Fiasco and DROPS provide a couple of mechanisms to obtain information about the per-thread CPU usage. The most straight-forward approach is to sample the consumed time of a thread using the *l4_thread_schedule()* system call. *l4_thread_schedule()* returns the accumulated time that has been consumed by a thread so far. However, to obtain the worst-case per-period CPU consumption of a real-time thread, this mechanism is not sufficient.

Fiasco's CPU usage reporting extension

Other established approaches on DROPS either require to modify the control flow of an application or to add an additional system call per period of a real-time thread. For sole measurement purposes, a transparent mechanism is preferable. An unpublished extension of Fiasco provides such a mechanism: Each thread can be assigned a *user thread control block* (UTCB) that holds a ring-buffer for scheduling notification events. Each time a thread switches to another timeslice or to another period, Fiasco writes a new entry in the ring-buffer. The entries contain information about the current timeslice and the CPU time consumed by this timeslice.

To obtain the per-period CPU usage of a real-time thread, a sample thread periodically traverses the ring-buffer in the UTCB of that real-time thread. As the ring-buffer can hold multiple entries, the sample thread does not need to traverse it in every period of the real-time thread, but instead can be activated in longer time intervals. Especially, the sample thread can be executed without a CPU reservation in many relevant measurement scenarios, not influencing the CPU scheduling of real-time threads. As such, the UTCB extension provides an efficient, nonintrusive CPU usage measurement mechanism. The CPU usage numbers presented in the following sections were all measured with the Fiasco UTCB extension.

7.4.2. Measurement setup

The measurements presented in Section 7.4 used a similar setup as the experiments in the previous sections: Five nodes were connected to the Level-One Fast Ethernet Switch. Figure 39 depicts the installation and shows the individual hardware configuration of the nodes. For the delay measurements in Section 7.4.6, nodes **F** and **J** were connected by a parallel cable for synchronization purposes.

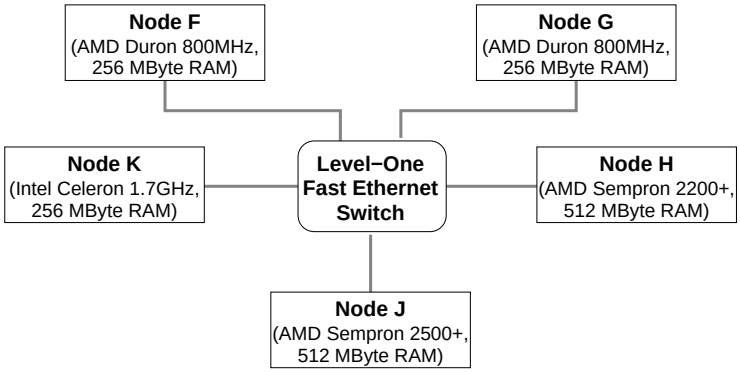

Figure 39: Setup for Section 7.4: five nodes (**F**, **G**, **H**, **J** and **K**) are connected to a Fast Ethernet switch.

In contrast to the previous measurements, each node executed the complete DROPS real-time system and the real-time network stack as described in Section 6 on page 40. The measurements were executed by dedicated measurement applications, which are described in detail in the following sections. In addition to the real-time network stack and the test applications, the nodes executed the DoPE window manager to visualize the measured data and a cache flooder to achieve a worst-case behavior of the network stack. The cache flooder continuously accessed the memory to flush the TLB, L1 instruction cache (trace-cache), L1 data cache and the L2 unified cache.

7.4.3. Transmit CPU costs

Equations 34 and 35 on page 47 define a model for the per-period CPU usage of the transmit thread of a real-time connection. The equations depend on the following node-specific and NIC-specific parameters:

cpu.tx_base – a base CPU usage that has to be paid for every periodic invocation. It is mainly determined by context switches and TLB and cache misses for the code pages and the data pages.

cpu.tx_packet – a cost to be paid for every sent packet. This mainly covers the access to the NIC. Sending a packet to the NIC involves putting a descriptor into the send ring of the NIC and trigger a new send operation at the NIC.

This section refines the CPU usage model by presenting measurements and determines parameter sets for predicting the CPU usage, depending on the host hardware at the nodes. To obtain the CPU usage parameters, one node after the other executed a client application that sent data over a real-time connection. In these experiments, the clients send their data using the blocking client model, the *TX threads* used the token-bucket shaper and the interrupt coalescing of the *IRQ thread* was enabled. A measurement instance at the nodes obtained

i) the per-period CPU usage of the according *TX thread*s, together with the number of packets the *TX threads* transmitted per period

ii) the per-period CPU usage of the *IRQ thread*, together with the number of packets the *IRQ thread* acknowledged per period
iii) the per-period CPU usage of the according notifier threads, together with the number of packets the notifier threads acknowledged per period and the number of client-request per period
iv) the per-period CPU usage of the client threads

The applications executed multiple measurement series, with each series having a fixed set of connection parameters. For each measurement series, the applications systematically varied one of the following parameters:

a) the transmission bandwidth of the connections,
b) the period length of the *TX threads* at the RT-Net server, and
c) the packet sizes the applications send data with

The applications selected the connection parameters as follows: 20 different bandwidths ranging from 8 MBit/s to 89 MBit/s per connection, three different *TX thread* periods ranging from 1 ms to 5 ms, 5 different packet sizes ranging from 300 byte UDP payload to 1472 byte UDP payload. Each individual measurement series consisted of at least 100,000 individual measurements. From the potentially 300 different parameter combinations, each of the five nodes executed at least 20 measurement series to obtain its CPU usage parameter set. In the following, I will discuss selected measurements in detail and will summarize the obtained results then.

Figure 40 shows the per-period CPU usage of the *TX threads* at node **J** for a packet size of 399 bytes. The graphs are linear in both the number of bandwidth and in the number of packets, with minor measurement deviations.

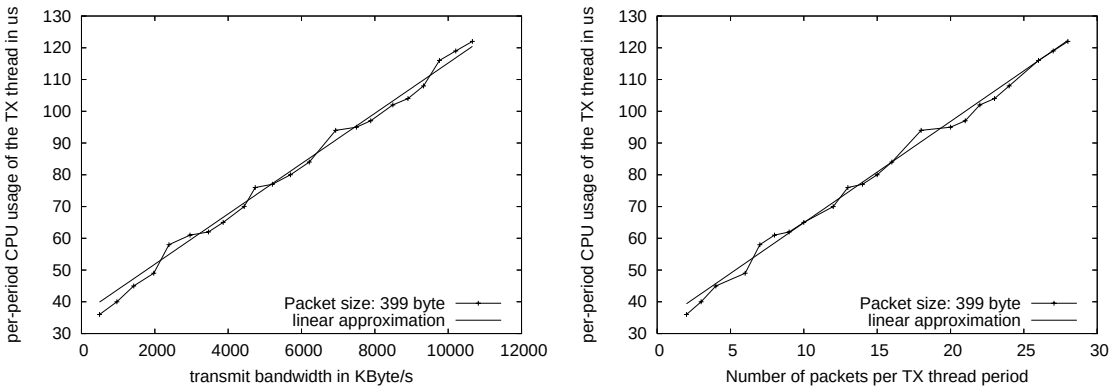

Figure 40: Per-period CPU usage of the *TX threads* at node **J** depending on the bandwidth (left figure) and the number of packets the *TX thread* sends to the NIC per period (right figure). The packet size was 399 bytes.

Figure 41 depicts the results of a more elaborate setup aimed to find out whether the *TX thread* CPU usage is more bandwidth- or packet-count dependent. In three experiment series the *TX*

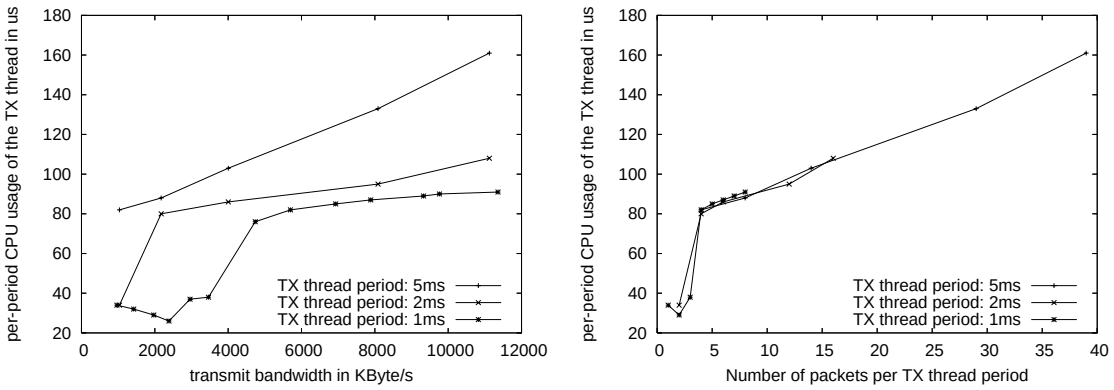

Figure 41: Per-period CPU usage of the *TX threads* at node **J** depending on the bandwidth (left figure) and the number of packets the *TX thread* sends to the NIC per period (right figure). The packet size was 1472 bytes.

threads were given different period lengths, and the packet size was set to 1472 bytes. Clearly, the right sub-figure demonstrates the dependency of the CPU usage on the number of packets instead of the bandwidth. Ignoring the steep gradient to the left of the graphs for the moment, the per-period CPU usage of the *TX threads* can be upper-bounded by a function that is linear in the number of packets to be transmitted per *TX thread* period.

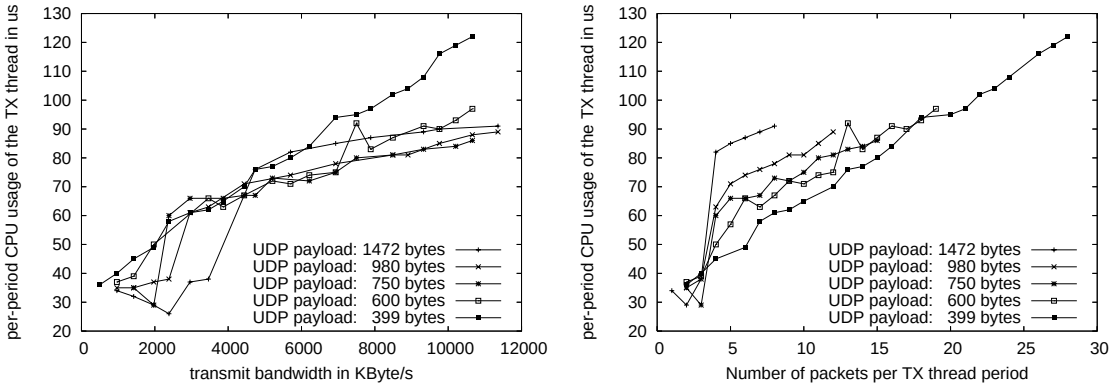

Figure 42: Per-period CPU usage of the *TX threads* at node **J** depending on the bandwidth (left figure) and the number of packets the *TX thread* sends to the NIC per period (right figure). The period length of the *TX threads* was 1 ms.

Figure 42 depicts the results of measurement series with varying packet sizes but constant *TX thread* periods of 1 ms. The figure shows that the *TX threads* consume less CPU with smaller packet sizes. However, the graphs of the right sub-figure approach each other for higher number of packets. Detailed measurements of the *TX thread*'s code-path revealed a phenomenon of the

used Intel EEPro100 NICs: For reasons most likely related to internal resource shortage of the NICs, the hardware access to enqueue a packet in a series of successively transmitted packets sometimes took significantly more time than the transmission of the other packets. For packet sizes of 1472 bytes, it was always the 4th packet. For smaller packets, the access times increased beginning with the 4th packet and approached that of the 1472 byte-packets later. The other nodes (**F**, **G**, **H** and **K**) showed a similar behavior. The gradient, however, is less significant for the slower nodes, and the actual CPU usage more resembles the linear model. As such, the worst-case per-period CPU usage of the *TX threads* indeed can be described by a function that is linear in the number of packets to be transmitted per *TX thread* period. Table 13 gives the resulting parameters for nodes **F** to **K** suitable for Equations 34 and 35 of Section 6.3.3 on page 47.

Node	IP	CPU Model	cpu.tx_base	cpu.tx_packet
F	100	AMD Athlon 800 Mhz	68 µs	3.4 µs
G	21	AMD Athlon 800 Mhz	68 µs	3.4 µs
H	53	AMD Sempron 2200+	77 µs	2.2 µs
J	118	AMD Sempron 2500+	73 µs	2.3 µs
K	21	Intel P4 Celeron 1700MHz	60 µs	3.4 µs

Table 13: CPU usage parameter set for the *TX threads* when transmitting data

In the experiments, nodes **F** and **G** showed an unexpected behavior: The NIC / memory bus / PCI bus infrastructure of **F** and **G** was incapable of transmitting the send data with a sufficiently high bandwidth from the main memory while the cache flooder at the host CPU was saturating the memory bus. Both nodes could not send data with bandwidths higher than 8000 KByte/s. Further, the NICs took up to 5 ms to transmit a single full-size packet after it had been enqueued into the empty hardware TX ring[9]. When the cache-flooder was disabled, no such delays and limitations could be observed. As such, the memory bus of both nodes is considered ineligible to support the DROPS real-time network stack for higher bandwidths.

The experiments were repeated with the strictly periodic shaper with data dependency. Due to the increased number of *TX thread* activations, the overall CPU consumption was significantly increased. Nonetheless, it was consistent with the parameters of Table 13, once the changed number of *TX thread* invocations were taken into account.

IRQ thread CPU costs Figure 43 shows the *IRQ thread* CPU usage for node **J** depending on the bandwidth and the number of packets the thread had to process per period in the previous experiment. Again, the linear model depending on the number of processed packets is more appropriate to predict the CPU usage than a model depending on the bandwidth. Table 14 shows the derived values to predict the *IRQ thread* CPU usage when transmitting data according to Equation 43 on page 54 and Equation 47 on page 55. Note that the table *cpu.irq_tx_base* to denote the constant (base) part of an IRQ activation. As Section 7.4.4 and Section 7.4.5 will show, that base part depends on whether data was only sent, whether data was only received or if data was both sent and received. Section 7.4.5 will refine the CPU usage model then.

[9]The bare medium-transmission time is about 123 µs.

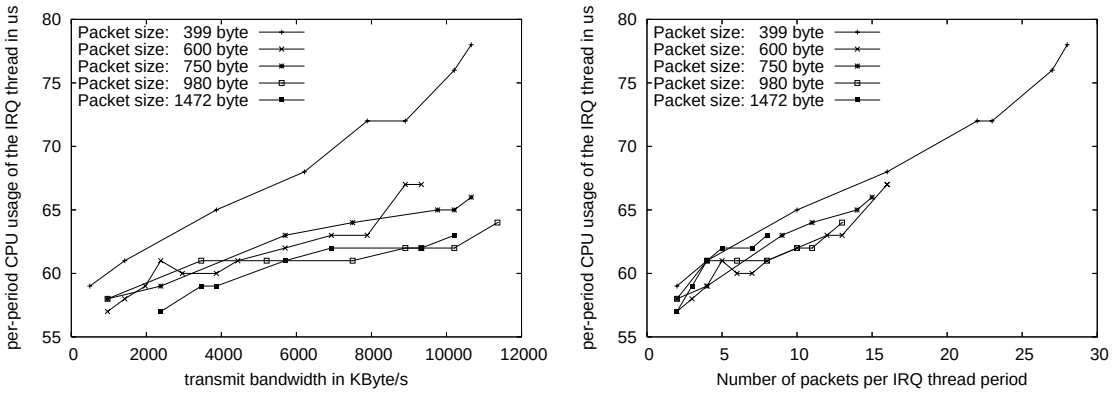

Figure 43: Per-period CPU usage of the *IRQ thread* at node **J** depending on the bandwidth (left figure) and the number of packets the *IRQ thread* processed per period (right figure) for different packet sizes.

Node	CPU Model	cpu.irq_tx_base	cpu.irq_tx
F	AMD Athlon 800 Mhz	90 µs	3.1 µs
G	AMD Athlon 800 Mhz	90 µs	3.1 µs
H	AMD Sempron 2200+	59 µs	0.8 µs
J	AMD Sempron 2500+	59 µs	0.7 µs
K	Intel P4 Celeron 1700MHz	73 µs	1.0 µs

Table 14: CPU usage paramete set for the *IRQ thread* when transmitting data

The experiments were repeated with interrupt coalescing disabled. As expected, the overall CPU usage of the *IRQ thread* increased due to the increased number of *IRQ thread* invocations. However, taking the increased number of invocations and the decreased number of sent notifications into account, the CPU usage was consistent with the obtained CPU usage parameter set.

Notifier thread CPU costs The experiments also confirmed, that the CPU usage of the notifier thread solely depended on the number of client wake-ups. As expected, it was especially independent on the number of packets that were signalled to the client per wake-up. Table 15 gives the according numbers[10].

Client thread CPU costs The CPU costs of the client application were measured too. In addition to the client threads CPU usage, the CPU cycles needed for filling the test-packets with data were measured and subtracted from the thread CPU usage.

However, the main purpose of the clients was to drive the RT-Net server and to gather and display the results. Naturally, the time needed to generate the data and submit it to the network stack

[10]The average CPU usage was between 30% and 60% of the given worst-case numbers.

Node	CPU Model	cpu.notify_base
F	AMD Athlon 800 Mhz	44 µs
G	AMD Athlon 800 Mhz	44 µs
H	AMD Sempron 2200+	28 µs
J	AMD Sempron 2500+	24 µs
K	Intel P4 Celeron 1700MHz	39 µs

Table 15: Per-client wakeup CPU usage for the notifier threads when sending data.

greatly depends on the actual application: It differs with the algorithms used to generate the data and their mutual interference to the numerous caches on modern host architectures. As such, the obtained worst-case numbers merely give a hint to the CPU usage of other client applications rather than a sound base for CPU usage prediction.

Table 16 presents the obtained results: *cpu.client_tx_packet* denotes the CPU time needed to generate a packet and submit it to the RT-Net server. *cpu.client_tx_wait* denotes the client threads CPU time consumed when waiting for free entries in the transmit ring buffer.

Node	CPU Model	cpu.client_tx_packet	cpu.client_tx_wait
F	AMD Athlon 800 Mhz	1.4 µs	46 µs
G	AMD Athlon 800 Mhz	1.4 µs	46 µs
H	AMD Sempron 2200+	1.0 µs	19 µs
J	AMD Sempron 2500+	0.6 µs	18 µs
K	Intel P4 Celeron 1700MHz	1.1 µs	23 µs

Table 16: Client thread CPU usage parameter set for data transmission.

7.4.4. Receive CPU costs

This section obtains the CPU usage parameters for the data reception path. In experiments, one node after the other executed a client application that received data on real-time connections. In most of the experiments, the data was generated by node **J**. For the measurement of the receive costs at node **J**, the data was generated by node **K**. For communication with the RT-Net server, the receiving clients used the blocking client model, and the interrupt coalescing of the *IRQ threads* was enabled. A measurement instance at the nodes obtained

i) the per-period CPU usage of the *IRQ thread*, together with the number of packets the *IRQ thread* received per period
ii) the per-period CPU usage of the according notifier threads, together with the number of packets the notifier threads copied per period and the number of client-request per period
iii) the per-period CPU usage of the client threads

As with the transmit path experiments, the nodes executed multiple measurement series, with varying connection parameters and varying incoming traffic characteristics. In detail, the following parameters were systematically modified:

a) the bandwidth offered to the nodes,
b) the size of the packet sent to the nodes, and
c) the period length of the *IRQ threads* at the RT-Net server.

Again, the memory infrastructure of nodes **F** and **G** turned out to be a bottleneck. Both nodes were incapable of transmitting the received data with a sufficiently high bandwidth to the main memory while the cache flooder at the host CPU was saturating the memory bus. Both nodes showed a substantial packet loss at moderate bandwidths[11]. Consequently, the following discussion only considers the results for nodes **H**, **J** and **K**.

IRQ thread CPU costs The CPU costs for the *IRQ thread*s when receiving data behave similar to the CPU costs when transmitting data. They can be upper-bounded by a function linear in the number of packets. Table 17 gives the parameter sets suitable for Equation 43 on page 54 and Equation 47 on page 55. As described in the previous section, the base CPU part of an *IRQ thread* depends on whether data was sent or received. As such, the table uses *cpu.irq_rx_base* to denote the base CPU part when receiving data.

Node	CPU Model	cpu.irq_rx_base	cpu.irq_rx_packet
H	AMD Sempron 2200+	65 µs	1.7 µs
J	AMD Sempron 2500+	62 µs	1.4 µs
K	Intel P4 Celeron 1700MHz	80 µs	2.3 µs

Table 17: CPU usage parameter set for the *IRQ thread* when receiving data

Again, the experiments were repeated with interrupt coalescing disabled. The overall CPU usage of the *IRQ thread* increased due to the increased number of *IRQ thread* invocations. However, taking the changed number of *IRQ thread* invocations and the changed number of received packets into account, the results were consistent with the obtained CPU usage parameter set.

Notifier thread CPU costs In contrast to the *IRQ thread*, the notifier threads access the received data to transmit it to the client applications. As such, the CPU costs for the notifier threads are expected to depend both on the number of packets and the amount of data to be transmitted per thread period. Table 18 shows the CPU usage parameter set suitable for Equation 55 on page 62, Equation 57 on page 63, Equation 75 on page 69 and Equation 82 on page 70.

Node	CPU Model	cpu.notify_base	cpu.notify_packet	cpu.notify_byte
H	AMD Sempron 2200+	28 µs	1.5 µs	4.2 ns
J	AMD Sempron 2500+	24 µs	1.2 µs	4.1 ns
K	Intel P4 Celeron 1700MHz	39 µs	2.1 µs	4.6 ns

Table 18: CPU usage parameter set for notify threads when receiving data

[11] Packet loss occurred for bandwidths over 3 MByte/s.

Additional experiments with the clients operating in polling mode showed no difference in the CPU costs of the notifier threads once the changed number of client notifications and signalled packets were taken into account.

Client thread CPU costs When data was received by the client applications, they only accessed the first bytes of the data packets to check for lost packets. As such, the CPU costs for the client applications should depend on the number of packets they received rather than the packet size. This expectation was verified by the results obtained at nodes **H**, **J** and **K**: the client thread CPU usage for receiving data was nearly linear in the number of packets received, and did not depend on the packet size. Table 19 shows the resulting numbers. The actual CPU reservation *cpu_client* to be done can be calculated as

$$cpu_client = cpu.client_base + n \cdot cpu.client_packet$$

with *n* denoting the expected number of packets per client thread period.

Node	CPU Model	cpu.client_base	cpu.client_packet
H	AMD Sempron 2200+	37 µs	1.1 µs
J	AMD Sempron 2500+	31 µs	1.0 µs
K	Intel P4 Celeron 1700MHz	44 µs	2.1 µs

Table 19: CPU usage parameter set for the *client threads* when receiving data

7.4.5. Costs for multiple connections

Using an experimental setup similar to that of the previous section, the actual CPU usage parameter sets applicable to the theory in Section 6.3 on page 44 and Section 6.4 on page 47 were obtained: Instead of just one connection, the nodes now sent and received data on multiple connections. Therefore, the experiments were repeated (1) with multiple send connections, (2) with multiple receive connections, and (3) a combination of multiple send and receive connections. The number of send and receive connections was between one and three.

In the experiments, the token-bucket traffic shaper was used, the *IRQ threads* enabled the interrupt coalescing, and the notification path used the blocking client mode.

For sole **data transmission**, the number of connections had no measurable influence on the CPU usage: The worst-case CPU usage of the client threads, the *TX threads* and the notifier threads of the individual connections was the same as that measured for a single connection with the same parameters (bandwidth, packet size, period length). The *IRQ thread*'s CPU usage was consistent with the accumulated number of packets of the individual connections. The wakeup of the notifier threads by the *IRQ thread* had no measurable influence to its worst-case CPU usage.

For sole **data reception**, the same behavior could be observed: The worst-case CPU usage of the client threads and the notifier threads corresponded to the setup with just one receive connection of the appropriate parameters. The *IRQ thread*'s CPU usage was consistent with the accumulated

number of packets of the individual connections. As with the data transmission, the wakeup of the notifier threads by the *IRQ thread* had no measurable influence to its worst-case CPU usage.

For a **combined transmission and reception**, the *IRQ thread*'s CPU usage was higher than the CPU usage for transmission and reception alone, but lower than the sum of both. The observed CPU usage can be expressed by the following formula:

$$\begin{aligned} CPU_{IRQ} = cpu.irq_base &+ cpu.irq_base_tx + cpu.irq_tx \cdot \overline{tx}_{IRQ} \\ &+ cpu.irq_base_rx + cpu.irq_rx \cdot \overline{rx}_{IRQ} \end{aligned} \tag{83}$$

cpu.irq_base covers the base cost for activating the *IRQ thread*, and results from the CPU cycles needed to execute the common code-path, and to handle the most TLB and cache misses.

cpu.irq_base_tx represents the additional constant costs to execute the transmit-specific path, which are mostly determined by the additional TLB and cache misses on that path. The *irq_base_tx* values in Table 14 correspond to $cpu.irq_base + cpu.irq_base_tx$ of Equation 83.

cpu.irq_base_tx represents those constant costs for the receive path. The *irq_base_rx* values in Table 17 correspond to $cpu.irq_base + cpu.irq_base_rx$ of Equation 83.

As such, the CPU usage calculation of Equation 43 on page 54 and Equation 47 on page 55 must be adapted as follows:

$$\begin{aligned} CPU_{IRQ} &= (cpu.irq_base + cpu.irq_base_tx + cpu.irq_tx) \cdot \overline{tx}_{IRQ} + \\ &\quad (cpu.irq_base + cpu.irq_base_rx + cpu.irq_rx) \cdot \overline{rx}_{IRQ} \end{aligned} \tag{84}$$

$$\begin{aligned} CPU_{IRQ}^{\langle IRQ \rangle} = cpu.irq_base &+ cpu.irq_base_tx \cdot \mathbf{1}_{cpu.irq_tx} + cpu.irq_tx \cdot \overline{tx}_{IRQ} \\ &+ cpu.irq_base_rx \cdot \mathbf{1}_{cpu.irq_tx} + cpu.irq_rx \cdot \overline{rx}_{IRQ} \end{aligned} \tag{85}$$

with $\mathbf{1}_x$ defined to be 1 for $x \neq 0$ and 0 for $x = 0$.

Table 20 shows the derived values for *cpu.irq_base*, *cpu.irq_base_tx* and *cpu.irq_base_rx* for nodes **H**, **J** and **K**.

Node	CPU Model	cpu.irq_base	cpu.irq_base_tx	cpu.irq_base_rx
H	AMD Sempron 2200+	42 µs	17 µs	23 µs
J	AMD Sempron 2500+	42 µs	17 µs	20 µs
K	Intel P4 Celeron 1700MHz	51 µs	22 µs	29 µs

Table 20: CPU usage parameter set for the *IRQ threads* when sending and receiving data

7.4.6. Application scenario

The application scenario in this section shows the end-to-end latency that can be achieved in practice with different server configurations. Node **F** periodically generated test packets and sent them to node **J**. To achieve low delays at **F**, it used the strictly periodic traffic shaper with data dependency. Nodes **G**, **H** and **K** generated bulk traffic using the token-bucket shaper. The bulk traffic was sent to node **J** as well and used a common destination UDP port different from the port for the data from **F**. Table 21 shows the connection parameters of the sending nodes.

For comparison, Node **J**'s *IRQ thread* was scheduled with interrupt coalescing disabled once and in another experiment with interrupt coalescing enabled. The client application at **J** receiving the traffic from **F** used the blocking client mode to achieve low delays on that connection. The client application receiving the bulk data from **G**, **H** and **K** used the polling client mode with a polling interval of 1 ms.

Node	CPU Model	M	bandwidth	T_i	D_i	burstiness b_i
F	AMD Athlon 800MHz	86 Bytes	62 kbyte/s	–	0.3 ms	104 bytes
G	AMD Athlon 800MHz	1514 Bytes	2500 kbyte/s	2.07 ms	0.5 ms	7939 bytes
H	AMD Sempron 2200+	1514 Bytes	4891 kbyte/s	2.09 ms	0.5 ms	14181 bytes
K	Intel P4 Celeron 1700MHz	1514 Bytes	3865 kbyte/s	2.05 ms	0.5 ms	11369 bytes

Table 21: Connection parameters of the transmitting nodes. T_i denotes the period length of the *TX threads* of the connections, and D_i their scheduling deadline. The burstiness parameter b_i results from Table 1.

Each node only sent on one connection. According to Equation 7 on page 22, the burstiness parameter of the individual connections thus did not increase at the sending nodes. According to Table 1 on page 38, the worst-case delay increase due to the traffic shaper at node **F** was **300 µs**. The network latency from node **F** to the switch was **7 µs**. Applying Equation 15 on page 23, the worst-case delay at the network switch resulted in **2575 µs**.

To calculate the resource requirements for node **J**, the traffic parameters of the flows entering **J** must be known. As the data of **G**, **H** and **K** was received by one connection at **J**, the burstiness of that data could be calculated from the aggregate of these flows at the switch: Instead of increasing the burstiness of the three flows separated and summing them up afterwards, the flow specifications were added first and that aggregate was subject to burstiness increase by the flow from **F** and t_{mux} at the switch output then. This aggregation results in a lower burstiness bound compared to the separated calculation. According to Equation 21 on page 25, the burstines parameter of the connection from node **F** to **J** was **263 Bytes** when entering **J**. The burstiness parameter of the aggregate of the connections from nodes **G**, **H** and **K** calculated to **34104 Bytes**.

To calculate the CPU reservation for the *IRQ thread* at **J**, the maximum number of packets that may arrive in a time interval must be calculated. As argued in Section 6.4.1, the relevant time interval is its scheduling period plus its deadline. denoted by $T_{IRQ} + D_{IRQ}$. During the experiment, T_{IRQ} was 1 ms. According to Equation 41 on page 53, the *IRQ thread* had to receive no more than 11 packets per millisecond (3 packets from **F**, and up to 8 packets from **G**, **H** and **K**).

With the *IRQ coalescing scheme*, this resulted in a worst-case execution time of 77 µs for the *IRQ thread*. As it was the highest priority thread in the system, 77 µs also was its scheduling deadline. Using Equation 41 again with $T_{IRQ} + D_{IRQ}$=1.077 ms verifies the maximum number of packets to be 11. The CPU reservation for the notifier thread of the connection from **F** resulted in 28 µs per ms, the CPU reservation for the client thread in 34 µs. The notifier thread for the data from **G**, **H** and **K** needed 83 µs in the worst case, and the corresponding client thread needs up to 39 µs. For the test packets from **F** this resulted in a worst-case scheduling delay of 222 µs, increased by T_{IRQ}=1 ms caused by the IRQ coalescing, giving **1222 µs**. The application-to-application delay bound for packets from node **F** to **J** thus was **4104 µs**.

With the *noncoalescing scheme*, the *IRQ thread* was activated whenever a packet was received. Per activation, no more than 4 packets needed to be handled then (3 from **F** and 1 from **G**, **H** and **K**), taking up 68 µs in the worst case. The *IRQ thread* could be activated up to 11 times per millisecond, resulting in a worst-case CPU reservation of **697 µs** per ms. The notifier thread for the test packets could be activated up to 3 times per ms taking up to 25 µs each. The corresponding client thread consumed up to 32 µs per activation. This resulted in a CPU-reservation of 75 µs for the notifier thread and in 96 µs for the client thread. The notifier thread of the other connection worked in the polling mode and was activated only once per millisecond. Its worst-case CPU usage resulted in 83 µs per ms, and the corresponding client thread needed not more than 39 µs per ms. The overall CPU reservation utilization is thus 99%. A time-demand analysis for the test packet's client thread gives a worst-case scheduling delay for the test packets of 887 µs.[12] However, the *IRQ thread* did not execute for 697 µs in a row, and applying this knowledge gives tighter bounds: Once a 1514 byte-packet arrived at the NIC, the next 1514 byte-packet could arrive not earlier than 123 µs later. The 86-byte packets arrived in intervals of 7 µs. Thus, after a 86-byte test packet was seen by the *IRQ thread*, no more than 2 further *IRQ thread* activations could happen until the client thread received the packet. This results in a worst-case scheduling delay for the test packets of **394 µs**, giving an application-to-application delay bound of **3276 µs** when IRQ coalescing is not used.

To measure the actual transmission delays, nodes **F** and **J** where synchronized with a parallel cable as described in Section 7.1.2 on page 85. Further, node **F** put two time-stamps into each test packet:

i) at the client application before it sent it to the *TX thread*
ii) at the *TX thread* when it removed the packet from the connection-specific send-ring and put it to send-ring of the NIC.

Upon reception, node **J** put another two time-stamps into the packet:

iii) when the *IRQ thread* got the packet from the NIC
iv) when the client thread finally received the packet.

The difference of time-stamps i) and iv) is the application-to-application delay, the difference of time-stamps ii) and iii) gives a hint on the actual network delay. The latter time difference does not include the delay of the traffic shaper and the scheduling delay of the notifier and the client thread. Although, it contains the time to enqueue a packet at the NIC and to schedule the *IRQ thread* after packet reception.

[12] priorities: IRQ > notify threads > test client thread > other client threads

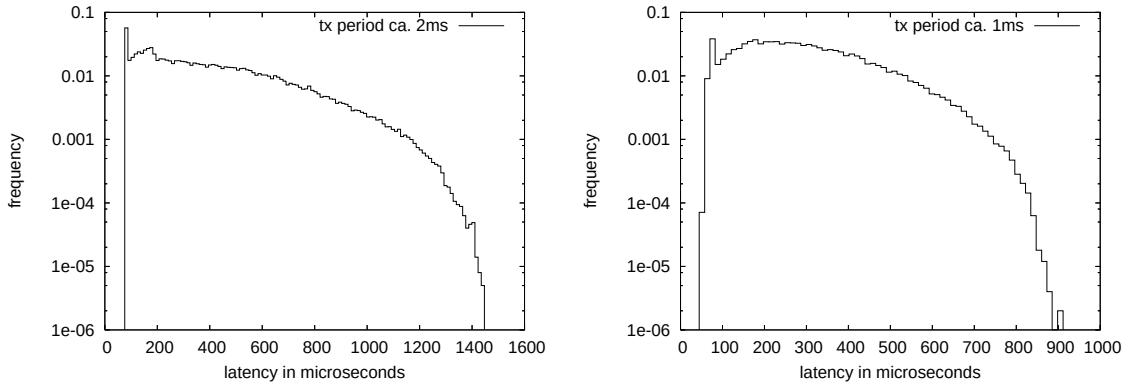

Figure 44: Distribution of app-to-app packet latencies from node **F** to **J**. The y-axis is in logarithmic scale.

The left part of Figure 44 shows the distribution of the measured test-packet application-to-application delays, without interrupt coalescing at node **J**. The maximum observed delay was 1441 ms, which is much less than the worst-case bound of 3276 µs. There are two reasons for this: First, the assumed network traffic model is overly pessimistic. The *TX threads* at nodes **F**, **G**, **H** and **K** suffered nearly no scheduling jitter, and as such their produced streams had a lower burstiness than expressed by the flow specifications. Second, the worst-case scheduling delays are pessimistic. This is caused by the deep and numerous caches of the x86 architecture, which almost always allow to execute a series of operations in a shorter time than the sum of their individual measured worst-case execution times. The parameter set for the linear CPU usage prediction model was obtained as the upper bound of many individual measurement results, and as such is pessimistic by nature.

The accumulated delay between the time-stamps i) and ii) and between time-stamps iii) to iv) was about 30 µs, thus the actual scheduling delays were much shorter than the theoretical worst-case bounds. The overall CPU consumption by the network stack and its clients at node **J** was about 50%, which is the half of the predicted worst-case CPU usage. The worst-case CPU consumption of the *IRQ thread* was 221 µs per ms, which is about a third of the CPU amount reserved. With the cache-flooder disabled, the overall CPU consumption at node **J** dropped to 25%.

In a second experiment, the *IRQ thread* at node **J** used the software interrupt coalescing feature. As expected, the application-to-application delays increased by T_{IRQ}=1 ms.

The right part of Figure 44 shows the end-to-end delays of another experiment: The period lengths of the *TX threads* at nodes **G**, **H** and **K** were reduced to 1.07 ms, 1.09 ms and 1.05 ms, and node **J** received the packets without interrupt coalescing. Due to the fine-granular traffic-shaping, the observed application-to-application delays of the test-packets were below 905 µs. In another experiment, with *TX thread* period lengths of 5.07 ms, 5.09 ms and 5.05 ms. In that experiment, the observed worst-case application-to-application-delay was 3082 µs.

Finally, the *IRQ thread* at node **J** enabled the IRQ coalescing feature again, but used a coalescing time of 10 ms. The application receiving the bulk data used a polling interval of 10 ms. Conse-

Figure 45: Setup for Section 7.4.7: two nodes (**L**, **M**) are connected to a Fast Ethernet switch. Node **L** executed L^{4}Linux: Once the standard L^{4}Linux with its native network stack ad NIC driver, another one it executed L^{4}Linux with the RT-Net driver stub. Node **M** generated test traffic to and received test traffic from **L**. For the RT-Net setup, it executed the bandwidth manager.

quently, the application-to-application delay was increased by another 10 ms, but the overall CPU reservation at node **J** was only 10%.

7.4.7. L^{4}Linux integration

To measure the performance of the L^{4}Linux stub described in Section 6.9.1 on page 81 and the performance of the one-shot reservation approach described in Section 6.8.1 on page 78, two nodes were connected two a switch as depicted in Figure 45.

In four measurement series, (1) the performance of the RT-Net stub was compared to the performance of standard L^{4}Linux for data reception, and (2) the performance of the once-shot reservation approach for data transmission as described in Section 6.8.1 on page 78 was analyzed and compared to that of standard L^{4}Linux. Therefore, node **L** executed L^{4}Linux version 2.2: Once the standard L^{4}Linux with its native network stack and standard NIC driver, another one **L** executed L^{4}Linux with the RT-Net driver stub. Node **M** generated test traffic to and received test traffic from **L**. For the RT-Net setup, **M** executed the bandwidth manager.

Data reception

For the data reception experiments, node **M** generated traffic with different bandwidths. Once **M** sent UDP packets, and in another series it sent TCP packets. At **L**, all packets were received by a Linux application. During the experiments, **L** measured its overall CPU load.

The left part of Figure 46 shows the CPU utilization of node **L** for both L^{4}Linux variants under different incoming TCP network loads. The depicted bandwidths are the effective TCP throughputs observed by the applications. As expected, the CPU usage with RT-Net stub is higher than those of standard L^{4}Linux for lower bandwidths. This can be explained by the increased overhead due to additional context switching and data copying. However, for higher bandwidths, the RT-Net stub outperforms the standard L^{4}Linux implementation. This can be explained by the packet coalescing of the RT-Net server at the notifier thread when communicating with L^{4}Linux. In contrast to the EEpro100 NIC driver in standard L^{4}Linux, the RT-Net server coalesced more packets in the case of higher load at L^{4}Linux. This resulted in fewer context switches to L^{4}Linux.

The right part of Figure 46 shows the CPU utilization of node **L** for both L^{4}Linux variants under different incoming UDP network loads. The depicted bandwidths are the effective UDP throughputs observed by the applications, which is nearly the overall network throughput as packet loss

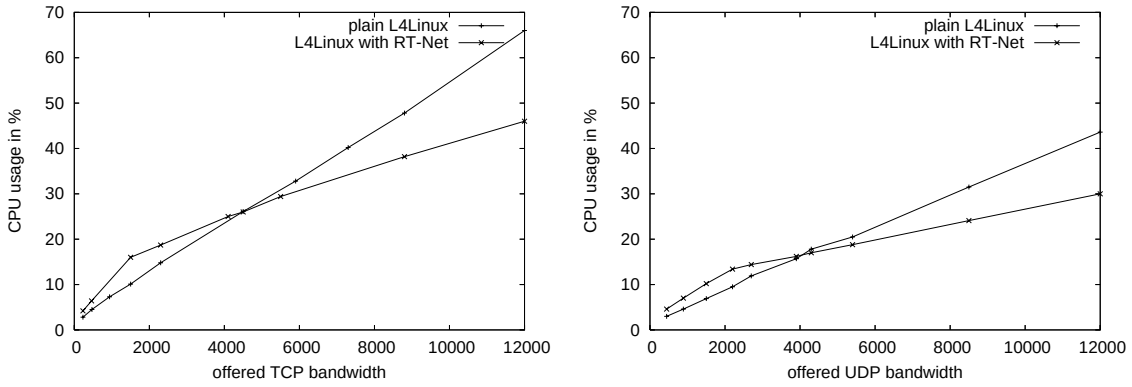

Figure 46: CPU usage of plain L^{4}Linux and L^{4}Linux with the RT-Net stub and when receiving data.

only happened occasionally. As with the TCP experiment, standard L^{4}Linux outperformed the RT-Net stub for lower bandwidths, but required more CPU with higher bandwidths.

Data transmission

For the data transmission experiments, a Linux application at node **L** transmitted TCP data with different TCP packet sizes as fast as possible to node **M**. The RT-Net server at **L** used one-shot reservations to adapt the bandwidth reservation to its best-effort clients needs. Therefore, the best-effort traffic shapers set a flag whenever they had to wait because of an empty bucket. A periodic thread polled this flag every 100 ms. If the flag was set, it asked the local network manager to increase the reserved bandwidth by a factor of 1.3. If the flag was not set, the bandwidth was decreased by a factor of 2.0, up to a minimum reservation of 200 Bytes/ms. Thus, a greedy sender could achieve the maximum Fast Ethernet throughput after 1.5 seconds, and the reservation fall back to its minimum value after 0.6 seconds of silence.

With these adaption parameters, L^{4}Linux achieved a TCP throughput of **9.7 MByte/s**. The bandwidth requirement for the reservation traffic to node **M** was 2.5 KByte/s.

Section 5.7 on page 36 discussed the minimum bucket size for the best-effort shaper to achieve a full network utilization. The experiments however showed, that bucket sizes greater than $B_i = r_t * T_i + M$ do not result in a higher throughput.

The left part of Figure 47 shows the CPU usage of node **L** for different TCP packet sizes. Clearly, the communication between the Linux client application and the L^{4}Linux significantly determines the overall CPU usage of the system. For comparison, the measurement was repeated with standard L^{4}Linux. With standard L^{4}Linux, the measurement application achieved a throughput of **10.5 MByte/s**. The overall CPU usage of node **L** is depicted in the right part of Figure 47. The higher CPU usage compared to the RT-Net experiment has two reasons: First, the transmitted bandwidth was increased by 8%. Second, the interrupt and packet coalescing of the RT-Net server

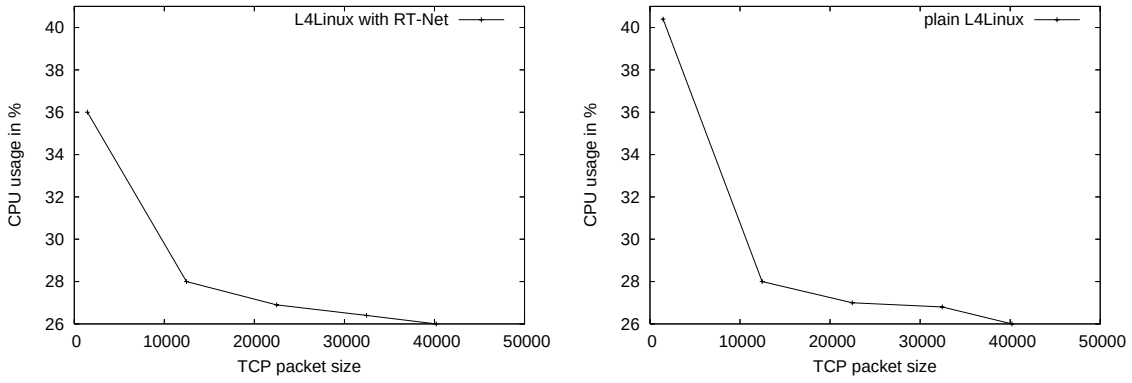

Figure 47: CPU usage of L^{4}Linux with the RT-Net stub and standard L^{4}Linux stub and when sending data.

resulted in in fewer context switches, and therefore in fewer cache-misses compared to the standard L^{4}Linux implementation.

8. Conclusion

This dissertation shows, both theoretically and in experiments, how traffic shaping can be used to achieve reliable application-to-application communication with bounded transmission delays on Switched Ethernet. The thesis comprises four main contributions and a number of auxiliary contributions.

8.1. Main contributions

Formal network model The dissertation applies network scheduling theory to Switched Ethernet and provides the model to calculate exact delay and buffer bounds for the switch and its attached nodes.

Dedicated traffic shapers To ensure that the nodes do not flood the network and obey previously acknowledged traffic contracts, they must shape their traffic accordingly. Different delay requirements and scheduling needs demand for dedicated shaper solutions. They differ in the properties of the generated data flows and in their implementation costs with respect to CPU utilization and requirement to the underlying CPU scheduler. As a general result, there is a trade-off between a low CPU usage by the traffic-shaping process on the one hand, and low delays at the Ethernet due to smoothly-shaped traffic flows on the other hand.

Operating-system requirements Section 6 identifies and analyzes the necessary operating system support for hard real-time communication on Switched Ethernet. It presents the design and implementation of a real-time network stack for the priority-based, dynamic real-time operating system DROPS. The network stack has the following properties:

- To meet different application QoS requirements, the network stack provides traffic models that differ in their used traffic shapers, achievable communication delays, guarantees on data delivery, and resource and scheduling requirements.
- It features a multi-threaded architectures to isolate connections with respect to CPU usage. The CPU reservation is based on an overall control- and data-flow analysis that takes the burstiness increase due to the inherent scheduling jitter into account.
- To reduce and bound the IRQ-related CPU consumption, the network drivers use software-based interrupt coalescing. Event-coalescing techniques reduce the CPU consumption for the coordination inside the network stack and the application interaction. Copying of data is avoided using zero-copying techniques, although data reception from the network needs to copy data once unless further hardware support is available.

Experimental verification Detailed measurements validated the starting hypothesis that with fine grained traffic shaping as only means of node cooperation, it is possible to achieve lower guaranteed delays and higher bandwidth utilization than with time-slotted and token-passing approaches:

- A special version of the network stack—having the highest priority in the system— demonstrated that sub-millisecond application-to-application delays can be guaranteed for both Fast and Gigabit Ethernet. The network utilization was 93% on Fast Ethernet and 49% on Gigabit Ethernet.

- The full version of the network stack—doing appropriate CPU reservation— requires a 10% CPU utilization to receive data with Fast Ethernet speed while giving a 10 ms application-to-application delay guarantee to a small fraction of the data and a 20 ms delay guarantee to the rest.
- With less aggressive interrupt and event coalescing, even modern and fast CPUs can be flooded with network traffic. With a 3 ms and 4 ms application-to-application delay guarantee, the previous setup resulted in a 99% CPU utilization.
- Slower nodes do not provide the memory bandwidth that is necessary for low-delay real-time communication on Fast Ethernet.

8.2. Auxiliary contributions

Nodes executing all-purpose operating systems such as Linux can also share a real-time network. Although no delay guarantees can be given to their applications, implemented traffic shapers prevent a flooding of the switches and allow these nodes to communicate with the network. Nonetheless, these traffic shapers sometimes generate data flows exceeding the specified parameters and advise to carefully analyze the traffic actually generated.

L^{4}Linux, a modified Linux kernel that allows to execute legacy Linux applications on DROPS, is one of the most important best-effort applications of that real-time system. I integrated L^{4}Linux into the real-time network stack to provide L^{4}Linux access to the real-time network and to show how best-effort network stacks can be layered on top of the real-time stack. Experiments demonstrated (1) that the one-shot reservation mechanism satisfies the changing bandwidth needs of best-effort network stacks, and (2) that the coalescing mechanisms are a sensible approach for layering network stacks on external network drivers in general and for providing L^{4}Linux access to the real-time network in particular. The performance and CPU usage are comparable to that of a standard L^{4}Linux implementation with integrated network drivers.

Operating systems with low scheduling jitters, such as time-triggered systems, allow a very regular application execution. Even in the presence of many jitter-sensitive tasks, these systems can schedule software-implemented traffic shapers in fixed time intervals. In comparison to the priority based system used in this dissertation, these systems allow to generate streams with lower burstiness, and thus can help to reduce the worst-case queueing delays in the network switches.

8.3. Suggestions for future work

This dissertation applies network scheduling theory to Switched Ethernet to formally describe the network traffic and to derive a model of a network with one switch and the nodes connected to it. Networks with multiple switches were not discussed as they are out of the scope of this thesis. An extension to multi-switched networks would open new application fields, and the steps necessary are outlined in Section 4.5.

The communicating nodes used in this thesis are modern workstations. Although even this sophisticated hardware experienced performance bottlenecks, an implementation on embedded systems

is very appealing, especially in environments with only moderate bandwidth requirements: Processors for embedded systems often apply only few performance enhancing technologies such as caches or deep pipelines. Hence their context-switching costs, the main source of the performance bottlenecks, are typically small or even negligible. Further, embedded systems often use specifically manufactured hardware, including the main processor and adapted communication devices on one chip (SoC). Using intelligent network adapters at these SoCs that disburden the main CPUs by offloading the traffic shaping of send data, and the demultiplexing of received data seems an obvious conclusion.

8.4. Concluding remarks

Traffic shaping on Switched Ethernet is a general approach that works with commodity hardware and requires no traffic monitoring or traffic control inside the network switches. The nodes must shape their traffic, but otherwise can send data whenever they want. During the doctorate process for this dissertation, I successfully filed an according patent [Loe02] on the principles of real-time communication on switched networks.

A. Derivations

A.1. Bounds for stair-case arrival curve and service curves

This section derives the backlog bounds (buffer bounds) and delay bounds for a system with a service curve in the form of a stair-case that is traversed by an arrival curve in the form of a stair-case, as given in Section 6.4.6 on page 69.

The arrival curve is denoted by $\alpha(t)$ and given by Equation 86. The service curve is denoted by $\beta(t)$ and given by Equation 87. Figure 48 shows the arrival and service curves.

$$\alpha(t) = \begin{cases} b + \left\lfloor \frac{t+m}{P} \right\rfloor \cdot \bar{b} & t > 0 \\ 0 & t = 0 \end{cases} \tag{86}$$

$$\beta(t) = \left\lfloor \frac{(t-D)+}{T} \right\rfloor \cdot C \tag{87}$$

It holds especially:

$$\bar{b} < b \qquad m \leq P \qquad D \leq T \qquad C = \alpha(T)$$

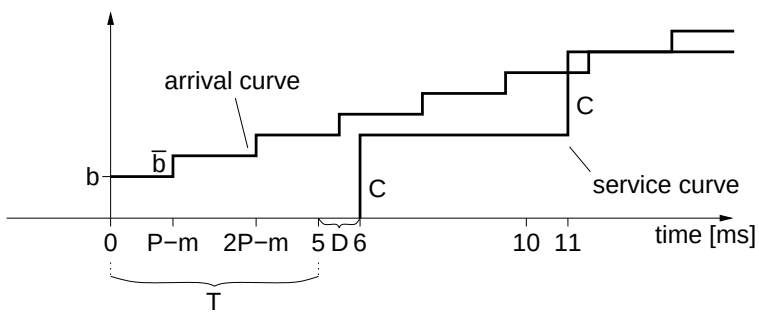

Figure 48: Arrival curve in the form of a stair-case and a service curve in the form of a stair-case.

Lemma A.1 *α is sub-additive, thus for all $x \geq 0$, $y \geq 0$ it is $\alpha(x) + \alpha(y) \geq \alpha(x+y)$.*

Proof of Lemma A.1:

If $x = 0$ or $y = 0$, $\alpha(x) + \alpha(y) \geq \alpha(x+y)$ follows immediately.

Let $x > 0$ and $y > 0$ hold. Further, let n_x and n_y be appropriate integer numbers and d_x and d_y be appropriate real numbers with $0 \leq d_x < 1$, $0 \leq d_y < 1$, so that

$$x = n_x \cdot P + d_x$$
$$y = n_y \cdot P + d_y$$

Then, for $\alpha(x)+\alpha(y)$ holds:

$$\begin{aligned}\alpha(x)+\alpha(y) &= 2b+\bar{b}\cdot\left(\left\lfloor\frac{n_x\cdot P+d_x+m}{P}\right\rfloor+\left\lfloor\frac{n_y\cdot P+d_y+m}{P}\right\rfloor\right)\\ &= 2b+\bar{b}\cdot(n_x+n_y)+\bar{b}\cdot\left(\left\lfloor\frac{d_x+m}{P}\right\rfloor+\left\lfloor\frac{d_y+m}{P}\right\rfloor\right)\end{aligned}$$

Prove for $d_x+m<P, d_y+m<P$:

If $d_x+m<P$ and $d_y+m<P$, it is $\lfloor\frac{d_x+m}{P}\rfloor=\lfloor\frac{d_y+m}{P}\rfloor=0$. Further, it is $d_x+d_y+m<2P$, and therefore $\lfloor\frac{d_x+d_y+m}{P}\rfloor\leq 1$.Thus,

$$\begin{aligned}\alpha(x)+\alpha(y) &= 2b+\bar{b}\cdot(n_x+n_y)\\ &\geq b+\bar{b}\cdot(n_x+n_y)+\bar{b}\cdot\left\lfloor\frac{d_x+d_y+m}{P}\right\rfloor\\ &\geq b+\bar{b}\cdot\left\lfloor\frac{n_x\cdot P+d_x+n_y\cdot P+d_y+m}{P}\right\rfloor\\ &\geq \alpha(x+y)\end{aligned}$$

Prove for $d_x+m\geq P$:

If $d_x+m\geq P$, it is $\lfloor\frac{d_x+m}{P}\rfloor\geq 1$. Further, it is $d_x+d_y+m<3P$, and therefore $\lfloor\frac{d_x+d_y+m}{P}\rfloor\leq 2$.Thus,

$$\begin{aligned}\alpha(x)+\alpha(y) &\geq 2b+\bar{b}\cdot(n_x+n_y)+\bar{b}\\ &\geq b+\bar{b}\cdot(n_x+n_y)+2\bar{b}\\ &\geq b+\bar{b}\cdot(n_x+n_y)+\bar{b}\cdot\left\lfloor\frac{d_x+d_y+m}{P}\right\rfloor\\ &\geq \alpha(x+y)\end{aligned}$$

q.e.d.

A.1.1. Buffer bound

The backlog bound B for the considered system is given by Equation 2 on page 18:

$$B=\sup_{s\geq 0}(\alpha(s)-\beta(s)) \tag{88}$$

With $\tilde{B}(t)$ defined as $\tilde{B}(t)=\alpha(s)-\beta(s)$, B can also be written as $B=\sup_{s\geq 0}\tilde{B}(t)$.

Lemma A.2 *B is given by $B=\alpha(D+T)$.*

Proof of Lemma A.2:

As α is wide-sense increasing, it follows from $\beta(t) = 0$ for all $t < D+T$ that $\tilde{B}(t) \leq \alpha(D+T)$ for all $t < D+T$.

β is constant in the interval $[T+D, T+2\cdot D)$ and has the value $C = \alpha(T)$. α is sub-additive, thus $\alpha(T+D+x) \leq \alpha(T+D)+\alpha(x)$ for all $x \leq 0$. Thus, $\alpha(T+2\cdot D) \leq \alpha(T+D)+\alpha(T) = \alpha(T+D)+C$. Consequently, $\tilde{B}(t) \leq \alpha(T+D)$ for all $T+D \leq T < T+2\cdot D$.

In further intervals $[T\cdot n+D, T\cdot(n+1)+D)$, with n=2,3,4,..... , β has the value $n\cdot C$. α is at most $\alpha(T+D)+n\cdot C$ due to its sub-additivity. q.e.d.

A.1.2. Delay bound

The delay bound d for the considered system is given by Equation 3 on page 18:

$$\begin{aligned} d &= \sup_s \delta(s) \\ \text{with } \delta(s) &= \inf\{\tau \geq 0 : \alpha(s) \leq \beta(\tau+s)\} \end{aligned}$$

Lemma A.3 *d is given by $d = D+T$.*

Proof of Lemma A.3:

As α is wide-sense increasing, it follows for all $0 \leq t \leq T$ that $\alpha(t) \leq \alpha(T) = \beta(D+T)$. As β is wide-sense increasing, it holds $\alpha(t) \leq \beta(D+T+t)$. Thus, $\delta(t) \leq D+T$ for all $0 \leq t \leq T$.

Further, it holds for all $0 \leq t \leq T$ that $\alpha(t+n\cdot T) \leq \alpha(T+n\cdot T)$ for n=1,2,3.... Due the sub-additivity of α, it is $\alpha(T+n\cdot T) \leq \alpha(T)+n\cdot\alpha(T)$. It further holds that $\beta(T+D+n\cdot T) = \alpha(T)+n\cdot C = \alpha(T)+n\cdot\alpha(T)$. Thus, $\alpha(t+n\cdot T) \leq \beta(T+D+n\cdot T)$. As β is wide-sense increasing, $\alpha(t+n\cdot T) \leq \beta(t+T+D+n\cdot T)$ holds. Thus, $\delta(t+n\cdot T) \leq D+T$ for all n=1,2,3... and all $0 \leq t \leq T$. q.e.d.

B. Abbreviations

API Application programming interface – an interface to be used by applications to communicate with a specific server application.

DDE Device Driver Environment – the driver environment used with DROPS. DDE provides abstractions and an according API to be used by device drivers such as interrupts, I/O ports and PCI memory. It is targeted to reuse Linux drivers in DROPS.

DROPS Dresden Real-time Operating System – a mikrokernel based real-time system developed at the Technische Universität Dresden. For details see Section 2.2.1.

DSI DROPS Streaming Interface – an asynchronous inter-task communication protocol for DROPS. It provides buffered communication between address spaces, and allows the communication between and among real-time and non–real-time applications [LRH01].

IP Internet protocol – a network communication protocol at layer 3 of the OSI network model. An IP address is a 32bit address identifying a node on the network. Actual network transmission at the MAC level requires a prior MAC lookup – the translation of an IP address into the MAC address of the corresponding node.

IPC Interprocess communication – in the context of L4 this is a mikrokernel abstraction of a synchronous message between two threads.

IRQ Interrupt request – an interrupt issued by some hardware device to asynchronously signal some event.

MAC Medium access control – the hardware level of network communication. A MAC address is an address than can be interpreted by network hardware such as switches and NICs.

NIC Network interface card – the network device at a node that sends network frames to and receives network frames from the physical network.

PCI Peripheral Controller Interface – a hardware bus specification used by hardware devices on many platforms, such as the Intel x86-based PC architecture.

QAS Quality assuring scheduling – one of the DROPS scheduling models. For details, see Section 2.2.2 at page 6.

QRMS Quality rate monotonic scheduling – one of the DROPS scheduling models. For details, see Section 2.2.2 at page 6.

TFTP Trivial file transfer protocol – an easy-to-implement, synchronous, IP-based protocol to obtain files from a TFTP file server. Standardized in RFC 783.

WCET worst-case execution time – the upper time bound of the execution of a specific piece of code

C. Glossary

leased line A theoretical model to describe a communication channel. A leased line i is a channel with a guaranteed bandwidth r_i and a guaranteed maximum delay d_i, capable of transporting messages up to size M_i.

leaky-bucket shaper, conforming to A theoretical model to describe traffic flows in a network. For details, see Section 4.2.2 at page 20.

task In the terminology of L4, a task is an address space containing virtual memory mappings, and, on the Intel x86 architecture, port I/O mappings. A task typically contains at least one *thread*.

thread In the terminology of L4, a thread is an activity inside an *address space*. A thread executes code and communicates with other threads using *IPC*s.

token-bucket shaper, conforming to A theoretical model to describe traffic flows in a network. For details, see Section 4.2.2 at page 20.

References

[APJ+01] Mohit Aron, Yoonho Park, Trent Jaeger, Jochen Liedtke, Kevin Elphinstone, and Luke Deller. The SawMill Framework for VM Diversity. In *Proc. 6th Australasian Computer Architecture Conference*, January 2001. Available at `ftp://ftp.cse.unsw.edu.au/pub/users/disypapersAron_PJLED_01.ps.gz`.

[BBC+98] D. Black, S. Blake, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An architecture for differentiated services. RFC 2475, December 1998.

[BBVvE95] A. Basu, V. Buch, W. Vogels, and T. von Eicken. U-net: A user-level network interface for parallel and distributed computing. In *Proc. of the 15th ACM Symposium on Operating Systems Principles*, Copper Mountain, Colorado, December 1995.

[BCS94] R. Braden, D. Clark, and S. Shenker. Integrated services in the internet architecture: An overview. RFC 1633, June 1994.

[Ber93] B. N. Bershad. Practical considerations for non-blocking concurrent objects. In Robert Werner, editor, *Proceedings of the 13th International Conference on Distributed Computing Systems*, pages 264–274, Pittsburgh, PA, May 1993. IEEE Computer Society Press.

[BH00] J.-Y. Le Boudec and G. Hebuterne. Comment on a deterministic approach to the end-to-end analysis of packet flows in connection oriented network. . *IEEE/ACM transactions on networking*, February 2000.

[Bor99] Martin Borriss. *Operating System Support for Predictable High-Speed Communication*. PhD thesis, TU Dresden, Dresden, Germany, January 1999.

[BT01] J.-Y. Le Boudec and P. Thiran. *Network Calculus*. Springer Verlag Lecture Notes in Computer Science volume 2050, July 2001.

[BZB+97] B. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin. Resource Reservation Protocol (RSVP) - Version 1 Functional Specification. RFC 2205, September 1997.

[CBM02] R. Caponetto, L. Lo Bello, and O. Mirabella. Fuzzy Traffic Smoothing: Another Step towards Real-Time Communication over Ethernet Networks. In *1st Intl Workshop on Real-Time LANs in the Internet Age*, Vienna, Austria, June 2002.

[CEB02] V. Cholvi, J. Echagüe, and J.-Y. Le Boudec. Worst Case Burstiness Increase due to FIFO multiplexing . *Performance Evaulation*, 49(1-4), November 2002.

[CKPW99] Dah Ming Chiu, Miriam Kadansky, Joe Provino, and Joseph Wesley. Experiences in Programming a Traffic Shaper. Technical Report SMLI TR-99-77, SUN microsystems, September 1999.

[CL01] N. Christin and J. Liebeherr. The QoSbox: A PC-Router for Quantitative Service Differentiation. Technical Report CS-2001-28, University of Virginia, November 2001.

[Cru91a] Rene L. Cruz. A calculus for network delay, part i: Network elements in isolation. *IEEE Transactions on Information Theory*, 37(1):114–131, January 1991.

[Cru91b] Rene L. Cruz. A calculus for network delay, part ii: Network analysis. *IEEE Transactions on Information Theory*, 37(1):132–141, January 1991.

[Dan99] Uwe Dannowski. ATM Firmware for DROPS. Master's thesis, TU Dresden, July 1999. Available from URL: `http://os.inf.tu-dresden.de/~ud3/papers/atm_firmware_for_drops.ps`.

[DH00] U. Dannowski and H. Härtig. Policing offloaded. In *Proceedings of the Sixth IEEE Real-Time Technology and Application Symposium*, Washington D.C., May 2000.

[DP93] Peter Druschel and Larry L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *Proceedings of the 14th ACM Symposium on Operating System Principles (SOSP)*, pages 189–202, Asheville, NC, December 1993.

[FSK05] M. Fidler, V. Sander, and W. Klimala. Traffic-shaping in aggregate-based networks: implementation and analysis. volume 28 of *Computer Communications*. Elsevier, March 2005.

[GKPR98] R. Guérin, S. Kamat, V. Peris, and R. Rajan. Scalable QoS Provision Trough Buffer Management. In *Proceedings of ACM SIGCOMM98*, Vancouver, Canada, August 1998.

[GLN+99] R. Guerin, L. Li, S. Nadas, P. Pan, and V. Peris. The Cost of QoS Support in Edge Devices: An Experimental Study. In *Proceedings of the IEEE Infocom*, New York, March 1999.

[HBB+98] H. Härtig, R. Baumgartl, M. Borriss, Cl.-J. Hamann, M. Hohmuth, F. Mehnert, L. Reuther, S. Schönberg, and J. Wolter. DROPS: OS support for distributed multimedia applications. In *Proceedings of the Eighth ACM SIGOPS European Workshop*, Sintra, Portugal, September 1998.

[HH01] M. Hohmuth and H. Härtig. Pragmatic nonblocking synchronization for real-time systems. In *USENIX Annual Technical Conference*, Boston, MA, June 2001.

[HHL+97] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. The performance of μ-kernel-based systems. In *Proceedings of the 16th ACM Symposium on Operating System Principles (SOSP)*, pages 66–77, Saint-Malo, France, October 1997.

[HHW98] Hermann Härtig, Michael Hohmuth, and Jean Wolter. Taming Linux. In *Proceedings of the 5th Annual Australasian Conference on Parallel And Real-Time Systems (PART '98)*, Adelaide, Australia, September 1998.

[HLR+01] C.-J. Hamann, J. Löser, L. Reuther, S. Schönberg, J. Wolter, and H. Härtig. Quality Assuring Scheduling - Deploying Stochastic Behavior to Improve Resource Utilization. In *22nd IEEE Real-Time Systems Symposium (RTSS)*, London, UK, December 2001.

[HWF05] Christian Helmuth, Alexander Warg, and Norman Feske. Mikro-SINA—Hands-on Experiences with the Nizza Security Architecture. In *Proceedings of the D.A.CH Security 2005*, Darmstadt, Germany, March 2005.

[JE04] J. Jasperneite and E. Elsayed. Investigations on a Distributed Time-triggered Ethernet Realtime Protocol Used by Profinet. In *Proceedings of RTN'04*, Catania, Italy, June 2004.

[KDK+89] Hermann Kopetz, Andreas Damm, Christian Koza, Marco Mulazzani, Wolfgang Schwabl, Christoph Senft, and Ralph Zainlinger. Distributed fault-tolerant real-time systems: the Mars approach. *IEEE Micro*, 9(1):25–40, February 1989.

[Ker03] Phill Kerr. Start of Distrubuted MIDI Stanard to Open Creative Possibilities for Musical Composition and Performance, 2003. `http://standards.ieee.org/announcements/p1639app.html`.

[KS00a] S.-K. Kweon and K. G. Shin. Achieving Real-Time Communication over Ethernet with Adaptive Traffic Smoothing . In *Proceedings of the Sixth IEEE Real-Time Technology and Application Symposium*, Washington D.C., May 2000.

[KS00b] S.-K. Kweon and K. G. Shin. Ethernet-Based Real-Time Control Networks for Manufacturing Automation Systems . In *Proceedings of the Seventh International Symposium on Manufacturing with Applications (WAC)*, 2000.

[KSZ99] Seok-Kyu Kweon, Kang G. Shin, and Qin Zheng. Statistical real-time communication over ethernet for manufacturing automation systems. In *Fifth IEEE Real-Time Technology and Applications Symposium (RTAS)*, Vancouver, Canada, June 1999.

[L4E] L4Env — an environment for l4 applications. Available at `http://os.inf.tu-dresden.de/l4env`.

[Lac04] Adam Lackorzynski. L^4Linux Porting Optimizations. Master's thesis, TU Dresden, March 2004.

[LH04a] J. Loeser and H. Haertig. Low-Latency Hard Real-Time Communication over Switched Ethernet. In *Proceedings of ECRTS'04, Euromicro Conference on Real-Time Systems*, Catania, Italy, June 2004.

[LH04b] J. Loeser and H. Haertig. Using Switched Ethernet for Hard Real-Time Communication. In *Proceedings of International Conference on Parallel Computing in Electrical Engineering Real-Time Systems (PARELEC)*, Dresden, Germany, September 2004.

[Lie95] J. Liedtke. On μ-kernel construction. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP)*, pages 237–250, Copper Mountain Resort, CO, December 1995.

[Liu00] J. W. S. Liu. *Real-Time Systems*. Prentice Hall, 2000.

[LMS05] Luciano Lenzini, Enzo Mingozzi, and Giovanni Stea. Delay bounds for fifo aggregates: a case study. volume 28 of *Computer Communications*. Elsevier, March 2005.

[Loe02] J. Loeser. *Verfahren zur verlustfreien Übertragung von Nachrichten in einem geswitchten Übertragungsnetzwerk*. Deutsches Patent- und Markenamt, Munich, Germany, February 2002. Patent no DE-10209787.

[Loe03a] J. Loeser. Buffer Bounds of a FIFO Multiplexer . Technical Report TUD-FI03-15, Technische Universität Dresden, November 2003.

[Loe03b] J. Loeser. Measuring Microsecond Delays . Technical Report TUD-FI03-16, Technische Universität Dresden, November 2003.

[LRH01] Jork Löser, Lars Reuther, and Hermann Härtig. A streaming interface for real-time interprocess communication. Technical Report TUD-FI01-09-August-2001, TU Dresden, August 2001. Available from URL: `http://os.inf.tu-dresden.de/~jork/dsi_tech_200108.ps`.

[LWD96] J. Liebeherr, D. E. Wrege, and Ferrari D. Exact Admission Control for Networks with a Bounded Delay Service. *IEEE/ACM Transactions on Networking*, 4(6), November 1996.

[LxH] `http://www.luxik.cdi.cz/~devik/qos/htb`.

[LxQ] `http://www.lartc.org`.

[MKT98] Frank W. Miller, Pete Keleher, and Satish K. Tripathi. General data streaming. In *19th IEEE Real-Time Systems Symposium (RTSS)*, Madrid, Spain, December 1998.

[Myr] Myricom Inc., 325 N. Santa Anita Ave, Arcadia, CA 91024. The GM message passing system. Available from `http://www.myri.com`.

[Oec97] P. Oechslin. Worst Case Arrivals of Leaky Bucket Constrained Sources: The Myth of the On-Off source . In *Proceedings of the IFIP Fifth International Workshop on Quality of Service*, New York, May 1997.

[PAG02] P. Pedreiras, L. Almeida, and P. Gai. The ftt-ethernet protocol: Merging flexibility, timeliness and efficiency. In *Proceedings of ECRTS'02, Euromicro Conference on Real-Time Systems*. IEEE Press, June 2002.

[PDZ00] Vivek S. Pai, Peter Druschel, and Willy Zwanenpoel. IO-Lite: A unified I/O buffering and caching system. *ACM Transactions on Computer Systems*, 18(1):37–66, February 2000.

[PL01] S. D. Patek and J. Liebeherr. Position Paper on Networs with Aggregate Qualitiy-of-Service . In *Proceedings of the SPIE Conference #4526*, October 2001.

[PLA03] P. Pedreiras, R. Leite, and L. Almeida. Characterizing the Real-Time Behavior of Prioritized Switched-Ethernet . In *2nd Intl Workshop on Real-Time LANs in the Internet Age*, Porto, Portugal, June 2003.

[Sch02] Martin Schwarz. Implementation of a ttp/c cluster based on commercial gigabit ethernet components. Master's thesis, Technische Universität Wien, 2002. Available from: `http://www.vmars.tuwien.ac.at/php/pserver/extern/docdetail.php?ID=1122&viewmode=thesis`.

[Ste04] Udo Steinberg. Quality-Assuring Scheduling in the Fiasco Microkernel. Master's thesis, TU Dresden, March 2004.

[Val95] John D. Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 214–222, Ottawa, Ontario, Canada, August 1995. Erratum available at `ftp://ftp.cs.rpi.edu/pub/valoisj/podc95-errata.ps.gz`.

[VC98] S. Varadarajan and T. Chiueh. EtheReal: A Host-Transparent Real-Time Fast Ethernet Switch. In *Proceedings of the Sixth International Conference on Network Protocols*, Austin, TX, October 1998.

[VcC94] Chitra Venkatramani and Tzi cker Chiueh. Supporting real-time traffic on ethernet. In *15th IEEE Real-Time Systems Symposium (RTSS)*, San Juan, Puerto Rico, December 1994.

[Wat02] Kym S. Watson. Network calculus in star and line networks with centralized communication. Technical Report IITB Report Number 10573, Fraunhofer IITB, Karlsruhe, Germany, April 2002.

[WJ03] Kym Watson and Jürgen Jasperneite. Determining end-to-end delays using network calculus. In *Proceedings of IFAC FET*, Aveiro, Portugal, 2003.

[WXBZ01] S. Wang, D. Xuang, R. Bettati, and W. Zhao. Providing Absolute Differentiated Services for Real-Time Applications in Static-Priority Scheduling Networks . In *Proceedings of the IEEE Infocom*, Anchorage, Alaska, April 2001.

[YC01] K. Yoshigoe and K. Christensen. Rate control for bandwidth allocated services in ieee 802.3 ethernet. In *IEEE 26th Conference on Local Computer Networks (LCN)*, Tampa, Florida, November 2001.