Report Number: WUCS-01-35

2001-11-13

Packet Scheduling for Link-Sharing and Quality of Service Support in Wireless Local Area Networks

Lars Wischhof and John W. Lockwood

The growing deployment of wireless local area networks (WLANs) in corporate environments, the increasing number of wireless Internet service providers and demand for quality of service support create a need for controlled sharing of wireless bandwidth. In this technical report a method for using long-term channel-state information in order to control the sharing of wireless bandwidth is studied. The algorithm enables an opera-tor of a WLAN to equalize the revenue/ cost for each customer in the network and to control the link-sharing based on a combination of user-centric and operator-centric fact-tors. As an example, we adapt one of the well-know...
Read complete abstract on page 2.

Packet Scheduling for Link-Sharing and Quality of Service Support in Wireless Local Area Networks

Lars Wischhof and John W. Lockwood

Complete Abstract:

The growing deployment of wireless local area networks (WLANs) in corporate environments, the increasing number of wireless Internet service providers and demand for quality of service support create a need for controlled sharing of wireless bandwidth. In this technical report a method for using long-term channel-state information in order to control the sharing of wireless bandwidth is studied. The algorithm enables an opera-tor of a WLAN to equalize the revenue/cost for each customer in the network and to control the link-sharing based on a combination of user-centric and operator-centric fact-tors. As an example, we adapt one of the well-know wireline schedulers for hierarchical link-sharing, the Hierarchical Fair Service Curve Algorithm (H-FSC), for the wireless environment and demonstrate through simulations that the modified algorithm is able to improve the controlled sharing of the wireless link without requiring modifications to the MAC layer. In addition, we present the design and implementation for a Linux based prototype. Measurements in a wireless testbed confirm that the scheduling scheme can perform resource-based link-sharing on currently available hardware conforming to the IEEE 802.11 standard.

Packet Scheduling for Link-Sharing and Quality of Service Support in Wireless Local Area Networks

Lars Wischhof
John W. Lockwood

WUCS-01-35

November 13, 2001

Department of Computer Science
Applied Research Lab
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130

Abstract

The growing deployment of wireless local area networks (WLANs) in corporate environments, the increasing number of wireless Internet service providers and demand for quality of service support create a need for controlled sharing of wireless bandwidth. In this technical report a method for using long-term channel-state information in order to control the sharing of wireless bandwidth is studied. The algorithm enables an operator of a WLAN to equalize the revenue/cost for each customer in the network and to control the link-sharing based on a combination of user-centric and operator-centric factors. As an example, we adapt one of the well-know wireline schedulers for hierarchical link-sharing, the Hierarchical Fair Service Curve Algorithm (H-FSC), for the wireless environment and demonstrate through simulations that the modified algorithm is able to improve the controlled sharing of the wireless link without requiring modifications to the MAC layer. In addition, we present the design and implementation for a Linux based prototype. Measurements in a wireless testbed confirm that the scheduling scheme can perform resource-based link-sharing on currently available hardware conforming to the IEEE 802.11 standard.

Contents

List of Figures

List of Tables

Listings

1. Introduction

In the last 10 years the usage of data services has increased tremendously: In 1993 1.3 million Internet hosts were registered, in September 2001 this number had increased to 123.3 million. From being a research project and medium for scientist, the Internet has become the world's largest public information network. As a result new types of applications were developed (e.g. multimedia streaming) with requirements which were not anticipated. Resource guarantees and performance assurance are not possible with the best-effort service the Internet is based on. Therefore, starting in the 1990s, a number of Quality of Service (QoS) architectures were developed. At the core they have an algorithm which distributes the available resources: the packet scheduler.

A relatively new trend is the usage of wireless technology to access Internet data services. The increasing deployment of wireless local area networks (WLANs) and emergence of wireless Internet providers create a need for providing mechanisms for controlled sharing of the wireless link and support of QoS. The unreliability of the wireless transmission media and its other characteristics make packet scheduling in the wireless environment a challenging problem. Recent research has made much progress in the development of algorithms which are able to guarantee the fair sharing of a wireless link and improve throughput.

1.1 Goals

The majority of the currently discussed wireless scheduling algorithms rely on the accurate knowledge of the quality of the wireless link (the channel-state) towards the mobile hosts at the moment when a packet is transmitted. In order to be able to access this information on time, the scheduling scheme has to be implemented at a low layer within the protocol stack, i.e. the link layer. This makes these approaches hard to deploy using currently available hardware. Furthermore link-sharing architectures can become quite complex which makes a more hardware independent configuration (at a higher layer which is part of the operating system) desirable.

Therefore this work investigates a different approach which is based on the long-term quality (“the average quality”) of the wireless link towards a specific mobile host. Since this information is less time-critical and updated in relatively large intervals, the wireless scheduling can be implemented above the link layer. Although with this limited information it is not possible to increase the available throughput to a mobile host by avoiding transmissions on a bad channel (this task should still be handled by the link layer) it is able to improve the controlled sharing of the wireless link.

The goal of this work is twofold: A hierarchical wireless link- sharing model based on the long-term channel state is developed and the approach is applied to an existing wireline scheduler. In contrast to the formerly available schedulers the algorithm can perform a resource-based sharing of the wireless link on currently available IEEE 802.11 hardware, since it is independent of the wireless link layer. In a second step we demonstrate how the model can be implemented within the traffic control architecture of an existing operating system. The correct performance of the algorithm is verified using simulations and early prototype results.

1.2 Organization of this Technical Report

This document is organized as follows: In Chapters 2 and 3 an overview of existing QoS architectures and currently discussed scheduling algorithms is given. Chapter 4 introduces the wireless link-sharing model which is applied to an existing scheduler, the Hierarchical Fair Service Curve (H-FSC) Algorithm, in Chapter 5. Its implementation within the Linux Traffic Control (TC) environment is described in Chapter 6. Results of simulations and measurements are presented in Chapter 7 and the main part concludes with a summary in Chapter 8.

Additional information about implemented testbed tools and testbed configurations are included in Appendix A. Selected source code listings are in Appendix B.

2. QoS Architectures

This chapter gives a brief overview of the most important currently discussed quality of service architectures for the Internet. Since guaranteeing quality of service in any QoS architecture is based on using an appropriate scheduling algorithm, it also motivates the development of schedulers for QoS/link-sharing and gives "the big picture" in which such an algorithm is used. Only the basic ideas of each concept will be outlined, a more complete overview can be found e.g. in [55].

2.1 Integrated Services

In 1994 the Integrated Services Working Group of the Internet Engineering Task Force (IETF) issued a memo proposing the Integrated Services (IntServ) architecture [6] as an extension to the existing Internet protocols to support real-time services. It is based on the per-flow reservation of resources in all routers along the path from traffic source to destination. In this model the following steps set up a connection:

- The application specifies the characteristics of the generated traffic and the required service.
- The network uses a routing protocol to find an optimal path to the destination.
- A reservation protocol, the Resource Reservation Protocol (RSVP) [7], is used to make reservations in each node along that path. (A host will only accept a RSVP request if it passes the admission control, which checks if sufficient resources are available.)

Two kinds of service models are supported: The Guaranteed Service [46] provides strict delay bounds and guaranteed bandwidth, whereas the Controlled Load Service [59] gives only statistical assurances (*better-than-best-effort* service). Since the Guaranteed Service requires the reservation of resources for the worst case it leads to a less efficient utilization

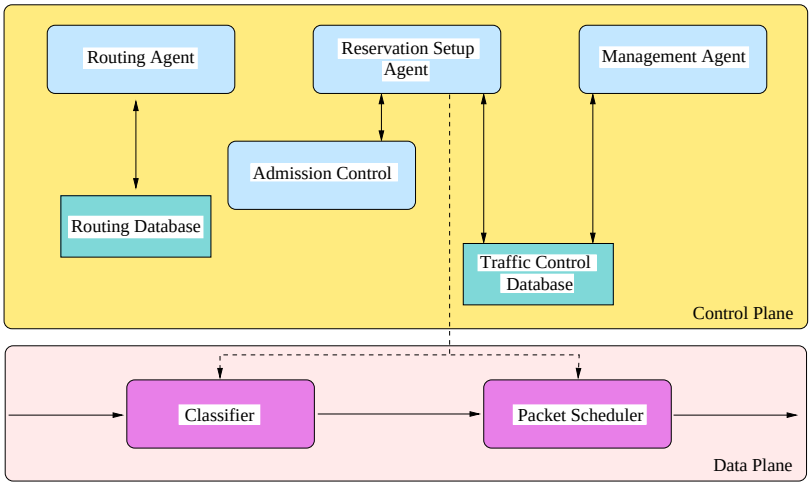


Figure 2.1: *Integrated Services reference model. [6], [55]*

of the available bandwidth. Reservations are enforced by a classifier which determines the flow/class a packet belongs to and the packet scheduler which serves each flow according to the resource reservations. Figure 2.1 illustrates the Integrated Services components in a node.

The main drawbacks of the IntServ architecture are the large overhead for setting up a connection and scalability problems since each interior node needs to implement per-flow classification and scheduling. These led to the development of alternative architectures like DiffServ.

2.2 Differentiated Services

Motivated by the demand for a more scalable Internet QoS architecture the development of the Differentiated Services (DiffServ/DS) architecture [5] started in 1997. Instead of resource management on a micro-flow[1] level it specifies a small number of forwarding classes. Packets are classified only at the edge of the network (where the number of micro-flows is relatively small). The packet's class is encoded in the DS field (the former *type of service* (TOS) field) of the IP packet header.

Nodes at the edge of the network are also responsible for traffic policing to protect the network from misbehaving traffic: A *meter* measures the amount of traffic submitted by each customer. The DS field of a packet is set to a specific *Differentiated Services Codepoint (DSCP)* by a *marker* depending on the conformance indicated by the meter and the result of the classification. Packets can also be marked by applications and will be remarked by the edge router in case of nonconformance. Instead of marking nonconforming packets with a different DSCP, they can be delayed by a *shaper* or dropped by a *dropper*. Figure 2.2 shows the classification and conditioning components.

[1] A *micro-flow* is a single instance of an application to application flow, usually characterized by the source/destination ports and addresses.

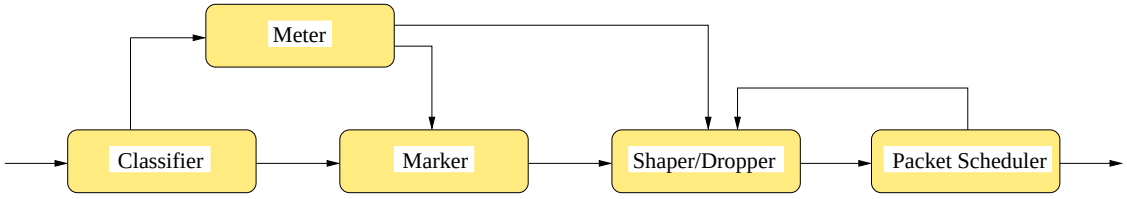


Figure 2.2: *DiffServ - traffic classification and conditioning components.*

Interior nodes forward packets according to the per-hop behavior (PHB) specified for the DSCP encoded in the packet header. Currently two PHBs have been standardized by the IETF: *Assured Forwarding (AF)* [19] and *Expedited Forwarding (EF)* [22]. AF defines four different forwarding classes with three drop precedences within each class. Each class has its own bandwidth allocation, a class exceeding these resources drops packets according to their drop priority. The EF PHB specifies a behavior similar to priority queuing: Traffic with the DSCP for EF can preempt other traffic as long as it does not exceed its assigned rate.

PHBs are implemented using buffer management and packet scheduling mechanisms. The packet scheduler distributes the available resources based on the rates specified for each class. Depending on the implementation it can also be responsible for parts of the traffic policing e.g. for shaping.

Since the Differentiated Services architecture does not require per micro-flow resource reservation setup and classification/conditioning is handled at the edge of the network, it has a high scalability. On the other hand it is harder to guarantee a specific behavior for individual flows, because the delay/rate experienced by a flow is determined by the aggregate traffic for a class and the amount of resources reserved for it.

2.3 Multiprotocol Label Switching

Originally developed as an alternative, efficient approach for IP over ATM, Multiprotocol Label Switching (MPLS) [44] has evolved to an IP based QoS architecture for the Internet. It combines the traditional datagram service with a virtual circuit approach. The basic idea of label switching is to encode a short, fixed-length label in the packet header on which all packet forwarding decisions are based. A label switched router (LSR) uses the incoming label to determine next hop and outgoing label. A label switched path (LSP) along which a packet is forwarded is set up by using a special signaling protocol, the label distribution protocol. Basically a LSP is a virtual circuit and allows the setup of explicit routes for packets of a class, a critical capability for traffic-engineering. Similar to the Differentiated Services approach the ingress LSR at the edge of a network classifies packets to classes and sets the initial label. MPLS also supports bandwidth reservation for classes, again enforced by a packet/cell scheduler. Techniques for supporting Integrated and Differentiated Services over a MPLS based core network are under development.

3. QoS Scheduling Algorithms

This chapter gives an introduction to the topic of scheduling algorithms in order to explain the foundations which this work is based on. Since nearly all scheduling schemes for the wireless channel are derived of wireline scheduling algorithms the most important of these are presented first. Next the special requirements for wireless scheduling are shown and some of the currently discussed algorithms for the wireless environment are introduced. Then the different scheduling schemes are compared regarding their structure and properties. Finally a related set of algorithms, the hierarchical link-sharing schemes, are presented which allow the distribution of (excess) bandwidth according to a hierarchical structure. Because of limited space the algorithms will no be covered in full detail, but the background is covered to understand the scope of this work. The reader who is familiar with the topic of packet scheduling can skip the first sections and should continue with Section 3.2.

The scenario assumed in the following is that of a packet scheduling algorithm operating on a computational node which is part of a larger Network (e.g. the Internet) and receives or generates packets which are forwarded to different destinations. These packets are buffered in one or more queues inside the network stack. The scheduling algorithm answers the question *Which packet should be sent next on the transmission medium?* This decision may be influenced by the state of the different queues, the flow or class a packet belongs to, the QoS level a destination subscribed to, the state of the transmission medium, etc. Figure 3.1 illustrates a simple downlink scheduling model.[1]

3.1 QoS Packet Scheduling in the Wireline Environment

Wireline scheduling algorithms have been developed and analyzed in the research community for some time. What makes scheduling in a wireline environment relatively easy

[1] Note that we do not consider the position of the QoS scheduler yet - it could be part of the network protocol stack of the operating system or part of the Medium Access Control (MAC) mechanism of the network interface itself. This issue will be considered in Chapter 4.

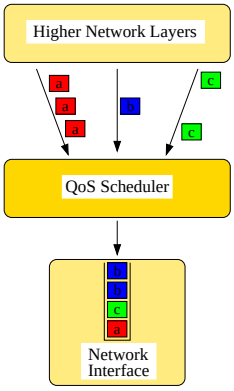


Figure 3.1: *Simple downlink scheduling model.*

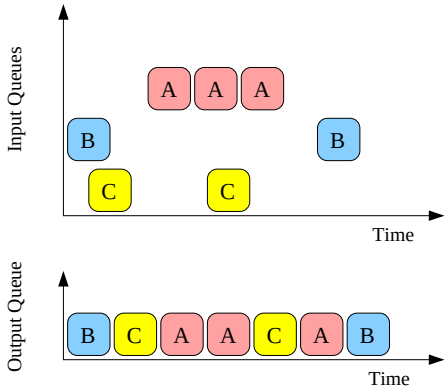


Figure 3.2: *Example for FIFO/FCFS scheduling.*

is that the capacity of the link and is constant. Therefore the algorithms aim at distributing this fixed bandwidth fairly among the different traffic flows - often taking into account the different quality of service requirements agreed on at admission-time of the flow.

3.1.1 First In First Out (FIFO) / First Come First Served (FCFS)

The simplest of all scheduling schemes called First In First Out (FIFO) or First Come First Served (FCFS) describes what happens in a system without explicit scheduling like most of the not QoS aware equipment in use today. All packets that arrive are enqueued in a single queue for each outgoing network interface card (NIC). If this queue is filled completely, all further packets are dropped until buffer space is available again. This property is called *tail dropping*. Whenever the network device is ready to transmit the next packet, the first packet in the queue, which is also called head-of-line (HOL) packet, is dequeued and transmitted. An an example for scheduling traffic from three different sessions in FCFS order is shown in Figure 3.2.

This “best effort” service is not suitable for providing any quality of service guarantees since flows are not isolated and the scheduling scheme does not take any of the specific requirements of a flow (e.g. delay, throughput) into account. For example, a low-throughput realtime connection like voice traffic can be starved by a high-throughput data transfer as shown in Figure 3.3.

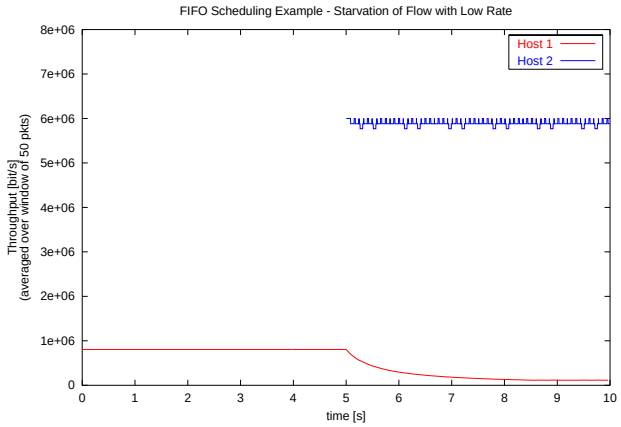


Figure 3.3: *A flow of Host 1 which is sent at a low rate is starved by a high-bandwidth flow of Host 2 which starts being transmitted 5 seconds later.*

Furthermore tail dropping is inadequate for delay-sensitive flows where packets become useless once they have been delayed in the network too long.

Scheduling schemes derived from FCFS are Last Come First Serve (LCFS) and Strict Priority. In LCFS the packet which arrived last is served first whereas in Strict Priority there exists a different queue for each packet priority. The next packet for transmission then is selected in FCFS fashion from the first non-empty queue of the highest priority.

3.1.2 Generalized Processor Sharing (GPS)

Generalized Processor Sharing (GPS) [40] is the ideal scheduling scheme which most other algorithms try to approximate as far as possible since GPS itself is not suitable for implementation in a packet network [2].

It is assumed that N flows are to be scheduled by the algorithm. In a GPS system every flow is assigned a weight ϕ_i which corresponds to its share of the total bandwidth in a way that the following condition holds:

$$\sum_{i \in [1,N]} \phi_i = 1 \tag{3.1}$$

If the amount of service that a flow i has received in the time interval $(\tau_1, \tau_2]$ is denoted by $S_i(\tau_1, \tau_2)$ then for every flow i *continuously backlogged*[3] within the interval a GPS server guarantees that

$$\frac{S_i(\tau_1, \tau_2)}{S_j(\tau_1, \tau_2)} \geq \frac{\phi_i}{\phi_j}; i, j \in [1, N] \tag{3.2}$$

[2] This is caused by the fact that GPS assumes work can be done in infinitely small steps which is not the case in a packet based network. Here the minimal amount of work is processing a packet of minimal packet size.

[3] A flow is called *backlogged* at time τ if there are packets stored in the queue of the flow.

Combining Equations 3.1 and 3.2 a server of rate r serves a flow i with a guaranteed service share of $g_i = \phi_i r$. In addition, if there is only a subset $B(\tau_1, \tau_2) \subset \{1, 2, \ldots, N\}$ of all flows backlogged, the remaining capacity will be distributed proportional to the weights ϕ_i

$$g_i^{inc} = \frac{\phi_i}{\sum_{j \in B} \phi_j} r;\ i \in B \tag{3.3}$$

A GPS server has the following properties which make it very suitable for traffic scheduling:

- It is work conserving[4].
- A flow is guaranteed a service share independent of the amount of traffic of other flows. The flows are isolated.
- It is very flexible. For example by assigning a small weight ϕ_i to unimportant background traffic, most of the channel capacity can be used by flows of higher priority (and a $\phi_j \gg \phi_i$) if they are backlogged while the remaining capacity will be used by the background traffic.
- The maximum delay a flow will experience is bounded by the maximum queue size Q_{max} and the maximum packet length L_{max} to

$$D_i^{max} = \frac{Q_{max} \cdot L_{max}}{g_i} = Q_{max} \cdot L_{max} \cdot \frac{\sum_{j \in [1,N]} \phi_j}{\phi_i \cdot r}; i \in [1, N] \tag{3.4}$$

Figure 3.4 illustrates GPS scheduling. Packets which are on one vertical line would need to be transmitted simultaneously on the network interface, which is not a feasible option in reality.

3.1.3 Weighted Round Robin (WRR)

Weighted Round Robin [40], [48] is the simplest approximation of GPS for packet based networks. It assumes that an average packet size is known. Every flow has an integer weight w_i corresponding to the service share it is supposed to get. Based on those weights a server with the total rate r pre-computes a service schedule (frame) which serves session i at a rate of $\frac{w_i}{\sum_j w_j} r$ assuming the average packet size. The server then polls the queues in sequence of the schedule, all empty queues are skipped. Figure 3.5 illustrates scheduling traffic from three sessions in WRR fashion. Although this scheme can be implemented very easily and requires only O(1) work to process a packet, it has several disadvantages: When an arriving packet misses its slot in the frame it can be delayed significantly in a heavy loaded system since it has to wait for the next slot of the flow in the pre-computed schedule. The algorithm is also not suitable if the average packet size is unknown or highly varying since in the worst-case a flow can consume $\frac{L_{max}}{L_{min}}$ times the rate that was assigned to it.

[4]The term *work conserving* describes the fact that the server is busy whenever there is a packet stored in one of the queues. No capacity is "lost" because of the scheduling.

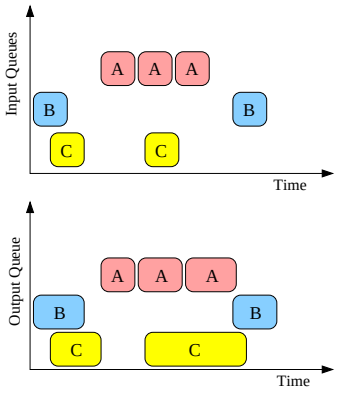


Figure 3.4: *Example for GPS scheduling with* $\phi_A = 0.5, \phi_B = 0.25, \phi_C = 0.25$.

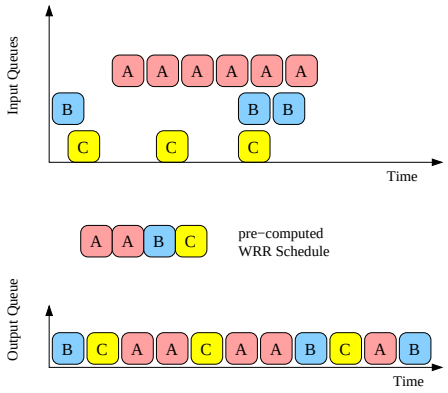


Figure 3.5: *Example for WRR scheduling with* $w_A = 2, w_B = 1, w_C = 1$.

3.1.4 Stochastic Fair Queuing (STFQ)

In order to avoid having a separate queue for each flow as in WRR, Stochastic Fair Queuing (STFQ) [29] was developed. In STFQ a hashing scheme is used to map incoming packets to their queues. Since the number of queues is considerably less than the number of possible flows, some flows will be assigned the same queue. Flows that collide this way will be treated unfairly - therefore the STFQ's fairness guarantees are probabilistic. The queues are served in round robin fashion (without taking the packet lengths into account). STFQ also uses a buffer stealing technique in order to share the available buffers among all queues: if all buffers are filled, the packet at the end of the longest queue is dropped.

3.1.5 Deficit Round Robin (DRR)

Because of the flaws WRR has in case of unknown or varying packet-sizes the Deficit Round Robin (DRR) scheduling scheme was suggested in [48] as an extension to STFQ. The basic idea is to keep track of the deficits a flow experiences during a round and to compensate them in the next round. Therefore, when a queue was not able to send a packet in the previous round because the packet size of its HOL packet was larger than the assigned slot, the amount of missed service is added in the next round.

This is implemented in the following way: Each queue has a value called *Quantum* which is proportional to the service share it is supposed to get and a state variable *DeficitCounter*. At the beginning of each round *Quantum* is added to the *DeficitCounter* of each queue. Then, if the HOL packet of the queue has a size less or equal to *DeficitCounter* it is processed and *DeficitCounter* is decreased by the packet size. In case of an empty queue the *DeficitCounter* is reset to zero.

Since DRR does not provide strong latency bounds, Shreedhar and Varghese [48] also introduced a variant called DRR+ which divides the flows in two classes, latency critical and best-effort flows. Latency critical flows are required not to send more packets than agreed on at admission time but are guaranteed not to be delayed more than one packet of every other latency critical flow.

3.1.6 Weighted Fair Queuing (WFQ)

Another often cited scheduling scheme which approximates GPS on a packet level is know as Weighted Fair Queuing (WFQ) or packet-by-packet Generalized Processor Sharing (PGPS) and was analyzed by Parekh and Gallager [40]. It can also be seen as the weighted version of a similar algorithm developed based on bit-by-bit round robin (BR) by Demers et al. [11], [24] as Fair Queuing.

The algorithm is based on the following assumptions:

1. Packets need to be transmitted as entities.
2. The next packet to depart under GPS may not have arrived yet when the server becomes free.

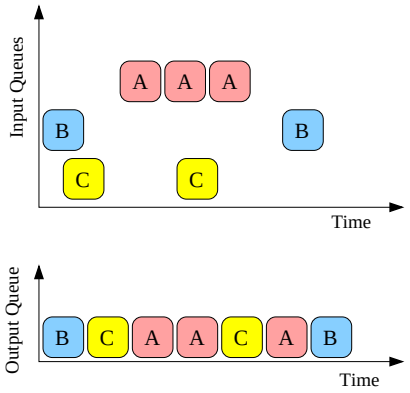


Figure 3.6: *Example for WFQ scheduling with* $\phi_A = 0.5, \phi_B = 0.25, \phi_C = 0.25$ *assuming a constant packet length* L_p*. Although the packets would have finished in a different sequence (B,C,A,A,A,C,B) under ideal GPS service no packet is delayed more than* $\frac{L_{max}}{r}$ *time units compared to its GPS departure time.*

3. Thus the server cannot be work conserving and process the packets in increasing order of their GPS departure time F_p.

The WFQ solution is to select that packet for transmission which would leave an ideal GPS server next *if no other packets were received.* Since both GPS and PGPS are work-conserving they must have the same busy periods. Therefore the maximum time which a packet can be delayed because of PGPS compared to GPS is

$$\hat{F}_p - F_p \leq \frac{L_{max}}{r} \tag{3.5}$$

where $\hat{F}_p$ is the PGPS departure time, F_p is the GPS finish time, L_{max} is the max. packet size and r is the server's rate. A PGPS system also does not fall behind the corresponding GPS system by more than one packet of maximum packet-size in terms of number of bits served $W_i(\cdot)$ for each session i:

$$W_{i,GPS}(0,\tau) - W_{i,WFQ}(0,\tau) \leq L_{max};\ \ i \in [1,N] \tag{3.6}$$

WFQ scheduling of packets from three different flows is shown in Figure 3.6.

Although a packet will not finish more than $\frac{L_{max}}{r}$ time units later, it can finish service much earlier than in GPS. An example would be a constellation in which there are 10 flows $i_1,\ldots,i_{10}$ with a small weight ϕ_i each backlogged with one packet and one flow j with $\phi_j = 10 \cdot \phi_i$ which is backlogged with 10 packets. In PGPS at least 9 packets of flow j will be processed first (with full server capacity!) before processing any packets of flows $i_{1,\ldots,10}$ since 9 packets of flow j have an earlier GPS departure time than the packets of flows $i_1,\ldots,i_{10}$. In the corresponding GPS system both would have been processed in parallel and the first 9 packets of flow j would have left much later. This specific issue is addressed by Worst-Case Fair Weighted Fair Queuing (WF^2Q) in Section 3.1.8.

3.1.7 Start-time Fair Queuing (SFQ)

A slightly different approach is taken by Start-time Fair Queuing (SFQ) [18] , which processes packets in order of their start-time instead of considering their finish tags. Therefore it is better suited than WFQ for integrated service networks because it offers a smaller average and maximum delay for low-throughput flows. It also has better fairness and smaller average delay than DRR and simplifies the calculation of the virtual time $v(t)$.

Upon arrival a packet p_j^f of flow f is assigned a start tag $S(p_j^f)$ and a finish tag $F(p_j^f)$ based on the arrival time of the packet $A(p_j^f)$ and the finish tag of the previous packet of the same flow. Their computation is shown in Equations 3.7 and 3.8.

$$S(p_j^f) = \max(v(A(p_j^f)), F(p_{j-1}^f)); j \geq 1 \tag{3.7}$$

$$F(p_j^f) = S(p_j^f) + \frac{L(p_j^f)}{r_f}; j \geq 1 \tag{3.8}$$

$$F(p_0^f) = 0 \tag{3.9}$$

The virtual time $v(t)$ of the server is defined as the start tag of the packet currently in service. In an idle period it is the maximum finish tag of all packets processed before. All packets are processed in increasing order of their start tags.

3.1.8 Worst-Case Fair Weighted Fair Queuing (WF^{2}Q)

In order to solve the problem that the service offered by WFQ can be far ahead of the GPS service it approximates (for an example see Section 3.1.6) it was extended to Worst-case Fair Weighted Fair Queuing (WF^{2}Q) in [3] which is neither ahead or behind by more than one packet of maximum packet-size. By avoiding oscillation between high service and low service states for a flow this way, WF^{2}Q is also much more suitable for feedback based congestion control algorithms.

In a WF^{2}Q system the server selects the next packets to process only among those packets which would have started service in the corresponding GPS system. In this group, the packet with minimum GPS departure time is chosen.

For WF^{2}Q the WFQ equations concerning delay and work completed for a session (Equations 3.5 and 3.6) are also valid. In addition, since a packet of flow *i* which leaves the WF^{2}Q system has at least started being processed in GPS it can be guaranteed that:

$$W_{i,WF^2Q}(0,\tau) - W_{i,GPS}(0,\tau) \leq (1 - \frac{r_i}{r})L_{i,max} \tag{3.10}$$

WF^{2}Q therefore is considered an optimal packet algorithm in approximation of GPS.

3.1.9 Worst-Case Fair Weighted Fair Queuing + (WF^{2}Q+)

The WF^{2}Q+ algorithm is an improved version of the WF^{2}Q (see 3.1.8) scheduling scheme with the same delay bounds but a lower complexity developed to support Hierarchical GPS (H-GPS) (Section 3.4.3, [4]). This is achieved by introducing a virtual time function $V_{WF^2Q+}(t)$ which eliminates the need to simulate the corresponding GPS system:

$$V_{WF^2Q+}(t+\tau) = \max(V_{WF^2Q+}(t) + W(t,t+\tau), \min_{i\in\hat{B}(t)}(S_i^{h_i(t)})) \tag{3.11}$$

in which $W(t,t+\tau)$ denotes the total amount of service in the period and $h_i(t)$ is the number of the head-of-line packet of queue i. Since at least one packet has a virtual start time lower or equal to the system's virtual time, the algorithm is work-conserving by selecting the packet with the lowest eligible start-time (like in WF^{2}Q) for transmission. The second modification to the original scheme is that WF^{2}Q+ simplifies the calculation of the start and finish tags of a packet. Instead of recalculating them for each packet after each transmission/packet arrival, they are only calculated when a packet reaches the head of its queue

$$S_i = \begin{cases} F_i & \text{if } Q_i(a_i^k-) \neq 0 \\ \max(F_i, V(a_i^k)) & \text{if } Q_i(a_i^k - 0) = 0 \end{cases} \tag{3.12}$$

where $Q_i(a_i^k-)$ is the queue size just before the arrival of packet k of flow i.

In [4] Bennett and Zhang show that with those modifications the algorithm is able to achieve the same worst-case fairness and bounded delay properties as WF^{2}Q.

3.2 Wireless QoS Packet Scheduling

The issue of packet scheduling within a wireless network is more complicated than in the wireline case since the wireless channel is influenced by additional factors which a wireless scheduling mechanism needs to take into account:

- location dependent errors (e.g. because of multi-path fading) as illustrated in Figure 3.7
- higher probability of transmission errors/error bursts
- interference of other radio sources
- dynamically increasing/decreasing number of stations (hand-over)

Most wireless scheduling algorithms assume that they schedule the downstream traffic in an access point and have full control over the wireless medium. Perfect knowledge on the states of the uplink queues of the mobile stations and the channel condition is also

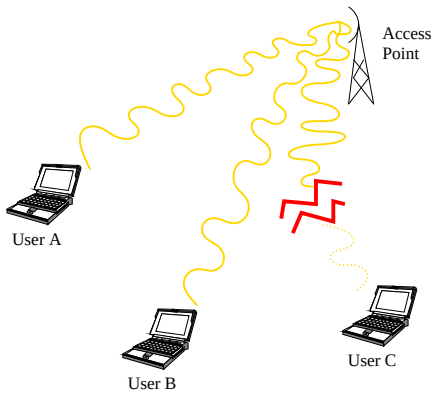


Figure 3.7: *Location dependent errors on the wireless channel. While users A and B have the full link capacity available, user C experiences an error burst.*

a precondition for the majority of the presented algorithms. For simulation purposes it is common to use a two state Markov chain model for the wireless channel known as Gilbert-Elliot Model [13] consisting of the two states “good channel” and “bad channel” and their transition probabilities.

Whereas in the case of a wireline network usually all stations experience the same channel quality, in the wireless network it is normal that some users have good channels while others do not and a fraction of time later a completely different set of users has capacity available. Therefore it might be necessary to provide users which experienced error bursts with additional channel capacity once they have good transmission conditions - this mechanism is known as *wireless compensation*.

3.2.1 Idealized Wireless Fair Queuing (IWFQ)

One adaptation of WFQ to handle location-dependent error bursts is Idealized Wireless Fair Queuing (IWFQ) [28], [35]. In IWFQ the error-free fluid service (GPS) for the flows is simulated as in WFQ. Each arriving packet is assigned a start tag $s_{i,n}$ and a finish tag $f_{i,n}$

$$s_{i,n} = \max(v(A(t_{i,n})), f_{i,n-1}); f_{i,n} = s_{i,n} + \frac{L_{i,n}}{r_i} \tag{3.13}$$

with $v(t)$ being computed from the set of backlogged flows $B(t)$ of the error-free service in the following way:

$$\frac{dv(t)}{dt} = \frac{C}{\sum_{i \in B(t)} r_i} \tag{3.14}$$

where C is the total capacity of the server and r_i is the rate of an individual flow i.

The algorithm then always selects the flow with the least finish tag which perceives a good channel for transmission. (Thus the selection process is comparable to WFQ - by limiting it to those packets who would have started transmission in the error-free reference model, it could also be adapted to a WF^{2}Q like variant.)

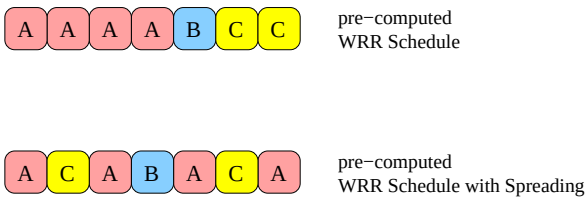


Figure 3.8: *In WPS* Spreading *is used to account for error bursts on the wireless channel.* $(w_A = 4, w_B = 1, w_C = 2)$

This scheme has an implicit wireless compensation: Since a flow which was not serviced for a while because of channel errors will have low finish tags, it will be given precedence in channel allocation for a short while once it is able to send again. Because this behavior violates the isolation property of GPS it is necessary to introduce an upper and a lower bound limiting the compensation. Therefore the start and finish tags are continously readjusted:

1. A flow i is only allowed to have an amount of B_i bits in packets with a finish tag lower than the virtual time $v(t)$. If this value is exceeded, the last recently arrived packets are discarded. [5]

2. A flow whose head of line packet has a start tag more than $\frac{l_i}{r_i}$ in the future will be adjusted to
$$s_{i,hol} = v(t) + \frac{l_i}{r_i}, f_{i,hol} = s_{i,hol} + \frac{L_{i,hol}}{r_i} \tag{3.15}$$

Packets in IWFQ can experience a higher (but bounded by $B = \sum_i B_i$) delay than in WFQ because of the wireless compensation:

$$d_{max,IWFQ} \leq d_{max,WFQ} + \frac{B}{C} \tag{3.16}$$

3.2.2 Wireless Packet Scheduling (WPS)

Wireless Packet Scheduling (WPS) [28] is an approximation of the IWFQ algorithm which uses Weighted Round Robin (WRR, see 3.1.3) as its error-free reference algorithm mainly because it is much simpler to implement. To account for the burstiness of wireless channel errors the basic WRR scheme was altered to use spreading of slots as illustrated in Figure 3.8 (generating a schedule equal to WFQ when all flows are backlogged). WPS assumes that all packets have the same size (slot), multiple slots form a frame.

WPS uses two techniques of wireless compensation:

[5] While this behavior seems appropriate for loss-sensitive traffic it is less suitable for delay-sensitive flows. In order to allow a flow to discard any packets from its queue without loosing the privileged access to the channel, IWFQ decouples finish tags and packets by having a slot queue and a packet queue.

1. *intra-frame swapping:* When the flow currently scheduled to send has an erroneous channel the algorithm tries to swap the slot with a later slot of a different flow currently having a good channel.

2. *credits/debits:* A backlogged flow unable to transmit during its assigned slot which also cannot swap slots, is assigned a credit (in bytes) if there are one or more other flows which are able to use the slot(s). In turn their credit is decremented. As in IWFQ both operations are bounded to a maximum credit/debt value.

The *effective weight* of a flow which is used when scheduling a new frame is the aggregate of its default weight and its credit/debit, thus lagging flows will receive more service than leading flows.

WPS uses a *one-step prediction* for channel estimation predicting that the channel will stay in the state in which it was when probed during the previous time slot.

3.2.3 Channel-Condition Independent Packet Fair Queueing (CIFQ)

The Channel-Condition Independent Packet Fair Queuing (CIF-Q) scheduling scheme is an adaption of Start-time Fair Queuing (SFQ, see 3.1.7) for the wireless channel [37], [38].

It was developed to provide

1. short-time fairness and throughput/delay guarantees for error-free sessions

2. long-time fairness for sessions with bounded channel error

3. graceful degradation of leading flows

An algorithm with these properties is called *Channel-Condition Independent Fair (CIF).*

The error-free service system used as a reference in CIF-Q is Start-time Fair Queuing (SFQ, see 3.1.7) in order to keep the implementation effort low, but any other wireline packet fair queuing scheme could also be used. CIF-Q schedules that packet for transmission which would be served next in the error-free reference model (i.e. in SFQ the one with the minimum start-time). In case that the selected flow is unable to transmit because it is in error or has to give up its lead, the service is distributed to another session but *it is charged to the session which would be serviced in the reference system.*[6] So in any case the virtual time of the selected flow is advanced in SFQ manner:

$$v_i = v_i + \frac{l_i^k}{r_i} \tag{3.17}$$

Thus the scheduling algorithm is self-clocked and there is no need to continously simulate the timing of a reference system. In order to keep track of the amount of traffic a session

[6] The virtual time of that flow which would transmit in the reference system is updated.

needs to be compensated for, a lag parameter is associated with every session which is increased for a session that was unable to transmit and decreased when it received additional service.

$$\sum_i lag_i = 0 \tag{3.18}$$

CIF-Q also avoids giving strict priority to the compensation of lagging flows by guaranteeing that leading flows will always receive a minimum amount of service of (1-α) times its service share (graceful degradation of leading flows). Excess bandwidth is distributed proportional to the weight of a flow.

The most important parts of the CIF-Q scheduling scheme are summarized in the following steps:

1. Select a flow *i* for transmission using corresponding reference system.

2. Advance the virtual time for flow *i* according to Equation 3.17.

3. If the flow is leading and has given at least a share of α percent for compensation of lagging flows or it is in-sync/lagging it is served.

4. If the selected flow is unable to send (channel error/not backlogged) or has to relinquish capacity for compensation the bandwidth is distributed among the lagging flows proportional to their rate.

5. If none of the lagging flows can send, flow *i* gets to transmit. If it is unable to transmit, distribute the excess bandwidth in proportion to the flows rates.

6. Adjust the lag variables if the flow selected is different from the flow which would send in the reference system.

When a lagging session wants to leave, the lag variable of the other active sessions is increased proportional to their rate so that Equation 3.18 still holds. A leading session is not allowed to leave until it has given up all its lead. In this way the algorithm has the CIF properties and is able to bound the delay that an error-free session will receive in an error system serving *n* active sessions to

$$d_{i,error-free} \leq (n-1)\frac{L_{max}}{R} + \frac{l_i^k}{R} + \frac{L_{max}}{r_i} \tag{3.19}$$

3.2.4 Wireless Fair Service (WFS)

A different algorithm that also supports the CIF properties is Wireless Fair Service (WFS) [27], which additionally decouples the bandwidth/delay requirements of flows in order to treat error- and delay-sensitive flows differently. The scheduling scheme also avoids disturbing flows which are *in sync* [7] because of wireless compensation if possible.

The corresponding error-free service model for WFS is an extension of WFQ (see Section 3.1.6): Each flow is assigned a weight for rate r_i and delay Φ_i. As in IWFQ (Section 3.2.1) an arriving packet creates a slot in the flows slot queue which has a start tag $S(p_i^k)$ and a finish tag $F(p_i^k)$

$$S(p_i^k) = \max(V(A(p_i^k)), S(p_i^k) + \frac{L_i^{k-1}}{r_i} \tag{3.20}$$

$$F(p_i^k) = S(p_i^k) + \frac{L_i^k}{\Phi_i} \tag{3.21}$$

Of all slots whose start tag is not ahead of the current virtual time by more than the schedulers lookahead parameter[8] ρ, WFS selects the slot with the minimum finish tag for transmission. The result is that packets will be drained into the scheduler proportional to r_i and served with rate Φ_i.

The separation of slots (right to access the channel) and packets (data to be transmitted) allows any kind of packet discarding scheme at a per flow level: For error-sensitive flows the scheduler can choose to drop all further packets once the queue is full, whereas for delay-sensitive flow a HOL packet which has violated its delay bounds can be dropped without loosing the right to access the channel.

The wireless compensation mechanism of WFS is rather complicated: Flows have a *lead counter* $E(i)$ and a *lag counter* $G(i)$ which keep track of their lead/lag in slots.[9] The algorithm performs compensation according to the following rules

1. If a flow is unable to transmit, it will try to allocate the slot first to a backlogged lagging flow, then to a backlogged leading flow whose lead is less than the leading bound $E_{max}(i)$, then to a in-sync flow and finally to any backlogged flow. If none of them has an error-free channel, the slot is wasted.

2. The lead/lag counter will only be modified if another flow is able to use the slot. Therefore the sum over all $E(i)$ is equal to the sum over all $G(i)$ at any time.

[7] A flow is called *in sync* when it is neither leading nor lagging compared to its error-free service.

[8] The *lookahead parameter* determines how far a flow may get ahead of its error-free service. If ρ is 0 the algorithm does not allow a flow to get ahead by more than one packet (like in WF^{2}Q), if ρ is ∞ then the selection mechanism is equal to Earliest Deadline First (EDF).

[9] Since either the lead counter or the lag counter of a flow is zero, one counter would be sufficient to keep track of the flows state. Despite this fact we chose to stay consistent with the algorithms description in [27].

3. A leading flow which has been chosen to relinquishing its slot to a lagging flow will regain its slot if none of the lagging flows perceives a clean channel.

4. If a lagging flow clears its queue (e.g. since the delay bounds were violated) its $G(i)$ counter is set to zero and the $E(i)$ counters of all leading flows will be reduced in proportion to their lead.

5. A leading flow is forced to give up a fraction of $\frac{E(i)}{E_{max}(i)}$ slots for wireless compensation (graceful degradation).

6. Slots designated for wireless compensation will be distributed in a WRR scheme to lagging flows, each flow has a compensation WRR weight of $G(i)$.

3.2.5 Wireless Worst-case Fair Weighted Fair Queueing (W^2F^2Q)

As the name suggests, the W^2F^2Q algorithm is an adaptation of Worst-case Fair Weighted Fair Queuing plus (W^2FQ+, see Section 3.1.9) for the wireless environment. It uses the same notion of virtual time as WF^2Q+ (Equation 3.11) which avoids the need to simulate a corresponding GPS reference system. In addition to the compensation inherited from WF^2Q+, which prefers flows recovering from an error-period since they have lower start tags, W^2F^2Q uses a wireless error compensation consisting of the following components:

1. **Readjusting:** If L denotes the length of a packet, each packet with a start tag more than $\frac{L}{\phi_i} + \delta_{i,max}$ ahead of the current virtual time is labeled with a start tag of $V + \frac{L}{\phi_i} + \delta_{i,max}$ (bounds leading flows). Each packet with a start tag more than $\frac{L}{\phi_i} + \gamma_{i,max}$ behind the current virtual time is assigned a start tag of $V - \frac{L}{\phi_i} + \gamma_{i,max}$ (bounds compensation received by lagging flows).

2. **Dynamic Graceful Degradation:** Instead of using a linear graceful degradation of leading backlogged flows as in CIF-Q (see Section 3.2.3), W^2F^2Q uses an exponential decrease. A leading flow which is more than one packet tranmission-length $\frac{L}{\phi_i}$ ahead will yield $(1-\alpha)$ of its service share to non leading flows. The parameter α is calculated depending on the number of lagging flows:

$$\alpha = \frac{1-\alpha_{min}}{e-1} e^{1-x} + \frac{e\alpha_{min}-1}{e-1} \tag{3.22}$$

   where x is the relation of combined weights of the flows in-sync to that of the flows not in-sync:

$$x = \frac{\sum_{\text{j in-sync}} \phi_j}{\sum_{\text{j not in-sync}} \phi_j} \tag{3.23}$$

3. **Handling of Unbacklogging in Lagging State:** In the case that a lagging flow becomes unbacklogged, meaning that it does not need as much service as the amount of lagging it has accumulated, the remaining lag is distributed to the other flows in

proportion to their weight as it is done in CIF-Q (Section 3.2.3) with the exception that instead of recalculating a *lag* variable the start tag is decreased:

$$s_j = s_j - \frac{\phi_i \cdot (V - s_i - \frac{L}{\phi_i})}{\sum_{k \in B} \phi_k} \tag{3.24}$$

3.2.6 Server Based Fairness Approach (SBFA)

A generalized approach for adapting wireline packet fair queuing algorithms to the wireless domain was developed by Ramanathan and Agrawal [43] as Server Based Fairness Approach (SBFA). It introduces the concept of a long-term fairness server (LTFS) which shares the bandwidth with the other sessions and is responsible for providing additional capacity to them in order to maintain the long-term fairness guarantees. SBFA also uses separate slot and packet queues as known from WFS (Section 3.2.4). Whenever a flow *i* is forced to defer the transmission of a packet because of an erroneous channel, a slot is inserted in the LTFS slot queue with tag *i*. Later, when the scheduler serves the queue of the LTFS the slot is dequeued and a packet of session *i* is transmitted. In this way the LTFS distributes an extra quantum of compensation bandwidth to lagging flows without degradation of the other flows. The same procedure happens if a flow needs retransmission of a packet.

Since sessions associated with a specific LTFS share its bandwidth it is beneficial to have more than one LTFS and assign sessions with similar requirements to the same LTFS. For example one LTFS could be used for real-time traffic and a different one for interactive traffic (e.g. WWW).

3.2.7 Channel State Independent Wireless Fair Queueing (CS-WFQ)

A quite different approach to wireless packet scheduling is taken by the Channel State Independent Wireless Fair Queuing (CS-WFQ) algorithm [26]. Whereas the schemes presented before use a two-state one-step-prediction for the channel state and try to maximize throughput for system experiencing a "good" channel,[10] CS-WFQ includes mechanisms in order to deal with channel errors (such as FEC) taking into account multiple levels of channel quality and tries to reach equal goodput at the terminal side. CS-WFQ is based on Start-time Fair Queuing (SFQ, see 3.1.7) which is extended in a way that each flow has a *time varying* service share. When the channel is in a bad condition the service share of the session is increased. This variation is bounded according to the QoS requirements of a flow: The higher the QoS level of a flow the larger are the bounds in which the share is allowed to vary.

Therefore a flow admitted to the CS-WFQ scheduler is assigned two values at call-admission time: a rate proportional fair share ϕ_i as in any wireline scheduler and a bound C_i^{th} for the instantaneous virtual goodput $C_i(t)$ of a flow. The scheduler will only compensate for a bad channel condition (e.g. by providing more bandwidth for FEC or ARQ schemes)

[10] This is referred to as *totalitarian situation* in [26] in contrast to an "egalitarian system" which tries to provide equal goodput for each user.

ATM class(es)	Priority	Scheduling
CBR (real-time, constant bit-rate)	1	Wireless Packet Scheduling (see 3.2.2)
rtVBR (real-time, variable bit-rate)	2	Wireless Packet Scheduling (see 3.2.2)
GFR, nrtVBR, ABR-MCR (non real-time, guaranteed bit-rate)	3	Weighted Round Robin (see 3.1.3)
ABR-rest (best-effort)	4	Recirculating FIFO (see 3.1.1)
UBR (pure best-effort)	5	FIFO (see 3.1.1)

Table 3.1: *MPFQ mapping of ATM classes to priorities/schedulers.*

as long as the channel quality results in a $C_i(t)$ better or equal to C_i^{th}. The time-varying service share ψ_i is computed as

$$\psi_i = \begin{cases} \frac{\phi_i}{C_i(t)} & \text{if } C_i(t) \geq C_i^{th} \\ \frac{\phi_i}{C_i^{th}} & \text{otherwise} \end{cases} \tag{3.25}$$

and thus limited by the threshold C_i^{th}.

3.2.8 Wireless Multiclass Priority Fair Queuing (MPFQ)

The Multiclass Priority Fair Queuing (MPFQ) which was presented in [30], [32], [33] and later improved with a wireless compensation mechanism in [8], [31] combines a class-based approach with flow based scheduling. It introduces a mapping of the ATM traffic classes in the wireline domain to the wireless channel which assigns each traffic class a specific scheduler (Table 3.2.8). Since MPFQ strictly separates the priorities, it can be seen as a combination of different independent schedulers.

The two real-time classes CBR and rtVBR both use a WPS algorithm with separate slot and packet queues for each flow. Thus, flows may drop packets according to different discard policies without losing the right to access the channel (decoupling of access-right and payload as in IWFQ, 3.2.1). The CBR traffic is given a higher priority than rtVBR traffic in order to provide a lower delay bound for this traffic class. The weight of a CBR/rtVBR flow is determined by $\phi_n = \frac{1}{maxCTD}$ so that it is inversely proportional to the maximum tolerated delay of the flow. [11]

The non-real-time classes with guaranteed bandwidth (including the minimum guaranteed rate of a ABR flow) are scheduled by a WRR scheduler at a medium priority level, since

[11] MPFQ assumes that monitoring and shaping will occur outside of the scheduler. Therfore e.g. a CBR flow will never exceed its specified rate.

the delay requirements are not that tight for these classes. The WRR algorithm serves the flows according to their MCR.

The fourth priority in the system is used for that amount of ABR traffic which exceeds the MCR. It uses a single, recirculating slot queue for all ABR flows. Since a slot contains only a pointer to the HOL packet of a queue, a slot of a packet which could not be transmitted due to channel errors can simply be re-inserted at the back of the queue. The lowest priority is used for traffic of unspecified bit-rate (UBR) which is serviced in a single best-effort FIFO queue without any wireless compensation.

In the error-free case traffic in the first priority level will have the same delay bounds as the WPS scheduling algorithm. The rtVBR traffic will additionally have to wait until all flows of priority 1 have cleared their queues but since proper admission control is assumed CBR flows will not have more than one packet in their queue.

In order to be able to determine the lead or lag of a flow MFPQ keeps track of two time variables: a current virtual time and a shadow virtual time. When a packet is skipped because the channel is in a bad state, the virtual time advances to the time of the packet transmitted instead. The shadow virtual time is updated to the value of the packet that should have been sent. Therefore the shadow virtual time keeps track of the time in a corresponding error-free system and the lead/lag of a flow is the difference of its HOL packet's tag to the shadow virtual time SV as shown in Equations 3.26, 3.27 and 3.28.

$$Lead(HOL_i) = ((f_{i,n} - \frac{L_p}{C}) - SV) \cdot \phi_{N_i} \tag{3.26}$$

$$Lag(HOL_i) = (SV - (f_{i,n} - \frac{L_p}{C})) \cdot \phi_{N_i} \tag{3.27}$$

$$\phi_{N_i} = \frac{\phi_i}{\sum_{j \in F} \phi_j} \tag{3.28}$$

Wireless compensation in MPFQ is also class-based. For CBR traffic a lagging flow is selected from a WRR schedule of lagging flows where each flow has a weight proportional to its lag. The rtVBR traffic is compensated by forcing the leading flows to give up a fraction of their bandwidth to the flow with the earliest finish time among all flows. For traffic of priority 3 the flow with the largest lag is chosen to receive compensation. The ABR traffic is compensated by re-insertion of the slot which could not be transmitted in the re-circulating packet queue. For UBR traffic no compensation is done at all.

3.3 Comparison of Scheduling Algorithms

This section compares the presented scheduling algorithms regarding different parameters and components. It provides a summary of this chapter and illustrates which algorithms have certain properties in common. On the other hand the listed aspects are rather arbitrary and cannot explain the capabilities of an algorithm in detail.

The following aspects are compared:

1. What is the algorithm's notion of (virtual) time? (Is a corresponding error-free reference-system simulated? How is the time variable updated?)
2. Are flows guaranteed a certain bandwidth (in the error-free case)? Does the algorithm provide separation/isolation of flows? In case of wireless scheduling algorithms this can be limited because of wireless compensation.
3. Does the algorithm provide wireless compensation?
4. Is the compensation of a flow which was unable to transmit in the past because of a bad channel bounded?
5. Are leading flows gracefully degraded?
6. Does the algorithm differentiate between slots and packets of a flow?
7. How complex is the implementation of the algorithm?
8. How high is the computational effort per scheduled packet?

Table 3.2: Comparison of Scheduling Algorithms

Algorithm	Virtual Time	Guaranteed Bandwidth	Compen-sation	Bounded Comp.	Graceful Degraded	Slot Queue	Impl. Effort	Comp. Effort	Additional Comments
Wireline Scheduling Algorithms									
FIFO	no	no	-	-	-	-	low	low	no notion of flows
GPS	no	yes	-	-	-	-	∞	∞	ideal algorithm
WRR	no	(yes)	-	-	-	-	medium	medium	bandwidth varies if pkt-length not const.
STFQ	no	proba-bilistic	-	-	-	-	medium	low	bandwidth varies if pkt-length not const.
DRR	no	proba-bilistic	-	-	-	-	medium	low	takes packet length into account
WFQ	GPS ref.	yes	-	-	-	-	high	high	-
SFQ	start-tag of pkt in service	yes	-	-	-	-	high	medium	-
WF^2Q	GPS ref.	yes	-	-	-	-	high	high	-
WF^2Q+	min. start-tag	yes	-	-	-	-	high	medium	ideal GPS approx.
Wireless Scheduling Algorithms									
IWFQ	GPS ref.	yes	implicit	no	no	no	high	high	-
WPS	WRR ref.	yes	yes	no	no	no	high	medium	-
CIF-Q	SFQ ref.	yes	yes	yes	yes	no	high	medium	-
WFS	WFQ ref.	yes	yes	yes	yes	yes	high	medium	-
W^2F^2Q	min. start-tag	yes	yes	yes	yes	no	high	medium	-
SBFA	depends	yes	LTFS	yes	yes	yes	high	medium	generic approach to extend wireline alg.
CS-WFQ	SFQ ref.	yes	yes	yes	yes	no	high	medium	egalitarian optimization
MPFQ	class dependent	class dependent	yes	yes	no	class dependent	high	medium	combines several algorithms

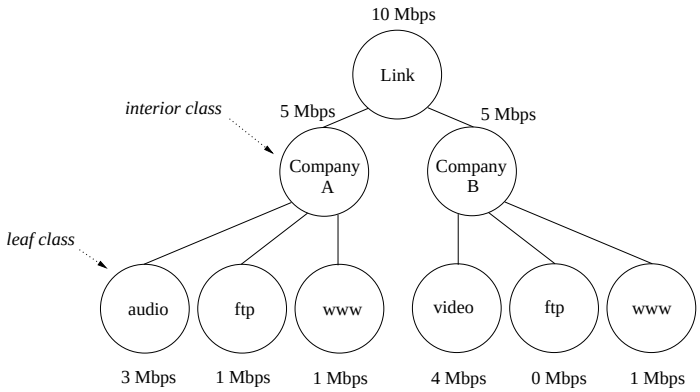


Figure 3.9: *Link-sharing between two companies and their traffic classes.*

3.4 Hierarchical Link-Sharing Algorithms

Very similar to the algorithms presented in the previous two sections are hierarchical link-sharing algorithms which allow packet scheduling based on a hierarchical link-sharing structure. A special focus of these algorithms is the sharing of excess bandwidth which - instead of simply being distributed proportional to the bandwidth share of a flow/class - is widely configurable. For example, in the situation that a link is shared by two customers bandwidth unused by one traffic class of a customer will first be available to other traffic classes of the same customer. Only if they do not have sufficient demand, the rest of the excess bandwidth is available to the other customer. This section gives a very brief overview of the two mainly used link-sharing algorithms: Class Based Queuing (CBQ) including a variant developed for wireless networks and the Hierarchical Fair Service Curve Algorithm (H-FSC).

3.4.1 Class Based Queuing (CBQ)

The class based queuing algorithm (CBQ) [16] presents a solution for unified scheduling for link-sharing and real-time purposes. It isolates real-time and best-effort traffic by allowing the separation of traffic in different traffic classes which are then treated according to their requirements, e.g. by giving priority to real-time traffic. Floyd and Jacobsen also develop a model for *hierarchical link-sharing* and show how the class based queuing algorithm can be used to set up hierarchical link-sharing structures which allow the controlled distribution of excess bandwidth. Although CBQ has the concept of using a *general scheduler* which is used in the absence of congestion and a *link-sharing scheduler* for rate-limiting classes, most implementations implement both mechanisms in a single (often WRR/DRR) scheduler. Furthermore, an *estimator* keeps track of the bandwidth a class is receiving by calculating an exponential weighted moving average over the discrepancy between the actual inter-departure time of two packets of a class to the inter-departure time corresponding to the classes rate.

An example for link-sharing is shown in Figure 3.9 where a 10 Mbps link is shared by two companies. In times of congestion each company should get the allocated 5 Mbps over a relevant time-frame and within a companies share the bandwidth is distributed to

Class Characteristic	Definition
regulated	Packets of the class are scheduled by the link-sharing scheduler.
unregulated	Packets of the class are scheduled by the general scheduler.
overlimit	The class has recently used more than its allocated bandwidth.
underlimit	The class has received less than its bandwidth-share.
at-limit	The class is neither over- nor underlimit.
unsatisfied	A leaf class which is underlimit and has persistent backlog or an interior class which is underlimit and has a child class with a persistent backlog.
satisfied	A class which is not unsatisfied.
exempt	The class will never be regulated.
bounded	The class is never allowed to borrow from ancestor classes.
isolated	A class which does not allow non-descendant classes to borrow excess bandwidth and that does not borrow bandwidth from other classes.

Table 3.3: *Definition of class charateristics in CBQ.*

the different traffic classes. If a leaf class has no rate assigned to it (e.g. the ftp traffic of Company B) it is not guaranteed any bandwidth at times of congestion. A list of class characteristics is listed in Table 3.4.1.

Floyd and Van Jacobsen define the following *link-sharing goals*:

1. Each interior or leaf class should receive roughly its allocated link-sharing bandwidth over appropriate time intervals, given sufficient demand.

2. If all leaf and interior classes with sufficient demand have received at least their allocated link-sharing bandwidth, the distribution of any 'excess' bandwidth should not be arbitrary, but should follow some set of reasonable guidelines.

Following these goals, a set of formal link-sharing guidelines[12] is derived which can be used to implement the desired behavior:

1. A class can continue unregulated if one of the following conditions hold:

   - The class is not overlimit, or
   - the class has a not-overlimit ancestor at level *i* and the link-sharing structure has no unsatisfied classes at levels lower than *i*.

[12] Actually the listed guidelines are called *Alternate link-sharing guidelines* in [16] since they are a variant of the formal link-sharing guidelines which avoids oscillation of a class between the regulated and unregulated state.

Otherwise, the class will be regulated by the link-sharing scheduler.

2. A regulated class will continue to be regulated until one of the following conditions hold:

   - The class is underlimit, or
   - The class has an underlimit ancestor at level *i*, and the link-sharing structure has no unsatisfied classes at levels lower than *i*.

Ancestor-Only link-sharing and Top Level link-sharing are two approximations to these formal link-sharing guidelines: In Ancester-Only link-sharing a class is only allowed to continue unregulated if it is not overlimit or if it has an underlimit ancestor. In Top Level link-sharing a *Top Level* variable is used in order to determine if a class may borrow bandwidth. A class must not be overlimit or it has to have an underlimit ancestor whose level is at most *Top Level* for not being regulated. The heuristics proposed by Floyd and Van Jacobsen for setting the *Top Level* variable are:

1. If a packet arrives for a not-overlimit class, *Top Level* is set to 1.
2. If *Top Level* is *i*, and a packet arrives for an overlimit class with an underlimit parent at a lower level than *i* (say *j*), then *Top Level* is set to *j*.
3. After a packet is sent from a class, and that class now either has an empty queue or is unable to continue unregulated, then *Top Level* is set to *Infinity*.

Top Level link-sharing has an additional overhead for maintaining the *Top Level* variable, but approximates the ideal link-sharing guidelines closer than Ancestor-Only link-sharing.

For scheduling real-time traffic CBQ allows the usage of different priorities for classes, which is able the reduce the delay for delay-sensitive traffic [15], [16]. Despite this fact, one major disadvantage of CBQ is the coupling of rate and delay within one priority class.

3.4.2 Enhanced Class-Based Queuing with Channel-State-Dependent Packet Scheduling

In [49] the CBQ scheme is extended for wireless link-sharing by making it suitable for variable rate links and combining it with Channel-State Dependent Packet Scheduling. The approach is to use the RTS-CTS handshake for sensing the quality of the link to a specific destination. If the link is in a “bad” state then a packet to a different destination is selected, thus avoiding the head-of-line blocking problem. In addition, the scheduler keeps track of the goodness of each link by using a parameter *g* which is increased inversely proportional to the number of RTS-CTS attempts which were done before successfully detecting a good link. The idea is that on a link which had good quality in the past one is more likely to detect a good link at a later time. Therefore the scheduler probes each destination up to *g* times, if a “good” channel is detected after less then *g* probes the

parameter is increased, if no CTS is received after g probes it is decreased. Also, the *estimator* component of CBQ is modified so that a class is unsatisfied if it has not received the allocated *percentage of the effective* throughput, thus taking the variable rate of the wireless link into account. Another change of the CBQ algorithm is that a restricted class is allowed to transmit even if unsatisfied classes exists if all of those unsatisfied classes experience a "bad" link state. Simulations show that by introducing these changes, the algorithm is able to achieve a higher throughput and a distribution of the goodput according to the allocated shares in a wireless environment.

3.4.3 Hierarchical Fair Service Curve (H-FSC)

A link-sharing scheduler which decouples delay and bandwidth allocation is the Hierarchical Fair Service Curve (H-FSC) Algorithm [51]. H-FSC is based on the concept of a *service curve* which defines the QoS requirements of a traffic class or single session in terms of bandwidth and priority (delay). A session i is guaranteed a service curve $S_i(t)$ if the following equation holds

$$S_i(t_2 - t_1) \leq w_i(t_1, t_2);\ \ t_1 < t_2 \tag{3.29}$$

in which t_2 is an arbitrary packet departure time at which the session is backlogged, t_1 is the start of this backlogged period and $w_i(t_1, t_2)$ is the amount of service received by session i in the interval $(t_1, t_2]$. Usually only linear or piece-wise linear service curves are used for simplicity. A concave[13] service curve results in a lower average and worst case delay than a convex curve with the same asymptotic rate. Figure 3.10 shows an example in which rate and delay are decoupled.

However, a server can only guarantee all service curves $S_i(t)$ if the service $S(t)$ it provides is always larger or equal to their sum:

$$S(t) \geq \sum_i S_i(t) \tag{3.30}$$

In H-FSC each class in the hierarchy has a service curve associated with it. Excess bandwidth is distributed according to the service curves and a service which received excess service will not be punished later (fairness properties). However, in their paper [51] Stoica et. al. prove that with non-linear service curves (which are necessary if bandwidth and delay properties are to be decoupled) it is impossible to avoid periods in which the scheduler is unable to guarantee the service curves of all classes or it is not able to guarantee both the service curves and fairness properties. Therefore the H-FSC algorithm only guarantees the service curves of leaf classes and tries to minimize the difference between the service an interior class receives and the amount it is allocated according to the fairness properties.

This is achieved by using two different criteria for selecting packets: A *real-time criterion* which is only used when the danger exists that the service curve of a leaf class is violated

[13] A service curve is *concave* if its second derivative is non-positive and not the constant function zero. Analogously, it is *convex* if its second derivative is positive and not the constant function zero.

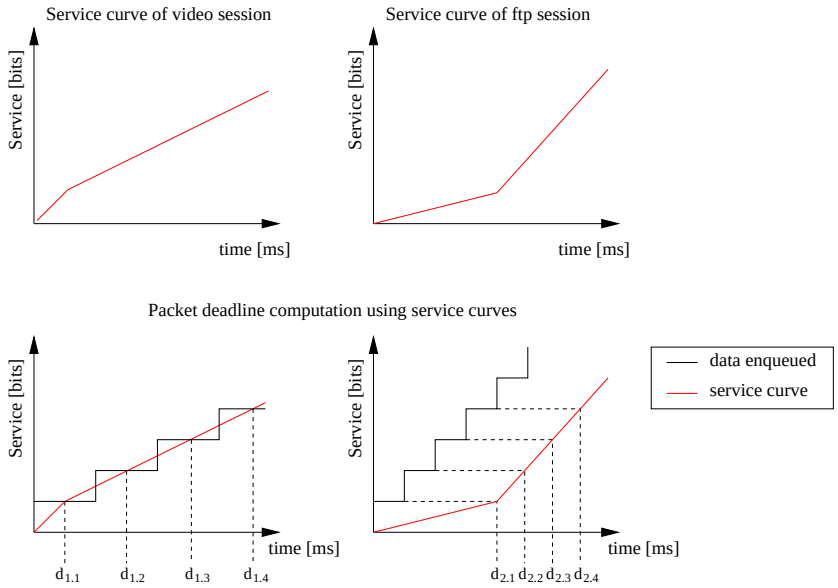


Figure 3.10: *An example for a service curve of a video and a FTP session. The (interactive) video session requires a low delay at a moderate rate, whereas the FTP session is guaranteed a high rate with a higher delay. [51]*

and a *link-sharing criterion* which determines the next packet otherwise. Therefore each leaf class has a triplet (e_i, d_i, v_i) associated with it which represents the eligible time, the deadline and the virtual time. An interior class only maintains its virtual time v_i. If, at time t, a leaf class with $e_i < t$ exists, there is danger of violating a service curve and the real-time criterion is used to select the packet with the minimal eligible time. Otherwise the link-sharing criterion searches the packet with the minimum virtual time, starting at the root class. Since the virtual time of a class represents the normalized amount of work received by a class, the goal of the link-sharing criterion is to minimize the difference between the virtual time v_i of a class and that of any sibling. For each leaf class the algorithm also maintains an eligible curve $E_i(a_i^k;\cdot)$ and a deadline curve $D_i(a_i^k;\cdot)$ where a_i^k is the start of active period number k of class i and a variable c_i, the amount of service a class received under the real-time criterion.

The deadline curve is initialized to the corresponding service curve of the class. At each point in time a_i^k when the class becomes active, the deadline curve is updated according to Equation 3.31, which by taking into account only the service received under real-time criterion does not punish a session for using excess service.

$$D_i(a_i^k;t) = \min(D_i(a_i^{k-1};t), S_i(t-a_i^k) + c_i(a_i^k)); \quad t \geq a_i^k \tag{3.31}$$

If a_i is the last time that session i has become active, then the eligible curve represents the maximum amount of service a class can receive under real-time criterion if it is continuously backlogged in the interval $(a_i, t]$ and is defined as

$$E_i(a_i;t) = D_i(a_i;t) + [\max_{\acute{t}>t}(D_i(a_i;\acute{t}) - D_i(a_i;t) - S_i(\acute{t}-t))]^+; \quad t \geq a_i \tag{3.32}$$

The real-time criterion guarantees that the deadline of a packet is not missed by more than the time needed to transmit a packet of the maximum size.

The system's virtual time v_i^s is computed as the mean of the minimal and maximal virtual times of any class by $v_i^s = (v_{i,\min} + v_{i,\max})/2$. Instead of keeping track of the virtual time v_i of each class the algorithm uses a virtual curve V_i which is the inverse function of v_i and computed as

$$V_i(a_i^k; v) = \min(V_i(a_i^{k-1}; v), S_i(v - v_{p(i)}^s(a_i^k)) + w_i(a_i^k)); \quad v \geq v_{p(i)}^s(a_i^k) \tag{3.33}$$

where v is the virtual time, $w_i(a_k^k)$ is equal to the total amount of service received and $v_{p(i)}^s(a_i^k)$ is the virtual time of the parent class.

In [36] the authors also demonstrate an approach how best-effort traffic scheduling can be optimized using H-FSC, avoiding the preferential treatment of long-term continuously backlogged traffic (e.g. file transfers) compared to bursty best-effort traffic (i.e. WWW).

4. Hierarchical Link-Sharing in Wireless LANs

In local area networks based on the IEEE LAN standards, mechanisms for hierarchical link-sharing and fluid fair queuing are implemented as part of the operating system or device driver, simply because the 802.X MAC and LLC do not provide the necessary mechanisms for a controlled sharing of the available bandwidth.[1] An advantage if a hierarchical link-sharing scheduler such as Class Based Queuing (CBQ, see Section 3.4.1) or the Hierarchical Fair Service Curve Algorithm (H-FSC, Section 3.4.3) is part of the operating system is its easy and flexible configurability and independence of specific networking hardware.

An important observation is that by implementing these mechanisms above the MAC layer, the scheduler does not see any MAC layer retransmissions and therefore schedules expected goodput.[2] Since the error probability in a wired LAN is low and all stations experience the same link quality, the assumption that goodput is equal to throughput does not create unfairness. The opposite is true in case of a wireless LAN. As mentioned in Section 3.2, the error probability on a wireless link is significantly higher which results in a larger number of retransmissions, need for adaptive modulation, etc. And since in this case the link quality is location-dependent, scheduling equal goodput to two mobile stations can lead to completely different shares of consumed raw throughput. While this

[1] An exception is the definition of different traffic classes in the ANSI/IEEE 802.1D Standard, 1998 Edition, Appendix H 2.2 which assigns different priorities to different traffic types. Although the future IEEE 802.11e standard will probably support these priorities, priorization does not reduce the need for controlled link-sharing.

[2] In this thesis the term *goodput* is used for the amount of data seen above the MAC layer, whereas the term *raw throughput* is used for the capacity of the wireless link. Although this is an approximation since successful delivery via the link layer is not guaranteed it is a valid estimation because we assume a MAC layer correcting the majority of the wireless link errors e.g. by using retransmissions as in the case of IEEE 802.11.

is desirable in some cases (e.g. in order to be able to compensate a user for a bad link quality) it might be inefficient/unwanted in others. [3]

Problem Example

In order to illustrate the problem this section presents a simplified example scenario which will also be used later on in order to show the benefits of our approach: An Internet Service Provider (ISP) offers services using currently available (e.g. IEEE 802.11) technology on a wireless link with a total capacity[4] of 6Mbit/s. The ISP has installed an access point in order to service two different companies: Company A has bought 85% of the bandwidth for $85 per month and Company B the remaining 15% for $15. For simplicity it is assumed that each company has only one mobile station using the access point, Mobile Station 1 (MS 1) belongs to company A and Mobile Station 2 (MS 2) to Company B. The correct sharing of the bandwidth is to be enforced at the access point using a link-sharing scheduler.

One can now describe the effort needed to transmit data to a mobile destination by using a variable g corresponding to the ratio of the amount of data received at a destination to the necessary resources. Thus, if a mobile station would switch to a modulation enabling it only to receive data at rate of $\frac{1}{11}$th of the full speed, its value of g would change to $g = \frac{1}{11}$. For a link-sharing scheduler implemented above the MAC layer, the only visible effect is a decrease of the total link bandwidth. It would schedule the remaining capacity according the 85% to 15% weights for both companies. This leads to unfairness in terms of costs per used resource for each customer as soon as one assumes different values of g for each mobile station, e.g. if MS 1 is located next to the access point (has a perfect channel, $g_{ms1} \approx 1$) and MS 2 experiences a bad channel quality making retransmissions or adaptive modulation necessary and resulting in a low value of g_{ms2}. Figure 4.1 shows an example[5] where the value of g for MS 2 was varied while MS 1 had a constant value of $g_{ms1} = 1$. As shown in Figure 4.1(b) the relative share of MS 2 of the available resources increases and the goodput for *both* mobile stations decreases (Figure 4.1(a)). Figure 4.1(c) illustrates the influence of g_{ms2} on the amount paid per consumed resource for each mobile station.

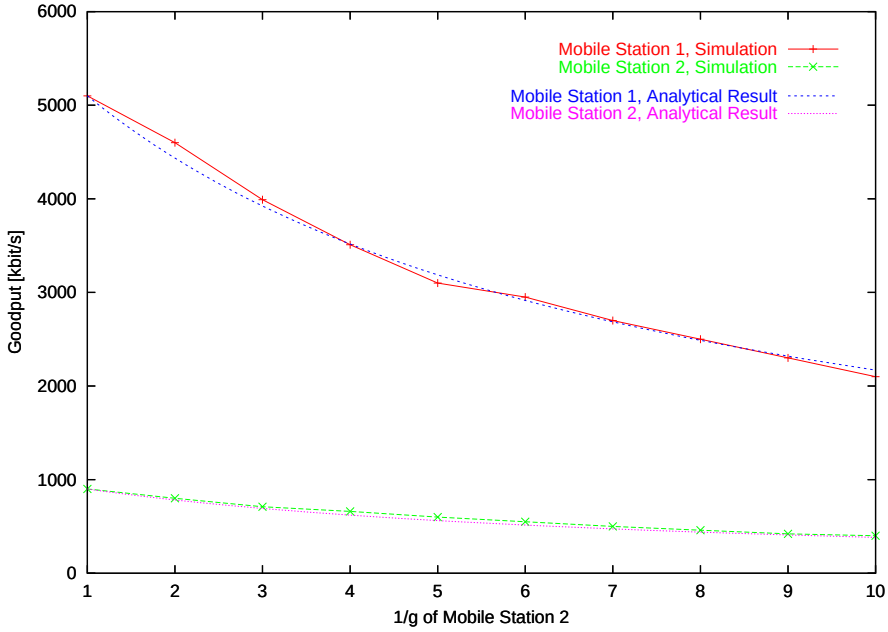
4.1 Purely Goodput-based Wireless Link Sharing

This section presents a brief mathematical analysis of the observed behavior. Most implementations of a link-sharing scheduler for CBQ are based on the DRR algorithm (Sections

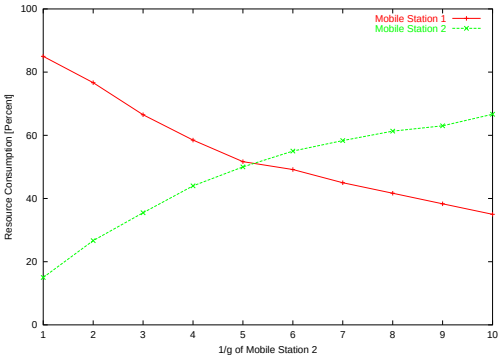
[3] An obvious solution to this problem would be to avoid MAC layer retransmissions and adaptive modulation completely. This is not desirable since most error bursts on the wireless medium have a very short duration [56], [57] and are best handled by MAC layer retransmissions.

[4] Although this is an arbitrary value, it is close to the maximal amount of goodput one usually achieves using 11MBit/s 802.11b WLAN cards.

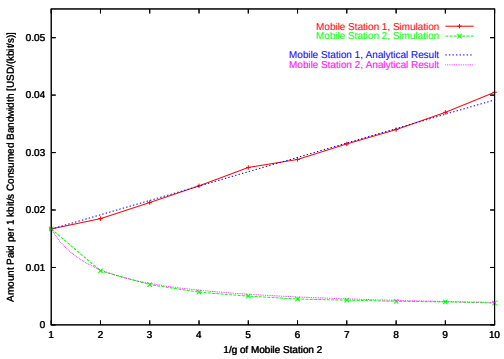
[5] The results were obtained using a simulation of Class-Based Queuing[16] with a separate class for each company. For more details on the simulation environment used refer to Chapter 6.



(a) The goodput of *both* stations is reduced. . .



(b) because MS 2 consumes an increasing share of the available resources.



(c) The amount paid per consumed resource by the customer with a perfect link (MS 1, $g_{ms1} \approx 1$) and the customer with decreasing link quality (MS 2) diverge.

Figure 4.1: *Effect of decreasing link quality of MS 2 ($\frac{1}{g_{ms2}}$ increases) on both mobile stations.*

3.1.3, 3.1.5). During the configuration process of the scheduler the rates assigned to different classes in the hierarchy are mapped to weights for the WRR scheduler. These are used to compute a round-robin schedule which distributes the available bandwidth proportional to the weights. Therefore, if the bandwidth available for the root class is constant and the same as assumed during the configuration process, each class will get its specified rate.

In case that the algorithm is scheduling for a wireless network interface, the available down-link bandwidth (in terms of goodput) and the effort necessary to transmit to different mobile stations vary. If, for simplicity, it is assumed that scheduling is done for N mobile stations belonging to N separate classes which are all at the same level of the hierarchy, Equation 4.1 must hold at any point in time where active classes exist.

$$\sum_{1 \leq i \leq N} \frac{b_{good,i}(t)}{g_i(t)} = B_{raw} \tag{4.1}$$

Here $b_{good,i}(t)$ is the rate of goodput scheduled for a mobile station i and B_{raw} corresponds to the total available resources of the wireless channel. Using the pre-computed WRR schedule the amount of available total goodput $B_{good}(t)$ at time t is still distributed according to the weights w_i of the mobile stations. Thus, if a class i is in the set of active classes $\mathcal{A}(\mathrm{t})$:

$$\frac{b_{good,i}(t)}{B_{good}(t)} = \frac{w_i}{\sum_{j \in \mathcal{A}(\mathrm{t})} w_j}; \quad i \in \mathcal{A}(\mathrm{t}) \tag{4.2}$$

Therefore the total amount of available goodput $B_{good}(t)$ is

$$B_{good}(t) = \frac{B_{raw}}{\sum_i \frac{w_i}{g_i(t)}}; \quad i \in \mathcal{A}(\mathrm{t}) \tag{4.3}$$

and the goodput scheduled for a single station i is:

$$b_{good,i}(t) = w_i \cdot \frac{B_{raw}}{\sum_i \frac{w_i}{g_i(t)}}; \quad i \in \mathcal{A}(\mathrm{t}) \tag{4.4}$$

Since the resource consumption of a mobile station i is equal to $b_{good,i}(t) \cdot \frac{1}{g_i(t)}$ and the amount paid by each customer is a constant C_i, the cost per consumed unit of resource can be calculated as $\frac{C_i \cdot g_i(t)}{b_{good,i}(t)}$. Figures 4.1(a) and 4.1(c) compare the results of the simulation of the example problem to these analytical results.

Summary

The previous example and this analysis demonstrate two effects which are unwanted for the ISP: 1) The behavior of Company A drastically influences the bandwidth available to Company B. 2) Although Company A only pays for 15% of the link bandwidth, it is able to utilize a much larger share of the available resources leading to unfairness in terms of costs per unit resource consumption.

In order to avoid such a scenario, the scheduler needs to be able to take into account the amount of resources (in terms of raw bandwidth) which are needed to provide a specific service to a user. This would enable an ISP to make Service Level Agreements (SLAs) not only based on goodput but also specifying the conditions under which this goodput can be expected, e.g. a user will be guaranteed a rate of 64kbit/s as long as his signal to noise ratio (SNR) is above a specified value. After this point the scheduled rate for him will be reduced in order to limit his consumption of the network resources. Note that such a limitation makes sense only in a situation where not enough bandwidth is available.

4.2 Monitoring per Destination Goodput to Throughput Ratio

A simple way for a packet scheduler above the MAC layer to estimate the raw throughput consumption caused by transmitting a specific amount of data to a destination is the concept of monitoring the goodput to throughput ratio (GTR) per destination. The idea is that an entity, the *Wireless Channel Monitor*, which is either part of the MAC layer, the lower OS layers or even the scheduler itself (a possible implementation is presented in Section 7.2) keeps a windowed average of the raw throughput needed per byte of goodput. The GTR is then used by the scheduler in order to estimate used raw bandwidth when handing data for a specific destination to the MAC layer. The advantage of this approach is that it allows a unified consideration of the effects of various MAC layer techniques as:

- retransmissions
- forward error control (FEC)
- adaptive modulation

The model assumes that the raw bandwidth available is constant for a specific device, e.g. 11MBit/s for a 802.11b WLAN NIC. If the card changes its modulation to transmit at 5.5Mbit/s this change is reflected by a corresponding change of the GTR. In [10], Choi develops a bandwidth management algorithm for adaptive QoS in a CDMA system which is based on the bandwidth usage efficiency of a connection, a property similar to the GTR. However, he assumes that only a number of discrete values can occur for this property and uses a set of adaptation rules in order to react to changes.

4.3 Wireless Link-Sharing Model

This section introduces a wireless link-sharing model which allows the specification of scheduling constraints based on combinations of user-oriented and resource-based criteria. It assumes that the scheduler is able to obtain information about the GTR for each mobile station.

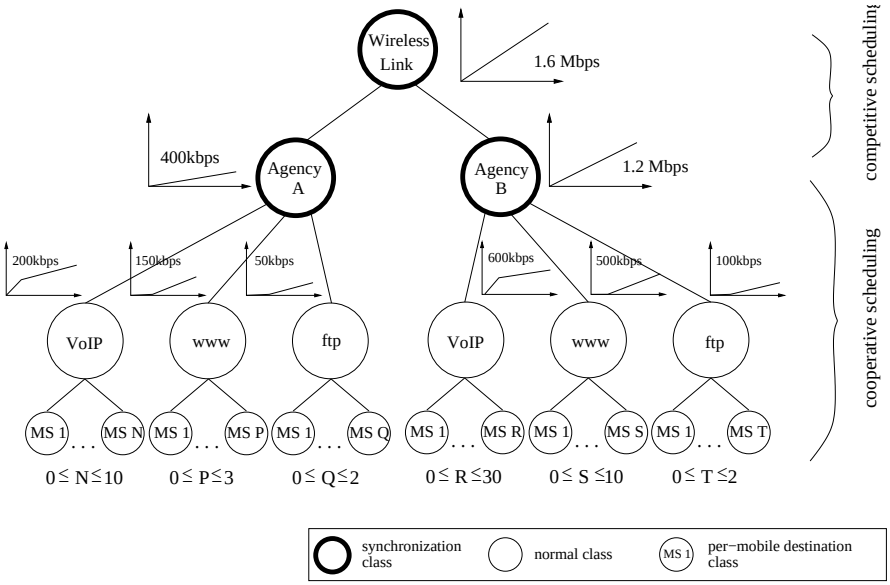


Figure 4.2: *Example for wireless link-sharing hierarchy.*

Agency	Traffic Type	m_1 [Mbit/s]	d [ms]	m_2 [Mbit/s]
A	VoIP	0.030	20	0.020
	WWW	0.000	20	0.050
	FTP	0.000	20	0.025
B	VoIP	0.030	20	0.020
	WWW	0.000	20	0.050
	FTP	0.000	20	0.050

Table 4.1: *Service curve parameters for leaf classes in the example scenario shown in Figure 4.2.*

Wireless Link-Sharing Example

The example (Figure 4.2) used to illustrate the behavior of the wireless link-sharing model and the modified H-FSC algorithm is an extended version of the scenario presented in the introduction of this chapter. Here the wireless link is shared between two agencies, each of them using Voice over IP, WWW and FTP services. It is also assumed that a leaf class has been created for every mobile destination using a specific service. Table 4.1 lists the service curve parameters for the leaf classes. Two piece-wise linear service curves [51] are used: m_1 is the slope of the first segment, d is the x-coordinate where the two pieces intersect and m_2 is the slope of the second segment.

4.3.1 Competitive vs. Cooperative Scheduling

A simple solution to avoid the unfairness caused by different goodput to throughput ratios of mobile stations would be to base the scheduling completely on the resource consumption of a mobile, i.e. by defining the service curve in the raw throughput domain.

However, this would lead to an unwanted behavior when scheduling traffic for different traffic classes within the subtree of each agency. A simple example would be that of two different mobile stations, each with a specified resource share of 20 kbit/s for VoIP. If MS 1 has a goodput to throughput ratio of $g_{ms1} = 1$ and MS 2 has $g_{ms2} = \frac{1}{10}$ and the scheduler has a share of 220 kbit/s of raw throughput available, in a purely resource-consumption based model it will schedule a fraction of 110 kbit/s of the available resources for each mobile leading to excess service for MS 1 and only 11 kbit/s MAC layer goodput for MS 2 causing it to drop its VoIP connection. The desired behavior would be to guarantee the minimum needed service for each mobile station before scheduling excess bandwidth.

Therefore a wireless link-sharing model has to be able to take resource-consumption based constraints *and* goodput based constraints into account. Thus one can define two types of scheduling which need to be supported by the algorithm:

- *Competitive Wireless Scheduling:* Resource-consumption based scheduling between competing entities (e.g. two different customers). Service curves specify the amount of resources in terms of raw bandwidth which the scheduler needs to provide for a class if it has demand. (In the example this corresponds to the scheduling between Agencies A and B.)
- *Cooperative Wireless Scheduling:* Goodput based scheduling between two classes (e.g. within one company). Service curves specify the amount of MAC layer goodput needed by a class. No excess bandwidth is scheduled to any class as long as another class in the subtree is unsatisfied. Excess bandwidth is scheduled independent of the GTR of a class.

4.3.2 Synchronization Classes

In order to integrate both scheduling types in one model, a class which synchronizes both approaches is needed. This type of class is called *synchronization class* in the remainder of this document. A possible algorithmic representation will be presented in Chapter 5, but usually cooperative scheduling is performed in the subtree below the synchronization class and the subtree is integrated in a competitive scheduling environment by using the synchronization class. The root class of the link-sharing tree is always a synchronization class since the raw throughput of the wireless device is assumed to be constant. In the example shown in Figure 4.2, the classes for Agency A and Agency B are both synchronization classes. This guarantees that the first Agency is able to use 75% and the second Agency 25% of the available resources at any point in time if they have sufficient demand.

5. A Wireless H-FSC Algorithm

In this chapter the previously defined wireless link-sharing model is applied the Hierarchical Fair Service Curve Algorithm (see Section 3.4.3).

First the cooperative scheduling within a subtree is described both in situations where enough bandwidth is available and under overload conditions, then competitive scheduling in the modified H-FSC is shown. (Note that it is not possible to completely avoid overload situations if a high utilization of the medium is demanded because of the non-predictable quality of the wireless medium. Roughly speaking, you can always have a user who moves into a position where the link-quality is worse than before, therefore requiring a larger amount of raw throughput to provide the same service.) Finally a simple way to support dropping of outdated packets for real-time classes is shown.

5.1 Cooperative Scheduling

As defined in Section 4.3.1 each class for which cooperative scheduling is performed is part of a subtree which has a synchronization class as its root. Cooperative scheduling within this subtree (e.g. the subtree of which the "Agency A" class is root in Figure 4.2) in situations where enough resources are available (Equation 3.30 is valid for the subtree) is similar to the behavior of an unmodified H-FSC scheduler: Service is provided without taking destination specific resource-consumption into account. The virtual time of a class within the subtree which determines the distribution of bandwidth using the link-sharing criterion corresponds to the amount of data handed down to the MAC layer. Implicitly, stations experiencing a bad link are allowed to use a larger share of the raw bandwidth in order to be able to compensate the low link quality e.g. by doing FEC, retransmissions or using a lower transmission rate.[1] Therefore, compared to an unmodified H-FSC scheduler

[1]The scheduler is only able to guarantee delay constraints with an accuracy of the time needed to transmit one maximum length packet via the MAC layer (including retransmissions, FEC, back-off etc). Various MAC layer mechanisms are proposed in order to provide differentiated services in the MAC layer, these are out of the scope of this paper but could be used in order to be able to give tighter delay guarantees.

implemented above the MAC layer the only difference is that the goodput scheduled is based on the amount of resources available for this subtree.

In an overload state the amount of resources available for a subtree s is insufficient to guarantee all service curves since it is less than the sum of the eligible curves of all active sessions in the subtree $\sum_{i\in\mathcal{A}_s(t)} E_i(a_i;t)$. Since the root of a cooperative scheduling subtree is a synchronization class, the service curve is resource-based and the total amount of resources available for the subtree within an arbitrary interval $(t_1,t_2]$ is $S_s(t_2)-S_s(t_1)$. Under overload these resources are completely consumed and the following equation must hold for all active leaf classes $\mathcal{A}_s(t)$:

$$S_s(t_2)-S_s(t_1)=\sum_{i\in\mathcal{A}_s(t_1)}\frac{1}{g_i\cdot d_i}\cdot(E_i(a_i,t_2)-E_i(a_i,t_1));\quad \mathcal{A}_s(t_1)=\mathcal{A}_s(t_2) \tag{5.1}$$

Here d_i is a class specific factor by which the service for this class is reduced and the goodput to throughput ratio $g_i(t)$ is approximated to the constant g_i within the interval $(t_1,t_2]$. If, analogously to the approach in the goodput domain, the raw bandwidth share of each class is to be degraded by a constant factor, then d_i has to be determined in a way that $\frac{1}{g_i\cdot d_i}=const$. The solution is to determine the d_i for each class as the fraction of the required GTR over GTR of the specific leaf class:

$$d_i=\frac{(\frac{K}{S_s(t_2)-S_s(t_1)})}{g_i}$$

where

$$K=\sum_{i\in\mathcal{A}_s(t_1)}(E_i(a_i,t_2)-E_i(a_i,t_1)) \tag{5.2}$$

Whereby $\frac{1}{g_i\cdot d_i}=\frac{S_s(t_2)-S_s(t_1)}{K}$ and the condition given in Equation 5.1 is satisfied.

Roughly speaking, this approach reduces the (goodput) service of each class in a way that the raw bandwidth available to each class is distributed proportional to the service curves and guarantees that the subtree consumes only the amount of resources available to the synchronization class which is its root.

5.2 Competitive Scheduling

The modified H-FSC scheduler performs competitive scheduling based on raw throughput (resource-consumption) among synchronization classes. Note that although they usually are root of a cooperative scheduling subtree as shown in Figure 4.2 they could also be leaf classes, e.g. if the desired behavior is to provide a fixed amount of resources to a leaf class. Since the algorithm defined in the previous section guarantees that a synchronization class is never allocated more than the amount of resource specified by its service curve by

using the real-time criterion, only the sharing of bandwidth using the link-sharing criterion needs to be considered. In contrast to cooperative scheduling, now the virtual time of a class is based on the resource consumption by incrementing it by the product of $\frac{1}{g_i}$ and the packet size whenever a packet is transmitted for a class. The following example illustrates the incrementation of virtual times: If one assumes that subclass "MS 1" of Agency B in Figure 4.2 transmits a packet of size s_p using the link-sharing criterion on a link with $g_{ms1} = \frac{1}{10}$, the virtual time of the class itself and of the VoIP class will be incremented by s_p and the virtual time of Agency B and of the wireless link root class will be incremented by $\frac{1}{g_{ms1}} \cdot s_p = 10 \cdot s_p$.

5.3 Delay

The original, wireline H-FSC algorithm guarantees that the deadline for a packet specified by the service curve is not missed by more than τ_{max} which is the time needed to transmit packet of the maximum size (Section 3.4.3). A similar guarantee can be derived for the wireless variant if the usage of an optimal channel monitor is assumed which has perfect knowledge of the GTR g_i for a mobile station i: If the time to transmit a packet with maximum length at the minimum GTR g_{min} is $\tau_{max,g_{min}}$, the deadline determined by the reduced service curve $\frac{1}{d_i} \cdot S_i(\cdot)$ is never missed by more than $\tau_{max,g_{min}}$.

In case that a suboptimal channel monitor is used, the maximal amount of time by which a deadline can be missed depends on the error of estimation of g_i and the delay with which a change of the channel quality is reflected by the monitor. (An example calculation is in Section 7.1.1.)

5.4 Packet Dropping

For a real-time class the reduction of its rate in an overload situation can lead to an unwanted accumulation of packets in its queue increasing the delay of each packet. For this traffic class a mechanism is needed which allows the dropping of outdated packets allowing the usage of the remaining capacity for packets which have not exceeded their maximal delay. A simple way to integrate this behavior in the H-FSC scheduler is the concept of a packet drop service curve. It is used to compute a hard deadline after which a packet is outdated and will be dropped from the queue in the same way that the transmission deadline is calculated using the regular service curve.

6. Implementation

This chapter describes the design and implementation details of a wireless scheduling architecture for the Linux traffic control environment. Although it was mainly developed in order to evaluate the wireless link-sharing model presented in Chapter 4 and the modified H-FSC scheduling scheme developed in Chapter 5 its use is not restricted to this specific algorithm. Unless denoted otherwise the given informations are valid for the simulation as well as for the prototype implementation. The remainder of the chapter is organized as follows: At first Section 6.1 briefly gives the necessary background information on the most relevant parts of the Linux kernel, introduces the various components of the Linux traffic control framework and presents a simple example for using it in a wire-line environment. Then our extensions for supporting wireless scheduling based on long-term channel state information within this framework are shown in Section 6.2. Finally implementation details of the modified H-FSC scheduler and the wireless channel monitor are given in Sections 6.4 and 6.5.

6.1 The Linux Traffic Control Framework

Linux is an open source UNIX variant which was started by Linus Torvalds in 1991 and has been continuously developed by the open source community since then. Its source code is freely available under the GNU License.[1] This and the fact that Linux is know for its highly flexible networking code which allows the configuration of state-of-the-art firewalls and routers (including support for virtually every known networking protocol, IP Masquerading/Network Address Translation, DiffServ, RSVP, etc.) were the main reasons that it was chosen as the basis for the implementation of the wireless scheduling testbed.

The part of the Linux operating system which is responsible for all bandwidth-sharing and packet scheduling issues is called Traffic Control (TC) and has been available since kernel 2.1.90. Beginning with the late 2.3 kernels it became part of the standard kernel.

[1] The GNU General Public License is published by the Free Software Foundation. License and Linux sources can be downloaded at [23].

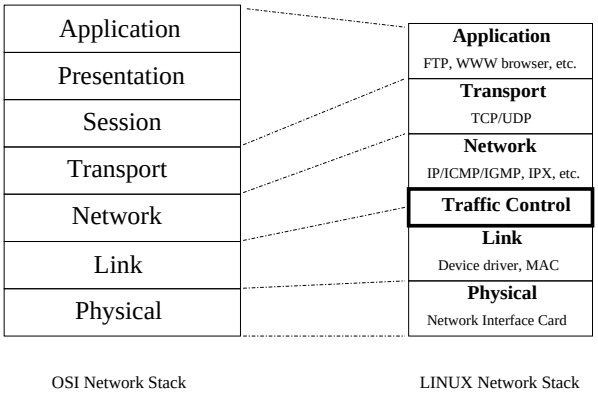


Figure 6.1: *Linux networking and the OSI reference model*

This description is based on kernel 2.4.13 which was the most recent kernel version at the time when this text was written.

Most parts of the kernel 2.4.13 traffic control (including Integrated Services and RSVP) were written by Alexey Kuznetsov [25], later support for Differentiated Services was added by Werner Almesberger [2]. Although Linux TC focuses on output traffic shaping, it has basic support for ingress traffic policing.

As in most UNIX operating systems, the implementation of the Linux networking code follows a simplified version of the OSI reference model [50] as shown in Figure 6.1. While the original Linux networking code was derived from BSD Unix, it was completely rewritten for the 2.2 kernel in order to be able to provide a better performance. Of the shown layers, only the data link layer, the network layer and the transport layer functions are executed in kernel space, the application layer protocols and the application itself are running in user space. The packet scheduling and traffic control code is implemented below the network layer directly above the device driver. Thus different network layer protocols as IP, IPX and Appletalk can be supported and the scheduler can reorder the packets right before they are sent down to the network interface card. Although thus the traffic control is implemented at the lowest level of the operating system which is not yet device specific (as the device driver) there are many situations - especially in a wireless environment - where TC could benefit from scheduling support in the lower layers.

Figure 6.2 shows the path of a data packet through the Linux network stack. When a packet arrives at a network interface and is passed to the network layer it can optionally be classified/tagged and policed (e.g. it will be dropped if a certain rate is exceeded). Then it is either passed to the higher layers or to the IP forwarding part where the next hop for the packet is determined. When it reaches the egress part of traffic control it is classified (possibly taking tags of the ingress classifier into account) and scheduled for transmission. The scheduling decision can include dropping of the packet, delaying it, immediately sending it and so on. A comprehensive discussion of the implemented components can be found in [1] and [42], the DiffServ components are presented in a separate paper by Almesberger [2].

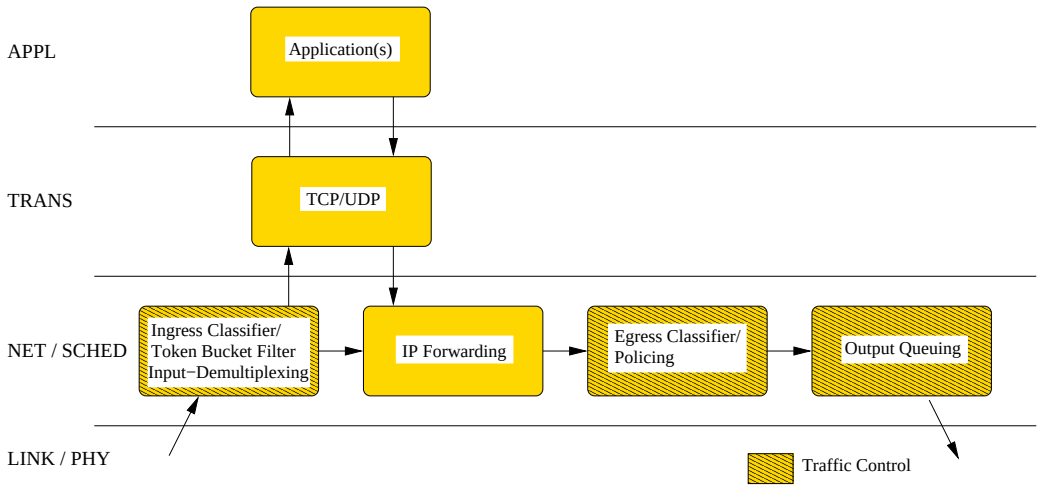


Figure 6.2: *Parts in the network stack where traffic control is involved*

6.1.1 Components of Linux Traffic Control

The high flexibility of the Linux traffic control architecture is achieved by providing a set of quality of service and link-sharing modules which can be combined in multiple ways to support the scheduling requirements needed at a specific node. These kernel modules can be configured with a user space control program which communicates with the kernel by using a special kind of network socket (called *netlink socket*). Besides passing parameters (e.g. bandwidth requirements) to the kernel the user space control program also specifies the structure of the specific QoS architecture at runtime of the node.

The basic components of the Linux QoS architecture are:

- queuing disciplines
- classes
- filter

All more complex scheduling setups are implemented by composing these three types of components in a way that the desired behavior is achieved. In the following after introducing the different kinds of components an example scenario is presented (including the necessary command options for the `tc` traffic control program) which illustrates the ways in which they can be combined.

6.1.1.1 Queuing Disciplines

Queuing Disciplines (*qdiscs*) have an enqueue and a dequeue function and perform reordering of the stored packets. In general the enqueue function is called whenever the network layer of the operating system wants to transmit a packet and the dequeue function is called when the device is able to transmit the next packet. A simple example is a single priority FIFO queue which would accept packets and dequeue them in the same

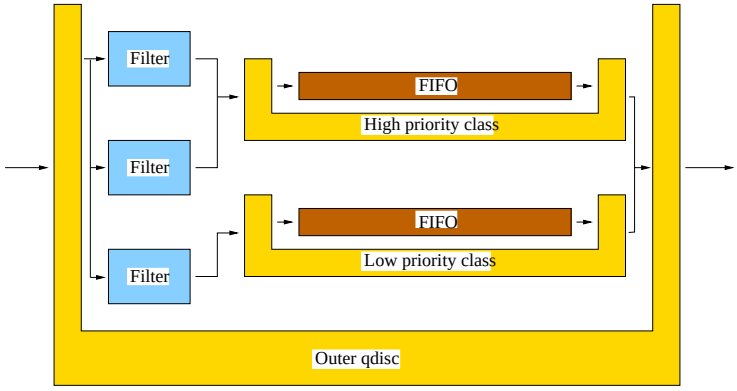


Figure 6.3: *Simple example of a qdisc which has inner classes, filters and qdiscs*

order. But qdiscs can also act as a container having filters and classes. A filter is a module which is used in order to determine the class of a packet. A class is a logical group of packets which share a certain property. What makes this concept very flexible is that a class usually again uses another queuing discipline in order to take care of storing the packets. In the example of a multi-priority queue a packet arriving at the qdisc would first be passed to a set of filters. These would return the class (in this case identical to the priority) of the packet. Each class then could use a single priority FIFO queue in order to actually store the packet.[2]

The available queuing disciplines can be divided in two groups: the simple qdiscs which have no inner structure (known as *queues* in the Linux TC terminology) and those which have classes (*schedulers*). The first group includes

- **pfifo_fast:** A 3-band priority FIFO queue. Packets of the lowest band will always be sent first, within each band usual FIFO scheduling is done. This queue is the default queue and is attached to a newly created class.
- **sfq:** A stochastic fair queuing discipline as explained in 3.1.4.
- **tbf:** A Token Bucket Filter (TBF) queue which passes packets only at a previously specified rate but has the possibility to allow short bursts.[3]
- **red:** Implements the Random Early Detection (RED) [17] behavior which starts dropping packets before the queue is completely filled in order to get TCP connections to adapt their transmission rate.
- **gred:** A generalized RED implementation used for DiffServ (see Section 2.2) support.

[2]This simple example is used to illustrate the interaction of modules. In reality it would be more efficient to use the standard 3-band priority queue (pfifo_fast) if 3 or less priority levels are to be managed.

[3]The model is that of a constant stream of tokens which ends in a bucket. Each packet needs a token to be dequeued - if the bucket is empty, meaning that the flow sends at a higher rate than allowed, the packet is delayed until a new token is generated. If the flow currently transmits at a rate lower than the rate at which tokens are generated, the surplus tokens are stored up to a specified limit (“the depth of the bucket”) and can be used at a later time.

- **ingress:** A queue used for policing ingress traffic.

The following queuing disciplines make use of classes

- **cbq:** Implementation of the class based queuing link-sharing scheme as described in Section 3.4.1.
- **atm:** A special qdisc which supports the re-direction of flows to ATM virtual channels (VCs). Each flow can be mapped in a separate VC or multiple flows can share one VC.
- **csz:** A Clark-Shenker-Zhang [47] scheduling discipline.
- **dsmark:** This queuing discipline is used for Differentiated Services (see Section 2.2, [2]) support. It extracts the DiffServ Codepoint (DSCP) which indicates the desired per-hop-behavior (PHB) and stores it in the packet buffer. After the packet has passed the inner classes, the DSCP is updated and the packet is sent.
- **wrr:** A Weighted Round Robin (WRR) scheduler[4] as explained in Section 3.1.3.

Each qdisc is assigned an unique id $X:Y$ composed of a major number X and a minor number Y. The root queuing discipline of a device always has the minor number 0.

Interface

As mentioned before the most important functions of a scheduler/qdisc are the enqueuing and dequeuing operations. Both work on the so-called `skbs` which are the Linux network buffers. Basically one can think of an `skb` as a structure which contains a packet with all of its headers and control information for handling it inside the network stack. Most operations also have a pointer to the scheduling discipline itself as an input, so that the function can access the private data of this instance of the queuing discipline. (This is a simple way to approximate an object oriented design using standard C.) All methods which are sending or receiving configuration data to/from user space also have a pointer to the Netlink buffer as parameter.

The interface of a queuing discipline is held in a structure `Qdisc_ops` (for qdisc operations) whose members are listed in Table 6.1.

6.1.1.2 Classes

A qdisc that supports classes usually has one root class which can have one or more subclasses. A class can be identified in two ways: Either by the user assigned *class id* which is composed of a major number corresponding to the qdisc instance it belongs to and an unique minor number or by its *internal id*. A class with one internal id can have one or more user assigned class ids. When a class is created, it has a **pfifo_fast** queuing discipline for storing its packets. This can be replaced by doing a so-called

[4]This queuing discipline is not part of the standard kernel but can be downloaded at [34].

Name	Description
`next`	A pointer to the next qdisc in the list.
`cl_ops`	A pointer to the class operations implemented by this queuing discipline. (see 6.1.1.2)
`id`	The unique name of this kind of queuing discipline.
`priv_size`	The size of the private data area.
`enqueue(*skb, *qdisc)`	The enqueue function.
`dequeue(*qdisc)`	The dequeue function, returns the dequeued skb.
`requeue(*skb, *qdisc)`	Re-enqueues a `skb` at the front of a queue when the network interface was unable to send the packet.
`drop(*qdisc)`	Drops a packet from the queue.
`init(*qdisc, *arg)`	Initializes the queuing discipline.
`reset(*qdisc)`	Resets the state.
`destroy(*qdisc)`	Frees all memory held by the qdisc.
`change(*qdisc, *arg)`	Changes the parameters.
`dump(*qdisc, *skb)`	Returns statistics.

Table 6.1: *Interface of a scheduler/qdisc in the Linux traffic control framework*

graft operation which attaches a different queuing disciple to the class. A class also supports a bind operation which is used to bind an instance of a filter to a class. Although classes and schedulers are two separate components, the methods for handling them are usually implemented in one kernel module. In contrast to that a filter is in most cases an independent module.

Interface

The methods offered by the interface of a class can be divided in two functional areas: Modifying the properties of the class itself and attaching/detaching filters and inner queuing disciplines. The most important of the first section is the `change` operation which is used in order to modify the properties of a class. If the class to be changed does not exist, it is created. Important functions of the second group are the `bind_tcf` function which binds a traffic control filter to a class and the `graft` operation. Table 6.2 gives an overview of all supported operations.

6.1.1.3 Filter

Queuing disciplines and classes use filters in order to classify incoming packets. They have a priority-ordered list of filters for each protocol which the packet is passed to until one of the filters indicates a match. Filters for the same protocol must have different priorities.

The following filter types are supported:

Name	Description
`graft(*sch, intid, *sch_new, *sch_old)`	Used to change the inner qdisc of a class.
`leaf(*sch, arg)`	Returns the leaf qdisc of a class.
`get(*sch, classid)`	Returns the internal id of a class and increments the classes usage counter.
`put(*sch, intid)`	Decrements the usage counter of a class. If no one uses the class anymore, it is destroyed.
`change(*sch, classid, parentid, **netlink, intid)`	Changes the properties of a class. (priority, bandwidth, etc.)
`delete(*sch, intid)`	Deletes the specified class.
`walk(*sch, *walker)`	Traverses all classes of a scheduler and invokes the specified function on each class.
`tcf_chain(*sch, arg)`	Returns the list of filters bound to this class.
`bind_tcf(*sch, classid)`	Binds a filter to a class.
`unbind_tcf(*sch, intid)`	Removes a filter from a class.
`dump(*sch, ...)`	Returns statistics for a class.

Table 6.2: *Interface of a class*

Name	Description
`*next`	Pointer to the next filter in the list.
`*id`	An unique id of the filter.
`classify(*skb, ..., *result)`	Classifies a network buffer.
`init(*tcf)`	Initializes a filter.
`destroy(*tcf)`	Destructor of a filter.
`get(*tcf, handle)`	Returns the internal id of a filter and increments its usage counter.
`put(*tcf, intid)`	Decrements a filters usage counter.
`change(*tcf, ...)`	Changes the properties of a filter.
`delete(*tcf, ...)`	Deletes a specific element of a filter.
`walk(*tcf, *walker)`	Traverses all elements of a filter and invokes the specified function on each element.
`dump(*tcf, ...)`	Returns statistics for a filter.

Table 6.3: *Interface of a filter.*

1. *Generic filters* can classify packets for all classes of a queuing discipline.

   - **cls_fw:** The packet is classified upon a tag assigned by the firewall/netfilter code [45].
   - **cls_route:** The packet is classified using routing table information [20].

2. *Specific* filters only classify packets for one specific class.

   - **cls_rsvp:** A filter for RSVP support (Section 2.1).
   - **cls_u32:** This filter classifies packets according to an arbitrary byte pattern in the packet header or payload.

When a new packet is to be enqueued in a class or qdisc, the filters bound to the class for the specific protocol of that packet are called in order of decreasing priority. (Lower priority values indicate higher priority.) The first filter which indicates a match classifies the packet.

Interface

The central function offered by the filter interface is the `classify` function which determines the class of a network buffer (packet). All the other operations are used to manage the assignment of filters to classes and their properties. Filters are stored in filter lists, each record in this list holds the information for a specific instance of a filter, for example the priority, the class id, and a pointer the structure with filter operations. These functions are listed in Table 6.3.

6.1.2 Managing and Configuring Linux TC

As stated in the introduction of this section, traffic control is part of all newer Linux kernels, so there is no need to apply any special patches if one is using a kernel of version 2.3.X or newer. The only precondition for using scheduling mechanisms is to ensure that the corresponding configuration constants for the desired traffic control components were set when the kernel was compiled. (A complete listing of all options can be found in Appendix A.2, Table A.2.)

Since traffic control is part of the kernel but has to be managed by programs running in user space, a standardized way was developed in which user programs can communicate with the operating system kernel: The information is encapsulated in special message format (Netlink-message) and sent on a special socket interface called Netlink-socket. The kernel receives and analyzes the message and answers with a confirmation message on the socket. Since each functional part of the kernel needs different parameters, a specific protocol is used for each area. In order to manipulate routing and traffic control information the *RTNetlink* protocol is used.

Although basically any user space program can generate Netlink messages (if it has the necessary permissions), the configuration program usually used for configuring the QoS components is the traffic control program `tc` which is part of the `iproute2` package [25]. Because of space limitations this document will not explain the detailed usage of `tc` (an interested reader will find more information in [20]) but the following example should make the basic principles clear and illustrate the combination of the basic modules to form a scheduling configuration for a specific purpose.

6.1.2.1 A Simple Example for Wire-line Traffic Control

A small company has a 10 Mbit/s link which connects its workstations and one FTP server to an Internet service provider. The FTP server is used to allow customers and employees to download and upload files via the Internet. The network is illustrated in Figure 6.4. Since bandwidth is a scarce resource the company wants to limit the share of the FTP traffic to 20 percent and at times where less bandwidth is needed by FTP the rest should be available for the workstations. On the other hand FTP traffic must never exceed its 20 percent share even if the rest of the bandwidth is currently unused because the companies ISP charges extra for any bandwidth consumed above a rate of 2 Mbit/s. In order to solve this problem, a Linux router is installed at the edge of the corporate network.

The first Ethernet interface of the Linux router (`eth0`) is connected to the ISP, the second interface (`eth1`) is the link towards the internal network. Since it is only possible to limit outgoing traffic, the setup consists of two parts: The CBQ configuration limiting the outgoing traffic on `eth0` (the “downstream” traffic from the internal network’s point of view) and a second part limiting outgoing traffic on `eth1` (the “upstream”).

The following `tc` commands would be used to configure the router for correct sharing of the link in the “upstream” direction:

1. At first the root queuing discipline currently attached to the network interface is deleted (not necessary if there is none attached yet):

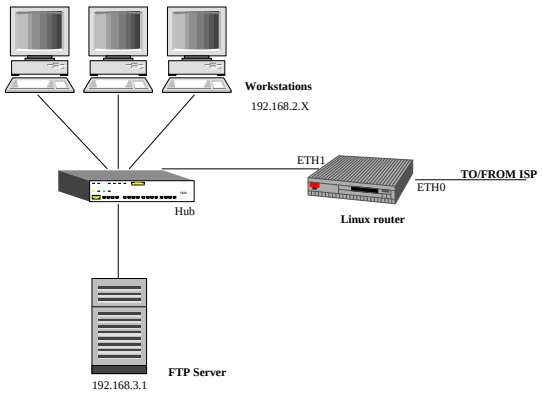


Figure 6.4: *Example network for wire-line bandwidth sharing*

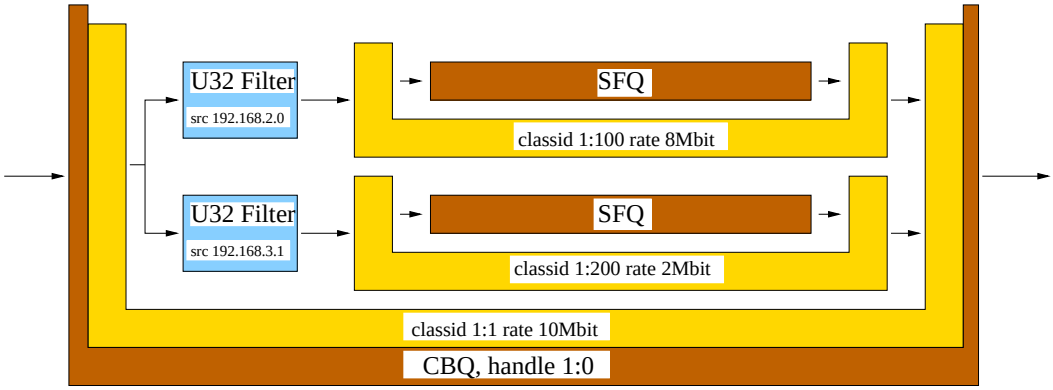


Figure 6.5: *The configuration of network interface eth0 in the example network*

```
# tc qdisc del dev eth0 root
```

2. Then a root qdisc is added which uses the class-based queuing scheduler:

```
# tc qdisc add dev eth0 root handle 1:0 cbq bandwidth 10MBit \
  avpkt  1000
```

3. The next step is to create a root class which contains two subclasses, one for the FTP traffic and one for the traffic coming from the workstations:

```
# tc class add dev eth0 parent 1:0 classid 1:1 cbq \
  bandwidth 10 MBit rate 10MBit allot 1514 weigth 1Mbit \
  prio 8 maxburst 20 avpkt 1000

# tc class add dev eth0 parent 1:1 classid 1:100 cbq \
  bandwidth 10MBit rate 8MBit allot 1514 weight 800kbit prio 5 \
  maxburst 20 avpkt 1000 isolated

# tc class add dev eth0 parent 1:1 classid 1:200 cbq \
```

```
bandwidth 10MBit rate 2MBit allot 1514 weight 200kbit prio 5 \
maxburst 20 avpkt 1000 bounded
```

The keyword `isolated` indicates that the workstation class does not borrow any bandwidth to other classes, the `bounded` option means that the class may never exceed its limit. If both keywords were omitted, surplus bandwidth would be shared in proportion to the weights of the classes, which could also be a desired behavior in our scenario.

4. Now the two CBQ subclasses are assigned a stochastic fair queuing discipline:

```
# tc qdisc add dev eth0 parent 1:100 sfq quantum 1514b perturb 15

# tc qdisc add dev eth0 parent 1:200 sfq quantum 1514b perturb 15
```

5. And the final step is to bind two U32 filters to the root qdisc, which decide based on the source address of the IP packet which class a packet belongs to. (For simplicity we assume that the FTP server is purely used for FTP services.)

```
# tc filter add dev eth0 parent 1:0 protocol ip prio 100 u32 \
  match ip src 192.168.2.0/24 flowid 1:100

# tc filter add dev eth0 parent 1:0 protocol ip prio 100 u32 \
  match ip src 192.168.3.1 flowid 1:200
```

In Figure 6.5 the created queuing configuration is shown schematically. The commands for setting up the “downlink” traffic scheduling are almost the same except that the filtering would be based on the destination IP address.

6.1.3 Traffic Control Next Generation (TCNG)

The Traffic Control Next Generation (TCNG) [1] project focuses on providing a more user-friendly configuration language and supporting hardware accelerators in traffic control. The two major components being developed are the Traffic Control Compiler (TCC) and the Traffic Control Simulator (TCSIM). Since this is work in progress only the basics will be presented in order to show the impact which this project has on our work.

The concept of the Traffic Control Compiler is to accept a wide variety of input languages (from user-friendly configuration languages to machine-friendly languages like XML) and to convert them to a unified internal data structure. This information is then sent to a variety of output processors e.g. `tc` using a Netlink socket to configure the kernel or a hardware accelerator configuration program. An output processor could also generate a new kernel module and provide the configuration data for it.

TCSIM is a event-driven simulation environment for Linux traffic control components. It operates on the original kernel code and allows to test the almost unmodified source code of a traffic control component while still running in user space. Event traces can be

generated and analyzed. By using the `tc` interface to configure the simulated modules, the communication via Netlink protocol can also be tested. Although TCSIM is part of TCNG it is able to simulate components running in the current TC architecture as well. Since the simulations are performed using almost exactly the same code that is used in the Linux kernel, the results are closer to the actual system behavior than those of an abstracted simulation model. The drawback of the TCSIM simulation environment is that it has only a vary basic support for traffic generation (only constant bit-rate) and statistical analysis.

6.1.4 TC API (IBM Research)

An alternative approach to configuring Linux traffic control is currently being developed by IBM Research. Instead of using the `tc` command line tool they propose a new traffic control Application Programming Interface (API) called *TC API*[39].[5] It provides a library which enables a user space application to configure the traffic control architecture of a host using a set of API functions. These generate the necessary netlink messages and send them to the kernel using a netlink socket. When the results are received, they are forwarded to the application as the return values of the called API function. By simplifying the development of QoS management applications for Linux in this way (since the API avoids the need to generate and parse `tc` arguments) they hope to encourage further research and development activity w.r.t. supporting QoS functionality on Linux.

6.1.5 Problems When Applying Standard Linux Traffic Control to Wireless Networks

While the Linux traffic control architecture is capable of supporting virtually all important wire-line scheduling scenarios (e.g. IntServ, DiffServ, MPLS, etc.) and many wire-line schedulers have been implemented, it has several disadvantages when used in a wireless environment. Taking the assumptions made by most wireless scheduling schemes presented in Chapter 3 into account, the main problems are:

1. Linux TC is an example for a scheduling architecture implemented above the MAC layer. Therefore, as described in Chapter 4, bandwidth regulation mechanisms are based on the assumption that the transmission medium has a constant bandwidth and are not able to take destination-specific resource-consumption into account.

2. Most wireless scheduling algorithms assume perfect knowledge of the channel-state. The concept of a channel monitor is completely missing in Linux TC and therefore a queuing discipline has no access to information on state of the wireless channel to a specific mobile station.

3. Queuing disciplines specifically developed to support scheduling in a wireless environment are not available.

[5] Version Beta 1.0 of the API is available since Oct. 2001 at [39].

In order to be able to evaluate the modified H-FSC algorithm described in Chapter 5 within the Linux traffic control framework it was extended to support long-term channel-state dependent and resource-consumption based scheduling. These modifications will be outlined in the following section.

6.2 Extending Linux Traffic Control for Wireless Scheduling

This section presents the extensions made to the existing Linux traffic control framework in order to be able to support the implementation of wireless scheduling. The main objectives for the design were:

- Support for the implementation of a wide range of wireless scheduling algorithms.
- Integration in the current framework:
  - Re-use existing parts/interfaces whenever possible.
  - Configuration of wireless scheduling components with the same mechanisms as used in wire-line setups (`tc`, netlink socket).
  - Usage of similar (internal) interfaces.
- Separation of channel monitoring and scheduling functionality.
- Minimize the necessary hardware dependent changes, i.e. the modification of device drivers.

Therefore a new component, the *wireless channel monitor*, was introduced, which is a loadable kernel module similar to the qdisc and filter/policier modules. But unlike the latter ones only maximal one wireless channel monitor is active per wireless device at any point in time. (The reason for this is that in many cases an accurate estimation of the channel conditions is not possible if two or more monitors are concurrently running on one device because each of them only sees a partial amount of the transmission attempts.) The channel monitor estimates the channel quality and capacity towards each mobile destination and keeps track of the last probing attempt (usually the last time data was sent to a destination). The way in which the monitor acquires its knowledge is implementation specific. But although it is possible to monitor channel qualities in a device independent way (e.g. by monitoring the time intervals between two dequeuing events) in general much more accurate estimations can be made by using information available in the device driver. Therefore a channel monitor can offer an interface on which a modified device driver can pass information it has about the channel state.

The channel monitor provides its information via a new interface towards the (wireless) qdisc. Therefore, whenever the queuing discipline needs information about the channel-state for scheduling a packet transmission, it requests it from the channel monitor. Because in this way the scheduling is decoupled from channel monitoring, the monitoring

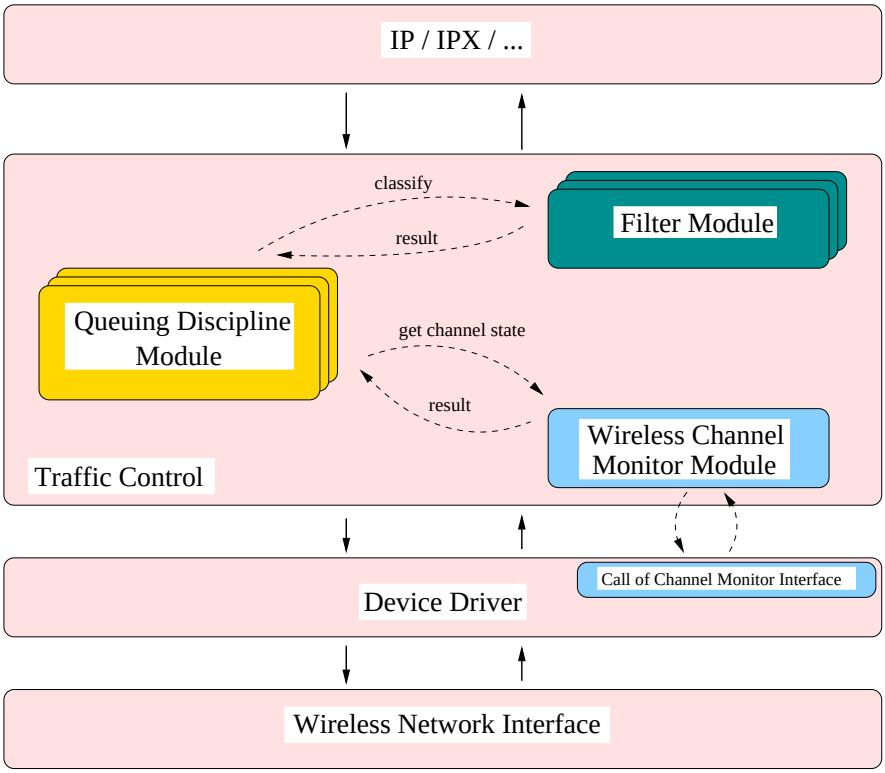


Figure 6.6: *Schematic overview of extended traffic control architecture.*

component can be replaced (e.g. if a wireless MAC supporting channel quality monitoring via RTS/CTS is available) without any modifications to the scheduler itself.

Figure 6.6 illustrates the extended architecture.

Three different types of wireless channel monitors were implemented:

- **dummy:** A dummy module which returns constant channel quality values for a mobile destination.
- **ratio:** A channel monitor which estimates the quality of a wireless channel by measuring the time between two dequeue events. This is not very accurate because of the interrupt driven nature of the dequeue event and the fact that most wireless devices use an internal buffer for storing multiple packets. The advantage of this method is that it does not require any modifications of the device driver.
- **driver:** This channel monitor implements the additional interface towards a wireless device driver in order to obtain the status information (for details see Section 6.5).

6.2.1 Channel Monitor Interfaces

Since the wireless channel monitor component needs to interact with its environment, three new interfaces were introduced: a kernel/channel monitor module interface, a chan-

nel monitor/scheduler interface and an interface between a channel monitor and a (modified) device driver. In addition the kernel/scheduler interface was extended to support the assignment of a channel monitor to a scheduler.

Like the interface of other components, the interface of a wireless scheduler is encapsulated within a characteristic structure of the type `wsched_channel_monitor`[6] which is shown in Table 6.4.

Scheduler Interface

This is the main interface of a wireless channel monitor. Most important are the `get_channel_state` and `get_capacity` functions with which a scheduler can get information about the quality/estimated goodput rate currently available to a specific mobile estimation. Since the channel monitor needs to keep track of transmission events, a wireless scheduler is also required to inform the channel monitor about the start/end of a transmission by using `start_transmit` and `end_transmit`. (The end of a transmission from the wireless schedulers point of view occurs when the next packet is dequeued.)

Kernel Interface

The main purpose of the kernel interface is to allow the registration of a monitor module once it is available and the scheduler initiated assignment of a monitor to a wireless device.[7] When a wireless channel monitor module is loaded (e.g. by issuing the `insmod` command) it will register itself using the `register_wsched_channel_monitor` function. After that point in time a wireless scheduler can request the assignment of this channel monitor to a device by using the `wsched_wchmon_add` call. When the scheduler does not need the monitor anymore (e.g. because it itself is removed/deactivated or because the user requested a different channel monitor) it signalizes this using `wsched_wchmon_del`. If a channel monitor is not in use, it can be removed (with an `rmmod` command). This causes the module to call the `unregister_wsched_channel_monitor` function which removes it from the kernels list of available monitors. Table 6.6 gives an overview of the exported functions.

Device Driver Interface

The device driver interface for channel-state signaling consists of only two functions: `wsched_data_link_transmit_status` and `wsched_data_link_receive_status` (Table 6.5). The first signals the result of a transmission attempt (many cards report the success/failure of a transmission in an interrupt when they are able to process the next packet). Unfortunately - since the card can have an internal buffer - this result may not correspond to the last packet handed down but to a previous packet. By taking the device buffer into account and averaging a channel monitor can still obtain some useful status information. The second function is used to signal link-status information received from the MAC controller, which many wireless cards report after the reception of a packet. This is usually the most reliable information currently available.

[6] defined in `../net/pkt_wsched.h`

[7] The functionality of this interface is implemented in the file `../net/sched/wchmon_api.c` of the kernel source tree.

Name	Description
`*id`	An unique identification string of this monitor.
`*next`	Pointer to the next monitor (if in list).
`init(*mon)`	Initializes a monitor.
`destroy(*mon)`	Destructor.
`skb2dst(*mon, *skb, *dst)`	Return destination id of an `skb`.
`get_channel_state (*mon, *dst)`	Returns the channel-state for the specified destination mobile.
`reset_channel_state (*mon, *dst)`	Resets all channel-state information.
`start_transmit (*mon, *dst, size)`	Signals the start of a transmission to the channel monitor.
`end_transmit(*mon, *dst)`	Signals the end of a transmission.
`abort_transmit(*mon, *dst)`	Transmission was aborted.
`channels_idle(*mon)`	Signals that scheduler has nothing to send.
`get_capacity(*mon, *dst)`	Get information about current channel capacity towards a destination (goodput in byte/s).

Table 6.4: *Channel monitor/scheduler interface.*

Name	Description
`data_link_transmit_status (*mon, *dst, status)`	Called by a device driver to inform the monitor about a successful/unsuccessful transmission.
`data_link_receive_status (*mon, *dst, status)`	Called by a device driver to signal received link status information.

Table 6.5: *Channel monitor/device driver interface.*

Name	Description
`register_wsched_channel_monitor (*mon)`	Announce the availablity of a monitor.
`unregister_wsched_channel_monitor (*mon)`	Remove a channel monitor from the list of available monitors.
`wsched_get_wchmon_by_id(*id)`	Search monitor specified by the id.
`wsched_wchmon_add(*dev, *id)`	Add a monitor to a device.
`wsched_wchmon_del(*dev)`	Remove a monitor from a device
`wsched_data_link_transmit_status (*dev, *dst, status)`	Wrapper for corresponding monitor function.
`wsched_data_link_receive_status (*dev, *dst, status)`	Wrapper for corresponding monitor function.

Table 6.6: *Additionally exported kernel functions.*

6.2.2 Wireless Queuing Disciplines

Since the integration of a wireless queuing discipline in the existing framework without requiring modifications of the higher layers is crucial, the standard interface of a scheduler in Linux traffic control as presented in 6.1.1.1 also has to be implemented by a wireless queuing discipline. Thus it will be possible to re-use existing components e.g. to set up configurations which make use of the available modules for packet classification and policing but use a wireless scheduler for actually scheduling the packets.

The only differences between the conventional wire-line queuing disciplines and their new variants for wireless scheduling is that a wireless queuing discipline takes the channel-condition into account when scheduling the next packet for transmission using the information provided by a channel monitor via the additional interface. A simple scheduler developed in order to test the framework and illustrate its usage is the wireless FIFO scheduler described in the following.

6.2.2.1 Example: Resource-Consumption Aware FIFO

One of the main problems in providing wireless quality of service is that because of the changing conditions in WLANs a realistic call admission mechanism will not be able to guarantee that the system will not go into an overload state. In this case the downlink scheduler is forced to drop packets to one or more destinations when its queues are filled. Rather than simply dropping all further packets once the queues in the access points are filled, a channel-state aware FIFO discipline could drop packets in a way which optimizes the total system goodput while still being very easy to implement. It also does not have the high scheduling overhead of a fair queuing discipline. In the following the idea of such a queuing discipline will be briefly presented, afterwards it is shown how such a discipline was implemented within the framework.[8]

In a wireless base station which uses conventional FIFO scheduling a station experiencing a bad channel will consume a higher share of the raw wireless bandwidth then a station with a good channel which is receiving data at the same rate because of MAC layer retransmissions and adaptive modulation. As illustrated in Chapter 4 this is the main reason why wire-line scheduling approaches at the network layer fail when applied to the wireless case since they do not have any control over the additional bandwidth consumed due to a bad channel-state.

In a system which uses a FIFO scheduler, a flow i having an arrival rate of $r_{a,i}$ can consume an amount of resources (in terms of raw bandwidth) b_{raw} of up to

$$b_{raw,i} = r_{a,i} \cdot \frac{1}{g_i} \tag{6.1}$$

where g_i corresponds to the GTR (as defined in Section 4.2) of the flow.

Under the assumption that all flows have the same priority and are sending at no more than their specified rate (or have been policed e.g. by a token bucket filter before) the number

[8] in `../net/sched/sch_csfifo.c`

Listing 6.1: *Wireless FIFO: enqueue a packet (pseudo code)*

```
 if ( queue->length == queue_limit )
    purge_one_packet_from_queue();
 else if ( queue->length ≥ drop_th )
 {
  x = random();
  if ( x * queue_limit < queue->length * drop_P )
  {
    purge_one_packet_from_queue();
  }
}

queue->enqueue(skb);
```

of flows whose specified rate is violated in an overload situation can be minimized by dropping packets of flows which are on the channel with the minimum GTR This also maximizes the total goodput of the system.

In order to dampen the influence of the selective dropping on the system and to avoid periodically entering and leaving the overload state, the scheduler allows the specification of a drop threshold $drop_{th}$ and a maximum drop probability $drop_P$. The algorithm starts dropping packets when the queue length exceeds the drop threshold parameter. After that the probability for dropping a packet increases linear with increasing queue length as shown in Listing 6.1.

The number of packets taken into account when trying to make room for the newly arrived packet is called the *lookahead_window* parameter of the scheduler. It is necessary to limit the computational overhead in cases where a long queue is to be used. In order to avoid the problem that a destination with a bad GTR is not able to update its status even if its channel quality improves since it is not allowed to send any packets, the scheduler has a *probe_interval* parameter which specifies the maximal duration a destination is not probed. All packets whose destination has not been probed for a longer time will be skipped when selecting a packet to drop.

The schedulers dequeue function (Listing 6.2) first signals the end of the previous transmission to the channel monitor. If the qdisc is backlogged, the channel monitor is notified of the next transmission and a packet is dequeued. Otherwise the idle state is signaled to the monitor. (Before the transmit call to the channel monitor which passes destination information, the function `skb2dst` is used to obtain the destination of the `skb` to be sent. This is necessary since the scheduler does not know in which way the channel monitor determines the destination of a packet. Usually this will be the MAC address but it could also be the IP address or other fields in the packet header/payload. E.g. in case of the scheduler running in the TCSIM simulation environment it has to be determined by the IP address since TCSIM does not support MAC header generation. A different approach would have been to handle the identifier of a destination only within the monitor and

Listing 6.2: *Wireless FIFO: dequeue a packet (pseudo code)*

```
monitor->end_transmit( monitor, 0 );
if ( packets_to_send() )
{
  monitor->skb2dst( monitor, skb, destination );
  monitor->start_transmit ( monitor, destination , length );
  skb = queue->dequeue();
}
else
{
  monitor->channels_idle( monitor );
  skb = 0;
}
return(skb);
```

passing the complete `skb` when calling `start_transmit` - with the disadvantage that a scheduler would not be able to make any destination based decisions.)

The `purge` function (Listing 6.3) demonstrates how a wireless queuing discipline can obtain information on the channel-state. In this case, starting with the most recently enqueued packet, the channel monitor information for all packets within the lookahead window is used in order to search the last recently enqueued packet on the channel with the worst GTR which is not due to be probed. This packet - if the search was successful - is dropped. (For simplicity the function gets channel-state information for every packet instead of caching results for a destination.)

If the number of packets in the lookahead window is equal to the total queue length, the probe interval is T_{probe}, the maximum packet size is L_p and the GTR of a destination i is g_i one can compute the maximum amount of raw bandwidth $r_{raw,max}$ used by the mobile station with the lowest GTR in an overload situation as:

$$r_{raw,max} = \frac{1}{T_{probe} \cdot g_i} * L_p \tag{6.2}$$

By choosing T_{probe} one is able to guarantee a minimum rate for giving a mobile station access to the channel. And since g_i and T_{probe} are independent of the rate of a flow, $r_{raw,max}$ can be as low as desired if one assumes that an upper limit for g_i can be determined (e.g. based on the maximum number of allowed retransmissions and the minimal modulation rate). Thus the influence of flow experiencing very bad channel conditions compared to the other channels on the system can be limited.

Therefore the channel-state aware FIFO algorithm is a very simple scheduler which demonstrates the usage of long-term channel-state information in order to limit the resource consumption of a mobile host. On the other hand it has severe disadvantages:

Listing 6.3: *Wireless FIFO: purge (pseudo code)*

```
skb = skb_peek_tail ( queue ) ;  /* look at last enqueued packet */
min_level = ∞ ;
drop_skb  = ∅;
range = min( lookahead_window, queue->length );

for ( i=0; i < range ; i++)
{
  /* query channel monitor for channel status information : */
  skb2dst ( monitor , skb , destination ) ;
  state = monitor->get_channel_state ( monitor , destination ) ;

  /* packet only dropped if the channel was recently probed */
  if ( jiffies - state ->last_probed < probe_interval ∧
       state ->level < min_level )
  {
       drop_skb  = skb;
       min_level = state ->level;
  }
  skb = skb->prev;       /* advance to previously enqueued packet */
}

if ( skb ≠ ∅ )
   drop_skb(skb) ;       /* unlink packet from queue and drop it */
```

- As in the case of a conventional FIFO queuing discipline, flows have to be rate-limited beforehand.
- The minimum bandwidth is the same for all destinations and cannot be adapted to the requirements of a specific flow. (QoS requirements of individual flows are not considered.)
- The isolation of flows is violated and the short-time bandwidth-share of a flow in a system in overload state becomes more bursty.
- Bandwidth and delay are not decoupled.

Therefore, in cases where a QoS criteria and resource-consumption oriented scheduling is demanded, the usage of a wireless fair queuing or link-sharing discipline is necessary, e.g. the modified H-FSC scheduler.

6.3 TC Simulation Environment Extensions

This section presents the modifications and extensions of the TCSIM Environment (Chapter 6.1.3) which were necessary in order to evaluate the wireless components by using simulations. Basically three parts were added/extended:

- simulation of wireless network interfaces
- traffic generation
- trace analysis

6.3.1 Wireless Channel Simulator

Since the chosen simulation environment TCSIM did not have support for simulation of wireless network interfaces/wireless channels an additional component had to be developed in order to add this functionality. One approach would have been to add a wireless network interface in the same way that is currently used for the wire-line interface simulation: Whereas a wired network interface is simulated by polling the queue(s) at a constant rate, the time between two transmissions on a wireless interface would vary - following a wireless channel model of some kind.

However, a different, more flexible approach was chosen. The assumption is that a wireless network interface also transmits at a constant raw bit-rate - only the achieved goodput is time-varying. Therefore one is able to approximate the behavior of a wireless network interface by using a special *wireless channel simulator queuing discipline* which is polled in regular intervals. The goodput of this queuing discipline follows the rules of an inner channel model, for simplicity the two-state Gilbert-Elliot Model was chosen, but any other model which can be simulated in a discrete event simulation is suitable.

The Gilbert-Elliot channel model (Figure 6.7) is a two state Markov model which simulates the wireless characteristics by assuming that the channel can be in only two states: a

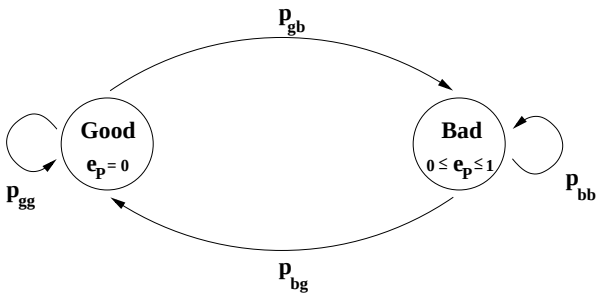


Figure 6.7: *Gilbert-Elliot model of a wireless channel.*

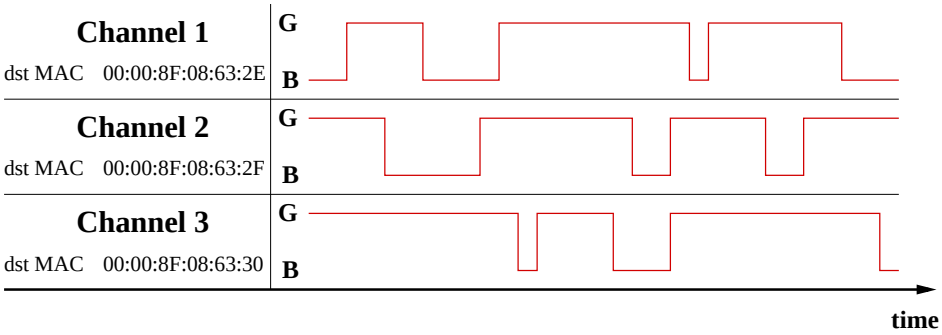


Figure 6.8: *State of wireless link from access point towards three mobile stations.*

good state, in which the probability for a packet error[9] is zero, and a bad state, in which a packet is corrupted with a probability $e_P, (0 \leq e_P < 1)$. The probability of changing from a good state to a bad state is given by the transition probability p_{gb}, the change from bad to good occurs with p_{bg}. Therefore the model is completely characterized by specifying e_P, p_{gb} and p_{bg}.

The wireless simulation queuing discipline was designed to simulate the behavior of a wireless LAN interface transmitting to mobile stations in different positions (and thus experiencing different channel characteristics):

- It allows the specification of e_P, p_{bg} and p_{gb} for the channel to each destination and for one default channel.
- The channel states are simulated independently of transmissions and separately for each channel as shown in Figure 6.8.
- The standard filters/classifiers are used in order to assign packets to simulated wireless channels, if no filter indicates a match, the default channel is chosen.
- An internal hardware queue of the wireless interface can be simulated, the queue length is configurable as a qdisc parameter.

[9] It was decided to use a packet based model rather than a bit-error based one since the position of a bit-error was not relevant in our simulations. In addition the overhead of a simulation is much less using a packet based model.

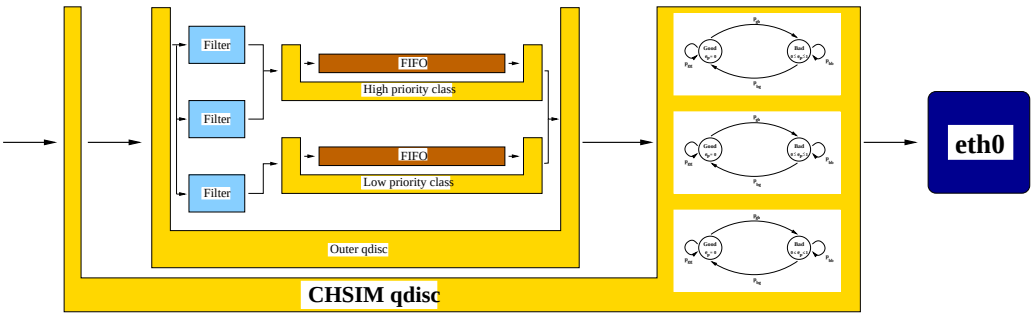


Figure 6.9: *Regular scheduler encapsulated in the wireless channel simulation scheduler which is running on a regular (TCSIM) network interface.*

- Automatic retransmissions (as of the 802.11 MAC) are simulated, the maximum number can be freely chosen.
- Different modulations are simulated by generating "useless" data before the packet so that the sending of a packet is delayed a time corresponding to the modulation rate.
- A simulated channel corresponds to a scheduling class. The probability values are configured using the `tc` program when a class is created. All parameters for the wireless simulation are set using the standard Linux traffic control interface.

Although the wireless simulation queuing discipline has the same interface of an usual scheduler, its behavior is quite different:

- It duplicates and delays packets (in case of retransmissions and adaptive modulation).
- It modifies packets. (When the simulated channel indicates that the packet would have been corrupted, the IP destination address is optionally deleted. This allows a detailed analysis of the resources consumed by corrupted packets.)
- Only one level of classes can be assigned (since a class always corresponds to a simulated channel) and there is only one inner qdisc which is the root scheduler running on the simulated wireless network interface. However, obviously this inner qdisc can have further classes and/or qdiscs.
- It classifies packets when they are *dequeued* not when they are enqueued as a normal qdisc would do. The reason for this behavior is that simulation only needs to know the wireless channel of a packet at the point in time when its transmission is simulated.

An example is shown in Figure 6.9. A packet will be processed according to the following steps in this design when its transmission on a wireless medium is simulated: When it is received from the network layer, it is handed to the root scheduler of the interface, which is

the wireless simulation queuing discipline `sch_chsim`. Here it is immediately handed to the inner scheduler whose behavior on the wireless link is to be examined. Whenever the network interface indicates it is able to send the next packet, the next packet is dequeued from the inner scheduler. If the simulated wireless interface has an internal queue it is completely filled. Then the head of line packet of the internal queue is classified by all registered filters in order to find out which channel this packet belongs to. (Usually one would use a classification based on the MAC or IP address of the packet.) If no match occurs, the default channel is assumed. The state of the channel is checked. If it is in state "good" the packet is transmitted immediately. Otherwise the packet is corrupted with probability e_P and sent. If the maximum number of retransmissions has been reached, the packet will be deleted from the internal queue of the simulated device, else it is kept there and a retransmission is done when the interface is ready again.

A detailed example script for simulating the usage of the wireless FIFO discipline described in Subsection 6.2.2.1 in a wireless environment can be found in the appendix, Section A.1. A side-effect of the way in which the wireless simulation was implemented is that it can also be executed within the network protocol stack of a running kernel (Appendix, Section A.4.1).

6.3.2 Traffic Generator (TrafGen)

A disadvantage of the TCSIM simulation environment is that only the simulation of constant-bit rate flows (using the *every* keyword, for an example see bottom half of Listing A.1 in the Appendix) and the sending of single packets at a specified point in time is supported. In order to support the simulation of Poisson distributed and bursty flows, a simple tool, the *TCSIM Traffic Generator (TrafGen)* was developed which creates trace files to be used in a simulation.

An example is shown in Listing 6.4, Listing 6.5 is an excerpt of the generated trace. Using this technique constant bit-rate, Poisson, uniform and bursty flows can be simulated.

6.3.3 Additional Trace Analysis Tools

Currently the support for trace analysis in TCSIM is limited. The main tools are the filter program `filter.pl` which is able to filter traces for source/destination addresses, ports and enqueue/dequeue events and the plot utility `plot.pl`. The supported plot types are: average rate, inter-arrival time, delay, and cumulative amount of dequeued data. Support for delay histograms, delay probabilities and inter arrival-time probabilities was added. Furthermore an averaging skript `avg_calc.pl` was implemented to calculate the maximum, minimum, average and standard deviation values over windows of trace data.

A usual TCSIM simulation therefore consists of the following steps:

1. Create *TrafGen* configuration (optional).
2. Generate traffic trace using *TrafGen* (optional).
3. Create TCSIM configuration.

Listing 6.4: *TrafGen configuration file.*

```
/*
 * Trafgen example
 *
 * Note: The macros PACKET(#) and PAYLOAD(#) are
 *       defined in the main simulation file.
 */

/* two Poisson distributed flows */
stream begin 0 end 60 type POISSON rate 50 "send PACKET(1) PAYLOAD(1)"
stream begin 10 end 60 type POISSON rate 50 "send PACKET(255) PAYLOAD(255)"

/* a constant bit-rate flow */
stream begin 10 end 60 type CBR rate 50 "send PACKET (2) PAYLOAD(2)"

/* a bursty flow */
stream begin 0 end 60 type BURST rate 50 burst_rate 200
p_nb 0.1 p_bn 0.3 "send PACKET(3) PAYLOAD(3)"

/* a uniform distributed flow */
stream begin 0 end 60 type UNIFORM rate 50 "send PACKET(1) PAYLOAD(1)"
```

Listing 6.5: *TrafGen trace file.*

```
time 0.000000s send PACKET(255) PAYLOAD(255)
time 0.000000s send PACKET(1) PAYLOAD(1)
time 0.000883s send PACKET(255) PAYLOAD(255)
time 0.004163s send PACKET(1) PAYLOAD(1)
time 0.007968s send PACKET(255) PAYLOAD(255)
time 0.014377s send PACKET(255) PAYLOAD(255)
time 0.020000s send PACKET(3) PAYLOAD(3)
time 0.020000s send PACKET (2) PAYLOAD(2)
...
```

4. Run simulation (with generated traffic trace/CBR flows).
5. Filter trace data using `filter.pl`.
6. Analysis using `plot.pl` and `avg_calc.pl`.

6.4 Implementation of Modified H-FSC Algorithm

This section describes the details about the implementation of a H-FSC algorithm which was extended to support the wireless scheduling model developed in Chapters 4 and 5. It is based on the H-FSC implementation by Carnegie Mellon University [51] distributed as part of Sony's ALTQ package for BSD UNIX [9]. Therefore, the task consisted of the following steps:

1. Port of H-FSC scheduler to Linux
   (a) in TCSIM environment
   (b) in "real" Linux kernel
2. Integration of new wireless scheduling model
   (a) in TCSIM environment
   (b) in "real" Linux kernel

The scheduler itself consists of two parts: a (user space) configuration module for the traffic control program `tc`[10] and a kernel module which implements the scheduling algorithm[11].

6.4.1 User Space Configuration Module

The user space part of the scheduler reads the configuration commands, stores the gathered data in the appropriate structures and passes it via the netlink socket to the kernel part. It also inquires status/statistic information and formats it for output.

Whereas the original H-FSC algorithm assumes that the curve used for link-sharing and the one used for real-time scheduling are always identical, the implementation is more flexible and allows the specification of separate curves for both purposes. By independently assigning the amount of service to be received under the real-time/link-sharing criterion, one can e.g. limit the maximal rate of a class by not specifying a link-sharing service curve (scheduler becomes non work-conserving!), or only provide resources if excess bandwidth is available by not assigning a real-time curve.

Service curves are two-piece linear and specified using the triplet [m_1 d m_2], where m_1 is the slope of the first segment, d is the projection of the intersection of both pieces on the

[10] in file `../tc/q_hfsc.[c,h]` of the `iproute2 directory tree`
[11] `../net/sched/sch_hfsc.c`

x-axis (corresponding to the length of the initial burst in ms) and m_2 is the slope of the second segment. Thus, if one chooses $d \leq \frac{L_{max}}{m_1}$ and L_{max} is the maximal packet length, the value of d determines the maximal delay for packets of a class in the wire-line case. In the wireless case the delay additionally depends on the amount of guaranteed resources for the subtree the mobile is in and on the GTR of the mobile itself. The value of m_2 specifies the long-term rate of the class. As outlined in Chapter 5 in the wireless scheduling model service curves in a cooperative scheduling subtree are in terms of goodput and service curves of competitive classes describe the amount of resources (raw bandwidth).

Global Scheduler Parameters

When the H-FSC qdisc is attached to a device the following parameters can be specified:

- **bandwidth *THROUGHPUT*:** The total bandwidth of the interface (for a wireless device this should be the expected average goodput).
- **varrate:** Use variable bandwidth adaption (in overload situations the scheduler adapts the service curves to the available bandwidth). [+]
- **estint *SIZE*:** Size of interval (in bytes of transmitted data) for bandwidth estimation/averaging when using "varrate". [+]
- **wireless:** Use the new hierarchical wireless scheduling model. [+]
- **wchmon *NAME*:** Install wireless channel monitor NAME on the target device. [+]
- **reducebad *PERCENT*:** When service curves have to be reduced because not enough bandwidth is available, the GTR of a class determines PERCENT of its reduction and all classes are reduced by (100-PERCENT) times the necessary reduction. For the wireless scheduling model PERCENT is 100 which is also the default value. [+]

Options marked with "[+]" were introduced with the modifications made for wireless scheduling.

Parameters for Individual Classes

When a class is created or modified, the following parameters can be specified:

- **sc *[m_1 d m_2]*:** Defines a service curve (short for using rt and ls with the same values for one class).
- **rt *[m_1 d m_2]*:** Defines a real-time curve.
- **ls *[m_1 d m_2]*:** Defines a link-sharing curve.
- **dc *[m_1 d m_2]*:** Defines a drop curve. [+]
- **grate *BPS*:** Defines a linear real-time service curve with a rate of BPS (equivalent to a linear [rt 0 0 M2] real-time curve).
- **default:** Specify this class as the (new) default class.
- **sync:** This class is a wireless synchronization class. [+]

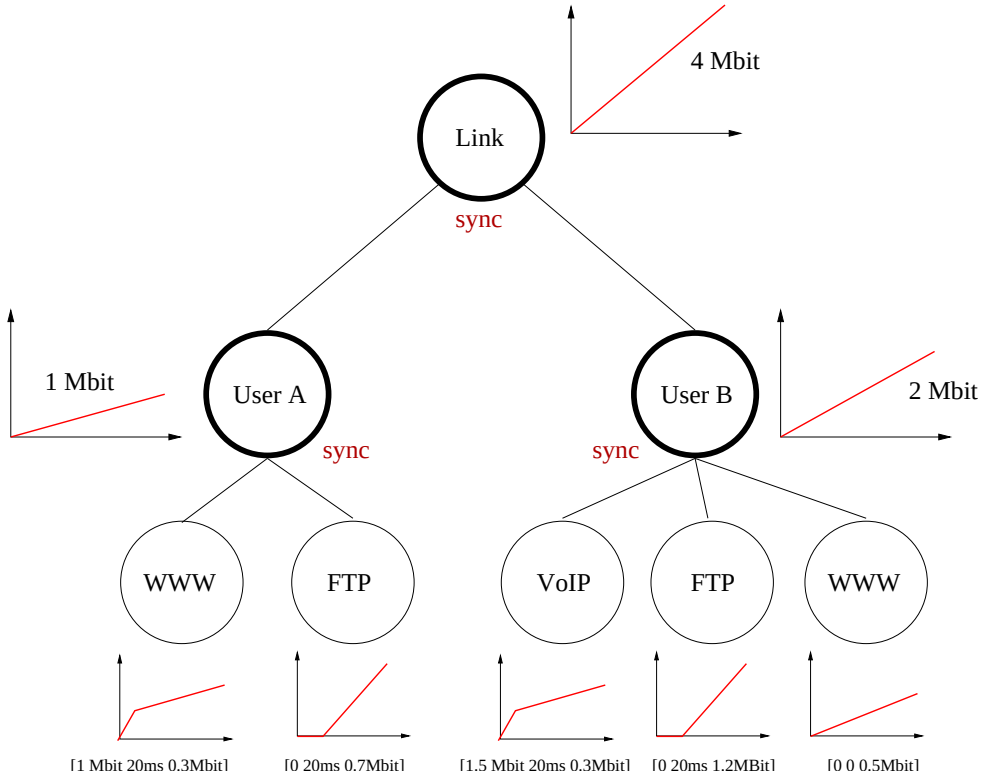


Figure 6.10: *Wireless link-sharing for two users with the modified H-FSC scheduler.*

An Example Setup

Listings 6.6, 6.7 and 6.8 show how the modified H-FSC scheduler would be configured for the wireless link-sharing structure shown in Figure 6.10.

The first step is to load a wireless channel monitor - in this case the type "`driver`" is used - and to add the qdisc (Listing 6.6).

Then the classes for both users are configured (Listing 6.7), competitive scheduling among both users is enforced by making both classes synchronization classes.

The last step is to add subclasses for each traffic type (Listing 6.8). Within the subtree of each user the scheduling is cooperative.

A more complicated example script which specifies the complete hierarchical link-sharing structure shown in Figure 4.2 for the modified H-FSC scheduler is part of the Appendix, Section A.4.4.

6.4.2 H-FSC Kernel Module

The kernel module `sch_hfsc` implements the core functionality of the (modified) H-FSC scheduler. In the following the most important aspects of the implementation will be described, for more details the reader is referred to the source code.

The scheduler uses two different kinds of data structures in order to store the data: a scheduler-specific and a class-specific structure. The first is used in order to keep track of

Listing 6.6: *Example Setup Part 1 of 3: Installing the channel monitor and setup of qdisc.*

```
#!/bin/bash
#
# simple example for configuring the modified H-FSC scheduler
#

# load a wireless channel monitor
insmod wchmon_driver

# configure root
tc qdisc add dev eth1 root handle 1:0 hfsc bandwidth 4Mbit \
 estint 32000b wireless wchmon driver
```

Listing 6.7: *Example Setup Part 2 of 3: Adding classes for both users.*

```
# add classes on first level

#  - user A:
tc class add dev eth1 parent 1:0 classid 1:10 hfsc [ sc 0 0 1 Mbit ] sync default

#  - user B:
tc class add dev eth1 parent 1:0 classid 1:20 hfsc [ sc 0 0 2 Mbit ] sync
```

Listing 6.8: *Example Setup Part 3 of 3: Adding classes for different traffic types.*

```
# add classes on second level - user A
#  - WWW
tc class add dev eth1 parent 1:10 classid 1:101 hfsc [ sc 1Mbit 20ms 0.3Mbit]
#  - FTP
tc class add dev eth1 parent 1:10 classid 1:102 hfsc [ sc 0 20ms 0.7Mbit ] default

# add classes on second level - user B
#  - VoIP
tc class add dev eth1 parent 1:20 classid 1:201 hfsc [ sc 1.5 Mbit 20ms 0.3Mbit]
#  - FTP
tc class add dev eth1 parent 1:20 classid 1:202 hfsc [ sc 0 20ms 1.2Mbit]
#  - WWW
tc class add dev eth1 parent 1:20 classid 1:203 hfsc [ sc 0 0 0.5 Mbit]

# now add the appropriate filters
tc filter add ...
```

Name	Description
*sched_rootclass	pointer to the schedulers root class
*sched_defaultclass	pointer to the default class
sched_requeued_skbs	a queue of processed but later requeued skbs
sched_classes	total number of classes in the tree
sched_packets	total number of packets stored in the tree
sched_classid	id number of scheduler
sched_eligible	eligible list
sched_filter_list	list of attached filters
wd_timer	watchdog timer (started to trigger the next dequeue event, if the scheduler has packets but is currently not allowed to send - e.g. because the flow is rate-limited)
sched_est_active	true if the scheduler currently estimates the dequeuing rate [+]
sched_est_time	time of last estimation [+]
sched_est_length	length of last dequeued packet [+]
sched_est_dfactor	current estimation of global degrade factor d [+]
sched_est_interval_log	length of estimation interval (bytes) [+]
sched_est_el_sum	sum of all active eligible curves [+]
sched_flags	scheduler flags
sched_est_required_cap	total capacity needed for real-time requests [+]
cur_dst	destination of packet currently scheduled [+]
sched_est_reducebad	percentage at which GTR of a flow is taken into account in overload situations [+]

Table 6.7: *Global data for a modified H-FSC scheduler instance.*

global scheduler information (Table 6.7, additional data necessary to implement the GTR based wireless scheduling model is marked with " [+]") and exists only once per scheduler instance. The most important parts are the *eligible list* which determines if the real-time criterion has to be used to select the next packet and the *list of filters* attached to the scheduler. It also holds the information necessary to estimate the current rate at which packets are dequeued. Although not necessary to implement the wireless scheduling model as described in Chapters 4 and 5, this was included in order to be able to support variable rate interfaces without wireless channel monitors. In this case, since no destination specific information is available, the service curves of all classes are reduced equally in an overload situation.

The second data structure used (Table 6.8) represents a class within the link-sharing hierarchy. Basically the information necessary to characterize the state of a class consists of: the three different service curve types (real-time, link-sharing and drop curve) with their current deadlines, the amount of service received for each curve and information how the class is integrated within the hierarchy. Each class also has a pointer to a synchronization

class. If the class is part of a subtree in which cooperative wireless scheduling is done, it points to the nearest synchronization class on the way to the root of the tree. In case the class is part of a competitive scheduling environment, it is a pointer to the class itself.

Internal Service Curve Representations

Internally the scheduler uses three different forms of service curves: The two-piece linear service curves specified by a user are transformed to an internal representation based on bytes per CPU clock count. (The CPU maintains a very accurate timer value with a resolution of about 10^6 tics per second.) In order to speed up the calculation, also the inverse values of the slope of both segments are pre-calculated. Whenever a class becomes active, the current runtime service curve for this class is calculated by shifting this representation to the current time/received service coordinates. For a simple form of admission control (checking that the sum over the service curves of all children does not exceed the parents service) a generalized service curve in form of a linked list is used which can consist of multiple pieces.

Synchronization Classes

The main tool in order to implement the resource-consumption aware scheduling model are the newly introduced synchronization classes. Classes which are part of a cooperative scheduling subtree will add their eligible curves upon activation to the sum of eligible curves `cl_sync_el_sum` of their parent synchronization class and the amount of demanded resources for the subtree will be calculated and stored in `cl_sync_required_cap`. The variable `cl_sync_required_cap` describes the average goodput in bytes per second which is required within a subtree in order to keep the service curves of all active sessions within the subtree. In an overload situation the scheduler then adapts the service rates of the individual classes as described in Section 5.1: The service for each class is reduced based on its resource-consumption so that the whole subtree does not consume more than the (resource-based) service curve of the synchronization class (which is its root) specifies. This is done by inquiring the current goodput for a class from the wireless channel monitor. This amount in relation to the required average goodput then determines the service curve reduction d_i of a class i as in Equation 5.1. Instead of actually adapting the service curves of a class each time its GTR changes, the implementation increases the amount of service required to transmit a packet correspondingly. E.g instead of reducing a service curve by 50% it will simply require two units of service for each byte to be sent since this requires much less computational effort.

As described in Section 5.2 a synchronization class is also required to adapt the virtual times used for the link-sharing criterion: Since within a cooperative subtree it is based on goodput it needs to be adapted to its resource-based amount whenever the “border” to a resource-based part of the link-sharing tree is crossed. Therefore the function responsible for updating virtual times is always called with specifying the goodput-based and the resource-based update values. It then updates the virtual times of each node corresponding to its scheduling mode.

Name	Description
cl_id	unique class id
cl_handle	unique class handle
*cl_sch	pointer to scheduler instance
cl_flag	flags (e.g. *sync*, *default*, ...)
*cl_parent	parent class
*cl_siblings	list of sibling classes
*cl_children	list of child classes
*cl_clinfo	class info (e.g. generalized service curve) for admission control
*cl_q	qdisc for storing packets of this class
*cl_peeked	next skb to be dequeued
cl_limit	maximal length of class queue
cl_qlen	number of currently stored packets
cl_total	total work received [bytes]
cl_cumul	work received under real-time criterion [bytes]
cl_drop_cumul	total packet drop service received [bytes]
cl_d	current deadline
cl_e	eligible time
cl_vt	virtual time
cl_k	packet drop time [+]
cl_delta_t_drop	difference between real time and drop service time [+]
*cl_rsc	real-time service curve
*cl_fsc	link-sharing service curve
*cl_dsc	packet drop service curve [+]
cl_deadline	deadline curve
cl_eligible	eligible curve
cl_virtual	virtual time curve
cl_dropcurve	packet drop curve
*cl_sync_class	parent synchronization class [+]
cl_sync_el_sum	sum of all eligible curves (sync class) [+]
cl_sync_required_cap	avg. required capacity [byte/s] (sync class) [+]
cl_vtperiod	virtual time period sequence number
cl_parentperiod	parent's vt period sequence number
cl_nactive	number of active children
*cl_actc	list of active children
cl_actlist	active children list entry
cl_ellist	eligible list entry
cl_stats	statistics collected for this class
cl_refcnt;	counts references to this class
cl_filters;	counts number of attached filters

Table 6.8: *Data of a class of the modified H-FSC scheduler.*

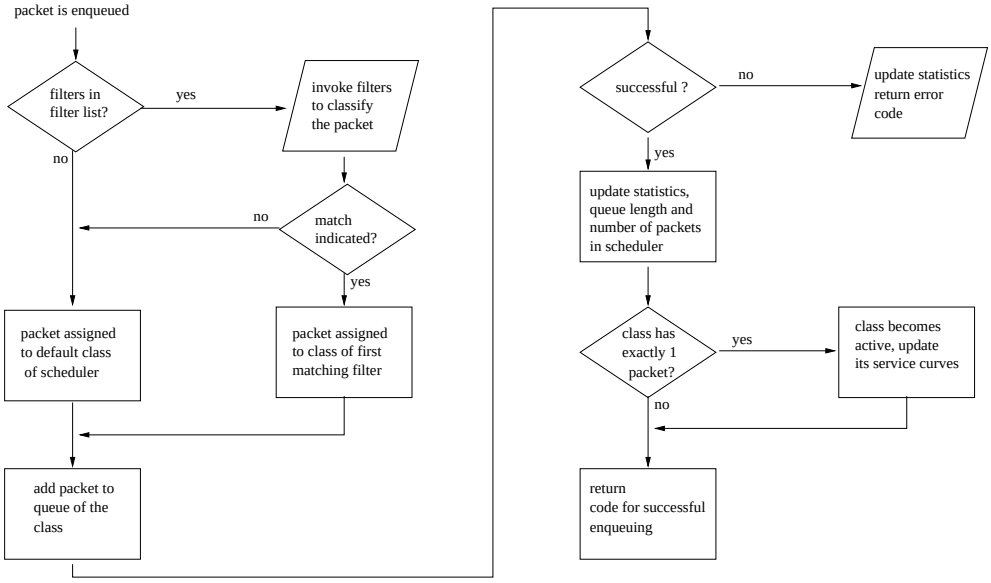


Figure 6.11: *Flow chart of packet enqueuing in the modified H-FSC implementation.*

Enqueue/Dequeue Functions

The most important functions of the scheduler are the `enqueue(...)` function which is called whenever the network protocol stack hands a packet down to the scheduler and the `dequeue(...)` function which is executed if the device is ready to send the next packet. The `enqueue` function is rather simple: Its main purpose is to determine the class a packet belongs to and to update the status of class and scheduler as shown in Figure 6.11.

The central function of the modified H-FSC scheduler is the `dequeue` Function. Its most important tasks are:

- Selecting the class of which the next packet is sent.[12]
- Estimating the available bandwidth and reacting to overload situations.
- Updating the service curves.

Figure 6.12 shows the reaction to a dequeue event within the `dequeue` function and its subfunctions as a process flow chart.

6.5 Long-term Channel-State Monitor

Since the prototype for the architecture had to be implemented on currently available hardware, where no explicit support for monitoring wireless channel quality towards a

[12]Before selecting a new class for transmission, the implementation has to check if any processed but later requeued packets are available. This happens if the device indicated that it is ready to accept the next packet (causing a packet to be dequeued and the service counters to be updated) but later signaled that it is unable to store the packet in its internal buffer, e.g. because of the packet's size. In this case the packet is temporarily stored in a special scheduler queue, the *requeued packets queue*, and sent to the device at the next dequeue event.

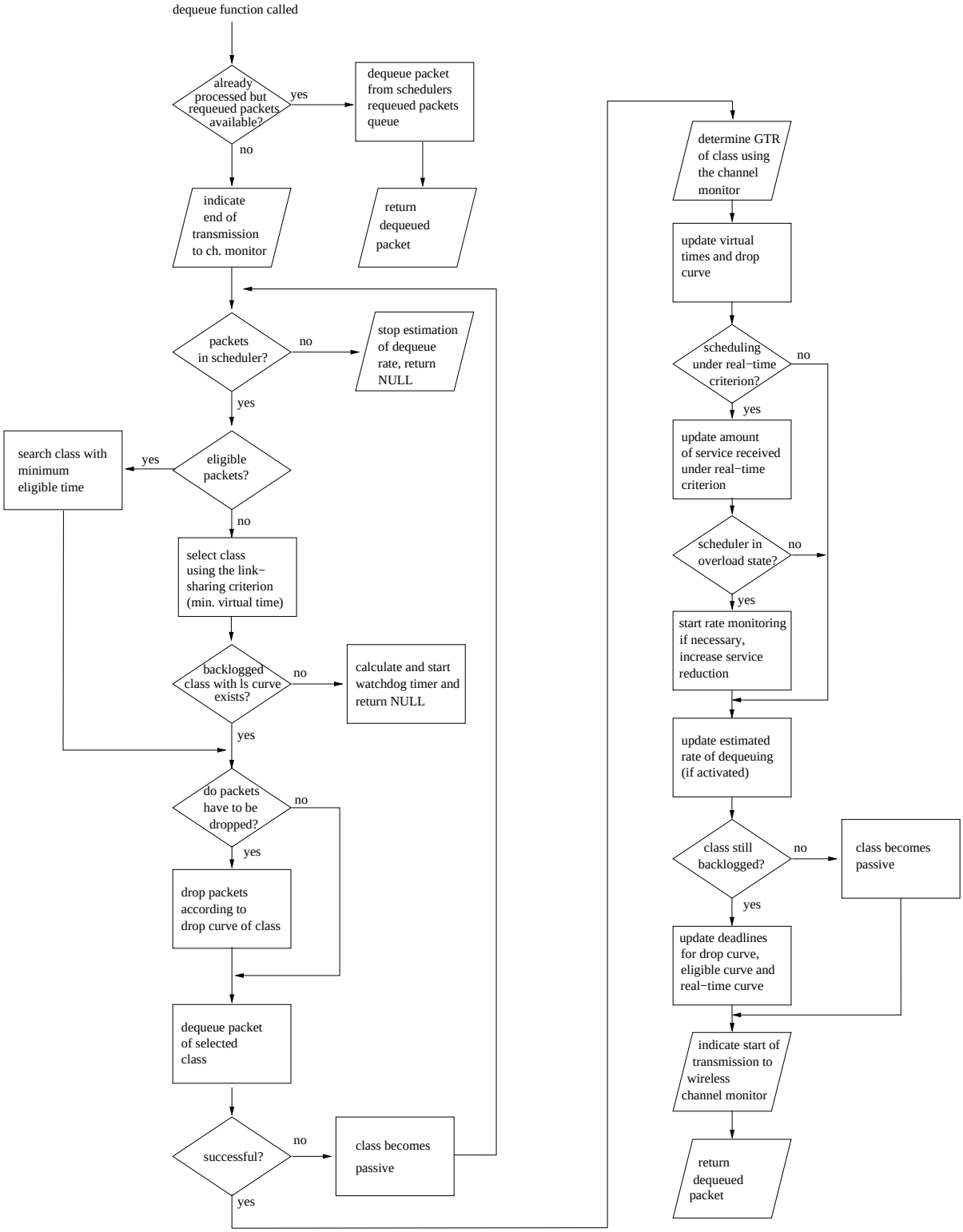


Figure 6.12: *Flow chart of packet dequeuing in the modified H-FSC implementation.*

specific mobile host is available in the data link layer, other options had to be considered: For the implementation of a wireless channel monitor there are several possible sources of information which can be used to determine the goodput to throughput ratio g_i for a given destination i: monitoring the time between dequeue events, the transmission interrupt of the wireless card, signal quality/signal strength and noise information reported by the MAC [12] and the currently used adaptive modulation of the MAC.

Signal Level, Noise Level and Signal Quality

In most currently available wireless chipsets (e.g. the Prism 2 by Intersil) an Automatic Gain Control (AGC) unit monitors the signal condition and adapts the RF circuit of the card. This information is also used to compute three values indicating the signal level, the noise level and the signal quality. (Some other designs only provide one value which indicates the signal level.) These values are stored in the Parameter Storage Area (PSA) of the wireless card and can be read by a device driver.

Measurements [12] [14] have demonstrated a strong correlation between the signal level reported by the card and the error-rate. Since the modulation technique used is adapted based on the amount of errors on the wireless link, we assumed that the available goodput-rate for a mobile destination is also correlated. Experiments with a Raylink 2 Mbit/s FHSS wireless LAN card [21] and a Lucent 11 MBit/s DSSS card [53] both conforming to the IEEE 802.11 standard confirmed this assumption (Figure 6.13). (Measurements were done with the testbed tools described in the Appendix, Section A.3.1. UDP packet size was 1024 bytes. Rates were calculated over a window of 100 packets. Estimated goodput was calculated as the average goodput of all windows for a signal level.)

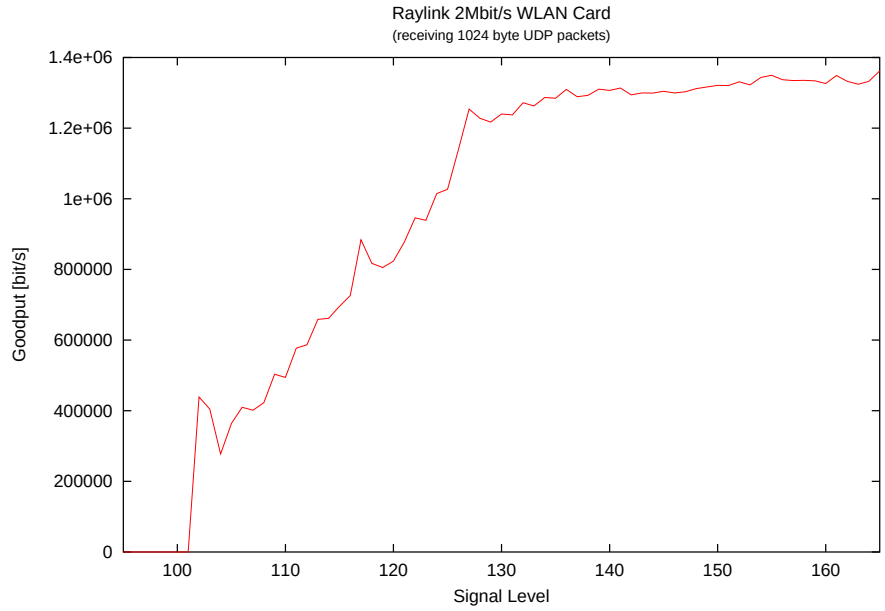
Therefore a long-term channel monitor was developed which obtains the signal quality/signal strength values (depending on the possibilities the card provides) and estimates the available goodput (in bytes/s) to a mobile destination.

Device Driver Modifications

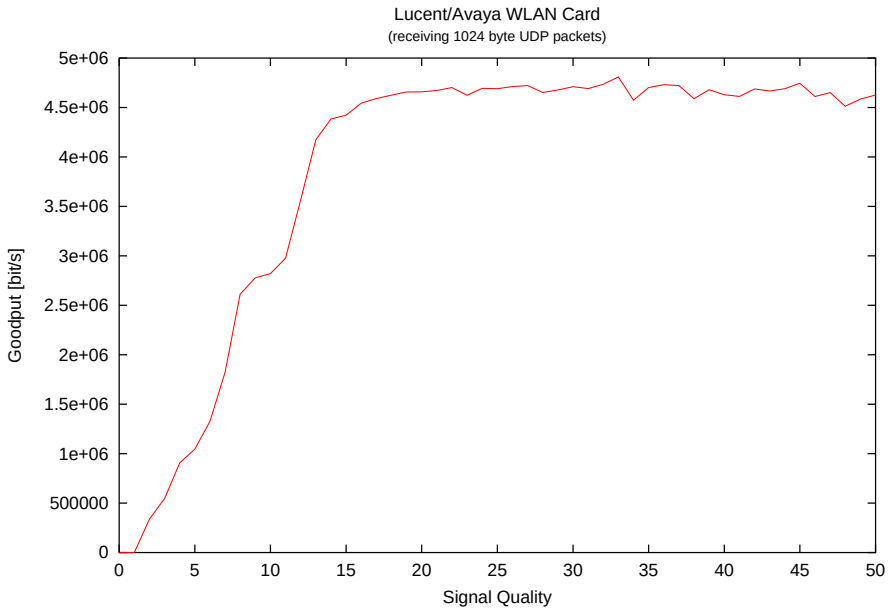
For the Linux operating system a standard interface called *Linux wireless extensions* [54] is available which allows user space programs to set wireless specific parameters of a device and obtain the three mentioned status bytes using a set of Input Output Control (IOCTL) calls. Unfortunately, since the wireless channel monitor is a kernel module, it is not able to use these functions directly. But since every wireless network card driver completely implementing the wireless extensions provides the status information and is similar in those parts, the necessary driver modifications are simple and similar for most of the drivers: After locating the parts where the level information for the wireless extensions are read from the card's PSA, a call to the `wsched_data_link_receive_status(...)` function exported by the kernel (Section 6.2.1) is inserted which relays this information to the attached wireless channel monitor.

Update of Channel-State Information

Because the MAC of the used wireless cards updates the signal strength information only upon reception of a packet, the current system forces the mobile stations to send packets



(a) Raylink 2 Mbit/s FHSS



(b) Lucent 11 Mbit/s DSSS

Figure 6.13: *Correlation between signal level/quality and goodput-rate. (Signal level and signal quality values are vendor specific and not comparable.)*

in regular intervals by sending *ICMP echo request* packets every second. This is done by starting a "ping" for each client on the AP, an improved approach would be to monitor the received traffic and only send echo requests if necessary. In a real-life situation the status information will be updated more often since most applications generate some kind of upstream traffic (e.g. the requests in case of WWW browsing, acknowledgments in case TCP is used, etc.).

Calibration of a Wireless Channel Monitor

The channel monitor module described in this section maps the link level for a specific mobile station to an expected goodput rate. One problem with this approach is that this mapping is vendor and technology specific as shown in Figure 6.13. Furthermore, it could also depend on the environment the access point is located in, the amount/characteristics of interference and the technique of adaptive modulation used. Therefore our prototype implementation uses the following calibration process: A monitoring tool developed for this purpose (detailed description in Section A.3.1) measures the achieved goodput for a mobile station in places with different signal levels. During this calibration process only one mobile station and the access point must be active. The access point generates UDP traffic at a high rate guaranteeing the link is always saturated. Goodput-rates for the different signal levels are recorded at the mobile host. The result of this process is a table which maps signal levels to expected goodput-rates for the used technology and environment.

The wireless channel monitor can be used on different wireless hardware (as long as the device driver provides the necessary status information) which could change even during the runtime of the AP. E.g. a user could decide to unplug the Raylink wireless card and install a Lucent card or the system might have two types of wireless cards. Therefore a flexible way was developed which allows the configuration/update of the level to goodput mapping during runtime and without need to recompile the channel monitor module: In the Linux operating system the `/proc` file system can be used to allow the communication of user space programs with the kernel by the means of regular file IO operations. The channel monitor creates a `wchmon_gtr_map` entry in `/proc/net/` where it exports the currently used table and is able to read a new signal to goodput mapping. Thus updating the wireless channel monitor with data available from the calibration utility is done by simple copying the generated file to `/proc/net/wchmon_gtr_map`.

Therefore the calibration process usually consists of the following steps:

- Deactivate all mobile stations except one.
- Start calibration tool on mobile station and packet generation utility on access point.
- Record data in a large variety of positions.
- Copy table with the recorded signal to goodput mapping to `wchmon_gtr_map` in the `/proc/net/` directory of the access point.

7. Results of Simulations and Measurements

This chapter presents the results of the conducted simulations and measurements.

7.1 Simulation

The described modifications of H-FSC were implemented and simulated in the TCSIM environment which was extended with a wireless component to be able to simulate the destination specific behavior of the wireless link (Section 6.3). Adaptive modulation is simulated by increasing the time needed to transmit each packet by a constant factor, dynamic changes of the channel quality are based on a Markov model (Section 6.3.1).

For the simulation the `ratio` wireless channel monitor is used, which estimates the channel quality based on the delay between two dequeue events. Since only minor changes were made influencing the behavior in a state where excess bandwidth is available, the simulated scenarios mainly concentrate on the properties of the scheduler in overload situations.

7.1.1 Scenario 1

The first simulated scenario demonstrates the benefits of a resource-consumption aware approach in a very simple link-sharing situation similar to the one presented as motivation in Chapter 4. A wireless link with a maximum (goodput) rate of 6 MBit/s is shared by two companies of which each has only a single mobile station. Company A has a Service Level Agreement (SLA) which guarantees a rate of 4424 kbit/s for mobile station 1 (MS 1) as long as its GTR g_1 is larger or equal to 0.9. Company B's SLA specifies a rate of 614 kbit/s for $g_2 \geq 0.5$. Therefore 4915 kbit/s (80%) of the total raw bandwidth are reserved for Company A and 1229 kbit/s (20%) for Company B. It is assumed that the costs for the wireless infrastructure are also shared in a 80/20 ratio. The setup is illustrated in Figure 7.1, the complete configuration script is part of the appendix (Section A.4.3).

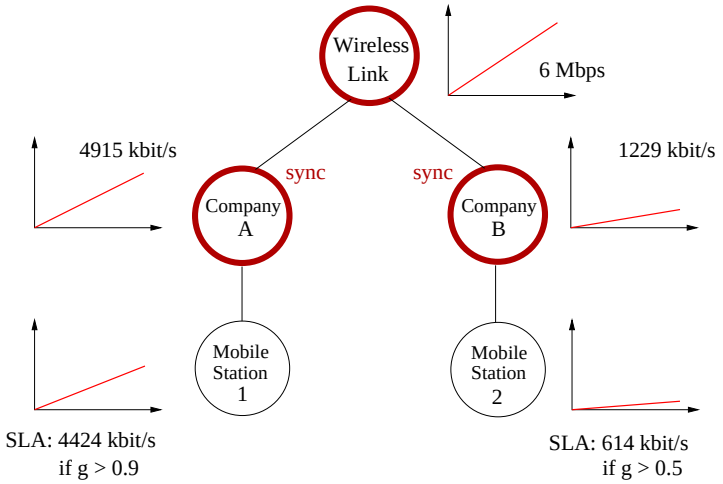


Figure 7.1: *Link-sharing hierarchy for Scenario 1.*

For comparison a regular CBQ scheduler above the link layer was configured to share the link in the same 80% to 20% ratio.

Both mobile stations receive UDP packets with a payload of 980 bytes. Traffic for MS 1 is generated at[1] a constant rate of 4887 kbit/s and for MS 2 at 607 kbit/s. The GTR g_1 of MS 1 is constantly 1.

Figure 7.2 compares the behavior of both schedulers when the modulation used by MS 2 is varied: As long as the inverse GTR of MS 2 is less or equal to 2 the rate of both stations is not limited since excess bandwidth is available. When $\frac{1}{g_2}$ is ≥ 3 the CBQ scheduler first reduces only the rate for MS 1 since it consumes more than 80% of the total goodput. Thus, although Company A does not consume its full share of resources, its rate is decreased in order to compensate for bad link quality of MS 2 since the unmodified CBQ scheduler is not aware of the resources consumed! The H-FSC scheduler modified according to the wireless link-sharing model on the other hand is able to guarantee a share of 80% of the resources for MS 1. Therefore it is able to keep the SLA for both companies independent of the modulation used by MS 2.

Whereas the amount of resources needed to transmit to both mobile stations was constant for each single simulation in the previous series (modulation was not varied *within* one simulation run) this changes when a Markov based channel simulation is used: The simulated wireless device is configured to perform up to 10 retransmissions, therefore the instantaneous GTR can vary between 1 and $\frac{1}{11}$. The average error burst length is chosen to be 5 packets, MS 1 constantly has a good channel and the probability for a bad channel state p_b of MS 2 is varied. Figure 7.3 illustrates that the wireless H-FSC scheduler still is able to perform the resource-based scheduling.

Figure 7.4 shows the influence of the probability for a bad channel state of MS 2 on the cumulative delay probability for packets sent to MS 1. For both schedulers increasing p_b

[1]The rate for MS 1 was chosen to exceed the rate specified in its SLA but to be less than the maximum rate possible with 80% of the link resources in a position with perfect link quality. This enables us to show that the rate scheduled for MS 1 takes the share of resources paid for by Company A into account.

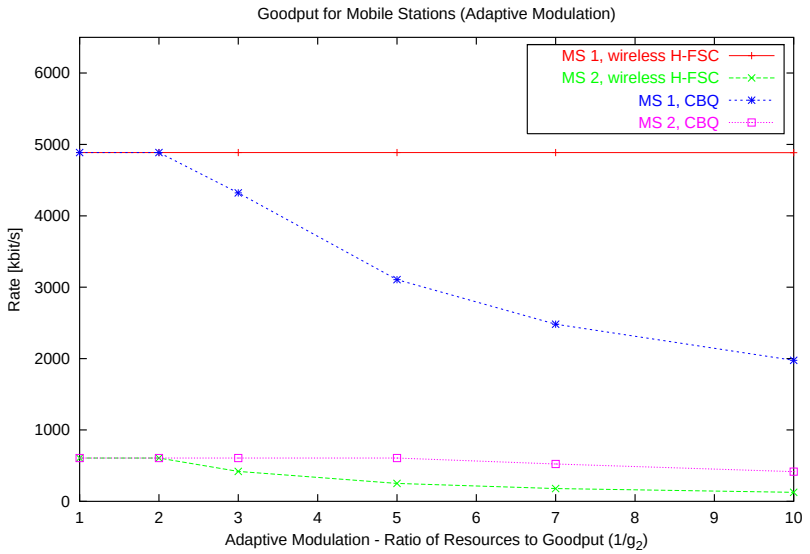


Figure 7.2: *Comparison of goodput of mobile stations in Scenario 1 for the modified H-FSC scheduler and a regular CBQ scheduler above the link layer when the modulation used by MS 2 is varied. (e.g. because MS 2 adapts to a decreasing link quality)*

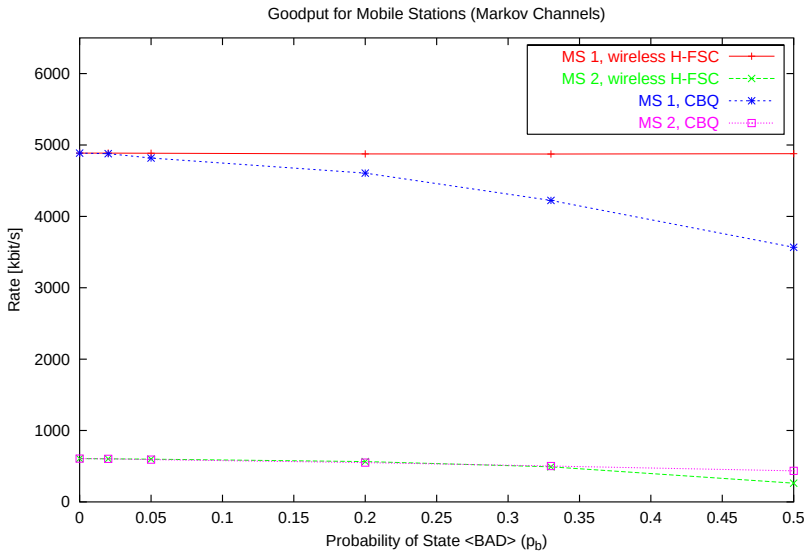
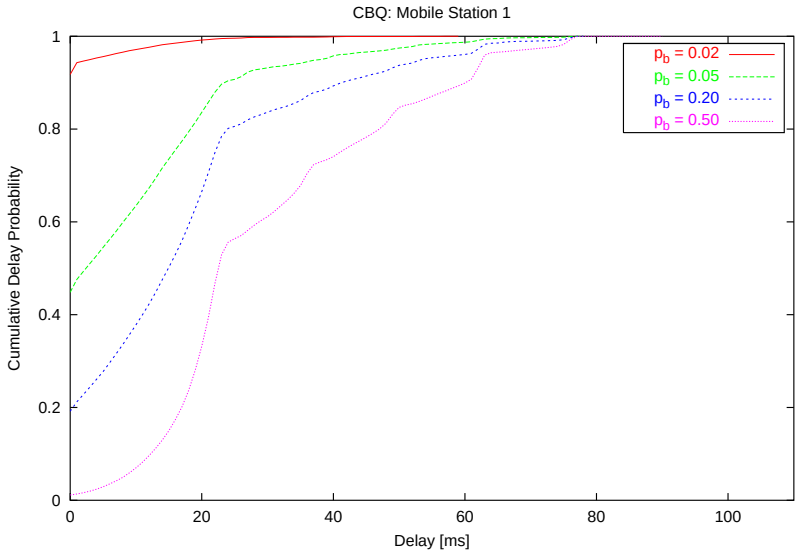
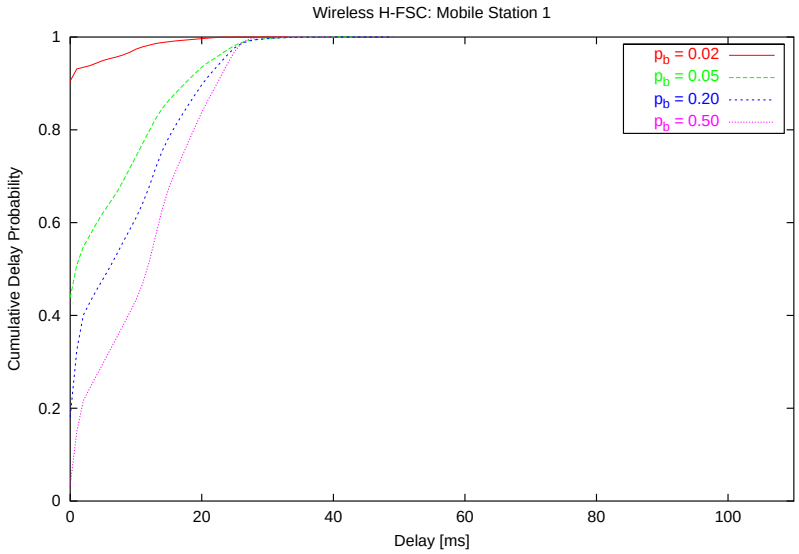


Figure 7.3: *Comparison of goodput of mobile stations in Scenario 1 for the modified H-FSC scheduler and a regular CBQ scheduler above the link layer. In this case a Markov model simulates the wireless channel and the probability for a bad channel state for MS 2 is varied.*



(a) CBQ



(b) wireless H-FSC

Figure 7.4: *Comparison of cumulative packet delay probabilities for MS 1 when the probability for a bad channel state p_b for MS 2 is varied. (Scenario 1)*

for the wireless channel of MS 2 increases the probability for higher delays for MS 1. However, Figure 7.4(b) indicates that the modified H-FSC scheduler is able to limit this influence and guarantees a delay of about 30 ms for MS 1 independent of the link quality for MS 2.

The reasons for this delay bound are the following: As explained in Section 5.3 the wireless H-FSC scheduler guarantees that the deadline for a packet is not missed by more than $\tau_{max,g_{min}}$, which is the time needed to transmit a maximum size packet at the minimum GTR if an ideal channel monitor is assumed. In the simulated scenario $\tau_{max,g_{min}}$ is the time needed to transmit a packet of maximum length including 10 retransmissions which is 14.3 ms. But the `ratio` channel monitor used in the simulation is not an ideal channel monitor. It estimates the GTR for a mobile station based on the time interval between two dequeue events. A change in channel quality is reflected by a longer time needed to transmit the packet because of link layer retransmissions. Unlike the ideal channel monitor which always estimates the current channel state correctly the `ratio` channel monitor updates the state with a delay of one packet transmission. Its delay bound therefore is twice as large as the one of the ideal monitor. The maximum delay for a packet to MS 1 is the delay guaranteed by the service curve plus twice the amount of $\tau_{max,g_{min}}$:

$$d_{max,1} = 1.9ms + 2 \cdot 14.3ms = 30.5ms \tag{7.1}$$

Agency	Traffic Type	m_1 [Mbit/s]	d [ms]	m_2 [Mbit/s]
A	VoIP	0.030	20	0.020
	WWW	0.000	20	0.050
	FTP	0.000	20	0.025
B	VoIP	0.030	20	0.020
	WWW	0.000	20	0.050
	FTP	0.000	20	0.050

Table 7.1: *Scenario 2 and 3: service curve parameters.*

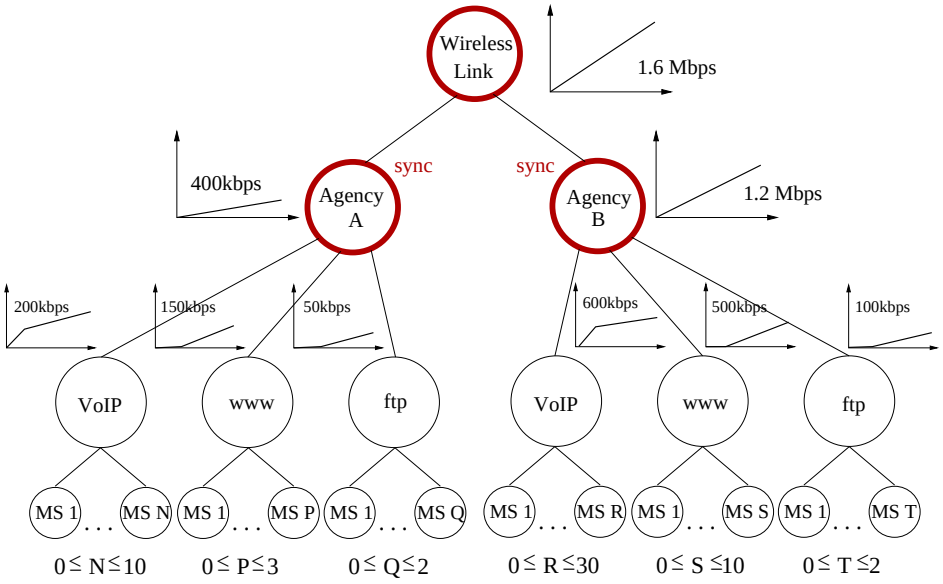


Figure 7.5: *Link-sharing hierarchy used in Scenario 2 and 3.*

7.1.2 Scenario 2

In the second scenario, all classes present in the hierarchy Figure 7.5 (which was also introduced in Chapter 4) are transmitting at their maximum rate. Table 7.1 lists the service curve parameters for the leaf classes. The VoIP traffic is enqueued at a constant rate of 20 kbit/s (similar to VoIP using a G.728 codec) and a packet a the head of line is dropped if it is more than 8 ms over its deadline (drop curve of 16 kbit/s). FTP/WWW traffic is enqueued at the maximum rate following a Poisson distribution. Table 7.2 lists the used packet sizes. All mobiles have a perfect channel (g_i=1) except for MS 2 of Agency A whose GTR is varied.

Figure 7.6 shows the influence of the transmission condition for a mobile in Agency A on the cumulative delay probability for VoIP packets transmitted to a mobile station in Agency B. For $g_{ms2} = 1$ the system is not in an overload condition and the scheduler is able to guarantee a delay below 20ms. With the decrease of g_{ms2} (the mobile station of Agency A needs more bandwidth in order to achieve its desired service) the scheduler has

Traffic Type	Packet Size [byte]
Voice over IP	64
WWW	512
FTP	1024

Table 7.2: *Packet sizes (at network layer, including IP header) of traffic types used in Scenario 2 and 3.*

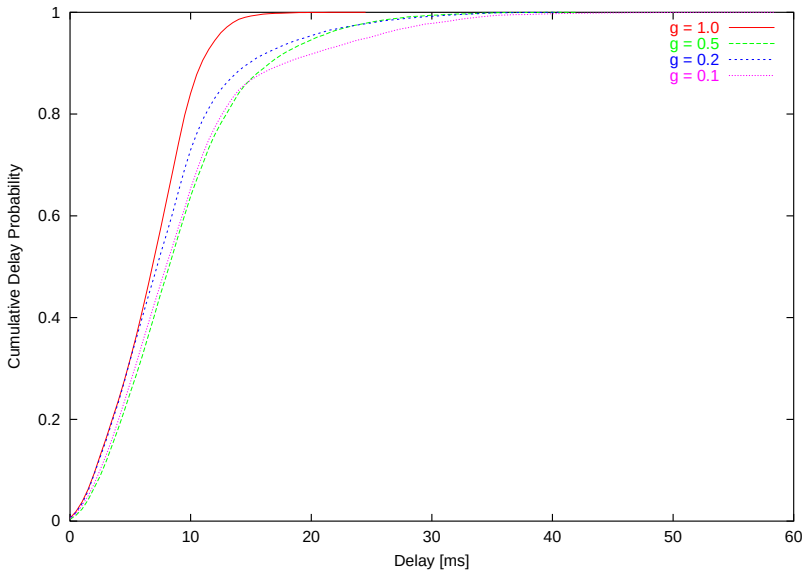


Figure 7.6: *Effect of g_{ms2} (MS 2, Agency A) on the packet delay of VoIP traffic of a mobile in Agency B. The delay of VoIP packets of the competing agency remains almost unchanged although the link quality for MS 2 in the other agency decreases dramatically.*

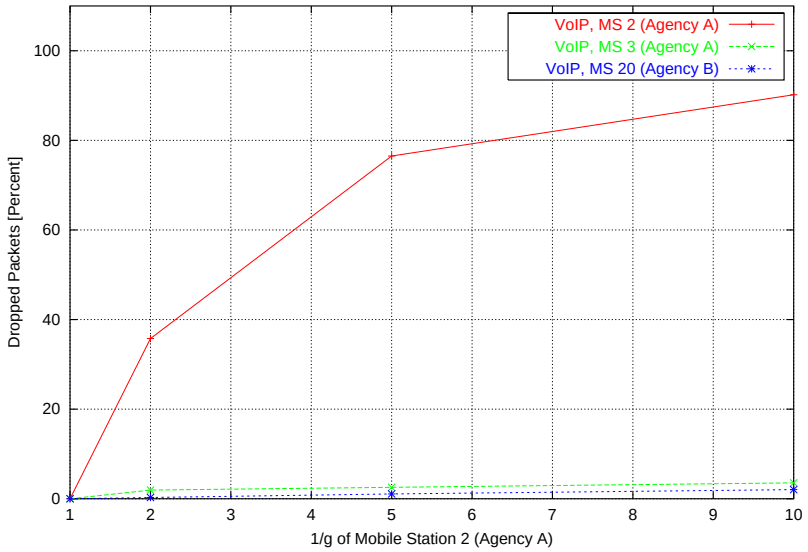


Figure 7.7: *Effect of g_{ms2} on the percentage of packets dropped for the mobile experiencing the bad link (MS 2), a mobile of the same Agency (MS 3) and a mobile of Agency B (MS 20) in an overload situation. MS 20 is not affected because of the competitive scheduling among the two agencies, and the service for MS 3 also is not reduced below its minimal share of resources with which in this case ($g_{ms3} = 1$) it is still able to send most packets on time.*

not enough resources available to guarantee all service curves. However, the simulation illustrates that the algorithm it is able to limit the impact on a mobile in the competing agency. Figure 7.7 compares the number of dropped packets for the two mobile stations. Since the scheduler is able to guarantee Agency B its share of the available resources, the number of dropped packets for a mobile station of Agency B does not significantly increase. In addition, since the scheduler in an overload state reduces the amount of service for subclasses within a cooperative scheduling subtree based on resource consumption, the service for a mobile in Agency A experiencing a perfect channel is also not reduced. (The way the hierarchical link-sharing is configured in this example assumes that it is better to significantly reduce the service for the customer experiencing the bad link (i.e. drop his connection) than to reduce the VoIP share of the other mobiles below their minimal service level.)

7.1.3 Scenario 3

This scenario demonstrates the effects of competitive and cooperative scheduling. For Agency A only three WWW traffic sources are transmitting (Poisson distributed, 160kbit/s), Agency B has two customers transmitting FTP traffic (Poisson distributed, 650 kbit/s). All other classes are idle. The mobile stations of Agency A and the first mobile station of Agency B have a GTR of $g_i = 1$, whereas for MS 2 and MS 3 of Agency A the long-term channel quality is varied. In Figure 7.9 the variation of the bandwidth available to one of the FTP customers and two WWW customers of which MS 1 is experiencing a constantly

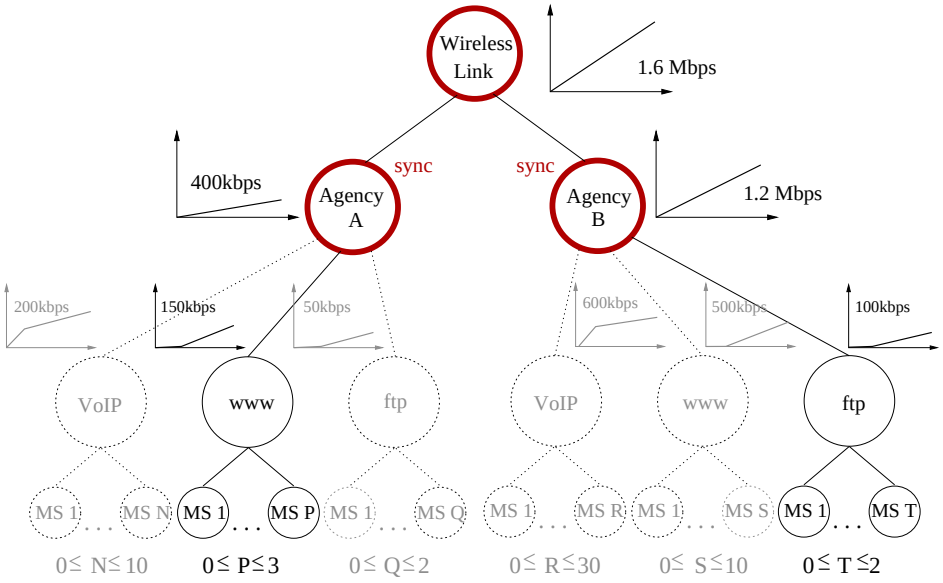


Figure 7.8: *Active classes in Scenario 3. Only WWW sources are transmitting in Agency A and only FTP flows are active in Agency B.*

good and MS 2 is experiencing a varying channel quality is shown. The competitive scheduling among the two agencies guarantees Agency B a share of 1200 Mbit/s of the raw bandwidth regardless of the channel quality which a mobile in Agency A experiences. Since both FTP customers of Agency B have perfect channels, each is able to achieve a goodput of 600 Mbit/s.

As long as $\frac{1}{g}$ of MS 2 (and MS 3, which is not shown in the graph since it receives the same goodput) is less than 4, excess bandwidth is available to the WWW customers of Agency A and because the scheduler is configured to use cooperative scheduling within an agency, it is distributed goodput-based and MS 1 and MS 2 achieve the same rate of goodput. (Therefore MS 2 and MS 3 are allowed to consume a larger share of resources than MS 1 in order to be able to compensate for the low link quality.) When $\frac{1}{g}$ becomes larger than 3, not enough bandwidth is available to guarantee the service curves of all WWW customers (50 kbit/s). Therefore the goodput of MS 2 is degraded based on its resource-consumption in order to avoid violating the service curve of MS 1 because of the bad channel conditions of MS 2 and MS 3.

7.2 Experimental Results

This section presents results of a prototype implementation of the modified algorithm in the Linux kernel (V. 2.4.13) as described in Section 6.2.

7.2.1 Wireless Testbed

The wireless testbed consisted of a desktop PC (AMD K6-II 500 Mhz, 64 MB RAM) serving as the access point (AP) and two mobile stations (laptops, Celeron 550 Mhz processor). Two different sets of wireless LAN cards were used:

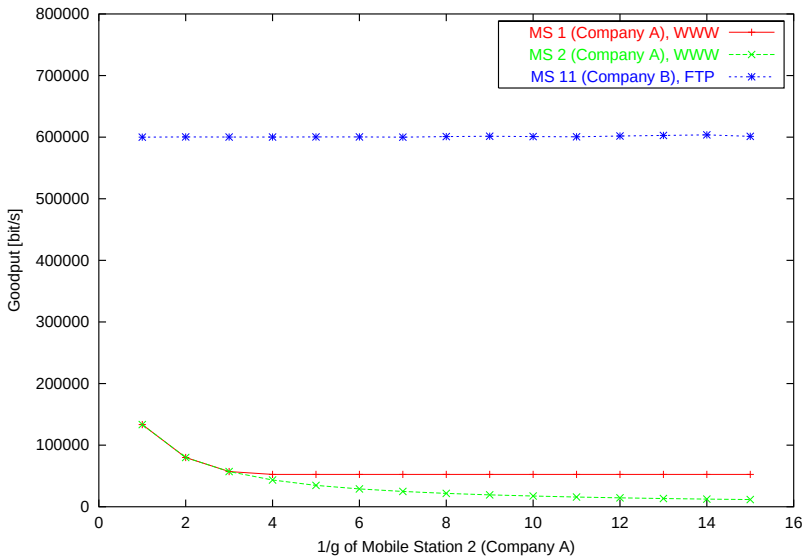


Figure 7.9: *Goodput available for selected mobile stations in Scenario 2. As the amount of resources needed to transmit to MS 2 (and MS 3) of Agency A increases, the goodput available for the competing Agency B is not affected. As long as excess bandwidth is available ($\frac{1}{g} < 4$) within Agency A, it is used to compensate the low link quality of MS 1 (and MS 3).*

1. Raylink 2 MBit/s FHSS [21]

2. Lucent/Avaya Gold 11 MBit/s DSSS [53] (128 bit RC4 WEP encryption)

The Raylink cards operate at 1 and 2 MBit/s according to the IEEE 802.11 standard, the Lucent cards additionally support the 5.5 and 11 Mbit/s modes specified in IEEE 802.11b. Both cards were available in a PCMCIA card version. The laptops had internal PCMCIA card slots whereas a PCMCIA to PCI adapter was used in the AP. For each test run the AP and both mobile stations were equipped with the same card type. For both WLAN cards the transmission rate was set to “auto” (card automatically selects rate/modulation), RTS/CTS was disabled, fragmentation was disabled and the sensitivity threshold was the factory default. All power management features of the cards were disabled. The WEP encryption was enabled for the Lucent/Avaya card only, since the Raylink card does not support any kind of encryption.

In addition to the downlink scheduling the access point also performed IP masquerading. The ARP cache on the access point was set up statically in order to avoid side effects on the measurements. The access point was connected to a 10 Mbit/s Ethernet in which a fixed host generated UDP traffic to each of the mobile destinations. For each data point a measurements of 300 seconds was done, the bit-rates were calculated over a window of one second. Minimum, maximum, average and standard deviation values are computed among the 300 windows.

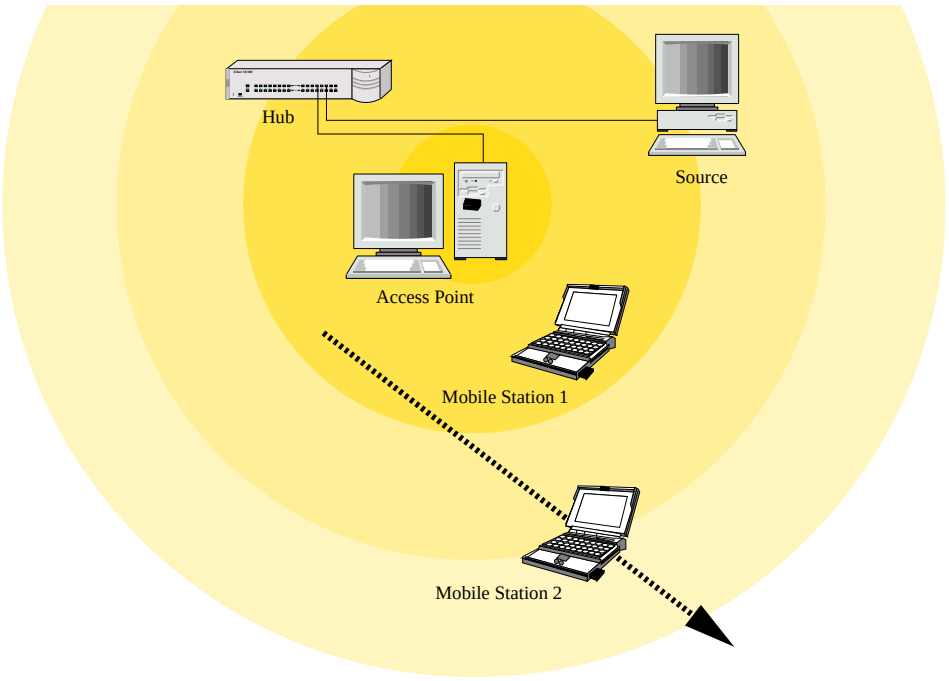


Figure 7.10: *The wireless testbed and main test scenario. While mobile station 1 (MS 1) was constantly in a perfect position, the position of mobile station 2 (MS 2) was varied during a measurement series.*

7.2.2 Test Scenario

The unique property of the proposed algorithm is that it allows a resource-consumption aware scheduling in a link layer independent way. By combining it with well known goodput oriented link-sharing the flexible hierarchical wireless link-sharing model presented in the previous chapters can be achieved. Therefore the focus of the testbed measurements is on demonstrating that resource-consumption oriented scheduling with the proposed algorithm can be done with currently available IEEE 802.11 hardware. To our knowledge none of the other currently discussed algorithms has shown this ability in a 802.11 testbed yet.

The main test scenario is rather simple and similar to the one presented in Section 7.1.1: The scheduler was configured to share the link on a *resource-based* criterion by using a separate synchronization class for each mobile/company: 80% of the link resources were assigned to MS 1 and 20% to MS 2. As a comparison, the same setup was used with the CBQ scheduler included in the kernel, configured to share the link in the same ratio.

The fixed host generated constant bit-rate traffic in UDP packets with a payload of 1024 bytes to both mobile stations at a high rate (3 Mbit/s for MS 1, 1 MBit/s for MS 2 for Raylink tests and 6 Mbit/s for MS 1, 1.5 MBit/s for MS 2 for Lucent/Avaya tests) guaranteeing that the classes of both mobile stations where always backlogged. This was also verified using the traffic control statistics.[2] MS 1 was constantly in a good position and

[2] command "`tc -s class show dev` *device*"

the position of MS 2 was varied. Since the wireless card only reports the link quality to a mobile station when a packet is received, the mobile clients were forced to send an *ICMP echo reply* every second. All packets correctly received at the mobile stations were recorded with timestamp and delay information. The setup is illustrated in Figure 7.10.

Optimal Behavior

The optimal behavior of a resource-based scheduler in this scenario is the following: Since both mobile stations constantly consume all resources available for them (queues are always backlogged) the goodput can be used to determine the amount of resources scheduled for a station: Since the position of MS 1 is not varied, its GTR is constant during the experiment. If a constant fraction of 80% of the available resources is scheduled for MS 1, its goodput should also be constant and independent of the position of MS 2.

Since the amount of resources scheduled for MS 2 is also constant but more resources are needed to transmit to a bad position, its goodput will decrease with decreasing signal quality. This decrease will be more significant than in a goodput oriented scheduling scenario which would allow MS 2 to increase its share of resources. Note that this does not mean that MS 2 is punished for its bad position: it gets a constant share of 20% of the available resources.

7.2.3 Raylink 2 MBit/s Cards (without calibration)

The majority of the tests were conducted using the 2 MBit/s Raylink WLAN cards. As described in Chapter 6 the wireless channel monitor estimates the consumed resources based on the signal level to a mobile destination. In an early stage of the project this was based on a few single goodput measurements. Figure 7.11 illustrates this estimation.

Even with this simple and inaccurate approach the new modified H-FSC scheduler is able to decrease the influence of the position of MS 2 on the amount of resources available for MS 1 as shown in Figure 7.12. (Note that in these first measurements 100 Byte UDP packets were used. Therefore the total achieved goodput is not directly comparable to later experiments, only the relative shares of total goodput achieved by MS 1 and MS 2.) Table 7.3 lists the average bit-rates, the minimum bit-rates and the standard deviation *s*.

7.2.4 Raylink 2 MBit/s Cards (with calibration)

Since the approach of "guessing" the signal to goodput mapping seemed too inaccurate, a method of measuring the mapping and calibrating a wireless channel monitor was introduced (see Section 6.5). Figure 7.13 shows the calibration profile obtained for the Raylink cards. Note that although - as shown in Chapter 6, Figure 6.13(a) - the goodput drops to ≈ 0 for levels below 100, it is mapped to a value of 5 kbit/s in order to avoid completely blocking the queue for MS 2. [3]

[3] This number can be chosen using the following rule of thumb: Assume that the wireless device needs k milliseconds before aborting the transmission of a packet to a mobile station and m milliseconds to transmit to a station under perfect conditions. Then the minimal goodput rate in the profile should be the maximal achievable goodput rate divided by $\frac{k}{m}$.

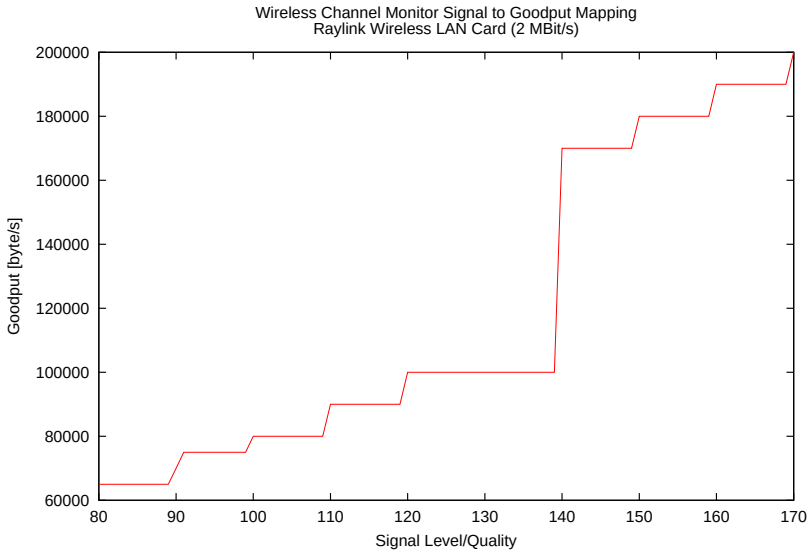


Figure 7.11: *Estimated signal level to goodput mapping for Raylink card (2 Mbit/s, FHSS) used in an early stage of the prototype.*

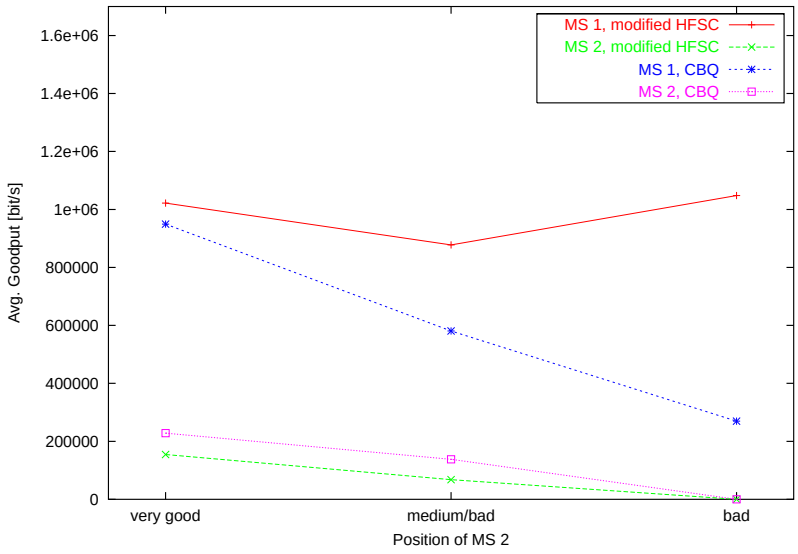


Figure 7.12: *Early prototype results without calibration. The resource-based scheduling decreases the influence of channel quality of MS 2 on the amount of resources available for MS 1.*

Position MS 2	Mobile Station	Scheduler	Average [kbit/s]	Min. [kbit/s]	Max. [kbit/s]	Std. Dev. [kbit/s]
good	1	mHFSC	1022	849	1212	42
		CBQ	949	870	1048	18
	2	mHFSC	154	80	228	30
		CBQ	228	198	257	10
med./bad	1	mHFSC	877	634	1233	80
		CBQ	580	384	727	58
	2	mHFSC	68	30	114	14
		CBQ	138	46	232	20
bad	1	mHFSC	1048	329	1203	108
		CBQ	270	215	321	19
	2	mHFSC	0	0	0	-
		CBQ	0	0	0	-

Table 7.3: *Comparison of first results obtained using an early prototype implementation of the modified H-FSC algorithm (mHFSC) without calibration with those of a not wireless aware CBQ scheduler.*

(In this and all following measurements 1024 byte UDP packets were used instead of the 100 byte packets of the first measurements in order to create a more realistic scenario. Although this does not influence the basic behavior of the scheduler, it increases the achievable total goodput since the relative length of the header is shorter.)

With the calibrated channel monitor the bandwidth available for MS 1 remains almost constant, indicating that the scheduler is able to schedule a constant fraction of 80% of the available resources for MS 1 as shown in Figure 7.13. The small but continuous increase of the rate for MS 1 with decreasing signal quality of MS 2 means that the channel monitor has a tendency to overestimate the amount of resources needed to transmit to MS 2. If the position in which MS 2 has no signal at all (and therefore will not mind getting less then its share) is not taken into account, the error is less than 6% for MS 1 which would mean an error of about 1.5% less than its fair goodput in the “medium/bad” position for MS 2. (In addition we assume that the results could be improved further by manually adapting the signal level to goodput mapping, e.g. by slightly increasing the values for low levels.)

If the CBQ scheduler is used (which does not take the resource-consumption into account) MS 2 can decrease the available goodput for MS 1 by up to 809 kbit/s which means that MS 2 can consume up to 77% of the available resources although it pays only for 20%. Therefore the resource-based scheduling possible with the modified H-FSC scheduler increases the available goodput for MS 1 by up to 340% compared to a CBQ scheduler above the link-layer in this simple scenario.

The detailed results are listed in Table 7.4.

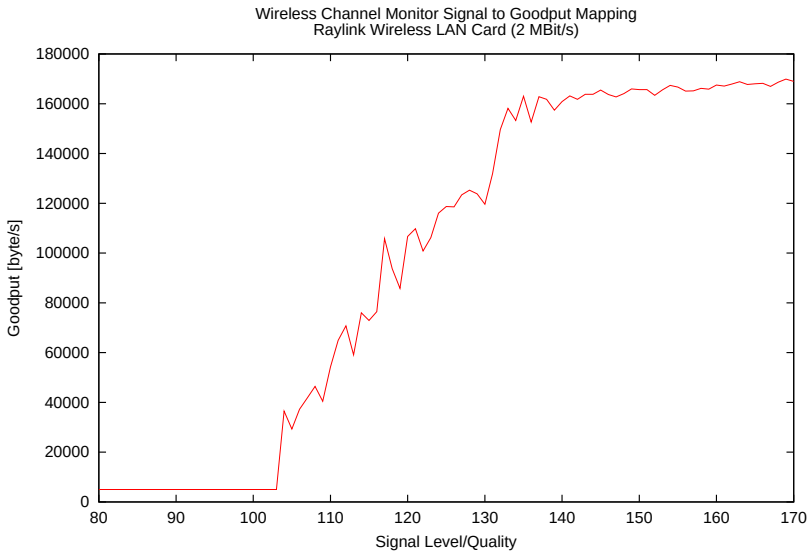


Figure 7.13: *Signal level to goodput mapping for Raylink card (2 Mbit/s, FHSS) obtained using the* WchmonSigMap *calibration tool described in Section 6.5.*

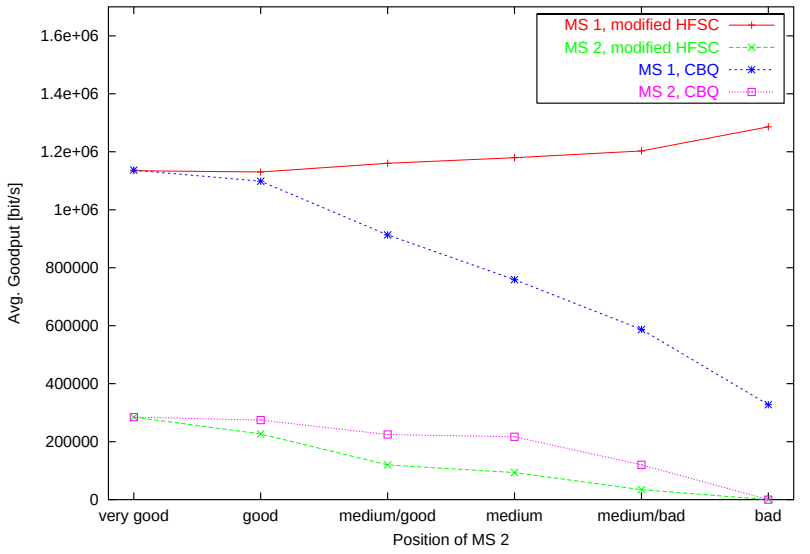


Figure 7.14: *Prototype results for Raylink with calibration. For the modified H-FSC algorithm the bandwidth available for MS 1 remains almost constant when the position of MS 2 is varied.*

Position MS 2	Mobile Station	Scheduler	Average [kbit/s]	Min. [kbit/s]	Max. [kbit/s]	Std. Dev. [kbit/s]
very good	1	mHFSC	1135	757	1372	35
		CBQ	1137	1035	1203	23
	2	mHFSC	285	260	496	14
		CBQ	285	252	480	17
good	1	mHFSC	1130	370	1363	89
		CBQ	1098	1002	1178	28
	2	mHFSC	227	42	934	68
		CBQ	274	244	421	15
med./good	1	mHFSC	1160	917	1380	96
		CBQ	913	151	1439	129
	2	mHFSC	120	17	362	47
		CBQ	225	126	67	63
medium	1	mHFSC	1180	564	1439	131
		CBQ	759	412	98	113
	2	mHFSC	94	8	404	55
		CBQ	217	110	547	36
med./bad	1	mHFSC	1203	732	1439	175
		CBQ	587	269	901	120
	2	mHFSC	35	8	328	33
		CBQ	120	8	244	42
bad	1	mHFSC	1286	1060	1439	48
		CBQ	328	17	766	71
	2	mHFSC	0	0	0	-
		CBQ	0	0	0	-

Table 7.4: *Detailed results for the 2 MBit/s Raylink cards with the calibrated channel monitor.*

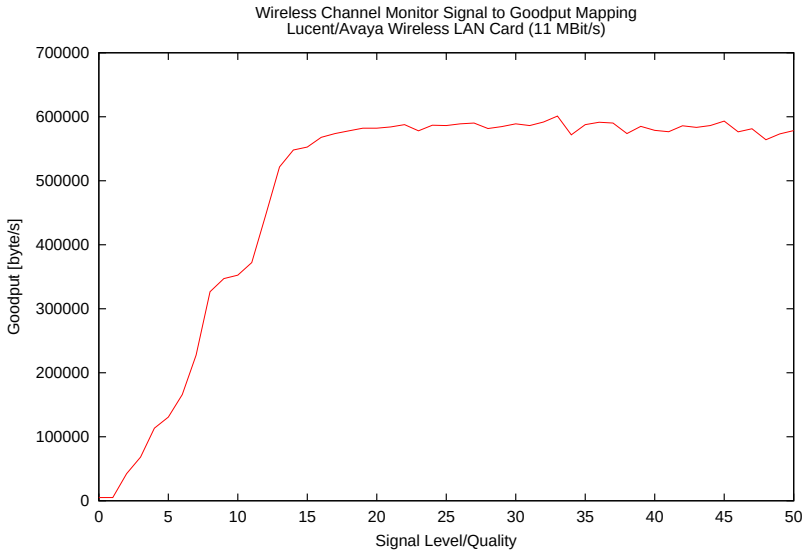


Figure 7.15: *Link quality level to goodput mapping for Lucent card (11 Mbit/s, DSSS) obtained using the* WchmonSigMap *calibration tool described in Section 6.5.*

7.2.5 Lucent/Avaya 11 MBit/s Cards (with calibration)

A limited series of tests was also run with the 11 MBit/s Lucent/Avaya cards. These cards use a different spread spectrum technology[4] and support higher data rates. Therefore the signal to goodput mapping shown in Figure 7.15 cannot be directly compared with that of the Raylink cards.

In addition to reporting the signal strength the Lucent/Avaya card also reports a link quality level which is computed as relation of signal level to noise level. This quality level was chosen as the basis for the calibration and goodput estimation since it was assumed it would be a more accurate indicator for the achievable goodput.

Figure 7.16 and Table 7.5 show that although the position of MS 2 has a significant influence on the goodput of MS 1, the wireless HFSC scheduler provides MS 1 with more than 220% of the goodput of a regular link-sharing scheduler above the link layer even in the "medium/bad" position. The reason for the still suboptimal behavior seems to be that the channel monitor underestimates the amount of resources needed to transmit to a mobile station in a less than optimal position. This could be caused by an inaccurate mapping of quality level to goodput or by the fact that the interval for the channel status update was chosen too large for the higher data rates.

Additional Scenario: Both Mobile Stations in Medium Position

An additional test was done in order to verify the behavior in the case where both mobile stations are in a similar position where they experience a limited signal. Both MS where placed in the "medium/good" position.

[4] direct-sequence spread spectrum (DSSS) instead of frequency-hopping spread sprectrum (FHSS)

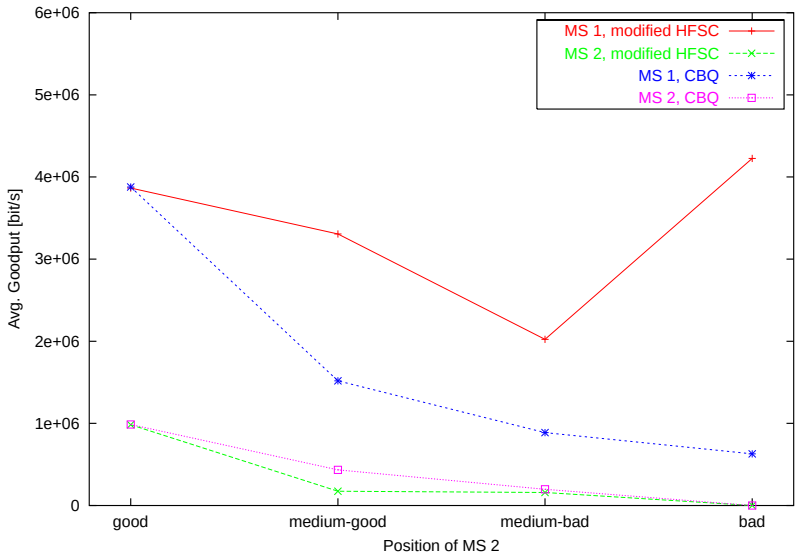


Figure 7.16: *Results for Lucent WLAN cards (11 Mbit/s, DSSS) with calibration.*

Position MS 2	Mobile Station	Scheduler	Average [kbit/s]	Min. [kbit/s]	Max. [kbit/s]	Std. Dev. [kbit/s]
good	1	mHFSC	3866	2608	4721	288
		CBQ	3878	2474	4780	288
	2	mHFSC	983	833	1018	27
		CBQ	986	825	1018	29
med./good	1	mHFSC	3306	1060	4914	578
		CBQ	1518	740	2449	245
	2	mHFSC	174	8	446	74
		CBQ	433	151	706	71
med./bad	1	mHFSC	2022	67	5310	848
		CBQ	888	513	1380	180
	2	mHFSC	158	8	337	66
		CBQ	197	8	412	87
bad	1	mHFSC	4225	437	5377	527
		CBQ	630	497	825	37
	2	mHFSC	0	0	0	-
		CBQ	0	0	0	-

Table 7.5: *Detailed results for the Lucent/Avaya 11 MBit/s wireless cards.*

In this case the modified H-FSC scheduler as well as CBQ scheduled a fraction of 80% of the total goodput for MS 1 and a fraction of 20% for MS 2. This is the expected behavior since in this case both stations have the same GTR. (As it was the case in the previous scenarios when both stations where in the “good” position.)

7.2.6 Summary of Experimental Results

The results demonstrate that the algorithm is able to perform resource-based scheduling on currently available IEEE 802.11 hardware. If an accurate mapping of signal strength to goodput is used, it can keep the average amount of resources available for a mobile station almost constant. Fine-tuning of factors such as the optimal update interval of the channel status, knowledge about the way in which the wireless card adapts its modulation scheme, etc. is likely to improve the the estimations of the wireless channel monitor and therefore the accuracy of the scheduler.

8. Summary and Conclusions

At the core of any QoS architecture is a scheduling algorithm which distributes the available bandwidth among companies, users and applications. Mechanisms for controlled link-sharing and quality of service support are currently implemented above the link layer in local area networks conforming to the IEEE LAN standards. This assumes that sending a specific amount of data consumes the same amount of resources for each destination node. While this is valid for wireline networks, in the case of wireless LANs the resource-consumption depends on the quality of the wireless link for a mobile station and is highly variable due to retransmissions, adaptive modulation and other error correction mechanisms. Therefore this approach is unable to guarantee a constant level of service for a mobile station since the scheduler is not aware of the amount of resources needed by the link layer to transmit to a mobile host.

The majority of the currently discussed packet scheduling algorithms for wireless networks probe the wireless channel towards a specific destination before sending a packet, deferring transmission if the channel is in a bad state. Meanwhile a mobile station which has a good channel is serviced. Various techniques have been developed to compensate at a later point in time for a station which experienced a bad channel in the past. In order to be able to perform channel probing and be aware of retransmissions, these algorithms have to be integrated in the link layer which makes them hard to deploy over currently available hardware. While it seems feasible that some kind of channel-state dependent scheduling will be integrated in future wireless link layers, the configuration of complex link-sharing hierarchies would best be handled in a more hardware independent way, i.e. as part of the operating system.

This technical report presented a new approach to use *long-term* information on the channel-state of a mobile station in order to improve the controlled sharing of a wireless link. The link layer independent scheduler estimates the resources needed to transmit to a given destination based on its recent knowledge on the channel capacities of each mobile station sharing the link. Using this information two types of wireless scheduling are possible: goodput-based (cooperative) scheduling and resource-consumption based

(competitive) scheduling. Since neither technique by itself is sufficient to describe wireless link-sharing hierarchies adequately, a new link-sharing model was introduced which makes it possible to integrate both types in a single hierarchy by using *synchronization classes*.

The Hierarchical Fair Service Curve Algorithm (H-FSC) was modified according to this wireless link-sharing model. The modified algorithm adapts the service curves for a mobile station depending on its resource-consumption and the resources consumed by other mobiles stations in the same cooperative scheduling subtree. Simulations confirm that the algorithm is able to guarantee a specific amount of resources to a customer and therefore can be used to equalize the cost per revenue for an ISP.

In order to demonstrate that the algorithm can be used with currently available wireless technology, a Linux based prototype was developed. The Linux traffic control architecture was extended to support the implementation of channel-state dependent scheduling for a wireless network interface. A wireless channel monitor component was introduced which monitors the capacity of a wireless link using signal strength information provided by a modified device driver. Since the relation of the signal strength reported by the device driver to the available capacity of the wireless channel is hardware dependent a method to calibrate a wireless channel monitor was developed.

Based on the code of the H-FSC scheduler available for FreeBSD the wireless H-FSC variant was implemented within this framework. The prototype implementation was tested with two different kinds of wireless network cards: The majority of measurements were done using *Raylink* 2 Mbit/s FHSS cards conforming to the original IEEE 802.11 standard. Additional results obtained with 11 Mbit/s *Lucent* DSSS cards using the higher data rates specified in IEEE 802.11b demonstrate that the algorithm does not depend on a specific type of wireless network card. Prototype results for a simple resource-based scheduling scenario are in accordance with simulation results and demonstrate that the algorithm is able to perform resource-based scheduling on standard hardware. Using the *Raylink* cards and a calibrated wireless channel monitor the scheduler is able to provide an almost constant fraction of resources to a customer. In test scenarios with a good and a very bad link this improves the goodput for the customer with a good link by up to 340% compared to a conventional goodput-based scheduling above the link layer.

Future Research

Our measurements indicate that in some cases - especially if the 802.11b *Lucent* cards are used - the estimations of our wireless channel monitor are not very accurate. Taking more information available at the device driver level (e.g. type of modulation, number of retransmissions) into account should improve the information provided by the channel monitor.

Furthermore a promising extension of the presented wireless link-sharing model would be to combine it with a short-term channel-state dependent scheduling at the link layer level: Most wireless network cards already buffer a small number of packets. Rather than send these packets in FIFO order, the network card could probe the channel and try to avoid

transmitting packets on channels in a bad state. This would require no modification of the proposed architecture - the hierarchical wireless link-sharing would still be enforced above the link layer - and is likely to result in increasing the total system goodput.

A. Kernel Configuration and Developed Testbed Tools

A.1 Access Point Setup - A To-Do List

The following list summarizes the necessary steps to set up a freshly installed Linux system[1] for using the developed kernel extensions and modified H-FSC scheduler:

1. Download and unpack the following packages:
   - Modified H-FSC scheduler and Linux wireless scheduling patches (the prototype implementation developed as part of this work) [58]
   - Linux 2.4.X kernel sources [23]
     (patch was developed for 2.4.13 but should work with all 2.4.X kernels)
   - iproute2 sources (Version 001007 or newer) [25]
   - PCMCIA card services (Version 3.1.29 or newer) [41]
     (Only necessary if the wireless channel monitor "driver" is to be used.)
   - optional for packet scheduling simulations: TCSIM [1]
2. Apply our kernel patch in the kernel source directory.
3. Configure (Section A.2), compile and boot the kernel.
4. Apply our `iproute2` patch in the `iproute2` directory.
5. Compile and install the `iproute2` package.
6. For wireless scheduling with device driver support:

[1] It is assumed that the wireless card is correctly installed and all other desired features besides the QoS scheduling (e.g. routing, DHCP, network address translation) are configured and tested.

- If a Lucent/Avaya or Raylink card is available replace the corresponding driver in the PCMCIA card services directory with their modified versions. If a different wireless card is used, modify its device driver to include calls to the wireless channel monitor interface. (see Table 6.5 in Chapter 6)
- Compile and install the PCMCIA card services.
  (One can verify that the correct version of the PCMCIA card services is used by the fact that a "`make config`" shows "*Wireless scheduling support is enabled.*" and "*Support for wireless channel monitoring is enabled.*" in the list of configured options.)
- Calibrate the used wireless card with the calibration tool. (Section A.3.1)

7. Configure the desired link-sharing hierarchy using the `tc` program. (command line options for the modified H-FSC algorithm are listed in Section 6.4)

A.2 Kernel Configuration

Table A.2 lists the settings of relevant configuration constants when compiling the kernel sources using `make config`, `make xconfig` or `make menuconfig`. The queue, scheduler and classifier options correspond to those listed in Section 6.1.1 and although it is necessary only to activate those which will later be used at the node, it is recommended to compile all of them as modules by selecting `[m]` whenever possible. Thus all components are available without wasting precious kernel memory and one does not have to compile the kernel again when a component is needed whose usage was not anticipated.

Options printed in *italics* are only available if the wireless scheduling patch developed as part of this project (see Chapter B) has been applied.

Clock Source Configuration

A configuration setting which currently cannot be selected via the well-known kernel configuration tools is the setting of the kernel clock source. This determines the way in which the kernel/a kernel module obtains information about the current time. It can be done by using the current `jiffie` count (a global kernel variable which is periodically increased by the timer interrupt), by using the `gettimeofday` routine or relying on the CPU timestamp function which is available on most modern processors (it was introduced with the Intel Pentium processor). The setting is configured in `./net/pkt_sched.h` in the kernel source directory and should be set to `PSCHED_CPU`. All other choices are not accurate enough to guarantee precise scheduling.

Question	Option	Setting
Prompt for development and/or incomplete code/drivers?	CONFIG_EXPERIMENTAL	[y]
Kernel/user netlink socket	CONFIG_NETLINK	[y]
Routing messages	CONFIG_RTNETLINK	[y]
TCP/IP networking	CONFIG_INET	[y]
QoS and/or fair queueing	CONFIG_SCHED	[y]
CBQ packet scheduler	CONFIG_NET_SCH_CBQ	[m] (or [y])
CSZ packet scheduler	CONFIG_NET_SCH_CSZ	[m] (or [y])
The simplest prio pseudoscheduler	CONFIG_NET_SCH_PRIO	[m] (or [y])
RED queue	CONFIG_NET_SCH_RED	[m] (or [y])
SFQ queue	CONFIG_NET_SCH_SFQ	[m] (or [y])
TEQL queue	CONFIG_NET_SCH_TEQL	[m] (or [y])
TBF queue	CONFIG_NET_SCH_TBF	[m] (or [y])
GRED queue	CONFIG_NET_SCH_GRED	[m] (or [y])
Diffserv field marker	CONFIG_NET_SCH_DSMARK	[m] (or [y])
Ingress qdisc	CONFIG_NET_SCH_INGRESS	[m] (or [y])
QoS support	CONFIG_NET_QOS	[y]
Rate estimator	CONFIG_NET_ESTIMATOR	[y]
Packet classifier API	CONFIG_NET_CLS	[y]
TC index classifier	CONFIG_NET_CLS_TCINDEX	[m] (or [y])
Routing table based classifier	CONFIG_NET_CLS_ROUTE4	[m] (or [y])
Firewall based classifier	CONFIG_NET_CLS_FW	[m] (or [y])
U32 classifier	CONFIG_NET_CLS_U32	[m] (or [y])
Special RSVP classifier	CONFIG_NET_CLS_RSVP	[m] (or [y])
Special RSVP classifier for IPv6	CONFIG_NET_CLS_RSVP6	[m] (or [y])
Traffic policing (needed for ingress/egress)	CONFIG_NET_CLS_POLICE	[m] (or [y])
H-FSC packet scheduler	CONFIG_NET_SCH_HFSC	[m] (or [y])
Wireless scheduling support	CONFIG_NET_WIRELESS_SCHED	[y]
channel-state aware FIFO queue (CSFIFO)	CONFIG_NET_SCH_CSFIFO	[m] (or [y])
simple wireless channel simulator (CHSIM)	CONFIG_NET_SCH_CHSIM	[m] (or [y])
dummy - wireless channel monitor (dummy)	CONFIG_NET_WCHMON_DUMMY	[m] (or [y])
ratio - wireless channel monitor (ratio)	CONFIG_NET_WCHMON_RATIO	[m] (or [y])
driver - wireless channel monitor (driver)	CONFIG_NET_WCHMON_DRIVER	[m] (or [y])

Table A.1: *Kernel Configuration Options*

A.3 Developed Testbed Tools

In the following the usage of a few additional tools developed for the wireless testbed is described in more detail.

A.3.1 Wireless Channel Monitor Calibration Tool

As mentioned in Section 6.5 the amount of goodput which can be achieved by a mobile station in a location where a specific signal level is indicated depends on many factors, e.g. the type of device which is used, its configuration settings, etc. In order to be able to parameterize a wireless channel monitor accurately a calibration tool was developed which records the goodput for a mobile station in relation to its link level. This tool, the *Wireless Channel Monitor Signal to Goodput Mapper (WchmonSigMap)*, and its usage are briefly presented in this section.

Installation

The tool is a Java 2 application and requires a working installation of Sun's Java 2 runtime environment [52]. After unpacking the source archive in a local directory, the tools can be compiled with `make all`. (It is assumed that the Java compiler `javac` and the virtual machine `java` are in the path - if this is not the case, adapt the corresponding variables within the `Makefile`.) This generates the necessary java `.class` files and a native library `WchmonSigMap_Wstats.so` which performs the IOCTL calls using the Linux wireless extensions [54] via the Java Native Interface (JNI). This library is copied to `/usr/lib` by the `Makefile`. The main application is started with `java WchmonSigMap.WchmonSigMap` in the application's directory, the simple UDP packet generation tool is executed with `java WchmonPackGen`.

Configuration

When the `WchmonSigMap` application is started, the main window appears as shown in Figure A.1. If signal level, quality and noise level are all 0 it means that the application was not able to read the statistics from `/proc/wireless`. In this case the program needs to be configured using the *File→Configure* option (Figure A.2).

The following options can be changed:

- **Interface:** The name of the wireless network interface. (default: `eth1`)
- **Signal Quality Indicator:** The property which characterizes the current link quality. In case the card provides a signal quality byte this usually is "quality". If the card provides only the signal level indicator (e.g. the Raylink card) this should be set to "signal". *This must be set to the same property which is sent by the modified device driver (see Section 6.5) to the wireless channel monitor.* (default: "quality")
- **Size of Avg. Window:** Number of packets over which the goodput-average is estimated. Increase this value if goodput measurements for the same quality level are highly varying, decrease the value for more accuracy in case of varying levels (e.g. the test mobile is moving). (default: 100)

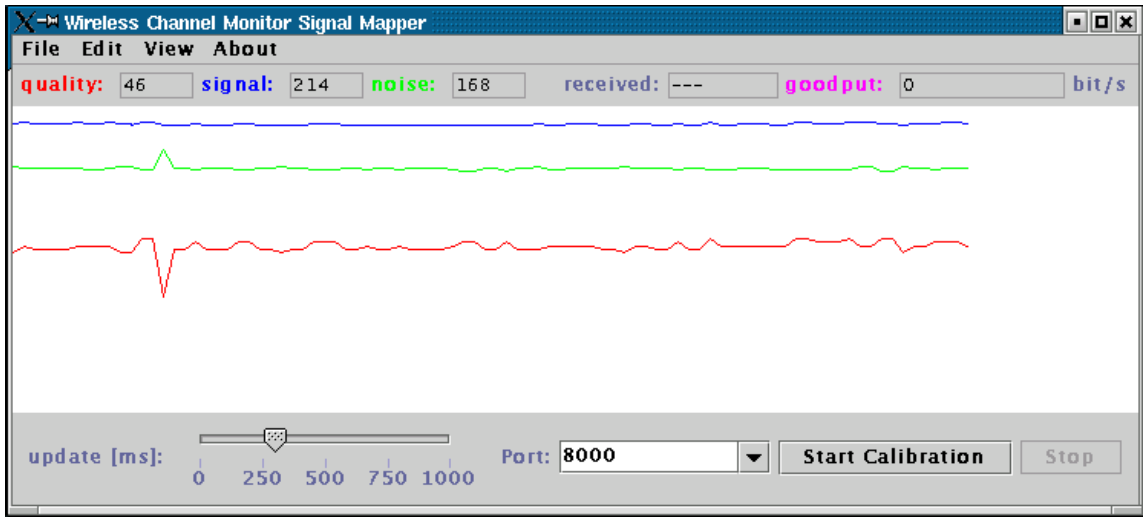


Figure A.1: *WchmonSigMap, the channel monitor calibration tool.*

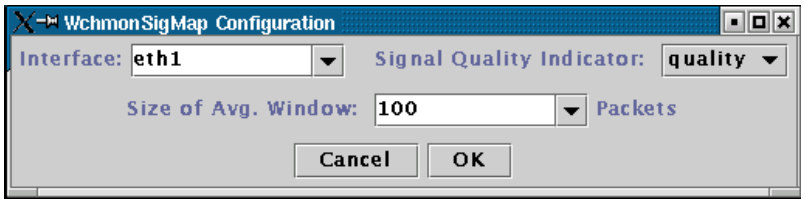


Figure A.2: *Configuring the calibration tool.*

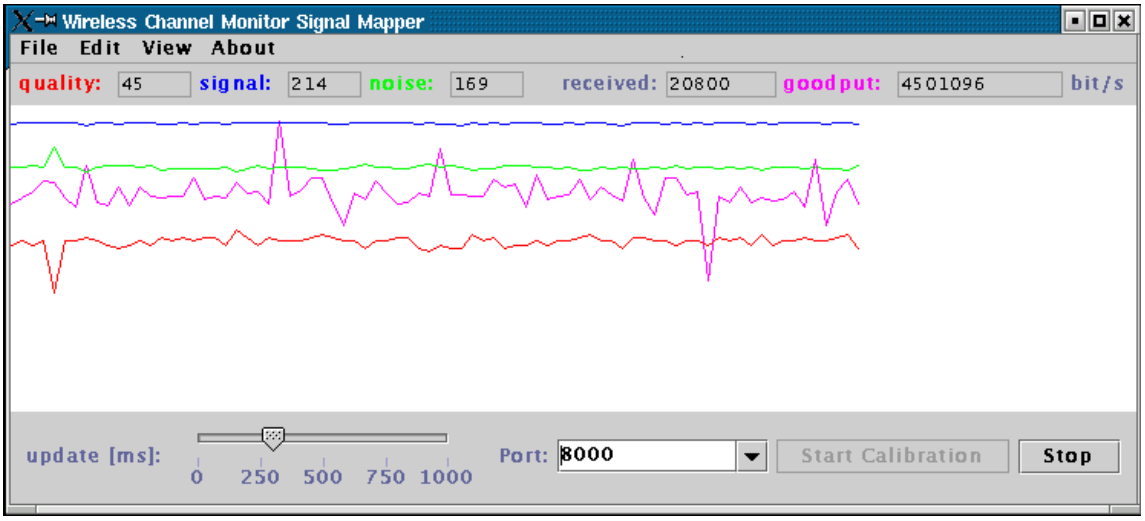


Figure A.3: *Measuring goodput using the calibration tool.*

If the configuration is valid it will be accepted upon pressing the "OK" button, otherwise an error message box is shown. Besides these options the main screen allows the adjustment of the port on which the application listens for incoming packets and the tuning of the time interval in which statistics are read from the wireless device.

Note: Signal Levels in Ad-Hoc Mode

Some wireless device drivers (e.g. the `wvlan_cs` driver for the Lucent cards) do not provide status information if the card is running in ad-hoc mode. In this case use the `iwspy` program of the wireless tools to add the IP address of your access point to the spy list: e.g. `iwspy eth1 +192.168.23.254`. Verify that status information is available with `iwconfig eth1`.

Measuring Goodput

Deactivate all mobile stations except the one which runs the calibration tool. In the *View* menu activate the goodput graph and any other information you would like to observe. Select the port on which the calibration tool is supposed to listen for incoming UDP packets (default: 8000) and start generating UDP traffic with the maximum rate at the access point to this port on the mobile station so that the wireless link is always saturated. For this purpose the `wchmonPackGen` tool described in A.3.2 can be used or any other similar tool (e.g. `mgen`). The size of the generated packets should be similar to the average packets used in your wireless applications. Then click on the "Start Calibration" button. The application will start displaying the number of received packets and the current goodput-rate in the upper right corner (Figure A.3). Move the mobile station to different locations with a wide range of signal levels (observe the level/goodput graphs). For reliable measurements the station should stay at each location for at least 20-30 Minutes. The measurements can be stopped/interrupted using the "Stop" button.

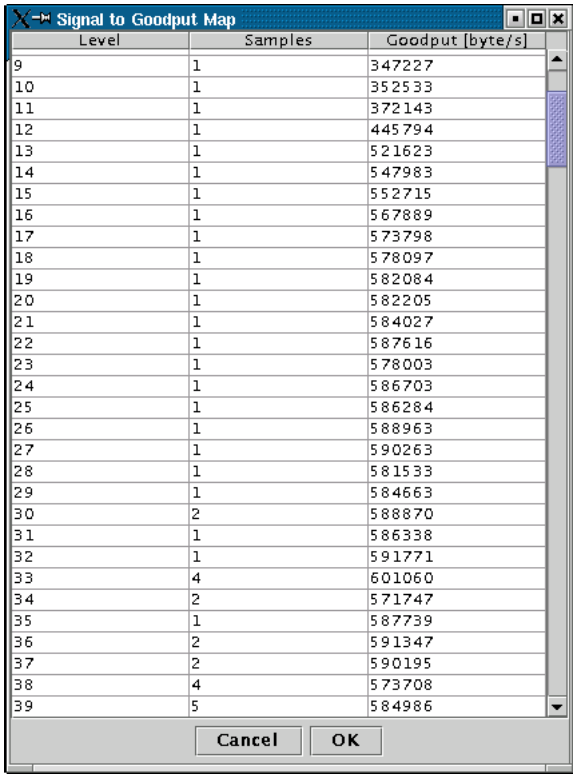


Figure A.4: *Editing the results.*

Manually Editing the Table

The results can be inspected and edited by selecting *Edit→Edit table...* (Figure A.4). Note that the tool assumes a minimal goodput of 5000 byte/s which is necessary in order to avoid that a scheduler completely blocks sending data to a mobile destination. In addition, if no data for a specific level is available (number of samples is 0), its goodput is assumed to be at least as good as the next available measurement of a lower level.

Saving, Loading and Exporting the Data

Tables are saved in the same format which is used by the wireless channel monitor `driver` (Section 6.5). Therefore they can simply be copied to `/proc/net/wchmon_gtr_map` in order to update the estimates of an installed wireless channel monitor module. Export in an ASCII format (e.g. for GnuPlot) is also possible.

A.3.2 Simple UDP Packet Generator

A simple tool which can be used to generate UDP packets to the mobile station running the calibration utility is the *Simple UDP Packet Generator* shown in Figure A.5. (For requirements and installation see Section A.3.1.) It is started with the command `java WchmonPackGen` and continuously generates UDP packets of the specified size at the maximum rate. The destination port setting must match the port setting of the calibration tool. 6.5

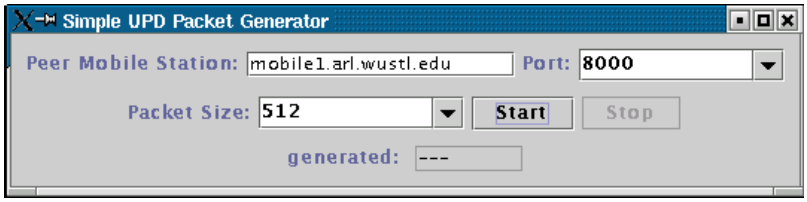


Figure A.5: *Simple UDP packet generation tool.*

A.4 Extended Simulation Environment - Examples/Additional Information

In the following additional examples and ideas concerning the simulation environment are presented.

A.4.1 Note: Kernel-level Simulation of Wireless Channels

This subsection describes an additional application of the wireless channel simulation queuing discipline which is a side-effect of the way it was implemented. It was not used as part of the thesis work and was only rudimentarily tested but we believe that this approach could simplify the testing of new wireless protocols implemented above or within the network layer.

The evaluation of the performance of network layer, transport layer or application layer protocols used over a wireless link is often very problematic because of the fast and uncontrollably changing conditions of the wireless environment. Since the exact conditions of a measurement setup are hard to reproduce, evaluation is often done by implementing them in a simulation environment approximating the behavior of the lower layer protocols and the wireless link.

Since the channel simulation queuing discipline presented in Section 6.3.1 has the interface of a regular queuing discipline and requires only that it is polled regularly by a constant bit-rate interface, it can also be used in a real kernel environment. If it is used as the root queuing discipline on a standard wire-line network interface, whose bandwidth is considerably larger than that of the simulated wireless device, the additional effects of the wire-line transmission are small. Such a setup has various advantages compared to the pure simulation or measurement approaches:

1. By parameterizing the wireless channel model exactly the desired link behavior can be simulated.
2. Channel conditions are reproducible.
3. The protocol does not have to be implemented separately for the simulation since the same implementation can be used for simulation and measurements.
4. In case of a transport or application layer protocol, the actual implementation of lower layer (network/network and transportation layer) protocols is used, not a separate simulated version.

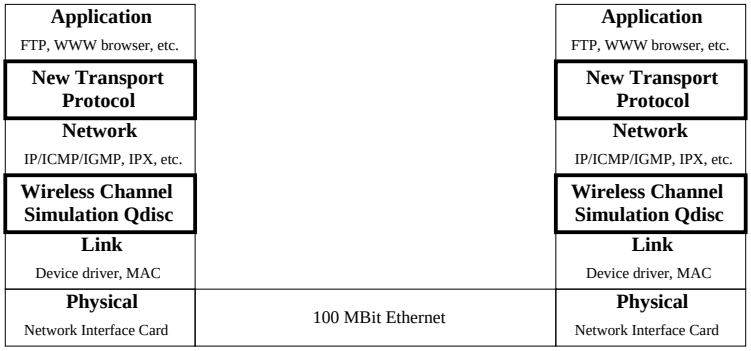


Figure A.6: *Example for the performance evaluation of a new/improved transport protocol using the wireless simulation queuing discipline.*

5. The simulated wireless device can be used to evaluate the behavior of the higher layer protocol on technologies not yet available.

An example for the performance evaluation of a transport protocol which is used on a simulated wireless link is shown in Figure A.6. While the basic idea of simulating a desired device behavior in the kernel is not new (e.g. the so called *softlink device* which is able to direct packets to user space for manipulation can be used for this purpose) using a qdisc for this purpose has two advantages: The different parts of traffic control can be flexibly combined to create the desired system behavior and the implemented channel model can be tested in user space (using TCSIM) as well as in kernel space. Furthermore the traffic control layer is the last device independent layer of the operating system and is destined for delaying and re-ordering of packets.

A.4.2 Example for Usage of Wireless Simulation Qdisc

Listing A.1 shows how the `chsim` queuing discipline is used in TCSIM to simulate scheduling in a wireless environment.

Listing A.1: *Example: Using the wireless channel simulation qdisc.*

```
/* csfifo_on_chsim
 *
 * Linux Wireless Scheduling Example
 *
 * goodput optimizing wireles fifo qdisc using the "ratio"
 * wireless channel monitor running on the wireless channel
 * simulation qdisc
 */

#include "ip.def"

/*
 * define packets
```

```
15  */

    #define PACKET(ms) UDP_PCK($src=10.0.0.1 $dst=10.0.0.##ms \
                                $sport=PORT_USER $dport=PORT_HTTP)

20  #define PAYLOAD(n) (n) x 980      /* +IP header = 1000 bytes */


    dev eth0  1400 /* 1.4 Mbps */

25  /* installing the module can only be done after the "dev"
       command! We are using the simple wireless channel monitor
       which estimates the channel quality based on the ratio of
       goodput/throughput                                        */

30  insmod /usr/src/tcng/tcsim/modules/wchmon_ratio.o

    /* setup TC architecture */

    /* this is for testing only ... just arbitrary values */
35
    /* first we set up the wireless channel simulation qdisc
       including the default wireless channel */

    tc qdisc add dev eth0 root handle 1:0 chsim limit 1 p_gb 0.04 \
40    p_bg 0.3 e_P 0.7 avg_size 1000 max_retrans 12 channel_rate \
      1400kbit clear_dst_addr

    /* Add another simulated channel, bad quality */
    tc class add dev eth0 parent 1:0 classid 1:30 chsim p_gb 0.1 \
45    p_bg 0.05 e_P 0.9

    /* And a third channel which has very good quality */
    tc class add dev eth0 parent 1:0 classid 1:20 chsim p_gb 0.0001 \
       p_bg 0.5 e_P 0.5
50
    /* Now the filters which select the packet for each channel are
       set up. While in a running system the selection would be based
       on the MAC address, we identify the different destinations by
       their IP address in the simulation. It does not matter, since
55     it is a one to one mapping of IP to MAC address by ARP anyway
       and we are able to use the existing filters this way. TCSIM
       also has no way to specify the MAC address of a packet that
       is sent.                                                  */

60  tc filter add dev eth0 parent 1:0 protocol ip u32 match \
       ip dst 10.0.0.1 flowid 1:20
    tc filter add dev eth0 parent 1:0 protocol ip u32 match \
```

```
    ip dst 10.0.0.2 flowid 1:30

/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    Now the setup of our simulated wireless environment is done
    and we start setting up the packet scheduling on top of it
    - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */

/* We add the csfifo qdisc to schedule on the simulated
    the wireless channel (note: limit for csfifo in packets!) */

/* a) the "real" CS-FIFO */
tc qdisc add dev eth0 parent 1:0 csfifo wchmon ratio \
  limit 20 lookahead 20 probe 500 threshold 10 probability 0.7

/* b) a usual FIFO (uncomment it for comparison)
tc qdisc add dev eth0 parent 1:0 pfifo limit 20 */

/* generate packets */

/* a) for a station on the default channel */
every 0.02 s send PACKET(255) PAYLOAD(255)

time 0.1 s

/* b) for the channel to mobile station 1 */
every 0.02 s send PACKET(1) PAYLOAD(1)

time 0.1 s

/* c) for the channel to mobile station 2 */
every 0.021 s send PACKET(2) PAYLOAD(2)

/* declare duration of traffic simulation */

time 120s
end 130s
```

A.4.3 Modified H-FSC - Simulation Scenario 1

Listing A.2 lists the script used to configure the link-sharing hierarchy simulated in Scenario 1 of the simulations.

Listing A.2: *Configuration of link-sharing hierarchy of Scenario 1*

```
/*
 * TCSIM: wireless H-FSC configuration for simulated Scenario 1
 */

```

```
#include "ip.def"

dev eth1 6144      /* 6 Mbit/s assumed max. goodput for 11 MBit/s card */

/* Device to be configured */
#define DEVICE eth1

/* max. queue length */
#define QUEUE_LENGTH 15

/* IP addresses of mobile hosts */
#define MS1IP 192.168.23.1
#define MS2IP 192.168.23.2

/* packets */
#define PAYLOAD(n) (n) x 980
#define PACKET(n) UDP_PCK($src=10.0.0.249 $dst=##n \
                          $sport=PORT_HTTP $dport=5000)

/* install wireless channel monitor module */
insmod /usr/src/tcng/tcsim/modules/wchmon_ratio.o

/* set up wireless simulation */

tc qdisc add dev DEVICE root handle 1:0 chsim limit 1 p_gb 0 p_bg 1 \
   e_P 0 avg_size 1000 max_retrans 10 channel_rate 6144 kbit \
    clear_dst_addr

tc class add dev DEVICE parent 1:0 classid 1:2 chsim p_gb 0 p_bg 1 \
   e_P 1 adapt_mod 1

tc filter add dev DEVICE parent 1:0 protocol ip u32 match \
   ip dst MS2IP flowid 1:2


/* add root queue disc for HFSC */
tc qdisc add dev DEVICE parent 1:0 handle 10:0 hfsc bandwidth 6Mbit \
wireless wchmon ratio reducebad 100

/* add class for company A */
/* SLA: 4424 kBit/s at g=0.9 -> 4915 kBit/s resources
   (80% of 6MBit) */
tc class add dev DEVICE parent 10:0 classid 10:1 hfsc \
[sc 0 0 4915 kbit ] sync

/* add class for company B */
/* SLA: 614 kBit/s at g=0.5 -> 1229 kbit/s of resources
   (20% of 6 MBit */
```

```
tc class add dev DEVICE parent 10:0 classid 10:2 hfsc \
[sc 0 0 1229 kbit ] sync

/* add class for mobile one */
tc class add dev DEVICE parent 10:1 classid 10:100 hfsc \
[sc 0 0 4424 kbit ]
/* set queue length for mobile one */
tc qdisc add dev DEVICE parent 10:100 pfifo limit QUEUE_LENGTH

/* add class for mobile two */
tc class add dev DEVICE parent 10:2 classid 10:200 hfsc \
[sc 0 0 614 kbit ]
/* set queue length for mobile two */
tc qdisc add dev DEVICE parent 10:200 pfifo limit QUEUE_LENGTH

/* define filters : */
tc filter add dev DEVICE parent 10:0 protocol ip prio 100 u32 match \
ip dst MS1IP flowid 10:100

tc filter add dev DEVICE parent 10:0 protocol ip prio 100 u32 match \
ip dst MS2IP flowid 10:200


/*
 * generate traffic : 4800 kbit/s to MS 1 and
 *                    614 kbit/s to MS 2
 *
 * (packet payload 980 bytes + header)
 */

every 0.00165s send PACKET(MS1IP) PAYLOAD(1)
every 0.01327s send PACKET(MS2IP) PAYLOAD(2)

time 180s
```

A.4.4 Modified H-FSC - Simulation Scenarios 2 and 3

Listing A.3 shows the configuration script used to configure the hierarchical link-sharing structure of Figure 7.5 used in the Scenario 2 and Scenario 3 of the simulation section.

Listing A.3: *Configuration of link-sharing hierarchy of Scenarios 2 and 3*

```
/*
 * Linux Wireless Scheduling
 *
 * example script for modified hierarchical fair service curve scheduler
 *
```

```
 * Simulation Scenario
 * ----------------------------
 *
 *  1 base station , 1.6 MBit downlink capacity to be scheduled
 *
 *  two agencies , each with VoIP, WWW, ftp
 *
 *  traffic generation done by " trafgen ": markov-modeled ON/OFF CBR
 */

#include " ip.def"

#define VOIP_QUEUE_LIMIT 3

#define PAYLOAD_VoIP(n) (n) x 44        /* +IP header = 64 bytes */
#define PAYLOAD_www(n) (n) x 492
#define PAYLOAD_ftp(n) (n) x 1004

#define PORT_VOIP 0x1111                /* port of VoIP application */
#define PORT_FTP 21

#define PACKET_www(ms) TCP_PCK($src=10.0.0.249 $dst=10.0.0.##ms \
                       $sport=PORT_HTTP $dport=PORT_HTTP)
#define PACKET_ftp(ms)  TCP_PCK($src=10.0.0.249 $dst=10.0.0.##ms \
                       $sport=PORT_FTP $dport=PORT_FTP)
#define PACKET_VoIP(ms) UDP_PCK($src=10.0.0.249 $dst=10.0.0.##ms \
                       $sport=PORT_VOIP $dport=PORT_VOIP)

/* configure wireless device */

dev eth1 1600 /* 1.6 Mbit max. capacity of wireless link */

insmod /usr/src/tcng/tcsim/modules/wchmon_ratio.o

/* setup TC architecture */

/* =============== start of wireless channel simulation ============= */

/* the default class is used for the 14 good channels */

tc qdisc add dev eth1 root handle 1:0 chsim limit 1 p_gb 0 p_bg 1 \
   e_P 0 avg_size 500 max_retrans 0 channel_rate 1600 kbit \
    clear_dst_addr

/* define class for the bad channel , adaptive modulation of 1/10 */
tc class add dev eth1 parent 1:0 classid 1:20 chsim p_gb 0 p_bg \
1 e_P 0 adapt_mod 10
```

```
tc filter add dev eth1 parent 1:0 protocol ip u32 match \
   ip dst 10.0.0.2 flowid 1:20

/* ------------------------------------------------------------------------

/* Now the setup of our simulated wireless environment is done and
   we start setting up the packet scheduling on top of it          */
/* ------------------------------------------------------------------------


/* root qdisc and parent class */
tc qdisc add dev eth1 parent 1:0 handle 10:0 hfsc bandwidth 2.1Mbit \
     estint 32000b wireless wchmon ratio reducebad 100


/* two interior classes for the two different agencies, both are
   wireless synchronization classes since agency A and B are
   competitors */
tc class add dev eth1 parent 10:0 classid 10:10 hfsc [ sc 0 0 .400 Mbit ] \
sync default /* [ dc 0 0 .064 Mbit ] */
tc class add dev eth1 parent 10:0 classid 10:20 hfsc [ sc 0 0 1.200 Mbit ] \
sync /* [ dc 0 0 .064 Mbit ] */

/* agency one, subclasses for VoIP, www, ftp */
tc class add dev eth1 parent 10:10 classid 10:1100 hfsc [ sc .300 Mbit 20ms .200Mbit]
     default
tc class add dev eth1 parent 10:10 classid 10:1200 hfsc [ sc 0 20 ms .150Mbit]
tc class add dev eth1 parent 10:10 classid 10:1300 hfsc [ sc 0 20 ms .050Mbit]

/* agency two, subclasses for VoIP, www, ftp */
tc class add dev eth1 parent 10:20 classid 10:2100 hfsc [ sc .900 Mbit 20ms .600Mbit]
tc class add dev eth1 parent 10:20 classid 10:2200 hfsc [ sc 0 20 ms .500Mbit]
tc class add dev eth1 parent 10:20 classid 10:2300 hfsc [ sc 0 20 ms .100Mbit]

/* customers of first agency, VoIP, 10 users */
tc class add dev eth1 parent 10:1100 classid 10:1101 hfsc [ sc 0.03 Mbit 20ms 0.02Mbit ] \
/* [ dc 0 0 .015 Mbit ] */   default
tc class add dev eth1 parent 10:1100 classid 10:1102 hfsc [ sc 0.03 Mbit 20ms 0.02Mbit ] \
... until
/* [ dc 0 0 .015 Mbit ] */
tc class add dev eth1 parent 10:1100 classid 10:1110 hfsc [ sc 0.03 Mbit 20ms 0.02Mbit ] \
/* [ dc 0 0 .015 Mbit ] */

/* adapt queue length for VoIP: 250ms*20kbit/s=625byte -> queue length
   of 9 packets is enough at maximum rate */

tc qdisc add dev eth1 parent 10:1101 pfifo limit VOIP_QUEUE_LIMIT
tc qdisc add dev eth1 parent 10:1102 pfifo limit VOIP_QUEUE_LIMIT
```

```
... until
tc qdisc add dev eth1 parent 10:1110 pfifo limit VOIP_QUEUE_LIMIT

/* customers of first agency, www, 3 users */
tc class add dev eth1 parent 10:1200 classid 10:1201 hfsc [ sc 0 20ms .050Mbit]
tc class add dev eth1 parent 10:1200 classid 10:1202 hfsc [ sc 0 20ms .050Mbit]
tc class add dev eth1 parent 10:1200 classid 10:1203 hfsc [ sc 0 20ms .050Mbit]

/* customer of first agency, ftp, 2 users */
tc class add dev eth1 parent 10:1300 classid 10:1301 hfsc [ sc 0 20ms .025Mbit]
tc class add dev eth1 parent 10:1300 classid 10:1302 hfsc [ sc 0 20ms .025Mbit]



/* customers of second agency, VoIP, 30 users */
tc class add dev eth1 parent 10:2100 classid 10:2101 hfsc [ sc 0.03Mbit 20ms 0.02Mbit ] \
/* [ dc 0 0 .015 Mbit ] */
tc class add dev eth1 parent 10:2100 classid 10:2102 hfsc [ sc 0.03Mbit 20ms 0.02Mbit ] \
... until
/* [ dc 0 0 .015 Mbit ] */
tc class add dev eth1 parent 10:2100 classid 10:2130 hfsc [ sc 0.03Mbit 20ms 0.02Mbit ] \
/* [ dc 0 0 .015 Mbit ] */

/* adapt max. queue length for all VoIP classes of second agency */
tc qdisc add dev eth1 parent 10:2101 pfifo limit VOIP_QUEUE_LIMIT
tc qdisc add dev eth1 parent 10:2102 pfifo limit VOIP_QUEUE_LIMIT
... until
tc qdisc add dev eth1 parent 10:2130 pfifo limit VOIP_QUEUE_LIMIT

/* customers of second agency, www, 10 users */
tc class add dev eth1 parent 10:2200 classid 10:2201 hfsc [ sc 0 20ms .050Mbit]
tc class add dev eth1 parent 10:2200 classid 10:2202 hfsc [ sc 0 20ms .050Mbit]
... until
tc class add dev eth1 parent 10:2200 classid 10:2210 hfsc [ sc 0 20ms .050Mbit]

/* customers of second agency, ftp, 2 users */
tc class add dev eth1 parent 10:2300 classid 10:2301 hfsc [ sc 0 20ms .050Mbit]
tc class add dev eth1 parent 10:2300 classid 10:2302 hfsc [ sc 0 20ms .050Mbit]


/* TRAFFIC FILTER SETUP */


/* add filters for agency 1 customers, 10 users, VoIP */
tc filter add dev eth1 parent 10:0 protocol ip u32 match \
   ip dst 10.0.0.1 match ip sport PORT_VOIP 0xffff flowid 10:1101
tc filter add dev eth1 parent 10:0 protocol ip u32 match \
   ip dst 10.0.0.2 match ip sport PORT_VOIP 0xffff flowid 10:1102
```

```
...until
tc filter add dev eth1 parent 10:0 protocol ip u32 match \
   ip dst 10.0.0.10 match ip sport PORT_VOIP 0xffff flowid 10:1110

/* add filters for agency 1 customers, www, 3 users */
tc filter add dev eth1 parent 10:0 protocol ip u32 match \
   ip dst 10.0.0.1 match ip sport PORT_HTTP 0xffff flowid 10:1201
tc filter add dev eth1 parent 10:0 protocol ip u32 match \
   ip dst 10.0.0.2 match ip sport PORT_HTTP 0xffff flowid 10:1202
tc filter add dev eth1 parent 10:0 protocol ip u32 match \
   ip dst 10.0.0.3 match ip sport PORT_HTTP 0xffff flowid 10:1203

/* add filters for agency 1 customers, ftp, 1 user */
tc filter add dev eth1 parent 10:0 protocol ip u32 match \
   ip dst 10.0.0.1 match ip sport PORT_FTP 0xffff flowid 10:1301
tc filter add dev eth1 parent 10:0 protocol ip u32 match \
   ip dst 10.0.0.2 match ip sport PORT_FTP 0xffff flowid 10:1302

/* add filters for agency 2 customers, 30 users, VoIP */
tc filter add dev eth1 parent 10:0 protocol ip u32 match \
   ip dst 10.0.0.11 match ip sport PORT_VOIP 0xffff flowid 10:2101
tc filter add dev eth1 parent 10:0 protocol ip u32 match \
   ip dst 10.0.0.12 match ip sport PORT_VOIP 0xffff flowid 10:2102
...until
tc filter add dev eth1 parent 10:0 protocol ip u32 match \
   ip dst 10.0.0.40 match ip sport PORT_VOIP 0xffff flowid 10:2130

/* add filters for agency 2 customers, www, 10 users */
tc filter add dev eth1 parent 10:0 protocol ip u32 match \
   ip dst 10.0.0.11 match ip sport PORT_HTTP 0xffff flowid 10:2201
tc filter add dev eth1 parent 10:0 protocol ip u32 match \
   ip dst 10.0.0.12 match ip sport PORT_HTTP 0xffff flowid 10:2202
...until
tc filter add dev eth1 parent 10:0 protocol ip u32 match \
   ip dst 10.0.0.20 match ip sport PORT_HTTP 0xffff flowid 10:2210

/* add filters for agency 2 customers, ftp, 2 user */
tc filter add dev eth1 parent 10:0 protocol ip u32 match \
   ip dst 10.0.0.11 match ip sport PORT_FTP 0xffff flowid 10:2301
tc filter add dev eth1 parent 10:0 protocol ip u32 match \
   ip dst 10.0.0.12 match ip sport PORT_FTP 0xffff flowid 10:2302


/* Voice over IP for all customers */

every 0.03 s send PACKET_VoIP(1) PAYLOAD_VoIP(1)
every 0.03 s send PACKET_VoIP(2) PAYLOAD_VoIP(2)
...until
```

```
every 0.03 s send PACKET_VoIP(40) PAYLOAD_VoIP(40)

/* ( remaining traffic generated using trafgen ) */
```

B. Selected Source Code Files

The implementation of the wireless scheduling extensions and the modified H-FSC scheduler within the Linux environment consists of various files within the kernel source tree. Table B.1 lists the added/modified files. Additionally the traffic control program `tc` of the `iproute2` package was extended to parse options for the new schedulers (Table B.2). The driver for the Raylink and Lucent/Avaya WLAN cards (which are part of the PCMCIA card services) were modified to provide information for wireless channel monitors, which is used e.g. by the `driver` channel monitor. Chapter 6 gives a detailed description of the design and implementation of the prototype.

In the following sections the most important source code files of the prototype implementation are listed: Section B.1 shows the wireless channel monitor "`driver`" which was used for the measurements in Section 7.2. An example for the implementation of a wireless FIFO qdisc in Section B.2 demonstrates the usage of the framework and can be used as a basis for the implementation of new wireless queuing disciplines. Section B.3 contains the sources for the wireless H-FSC scheduler.

[1] Because of space limitations not all source files are included in form of a listing in this document. The complete sources including the patch for the Linux 2.4.X kernel are distributed under the GNU Public License and available at [58].

Path	File	Change /New	Description	Listing
/net	netsyms.c	C	Added export of channel monitor interface.	no[1]
/net/sched	Config.in	C	Added configuration options.	no[1]
/net/sched	sch_chsim.c	N	Wireless channel simulation qdisc.	no[1]
/net/sched	sch_csfifo.c	N	Example qdisc: wireless FIFO.	B.2
/net/sched	sch_hfsc.c	N	Wireless H-FSC qdisc.	B.4
/net/sched	sch_hfsc.h	N	Data structures for wireless H-FSC.	B.3
/net/sched	wchmon_api.c	N	Wireless channel monitor interface.	no[1]
/net/sched	wchmon_driver.c	N	NIC driver based channel monitor.	B.1
/net/sched	wchmon_dummy.c	N	Dummy channel monitor.	no[1]
/net/sched	wchmon_ratio.c	N	Dequeuing delay based channel monitor.	no[1]
/net/sched	Makefile	C	Added new modules.	no[1]
/include/net	pkt_sched.h	C	Clock source changed. (see Section A.2)	no[1]
/include/net	pkt_wsched.h	N	Declarations for wireless scheduling.	no[1]
/include/linux	pkt_sched.h	C	Added netlink structures for H-FSC.	no[1]
/include/linux	pkt_wsched.h	N	Wireless specific netlink structures.	no[1]
/include/linux	netdevice.h	C	Channel monitor added to network device structure.	no[1]

Table B.1: *Added/modified files in Linux kernel source tree.*

Path	File	Change /New	Description	Listing
/iproute2/tc	q_chsim.c	N	Parsing options for chsim qdisc.	no[1]
/iproute2/tc	q_csfifo.c	N	Parsing options for csfifo qdisc.	no[1]
/iproute2/tc	q_hfsc.c	N	Parsing options for hfsc qdisc.	no[1]
/iproute2/tc	wsched_util.c	N	Common functions for parsing of wireless scheduler options.	no[1]
/iproute2/tc	wsched_util.h	N	Declarations for parsing of wireless scheduler options.	no[1]

Table B.2: *Added/modified files in* iproute2 *source tree.*

Path	File	Change /New	Description	Listing
/pcmcia-cs/wireless	ray_cs.c	C	Raylink driver - added calls to channel monitor interface.	no[1]
/pcmcia-cs/wireless	wvlan_cs.c	C	Lucent/Avaya driver - added calls to channel monitor interface.	no[1]
/pcmcia-cs	Configure	C	Added wireless configuration.	no[1]

Table B.3: *Added/modified files in* pcmcia-cs *source tree.*

B.1 Wireless Channel Monitor

Listing B.1: *The "*`driver`*" wireless channel monitor (*`wchmon_driver.c`*).*

```
/* $Id: wchmon_driver.c,v 1.1.2.1 2001/11/12 20:37:38 wischhof Exp $
 *
 * Wireless Channel Monitor: based on currently available data link
 *                           layer information of non-QoS aware wireless NIC
 *
 * This wireless channel monitor evaluates the following informations
 * available from a modified wireless device driver:
 *
 *   a) success/failure of a transmission attempt
 *   b) status byte appended to received packet by wireless MAC
 *
 * module parameters:
 *
 * max_monitored        max. number of monitored channels (at one time)
 * dst_indicator        specifies how different destinations/channels
 *                      should be identified
 *                      (0: MAC address, 1: IP address, default is 0)
 * module_debug         level of debugging if compiled with debug support
 *                      (0: no debug messages, default)
 * monitor_source       the source used for monitoring the channel quality
 *                      0: no monitoring at all
 *                      1: packet transmission interrupt result
 *                      2: MAC controller channel status byte of received
 *                         packets
 *                      3: combination of options 1 and 2
 * max_open_slots       maximum number of open slots (packets stored
 *                      in the wireless NICs internal buffer for which
 *                      no tx status confirmation has been received)
 *
 * max_unprobed         time the status info for a channel is valid [in ms]
 *                      (default is 3 seconds)
 *
 *
 * author:      Lars Wischhof <wischhof@ieee.org>
 *              Washington University, St. Louis, MO
 *              Technical University of Berlin, Germany
 *
 *
 *      This program is free software; you can redistribute it and/or
 *      modify it under the terms of the GNU General Public License
 *      as published by the Free Software Foundation; either version
 *      2 of the License, or (at your option) any later version.
 *
 * $Log: wchmon_driver.c,v $
 * Revision 1.1.2.1    2001/11/12 20:37:38    wischhof
 * Monday afternoon version - final
 *
 * Revision 1.1    2001/10/31 07:19:22    wischhof
 * first version on SF, transition 2.4.5 to 2.4.13
 *
 */


/*
 * constants
 */

#define MIN_TIME_BEFORE_FREED 120 /* minimal time which must pass before record is freed */

#define IDLE_TIME       (HZ>>1)  /* if there was no transmission for
                                    IDLE_TIME jiffies, the devices internal buffer
                                    is assumed empty. this value is equal to
                                    0.5s which is enough to clear a full 64kb
                                    buffer at 1MBit/s */

#define DST_MAC                 0  /* destination is identified by MAC address */
#define DST_IP                  1  /* destination is identified by IP address */

#define MONITOR_NONE    0
#define MONITOR_TX      1
#define MONITOR_RX      2
#define MONITOR_TX_RX   3

#include <linux/config.h>
#include <linux/module.h>
#include <linux/proc_fs.h>
#include <asm/uaccess.h>
#include <linux/netdevice.h>
#include <linux/if_ether.h>
```

```
#include <linux/skbuff.h>
#include <linux/ip.h>
#include <linux/rtnetlink.h>
#include <linux/pkt_sched.h>
#include <net/pkt_sched.h>
#include <net/pkt_wsched.h>


/*
 * module interface
 */

MODULE_AUTHOR("Lars Wischhof <wischhof@ieee.org>");
MODULE_DESCRIPTION("Driver Channel Monitor (Wireless Scheduling) V0.1");

MODULE_PARM(module_debug, "i");  /* debugging level (if debug macros defined) */
MODULE_PARM(max_monitored, "i"); /* max. number of monitored channels */
MODULE_PARM(dst_indicator, "i");    /* destination indicator */
MODULE_PARM(monitor_source, "i"); /* source for channel-state information */
MODULE_PARM(max_unprobed, "i");  /* max. time channel info is valid */

/* default values for kernel module */
static unsigned int module_debug = 0;                /* no debugging messages */
static unsigned int dst_indicator = 0;               /* destination indicated by MAC address */
static unsigned int max_monitored = 20;              /* maximum number of monitored channels */
static unsigned int monitor_source = MONITOR_RX; /* data source for channel monitoring,
                                                    0: none, 1: tx interrupt,
                                                    2: receive level, 3: combines 1 and 2 */
static unsigned int max_open_slots = 14;             /* maximum number of open slots allowed,
                                                    this number should be chosen taking the
                                                    size of the network cards internal buffer
                                                    into account */

static unsigned int max_unprobed = 3000;             /* time for which status information for a
                                                    channel is valid (in ms) */
static u64          max_unprobed_jiffies = 0;        /* caches converted result */

/*
 * debug macros
 */

#define DEBUG_CALIBRATE     3
#define DEBUG_STATE         4
#define DEBUG_INFO          5
#define DEBUG_FCT_TRACE     7

#if 1
#define DEBUG(n, args...) if (module_debug >=(n)) printk(KERN_DEBUG args)
#define ASSERT(x) if (!(x)) { printk("KERNEL: assertion (" #x ") failed at " __FILE__ "(%d):" __FUNCTION__ "\n",
    __LINE__); }
#define FCT_IN if( module_debug >= DEBUG_FCT_TRACE )printk(KERN_DEBUG "->" __FUNCTION__ " (" __FILE__ ":
    line %d)\n", __LINE__);
#define FCT_OUT if( module_debug >= DEBUG_FCT_TRACE )printk(KERN_DEBUG "<-" __FUNCTION__ " (" __FILE__ "
    :line %d)\n", __LINE__);
#else
#define FCT_IN
#define FCT_OUT
#define DEBUG(n, args...)
#define ASSERT(x)
#endif

static char wsche_channel_monitor_id[] = "driver";

/* definitions for proc filesystem */
#define PROC_GTR_MAP_NAME "net/wchmon_gtr_map"

/*
 * capacity map
 *
 * Maps signal-quality to expected channel capacity. This table usually is specific for a
 * wireless hardware - it it can be changed without recompiling the module by writing to the
 * wchmon_driver file in the /proc/net directory
 */

#define CAPACITY_MAP_MIN_LEVEL 0
#define CAPACITY_MAP_MAX_LEVEL 255

/* Mapping a measured link-level to an expected goodput rate
 * is done by using the following table. Since the ralation of
 * goodput to link-level is highly device and technology dependend
 * the table is different for every device.
 *
 * Therefore, before using a new device with the channel monitor,
 * a valid table must be obtained e.g. by using the WchmonSigMap
```

```
 * calibration utility also developed as part of this project.
 * The table used by an installed wireless channel monitor
 * can easily be updated at runtime by copying the calibration file to
 * /proc/net/wchmon_driver. Another solution is to include it in this
 * source file and to recompile it - which makes it the default
 * mapping which is used if no update was written to /proc/wchmon_driver.
 *
 *
 * the map starts with MIN_LEVEL and ends with a value for MAX_LEVEL
 * therefore the size of the map must be [CAPACITY_MAP_MAX_LEVEL - CAPACITY_MAP_MIN_LEVEL + 1]
 *
 * Note: The first value in the map is also used if the destination is not
 *       responding at all. It is chosen so that in this case it will not influence
 *       the performance of other mobiles too badly.
 */

/* this table is for the Raylink 2MBit cards,
   the channel bandwidth is measured in byte/s */

unsigned int capacity_map[((CAPACITY_MAP_MAX_LEVEL-CAPACITY_MAP_MIN_LEVEL)+1)] =
{
  5000,   5000,   5000,   5000,   5000,   5000,   5000,   5000,   5000,   5000,
  5000,   5000,   5000,   5000,   5000,   5000,   5000,   5000,   5000,   5000,
  5000,   5000,   5000,   5000,   5000,   5000,   5000,   5000,   5000,   5000,
  5000,   5000,   5000,   5000,   5000,   5000,   5000,   5000,   5000,   5000,
  5000,   5000,   5000,   5000,   5000,   5000,   5000,   5000,   5000,   5000,
  5000,  20000,  30000,  40000,  50000,  50000,  50000,  50000,  50000,  50000,  /*      - 60 */  /* 1 MBit modulation */
 55000,  55000,  55000,  55000,  55000,  55000,  55000,  55000,  55000,  55000,  /*  60 - 70 */
 60000,  60000,  60000,  60000,  60000,  60000,  60000,  60000,  60000,  60000,  /*  70 - 80 */  /*              assumed */
 65000,  65000,  65000,  65000,  65000,  65000,  65000,  65000,  65000,  65000,  /*  80 - 90 */
 70000,  75000,  75000,  75000,  75000,  75000,  75000,  75000,  75000,  75000,  /*  90 - 100 */
 80000,  80000,  80000,  80000,  80000,  80000,  80000,  80000,  80000,  80000,  /* 100 - 110 */
 90000,  90000,  90000,  90000,  90000,  90000,  90000,  90000,  90000,  90000,  /* 110 - 120 */
100000, 100000, 100000, 100000, 100000, 100000, 100000, 100000, 100000, 100000,  /* 120 - 130 */
100000, 100000, 100000, 100000, 100000, 100000, 100000, 100000, 100000, 100000,  /* 130 - 140 */
170000, 170000, 170000, 170000, 170000, 170000, 170000, 170000, 170000, 170000,  /* 140 - 150 */  /* 2 MBit modulation */
180000, 180000, 180000, 180000, 180000, 180000, 180000, 180000, 180000, 180000,  /* 150 - 160 */
190000, 190000, 190000, 190000, 190000, 190000, 190000, 190000, 190000, 190000,  /* 160 - 170 */
200000, 200000, 200000, 200000, 200000, 200000, 200000, 200000, 200000, 200000,  /* 170 -     */
200000, 200000, 200000, 200000, 200000, 200000, 200000, 200000, 200000, 200000,
200000, 200000, 200000, 200000, 200000, 200000, 200000, 200000, 200000, 200000,
200000, 200000, 200000, 200000, 200000, 200000, 200000, 200000, 200000, 200000,
200000, 200000, 200000, 200000, 200000, 200000, 200000, 200000, 200000, 200000,
200000, 200000, 200000, 200000, 200000, 200000, 200000, 200000, 200000, 200000,
200000, 200000, 200000, 200000, 200000, 200000, 200000, 200000, 200000, 200000,
200000, 200000, 200000, 200000, 200000, 200000
};

/*
 * channel monitor interface structure
 */
static struct wsched_channel_monitor *wchmon_driver = NULL;

/*
 * slot entry
 */
struct wchmon_driver_slot
{
  struct wchmon_driver_slot *next;
  struct wchmon_driver_slot *prev;
  struct wchmon_driver_entry *channel;
};

/*
 * channel entry in list of monitored channels
 */
struct wchmon_driver_entry
{
  struct wchmon_driver_entry *next;       /* next entry in list */

  struct wsched_dest dst;                 /* destionation (unique id) of this channel */
  struct wsched_chstate * state ;         /* current information on this channel */
};

/*
 * private data structure
 */

struct wchmon_driver_data
{
```

```
    struct wchmon_driver_entry *channel_list; /* pointer to list of monitored channels */
    struct wchmon_driver_slot *slot_list; /* pointer to list of transmission slots */
    unsigned long nr_channels; /* number of currently monitored channels */
    u32 max_level;
    u32 min_level;
    unsigned int slot_count;
    unsigned long tx_in_interval; /* counts transmissions in one transmission
                                     interval (used to estimate number of missed
                                     rx interrupts */
    unsigned long miss_tx_stat; /* estimate of correct slots for each missed
                                   tx status info call */
    unsigned long last_tx; /* start of last transmission in jiffies */
    u32 current_trans_stat; /* currently estimated transmission status */
    atomic_t last_trans_stat; /* last transmission status received from NIC driver */
    atomic_t trans_stat_count; /* number of transmission status notifications since last flush */
    spinlock_t slot_lock; /* lock for access to transmission slots */
    spinlock_t entry_lock; /* lock for access to channel entries */
};

/*
 * function declarations
 */
void wchmon_driver_reset_channel_state ( struct wsched_channel_monitor *mon, struct wsched_dest *dst );
void delete_entry ( struct wsched_channel_monitor *mon, struct wsched_dest *dst );
void delete_slot (struct wsched_channel_monitor *mon, struct wchmon_driver_slot ** slot );
void wchmon_driver_transmission_status ( struct wsched_channel_monitor *mon, char *dst,
                                         unsigned int status );
void wchmon_driver_channel_status ( struct wsched_channel_monitor *mon, char *dst,
                                    unsigned int status );

/*
 * --- private functions of this module ---
 */

/*
 * look up channel information for a given destination
 * (@@@ this is the slow brute force method, use hashing?)
 */
struct wchmon_driver_entry *find_entry ( struct wsched_channel_monitor *mon, struct wsched_dest *dst )
{
 struct wchmon_driver_entry *c=((struct wchmon_driver_data *) mon->data)->channel_list;

 FCT_IN;
 if ( ! dst )
   return NULL;

 while( c && (memcmp(dst->addr, c->dst.addr, WCHMON_MAX_ADDR_LEN) != 0))
   c = c->next;

 FCT_OUT;
 return c;
}

/*
 * add channel entry
 * (pre-condition: entry spinlock is taken)
 */
struct wchmon_driver_entry *add_entry( struct wsched_channel_monitor *mon, struct wsched_dest *dst )
{
  struct wchmon_driver_entry *c_new=NULL, *c=NULL;
  struct wchmon_driver_data *data =
    (struct wchmon_driver_data *) mon->data;

  FCT_IN;
  if ( data->nr_channels == max_monitored )
  {
    /*
     * delete oldest entry in list
     */
    unsigned long oldest_probe = data->channel_list->state->last_probed;

    for ( c = data->channel_list; c; c = c->next)
      if ( c->state && (c->state->last_probed < oldest_probe ) )
      {
        c_new = c;
        oldest_probe = c->state->last_probed;
      }
    delete_entry ( mon, &c->dst );
  }

  if (!( c_new = kmalloc ( sizeof ( *c_new ), GFP_ATOMIC )))
   {
    printk ("wchmon_driver: out of memory!\n");
```

```
        return NULL;
      }
    memset( c_new , 0, sizeof (*c_new));

    if (!( c_new->state = kmalloc ( sizeof ( *c_new->state), GFP_ATOMIC )))
    {
        printk ("wchmon_driver: out of memory!\n");
        return NULL;
    }
    memset( c_new->state , 0, sizeof ( *c_new->state ));

    data->nr_channels++;
    c_new->next = data->channel_list;
    data->channel_list = c_new;

    memcpy( c_new->dst.addr, dst->addr, WCHMON_MAX_ADDR_LEN);
    FCT_OUT;
    return c_new;
}

/*
 * delete channel entry
 * ( pre-condition: entry spinlock is taken)
 */
void delete_entry ( struct wsched_channel_monitor *mon, struct wsched_dest *dst )
{
  struct wchmon_driver_entry *c_old=NULL;
  struct wchmon_driver_data *data =
        (struct wchmon_driver_data *) mon->data;
  struct wchmon_driver_entry *c;
  struct wchmon_driver_slot * slot = data-> slot_list ;
  struct wchmon_driver_slot * next_slot = NULL;
  int i;

  FCT_IN;
  c = data->channel_list;

  for ( i=0; c && (i<WCHMON_MAX_ADDR_LEN); i++)
    if ( dst->addr[i] != c->dst.addr[i] )
    {
      c_old=c;
      c = c->next;
      i = 0;
      continue;
    }

  if ( c )
  {
    if ( c_old ) {
      c_old->next = c->next; }
    else
        data->channel_list = NULL;
    if ( c->state )
      kfree ( c->state ) ;

    /*
     * delete all slots of this channel
     */
    spin_lock(&data->slot_lock);
    while( slot )
    {
      next_slot = slot->next;
      if ( slot->channel == c)
          delete_slot (mon, &slot);
      if ( next_slot == data-> slot_list )
        break;
      slot = next_slot ;
    }
    spin_unlock(&data->slot_lock);
    kfree ( c ) ;
  }
  else
    DEBUG( DEBUG_INFO, "wchmon_driver: Tried to delete non-existing entry!\n" );
  FCT_OUT;
}

/*
 * delete a packet transmission slot
 * ( precondition : slot_lock has been successfully obtained before)
 */
void delete_slot ( struct wsched_channel_monitor *mon, struct wchmon_driver_slot ** slot )
{
  struct wchmon_driver_slot *prev;
  struct wchmon_driver_slot *next;
```

```
410   struct wchmon_driver_data *data =
        (struct wchmon_driver_data *) mon->data;

      FCT_IN;

415   ASSERT( slot != NULL );
      if ( (* slot )->prev == *slot )
      {
        /*
         * deleting last slot
420      */
        ASSERT(data->slot_list == *slot ) ;
        data-> slot_list = NULL;
      }
      else
425   {
        /*
         * update links of previous and next slot
         */
        prev = (* slot )->prev;
430     next = (* slot )->next;
        if ( prev )
          prev->next = next;
        if ( next )
          next->prev = prev;
435     /*
         * special case : delete HOL slot
         */
        if ( * slot == data-> slot_list )
            data-> slot_list = prev;
440   }
      kfree (* slot ) ;
      *slot = NULL;
      data->slot_count--;

445   FCT_OUT;
    }

    /*
     * --- Define methodes exported for usage by modified device driver ---
450  */

    /*
     * notify channel monitor about the result of a transmission
     *
455  * !!! can be called in interrupt context !!!
     *
     * ( since the driver usually does not know the destination corresponding
     *   to this received interrupt because of the device's internal buffer ,
     *   dst may be NULL - in this case the channel monitor will keep track of it )
460  *
     * note : The current level is an exponential average over the last slots since
     *          we cannot be shure to always hit the right slot . The long-term level
     *          is an exponential average over the last status updates _for this channel_.
     */
465 void wchmon_driver_transmission_status ( struct wsched_channel_monitor *mon, char *dst,
                                             unsigned int status )
    {
      struct wchmon_driver_data *data;
      struct wchmon_driver_entry          *c      = NULL;
470   struct wchmon_driver_slot * slot = NULL;
      unsigned int stat_nr ;
      unsigned int transmission_status ;

      FCT_IN;
475
      if ( !( monitor_source & MONITOR_TX))
        return;

      data    = ( struct wchmon_driver_data *)mon->data;
480
       /*
        * update status of channel
        * ( currently we ignore the destination information provided by the driver
        *   since it is NULL (unknown) in case of all current drivers and keep
485     *   track of it ourselves )
        */
      if (! spin_trylock (&data->slot_lock))
      {
          /*
490        * unable to update status , delay it
           */
          atomic_inc(&data->trans_stat_count) ;
          atomic_set(&data-> last_trans_stat , status ) ;
```

```
        DEBUG( DEBUG_INFO, "status␣update␣delayed!" );
495     FCT_OUT;
        return;
    }

    if ( !( data->slot_list ))
500 {
      DEBUG( DEBUG_INFO, "Slot␣list␣empty,␣estimate␣updated!\n" );
      data->miss_tx_stat++;
      spin_unlock(&data->slot_lock);
      return;
505 }

    /*
     * check if it is time to skip a slot
     */
510 if (data->miss_tx_stat && ((data->tx_in_interval % data->miss_tx_stat) == 0) )
    {
      if ( data->slot_count > 1 )
      {
        slot = data->slot_list ;
515     delete_slot (mon, &slot);
        DEBUG( DEBUG_INFO, "skipped␣a␣slot\n" );
      }
      else
      {
520     data->miss_tx_stat++;
      }
    }

    /*
525  * now that we have the spinlock, update status info
     */

    if ( ( stat_nr = atomic_read(&data->trans_stat_count)) )
    {
530   /*
       * update delayed status infos
       * @@@ since only the last status is available, we update all with it
       *       (this should not happen often on an average fast system ...)
       */
535   DEBUG( DEBUG_INFO, "delayed␣status␣info␣available␣for␣%i␣slots!\n", stat_nr );

      atomic_set(&data->trans_stat_count, 0) ;
      transmission_status = atomic_read(&data->last_trans_stat );
      for ( ; stat_nr && (slot=data->slot_list ) ; stat_nr-- )
540   {
        c = slot->channel;

        ASSERT(c != NULL);

545     c->state->last_updated = jiffies ;
        data->current_trans_stat = (data->current_trans_stat + ( transmission_status «7) ) »1;
        c->state->level = data->current_trans_stat ;
        c->state->longterm_level =
            (3 * c->state->longterm_level + c->state->level) »2;
550
        /*
         * update channel capacity:
         *
         * (Note: This is just a first experimental approach,
555      *          a better solution would be i.e. to try to read
         *          the modulation rate from the MAC and - using the
         *          knowledge on the length of the packet - to update
         *          the channel capacity.)
         */
560     if ( transmission_status > 1 ) {
                c->state->capacity += ((capacity_map[(CAPACITY_MAP_MAX_LEVEL-CAPACITY_MAP_MAX_LEVEL)
                        ] -
                                capacity_map[0])/1000);
        }
        else
565             c->state->capacity -= ((capacity_map[(CAPACITY_MAP_MAX_LEVEL-CAPACITY_MAP_MAX_LEVEL)
                        ] -
                                capacity_map[0])/1000);
        if ( c->state->capacity < capacity_map[0] )
            c->state->capacity = capacity_map[0];
        if ( c->state->capacity > capacity_map[(CAPACITY_MAP_MAX_LEVEL-CAPACITY_MAP_MAX_LEVEL)] )
570         c->state->capacity = capacity_map[(CAPACITY_MAP_MAX_LEVEL-CAPACITY_MAP_MAX_LEVEL)];

        delete_slot (mon, &slot);
      }
    }
575
```

```
    /*
     * check for errors , we should at least have one slot left ,
     * else we skipped too many slots and need to update the estimate
     */
    if ( !( data->slot_list ))
    {
      DEBUG( DEBUG_INFO, "Slot list empty!\n" );
      data->miss_tx_stat++;
      spin_unlock(&data->slot_lock);
      return;
    }

    /*
     * now we have valid status info , update channel information
     */
    slot = data->slot_list ;
    c = slot->channel;

    ASSERT(c != NULL);

    c->state->last_updated = jiffies ;
    data-> current_trans_stat = ( data-> current_trans_stat + ( status «7) ) »1;
    c->state->level = data-> current_trans_stat ;
    c->state->longterm_level =
       (3 * c->state->longterm_level + c->state->level) »2;

    /*
     * debug output to monitor channel states
     */
    if ( dst_indicator == DST_MAC )
      DEBUG( DEBUG_STATE, "channel %02X:%02X:%02X:%02X:%02X:%02X: level %d longterm-level %d\n",
             c->dst.addr [0], c->dst.addr [1], c->dst.addr [2], c->dst.addr [3], c->dst.addr [4],
             c->dst.addr [5], c->state->level , c->state->longterm_level );
    if ( dst_indicator == DST_IP )
      DEBUG( DEBUG_STATE, "channel %i.%i.%i.%i: level %d longterm-level %d\n",
             c->dst.addr [0], c->dst.addr [1], c->dst.addr [2], c->dst.addr [3], c->state->level,
             c->state->longterm_level );

    delete_slot (mon, &slot);
    spin_unlock(&data->slot_lock);

    FCT_OUT;
}

/*
 * notify channel monitor about the received status byte for a channel
 * ( signal strength )
 */
void wchmon_driver_channel_status ( struct wsched_channel_monitor *mon, char *dst,
                                    unsigned int status )
{
    struct wchmon_driver_entry *c;
    struct wchmon_driver_data *data;

    if (!( monitor_source & MONITOR_RX))
      return;

    FCT_IN;
    data = ( struct wchmon_driver_data *) mon->data;
    if (! spin_trylock (&data->entry_lock))
    {
      DEBUG( DEBUG_INFO, "Could not update status info - spinlock busy.\n");
      return;
    }
    c = data->channel_list;

    while ( c && (memcmp(dst, c->dst.addr, WCHMON_MAX_ADDR_LEN) != 0))
      c = c->next;

    if ( c == NULL)
    {
     DEBUG( DEBUG_INFO, "channel unknown\n" );
     spin_unlock(&data->entry_lock);
     return;
    }

    /*
     * update status information
     */

    c->state->last_updated = jiffies ;
    c->state->level = status ;
    c->state->longterm_level =
       (3 * c->state->longterm_level + c->state->level) »2;
```

```
660
    /*
     * map status byte to expected capacity
     * Note: An averaging mechanism here could help to dampen the influence
     * on the schedulers transmission rate.
665  */
    if ( status > CAPACITY_MAP_MAX_LEVEL )
        status = CAPACITY_MAP_MAX_LEVEL;
    if ( status < CAPACITY_MAP_MIN_LEVEL )
        status = CAPACITY_MAP_MIN_LEVEL;
670 c->state->capacity = capacity_map[( status -CAPACITY_MAP_MIN_LEVEL)];
    spin_unlock(&data->entry_lock);

    /*
     * debug output to monitor channel states
675  */
    if ( dst_indicator == DST_MAC )
      DEBUG( DEBUG_STATE, "channel %02X:%02X:%02X:%02X:%02X:%02X: level %d longterm-level %d\n",
             c->dst.addr [0], c->dst.addr [1], c->dst.addr [2], c->dst.addr [3], c->dst.addr [4],
             c->dst.addr [5], c->state->level, c->state->longterm_level );
680 if ( dst_indicator == DST_IP )
      DEBUG( DEBUG_STATE, "channel %i.%i.%i.%i: level %d longterm-level %d\n",
             c->dst.addr [0], c->dst.addr [1], c->dst.addr [2], c->dst.addr [3], c->state->level,
             c->state->longterm_level );

685
    FCT_OUT;
}


690 /*
 * --- Define channel monitor methods ---
 */

/*
695 * Initialize a newly attached monitor
 */
int wchmon_driver_init ( struct wsched_channel_monitor *mon)
{
    struct wchmon_driver_data * data;
700 struct wchmon_driver_entry *c;
    struct wsched_dest *broadcast_dst;

    FCT_IN;
    MOD_INC_USE_COUNT;
705
    if ( !( data = kmalloc( sizeof ( *data ), GFP_KERNEL )))
        return (-ENOMEM);

    memset( data, 0, sizeof ( *data ) );
710  spin_lock_init ( &data->slot_lock );
     spin_lock_init ( &data->entry_lock );
    mon->data = (void *) data;

    /*
715  * add a persistent channel for broadcast traffic
     */
    if (!( broadcast_dst = kmalloc( sizeof ( * broadcast_dst ), GFP_KERNEL )))
            return (-ENOMEM);

720 memset( broadcast_dst, 0, WCHMON_MAX_ADDR_LEN );

    if ( dst_indicator == DST_MAC) {
          memset( broadcast_dst, 255, 6 ) ;
    }
725 if ( dst_indicator == DST_MAC) {
          memset( broadcast_dst, 255, 4 ) ;
    }

    if (!( c = add_entry ( mon, broadcast_dst )))
730 {
      printk ("wchmon_driver_start_transmit: cannot add channel for broadcast traffic \n");
      return -ENOMEM;
    }

735 /*
     * broadcast traffic has always perfect channel
     * and is never updated
     */

740 c->state->level = ~0;
    c->state->longterm_level = ~0;
    c->state->last_updated = ~0;
    c->state->last_probed = ~0;
```

```
    c->state->capacity = capacity_map[(CAPACITY_MAP_MAX_LEVEL - CAPACITY_MAP_MIN_LEVEL)];

    FCT_OUT;
    return 0;
}

/*
 * Free all data held by a monitor when it is detached from
 * the wireless device
 */
void wchmon_driver_destroy(struct wsched_channel_monitor *mon)
{
    struct wchmon_driver_data *data =
        (struct wchmon_driver_data *) mon->data;
    struct wchmon_driver_entry *cl =
        data->channel_list;
    struct wchmon_driver_entry *cl_next;
    struct wchmon_driver_slot * slot;

    FCT_IN;

    /*
     * free all stored channel data
     */
    while( cl )
    {
      cl_next = cl->next;
      if ( cl->state )
        kfree(cl->state);
      kfree(cl);
      cl = cl_next;
    }

    /*
     * delete all slots
     */
    spin_lock(&data->slot_lock);

    while(( slot = data-> slot_list ))
    {
        delete_slot (mon, &slot);
    }
    spin_unlock(&data->slot_lock);

    MOD_DEC_USE_COUNT;
    FCT_OUT;
}

/*
 * get information about current channel state to
 * a given destination
 */

struct wsched_chstate * wchmon_driver_get_channel_state(struct wsched_channel_monitor *mon,
    struct wsched_dest *dst )
{
    struct wchmon_driver_data *data =
        (struct wchmon_driver_data *) mon->data;
    struct wchmon_driver_entry *c;

    FCT_IN;
    c = find_entry ( mon, dst );

    if ( ! c )
      return NULL;

    if ( ( c->state->last_updated + max_unprobed_jiffies ) < jiffies )
    {
        printk ( "wchmon_driver_get_channel_state: channel state too old\n" );
        wchmon_driver_reset_channel_state ( mon, dst );
        return NULL;
    }
    c->state->max = data->max_level;
    c->state->min = data->min_level;
    DEBUG( DEBUG_INFO, "%ld.000000 state of %X is %u \n", jiffies, (unsigned int) c, c->state->level);
    FCT_OUT;
    return c->state;
}

/*
 * reset all stored information on a channel
 */
```

```
void wchmon_driver_reset_channel_state ( struct wsched_channel_monitor *mon,
    struct wsched_dest *dst )
{
  struct wchmon_driver_entry *c;

  FCT_IN;

  c = find_entry ( mon, dst ) ;

  if ( ! c )
    return;

  c->state->level = 0;
  c->state->longterm_level = 0;
  c->state->capacity = 0;

  if ( ( jiffies - c->state->last_probed)> (MIN_TIME_BEFORE_FREED*HZ)) {
        delete_entry ( mon, dst ) ;
  }
  else
    DEBUG( DEBUG_INFO, "wchmon_driver_reset_channel_state: channel not deleted because recently accessed\n");
  FCT_OUT;
}


/*
 * signals start of transmission on the given channel
 */
void wchmon_driver_start_transmit ( struct wsched_channel_monitor *mon,
  struct wsched_dest *dst , u32 size )
{
  struct wchmon_driver_data *data =
      ( struct wchmon_driver_data *) mon->data;
  struct wchmon_driver_entry *c;
  struct wchmon_driver_slot * slot = NULL;

  FCT_IN;

  if ( ! dst )
    return;

  spin_lock(&data->entry_lock);
  c = find_entry ( mon, dst ) ;

  if ( ! c )
  {
    if (!( c = add_entry ( mon, dst )))
    {
      printk ("wchmon_driver_start_transmit : cannot add new channel - limit exceeded?\n");
      spin_unlock(&data->entry_lock);
      return;
    }
  }

 /*
   * update " last probed" information
   */
  c->state->last_probed = jiffies ;

  /*
   * return spinlock
   */
  spin_unlock(&data->entry_lock);

  /*
   * in case of RX monitoring, we are done now;
   * for TX monitoring we add a transmission slot
   */

  if (!( monitor_source & MONITOR_TX))
    return;

 /*
  * check if we have been in an indle period
  */
 if ( (( jiffies - data->last_tx)>=IDLE_TIME) && data->slot_count)
 {
   /*
    * yes , update estimation of missed transmission status IRQs
    */
   data->miss_tx_stat = ( data->miss_tx_stat + ( data->tx_in_interval / data->slot_count)) >>1;
   DEBUG( DEBUG_INFO, "new estimation: miss one tx status every %ul slots\n", data->miss_tx_stat);
   /*
    * delete all old slots
```

```
        */
       spin_lock(&data->slot_lock);

915    while(( slot = data-> slot_list ))
       {
           delete_slot (mon, &slot);
       }
       spin_unlock(&data->slot_lock);
920  }
     /*
      * add slot for packet and fill it
      */
     DEBUG( DEBUG_INFO, "currently %i open slots\n", data->slot_count);
925  if ( data->slot_count >= max_open_slots )
     {
        printk ("wchmon: Maximum number of open slots reached, updating skip estimation!\n");
       /*
        * In normal operation the upper slot limit should never be reached
930     * therefore we are not getting enough tx status information from the
        * wireless NIC driver . The estimate is updated so that less status
        * updates are expected.
        */
       if (data->miss_tx_stat > 1)
935    {
          data->miss_tx_stat--;
       };
       spin_lock(&data->slot_lock);
       slot = data-> slot_list ;
940    delete_slot (mon, &slot);
       spin_unlock(&data->slot_lock);
     }
     if (!( slot = kmalloc( sizeof (* slot ) , GFP_ATOMIC)))
     {
945    printk ("wchmon: Unable to allocate slot!\n");
       return;
     }
     memset(slot , 0, sizeof (* slot ));
     slot ->channel = c;
950  spin_lock(&data->slot_lock);

     if ( data-> slot_list )
     {
       /*
955     * append new slot to list of slots
        */
       slot ->prev = data-> slot_list ;
       slot ->next = data-> slot_list ->next;
       if ( data-> slot_list ->next )
960       data-> slot_list ->next->prev = slot;
       data-> slot_list ->next = slot ;
     }
     else
     {
965    /*
        * this is the first slot in the list
        */
       slot ->prev = slot ;
       slot ->next = slot ;
970    data-> slot_list = slot ;
     }
     data->slot_count++;
     data-> tx_in_interval ++;
     data->last_tx = jiffies ;
975
     spin_unlock(&data->slot_lock);

     FCT_OUT;
   }
980
   /*
    * signals end of transmission on the given channel
    */
   void wchmon_driver_end_transmit( struct wsched_channel_monitor *mon,
985    struct wsched_dest *dst )
   {
     FCT_IN;
     /*
      * there is nothing to do here , all work is done when tx/rx info
990   * is received from the modified device driver
      */
     FCT_OUT;
   }

995 /*
```

```
 * signals that the transmission was aborted and that
 * the qdisc does not want to monitor this transmission
 */

void wchmon_driver_abort_transmit( struct wsched_channel_monitor *mon,
    struct wsched_dest *dst )
{
  FCT_IN;
  /* @@@ delete last queued slot ? */
  FCT_OUT;
}

/*
 * wchmon_driver_get_capacity
 *
 * Return an estimated value for the available bandwidth (goodput) for
 * the specified destination.
 *
 * Note: Right now, in case we do not have any information we simply
 *       return the minimum capacity - it could be beneficial to be
 *       able to signal this fact to the scheduler.
 */

u32 wchmon_driver_get_capacity( struct wsched_channel_monitor *mon,
  struct wsched_dest *dst )
{
  struct wchmon_driver_entry *c;

  FCT_IN;
  c = find_entry( mon, dst );

  if ( !c )
    return capacity_map[0];

  if ( (( c->state->last_updated + max_unprobed_jiffies ) < jiffies ) &&
       (c->state->last_probed != ~0) )
  {
    DEBUG( DEBUG_INFO, "wchmon_driver_get_capacity: channel state too old, assuming %i \n",
        capacity_map[0] );
    wchmon_driver_reset_channel_state( mon, dst );
    return capacity_map[0];
  }

  DEBUG( DEBUG_INFO, "wchmon_driver_get_capacity: channel state valid, %Lu jiffies old (updated: %Lu now: %Lu)\
      n",
          jiffies -c->state->last_updated, c->state->last_updated, jiffies );

  if ( c->state->capacity == 0 ) {
          FCT_OUT;
          return capacity_map[0];
  };

   /*
    * debug output to monitor channel capacity
    */
  if ( dst_indicator == DST_MAC )
    DEBUG( DEBUG_STATE, "channel %02X %02X %02X %02X %02X %02X: capacity %d\n",
          c->dst.addr[0], c->dst.addr[1], c->dst.addr[2], c->dst.addr[3], c->dst.addr[4],
          c->dst.addr[5], c->state->capacity );
  if ( dst_indicator == DST_IP )
    DEBUG( DEBUG_STATE, "channel %i.%i.%i.%i: capacity %d\n",
          c->dst.addr[0], c->dst.addr[1], c->dst.addr[2], c->dst.addr[3], c->state->capacity );

  FCT_OUT;
  return c->state->capacity;
}

/*
 * the scheduler signals that it has no more data to send
 * (we simply ignore this info)
 */
void wchmon_driver_channels_idle( struct wsched_channel_monitor *mon )
{
  FCT_IN;
  FCT_OUT;
  return;
}


/*
 * returns the channel monitors notion of the destination of
 * this skb
 *
```

```
 * Determination of destination is flexible . Currently IP/MAC based separation
 * of wireless channels is supported in the kernel and only IP based separation
 * is supported for simulations .
 * (Note: using IP should be avoided if possible , since it creates a new
 *  channel for each destination address the host is transmitting to)
 */

void wchmon_driver_skb2dst( struct wsched_channel_monitor *mon,
    struct sk_buff *skb , struct wsched_dest * result )
{
  FCT_IN;
  if ( (( skb->nh.raw) && (skb->nh.iph) && (dst_indicator == DST_IP)) ||
       ((skb->mac.raw) && (skb->mac.ethernet) && (dst_indicator == DST_MAC)))
  {
    memset(( void *) result ->addr , 0, WCHMON_MAX_ADDR_LEN );
    if (   ( sizeof ( skb->nh.iph->daddr )> WCHMON_MAX_ADDR_LEN)
        || ( ETH_ALEN > WCHMON_MAX_ADDR_LEN ))
    {
      printk ("wchmon_driver_skb2dst: WCHMON_MAX_ADDR_LEN too small!\n");
      return;
    }
    switch ( dst_indicator )
    {
      case DST_MAC:
        memcpy( ( void *) result ->addr, (( struct ethhdr *)skb->data)->h_dest, ETH_ALEN );
        break;
      case DST_IP:
        memcpy(( void *) result ->addr , ( void *) &(skb->nh.iph->daddr), sizeof ( skb->nh.iph->daddr ));
        break;
      default:
        printk ("wchmon_driver: ERROR - unknown destination address indicator/format\n");
        break;
    }
  }
  FCT_OUT;
  return;
}

/*
 * /proc file system related functions :
 * ------------------------------------
 *
 * These functions make the currently used signal-level to expected goodput
 * mapping available at "/proc/net/wchmon_driver". From there it can be read,
 * edited e.g. using the WchmonSigMap calibration utility or updated by copying
 * a new file to it .
 *
 * ---
 * For details on the implementation of "/proc" filesystem related module
 * functions see "Linux Kernel Module Programming Guide (by Ori Pomerantz)"
 * although a lot has changed in the 2.4 kernels !
 *
 */

#define MAP_RECORD_LEN 12

/*
 * output function
 *
 * called when a user process reads the "/proc/net/wchmon_driver" file
 */

static ssize_t wchmon_driver_proc_output( struct file * file , char *buf,
                                          size_t len , loff_t * offset )
{
        int my_len, i ; /* number of bytes used in our output buffer */
        char my_buffer[MAP_RECORD_LEN+2];
        static int finished =0;
        unsigned pos_in_map = 0;

        FCT_IN;

        /*
         * user process reads until it gets 0
         */
        if ( finished )
        {
                finished = 0;
                return 0;
        }

        if (( offset != NULL) && (*offset > 0))
                pos_in_map = ((unsigned long int )(* offset ))/MAP_RECORD_LEN;
```

```
        if (pos_in_map > (CAPACITY_MAP_MAX_LEVEL - CAPACITY_MAP_MIN_LEVEL) )
                return 0;

        my_len = sprintf (my_buffer, " %.10u ", capacity_map[pos_in_map]);

        /*
         * adapt length
         */
        if (my_len > len)
                my_len = len;
        if (my_len < 0)
                my_len = 0;
        /*
         * copy to user space
         */
        for ( i=0; i<my_len; i++ )
                put_user(my_buffer[i], buf+i);

        if ( pos_in_map >= ((CAPACITY_MAP_MAX_LEVEL - CAPACITY_MAP_MIN_LEVEL) + 1 ))
                finished = 1;

        if ( offset != NULL )
                (* offset ) += my_len;

        FCT_OUT;
        return my_len;
}

/* input function
 *
 * called when user process writes to "/proc/net/wchmon_driver",
 * reads input and updates the link-level to goodput mapping
 *
 * (input has to be in fixed size records of MAP_RECORD_LEN)
 */

static unsigned int current_pos_in_buffer = 0;      // current position in read buffer

static ssize_t wchmon_driver_proc_input( struct file * file , const char *buf, size_t len, loff_t * offset )
{
        unsigned pos_in_map = 0;
        int i=0;
        static char my_buffer[MAP_RECORD_LEN+2]; /* has to be static so we can continue processing */
        char **read_pos=NULL;

        FCT_IN;

        if ( offset != NULL )
                pos_in_map = ((unsigned long int)(* offset ))/MAP_RECORD_LEN;

        while( i<len ) {
                if (pos_in_map > (CAPACITY_MAP_MAX_LEVEL - CAPACITY_MAP_MIN_LEVEL) ){
                        printk ("wchmon_driver_proc_input: warning, GTR map was too large\n");
                        i=len;
                        break;
                }

                get_user(my_buffer[ current_pos_in_buffer ], buf+i);
                current_pos_in_buffer ++;
                if ( current_pos_in_buffer == MAP_RECORD_LEN ){
                        /*
                         * received number, try to read
                         */
                        capacity_map[pos_in_map] = simple_strtoul (&(my_buffer[1]), read_pos, 10) ;
                        if ( ( read_pos != NULL) && (*read_pos != &(my_buffer[MAP_RECORD_LEN-1])) )
                                printk ("wchmon_driver_proc_input: record for capacity map has invalid format!\n");
                        DEBUG(DEBUG_INFO, "Received %u for position %u in map.\n", capacity_map[pos_in_map],
                                pos_in_map);
                        current_pos_in_buffer = 0;
                        pos_in_map++;
                }
                i++;
        }

        if ( offset != NULL )
                * offset += i;

        FCT_OUT;
        return i; /* returns number of input bytes */
}

/*
 * proc file is opened
 */
```

```
int wchmon_driver_proc_open(struct inode *inode, struct file * file ) {
  MOD_INC_USE_COUNT;
  return 0;
}

/*
 * proc file is closed
 */
int wchmon_driver_proc_close(struct inode *inode, struct file * file ){
  MOD_DEC_USE_COUNT;
  return 0;
}

/*
 * proc file permissions are requested
 *
 * the operations are: 0 - execute
 *                     2 - write
 *                     4 - read
 */
static int wchmon_driver_proc_permission( struct inode *inode, int op) {
  /* allow supeuser write access only */
  if (op ==4 || ( op==2 && current->euid==0))
      return 0;
  /* for anything else the access is denied */
  return -EACCES;
}

/*
 * structure with file operations
 */
static struct file_operations wchmon_driver_proc_fops =
{
read:wchmon_driver_proc_output,
write:wchmon_driver_proc_input,
open:wchmon_driver_proc_open,
release:wchmon_driver_proc_close,
};

/*
 * structure with inode operations
 */
static struct inode_operations wchmon_driver_proc_iops =
{
  permission: wchmon_driver_proc_permission
};

/* Kernel Module Management Functions
 * ==================================
 *
 * The following functions are ivoked when the module is
 * loaded/removed and take care of exporting the wireless
 * channel monitor interface, registering the monitor and
 * registering the /proc filesystem functions.
 */

#ifdef MODULE
/*
 * module initialization.
 * This function is called when the module is being installed.
 */
int init_module(void)
{
struct proc_dir_entry *wchmon_driver_proc_entry=NULL;

FCT_IN;
printk("Wireless Channel-Monitor (driver), LW [" __DATE__ " " __TIME__ "]\n");
/*
 * Allocate and fill channel monitor struct
 */
wchmon_driver = kmalloc(sizeof(*wchmon_driver), GFP_KERNEL);
if ( !wchmon_driver )
{
   printk ( "wchmon_driver: out of mem\n" );
  return(1);
}

memset( wchmon_driver, 0, sizeof(*wchmon_driver));

wchmon_driver->id                  = wsche_channel_monitor_id;

wchmon_driver->init                = wchmon_driver_init;
wchmon_driver->destroy             = wchmon_driver_destroy;
wchmon_driver->get_channel_state   = wchmon_driver_get_channel_state;
```

```
1330 wchmon_driver->reset_channel_state = wchmon_driver_reset_channel_state ;
     wchmon_driver->start_transmit       = wchmon_driver_start_transmit ;
     wchmon_driver->end_transmit         = wchmon_driver_end_transmit;
     wchmon_driver->abort_transmit       = wchmon_driver_abort_transmit;
     wchmon_driver->channels_idle        = wchmon_driver_channels_idle;
1335 wchmon_driver->get_capacity          = wchmon_driver_get_capacity;
     wchmon_driver->skb2dst              = wchmon_driver_skb2dst;
     wchmon_driver->data_link_receive_status = wchmon_driver_channel_status;
     wchmon_driver->data_link_transmit_status = wchmon_driver_transmission_status ;

1340 /*
      * register channel monitor
      */

     register_wsched_channel_monitor ( wchmon_driver );
1345
     /*
      * register proc file
      */

1350   wchmon_driver_proc_entry =
               create_proc_entry (PROC_GTR_MAP_NAME, S_IFREG | S_IRUGO | S_IWUSR, NULL /* &proc_root */);
       if ( ! wchmon_driver_proc_entry )
               return -ENOMEM;
       wchmon_driver_proc_entry->proc_fops = &wchmon_driver_proc_fops;
1355   wchmon_driver_proc_entry->proc_iops = &wchmon_driver_proc_iops;

     /*
      * max. unprobed: convert ms to jiffies
      * (to avoid this calculation during scheduling)
1360  */
     max_unprobed_jiffies = (u64) ((( unsigned int )(max_unprobed*HZ))/1000);

     DEBUG( DEBUG_INFO, "max. age of valid channel info: %i ms (%Lu jiffies)\n", max_unprobed, max_unprobed_jiffies );

1365 FCT_OUT;
     return(0);
     }

     /*
1370  * clean-up function
      *
      * called when the module is unloaded from kernel space
      */

1375 void cleanup_module(void)
     {
      FCT_IN;
      /*
       * remove channel monitor from kernel list
1380   */
      unregister_wsched_channel_monitor ( wchmon_driver );

      /*
       * remove proc file
1385   */
       remove_proc_entry(PROC_GTR_MAP_NAME, NULL);

      if ( wchmon_driver->refcnt )
      {
1390     printk ( "wchmon_driver: Unable to free channel monitor memory - still in use!\n" );
      }
      else
        kfree ( wchmon_driver );
      FCT_OUT;
1395  }
     #endif /* MODULE */

     /* $Id: wchmon_driver.c,v 1.1.2.1 2001/11/12 20:37:38 wischhof Exp $ */
```

B.2 Example for a Simple Wireless Queuing Discipline (CSFIFO)

Listing B.2: *The CS-FIFO kernel module (*`sch_csfifo.c`*).*

```
/*
 * net/sched/sch_csfifo.c          A channel-state aware FIFO queuing discipline
 *
 *
 *              This program is free software; you can redistribute it and/or
 *              modify it under the terms of the GNU General Public License
 *              as published by the Free Software Foundation; either version
 *              2 of the License, or (at your option) any later version.
 *
 * Authors:     Lars Wischhof <wischhof@ieee.org>,
 *              Washington University, St. Louis, MO, USA
 *              Technical University of Berlin, Germany
 *
 *
 * $Log: sch_csfifo.c,v $
 * Revision 1.1.2.1    2001/11/12 20:37:37    wischhof
 * Monday afternoon version - final
 *
 * Revision 1.2    2001/11/02 19:49:18    wischhof
 * release 0.5.1
 *
 */

#include <linux/config.h>
#include <asm/uaccess.h>
#include <asm/system.h>
#include <asm/bitops.h>
#include <linux/types.h>
#include <linux/kernel.h>
#include <linux/sched.h>
#include <linux/string.h>
#include <linux/mm.h>
#include <linux/socket.h>
#include <linux/sockios.h>
#include <linux/in.h>
#include <linux/errno.h>
#include <linux/interrupt.h>
#include <linux/if_ether.h>
#include <linux/inet.h>
#include <linux/netdevice.h>
#include <linux/etherdevice.h>
#include <linux/notifier.h>
#include <net/ip.h>
#include <net/route.h>
#include <linux/skbuff.h>
#include <net/sock.h>
#include <net/pkt_sched.h>
#include <linux/module.h>


// # define TCSIM_COMPATIBLE /* define this variable in order to stay compatible to TCSIM */

MODULE_AUTHOR("Lars Wischhof <wischhof@ieee.org>");
MODULE_DESCRIPTION("CS-FIFO queuing discipline (Wireless FIFO) V0.1");


/*
 * debug macros
 */

MODULE_PARM(csfifo_debug, "i");
#ifdef TCSIM_COMPATIBLE
static unsigned int csfifo_debug = 100;          /* default for TCSIM is debugging */
#else
static unsigned int csfifo_debug = 0;
#endif

#define DEBUG_DROP 4  /* dropped packets are logged */
#define DEBUG_INFO 5  /* info debugging messages    */
#define DEBUG_FCT_TRACE 7 /* function call traces       */

#if 1
#define DEBUG(n, args...) if ( csfifo_debug >=(n)) printk (KERN_DEBUG args)
#define ASSERT(x) if (!(x)) { printk ("KERNEL: assertion (" #x ") failed at " __FILE__ "(%d):" __FUNCTION__ "\n",
        __LINE__); }
```

```
#define FCT_IN if( csfifo_debug >= DEBUG_FCT_TRACE )printk(KERN_DEBUG "->" __FUNCTION__ "␣(" __FILE__ ":␣
    line␣%d)\n", __LINE__);
#define FCT_OUT if( csfifo_debug >= DEBUG_FCT_TRACE )printk(KERN_DEBUG "<-" __FUNCTION__ "␣(" __FILE__ ":␣
    line␣%d)\n", __LINE__);
#else
#define FCT_IN
#define FCT_OUT
#define DEBUG(n, args...)
#define ASSERT(x)
#endif


/*
 * constants
 */

#define DEFAULT_PROBE_INTERVAL (5*HZ) /* max. time (in ms) a destination is unprobed */
                                      /* ( should be _larger_ than time needed to transmit
                                           packets in queue) */

                                      /* default drop probability */
#define DEFAULT_DROP_P        ((prob_t) 1 << ( PROB_BITS-1))

/*
 *  private data of queuing discipline
 */
struct csfifo_sched_data
{
        unsigned limit ;              /* max. number of packets in queue */
        struct wsched_dest cur_dst ;  /* destination of packet currently scheduled */
        struct wsched_chstate cur_state ; /* state of packet currently scheduled */
        unsigned long probe_interval ;/* min. interval of probing of a channel */
        unsigned lookahead_window;    /* window which is looked at when trying to
                                         find a packet to drop */
        unsigned drop_th ;            /* drop threshold : packet dropping starts if
                                         this threshold is reached */
        prob_t drop_P;                /* packet drop probability for a packet on a bad channel
                                         in the lookahead window */
};

/*
 *  purge_queue - drop one packet on a bad channel
 *
 *  The big difference compared to a usual FIFO discipline is
 *  that the qdisc will notice if the queue is very full .
 *  In this case it will examine the packets in the LOOKAHEAD_WINDOW
 *  at the head of the queue and select the packet with the worst
 *  link level which has been probed during the PROBE_INTERVAL.
 *
 *  Thus, these two parameters can be used to tune the behavior of
 *  this scheduler .
 *
 *  The effect of this scheduling mechanism is, that in situation when
 *  bandwidth is very scarce , it will preferably drop packets to destinations
 *  which need much throughput to achive low goodput.
 *
 *  For simplicity we currently always try to drop a packet on the worst
 *  channel which currently does not need any probing . This rule is not
 *  fair since a channel with a slightly better quality will not get dropped
 *  at all . However, inaccuracy of channel quality measurements seem to
 *  equalize this fact - but still there is a lot of room for improvements.
 *
 *
 */
static void
purge_queue(struct Qdisc* sch)
{
  struct wsched_channel_monitor *mon=sch->dev->wsched_chmon;
  struct csfifo_sched_data *data = ( struct csfifo_sched_data *) sch->data;
  struct sk_buff *cur_skb , * drop_skb;
  struct wsched_chstate * cur_state ;
  int i ;
  int range=0;
  u32 cur_level = ~0;

  FCT_IN;
  drop_skb = NULL;
  cur_skb = skb_peek_tail (&sch->q);
  if ( ! cur_skb )
    return;

  range = ( data->lookahead_window > skb_queue_len(&sch->q) )?
          skb_queue_len(&sch->q) : data->lookahead_window;
  for ( i =0; i < range ; i++)
```

```
  {
    DEBUG( DEBUG_DROP, "examining␣skb␣%i\n", i);
    if ( cur_skb == ( struct sk_buff *) &sch->q )
    {
      /* head of the queue, skip it */
      if ( cur_skb->prev ) {
        cur_skb = cur_skb->prev; }
      else
      {
        printk (" csfifo :␣purge_queue␣-␣queue␣seems␣to␣be␣corrupted!\n");
        break;
      }
    }
    mon->skb2dst(mon, cur_skb, &data->cur_dst);
    cur_state = mon->get_channel_state( mon, &data->cur_dst);

    if ( ! cur_state )
      DEBUG( DEBUG_DROP, "monitor␣has␣no␣status␣info\n");

    if ( cur_state )
    {
      if ( ( jiffies -cur_state->last_probed) >= data->probe_interval )
        DEBUG( DEBUG_INFO, "skipping␣skb␣probe␣necessary\n");
      DEBUG( DEBUG_DROP, "skb␣has␣level␣%i␣(cur_level␣is␣%i)", cur_state->longterm_level, cur_level);
    }
    if ( cur_state && ( cur_state ->longterm_level < cur_level ) &&
      (( jiffies -cur_state->last_probed) < data->probe_interval ) )
      {
        drop_skb = cur_skb;
        cur_level = cur_state ->longterm_level;
        DEBUG( DEBUG_DROP, "updating␣cur_level␣to␣%d\n", cur_level );
      }
    if ( cur_skb->prev ) {
      cur_skb = cur_skb->prev; }
    else
    {
      DEBUG( DEBUG_DROP, "csfifo:␣purge_queue␣-␣queue␣seem␣broken!\n");
      break;
    }
  }
  if ( ! drop_skb )
  {
    printk (" csfifo :␣purge_queue␣has␣found␣no␣skb!\n");
    FCT_OUT;
    return;
  }

  mon->skb2dst(mon, drop_skb, &data->cur_dst);
  DEBUG( DEBUG_DROP, "DROP:␣channel␣%02X␣%02X␣%02X␣%02X␣%02X␣%02X:␣longterm-level␣%d\n",
         data->cur_dst.addr [0], data->cur_dst.addr [1], data->cur_dst.addr [2],
         data->cur_dst.addr [3], data->cur_dst.addr [4], data->cur_dst.addr [5],
         cur_level );

  sch->stats.drops++;
  sch->stats.backlog -= drop_skb->len;
  __skb_unlink(drop_skb, &sch->q);
  kfree_skb(drop_skb);
  FCT_OUT;
} /* purge_queue */


/*
 * enqueue a new packet at end of queue
 *
 * If the queue is filled , we try to drop a packet depending on
 * the channel condition by calling purge_queue. Only if this
 * is not successful , the new packet is dropped.
 */
static int
csfifo_enqueue(struct sk_buff *skb, struct Qdisc* sch)
{
        struct csfifo_sched_data *q = ( struct csfifo_sched_data *) sch->data;
        prob_t u;

        FCT_IN;
        DEBUG( DEBUG_INFO, "queue␣length:␣%d\n", skb_queue_len(&sch->q) );
        if ( skb_queue_len(&sch->q) >= q->limit)
        {
            DEBUG( DEBUG_INFO, "queue␣is␣larger␣than␣limit␣-␣unconditional␣purge␣of␣queue!\n" );
            purge_queue( sch );
        }
        else
        {
                if ( skb_queue_len(&sch->q) >= q->drop_th)
```

```
                {
                  /*
                   * drop a packet from a bad channel with
                   * a probability depending on drop_P and filling state
                   */
                  get_random_bytes( &u, PROB_BITS>>3);
                  if ( ( u*q->limit) < ( skb_queue_len(&sch->q) * q->drop_P ))
                      purge_queue( sch ) ;
                }
        }
        if ( skb_queue_len(&sch->q) < q->limit) {
                __skb_queue_tail(&sch->q, skb);
                sch->stats.backlog += skb->len;
                sch->stats.bytes += skb->len;
                sch->stats.packets++;
                FCT_OUT;
                return 0;
        }
        sch->stats.drops++;
#ifdef CONFIG_NET_CLS_POLICE
        if ( sch->reshape_fail==NULL || sch->reshape_fail(skb, sch))
#endif
                kfree_skb(skb);
        FCT_OUT;
        return NET_XMIT_DROP;
}

/*
 * re-queue a packet which could not be transmitted after being dequeued
 */
static int
csfifo_requeue ( struct sk_buff *skb, struct Qdisc* sch)
{
        FCT_IN;
        __skb_queue_head(&sch->q, skb);
        sch->stats.backlog += skb->len;
        FCT_OUT;
        return 0;
}

/*
 * dequeue the next packet from head of queue
 *
 *   Since this qdisc is to be used with the " ratio " wireless
 *   channel monitor, it does not defer the sending of an skb but
 *   always sends the HOL skb. Deferring makes no sense since the
 *   channel monitors information will not be updated without sending
 *   something on a channel.
 *
 *   Instead, packets for bad channes are selectively dropped on
 *   the enqueue event, see csfifo_enqueue.
 */
static struct sk_buff *
csfifo_dequeue( struct Qdisc* sch)
{
        struct sk_buff *skb;
        struct wsched_channel_monitor *mon=sch->dev->wsched_chmon;
        struct csfifo_sched_data *q = ( struct csfifo_sched_data *)sch->data;

        FCT_IN;

        skb = __skb_dequeue(&sch->q);

        if (mon)
        {
              mon->end_transmit(mon, NULL);

              if ( skb )
              {
                /*
                 * get destination of skb, signal start of
                 * transmission to channel monitor
                 */
                mon->skb2dst(mon, skb, &q->cur_dst);
                mon->start_transmit(mon, &q->cur_dst, skb->len);
              }
              else
              {
                /*
                 * we have nothing to send right now,
                 * signal idle period to channel monitor
                 */
                mon->channels_idle(mon);
              }
```

```
325             }
                else
                        printk("dequeue: channel monitor is missing!\n");

                if (skb)
330                     sch->stats.backlog -= skb->len;
                FCT_OUT;
                return skb;
}

335 /*
 * drop one packet at end of the queue
 */
static int
csfifo_drop (struct Qdisc* sch)
340 {
                struct sk_buff *skb;

                FCT_IN;
                skb = __skb_dequeue_tail(&sch->q);
345             if (skb) {
                        sch->stats.backlog -= skb->len;
                        kfree_skb(skb);
                        FCT_OUT;
                        return 1;
350             }
                FCT_OUT;
                return 0;
}

355
/*
 * reset queuing discipline
 */
static void
360 csfifo_reset (struct Qdisc* sch)
{
                FCT_IN;
                skb_queue_purge(&sch->q);
                sch->stats.backlog = 0;
365             FCT_OUT;
}

/*
 * initialize queuing discipline
370 */
static int csfifo_init (struct Qdisc *sch, struct rtattr *opt)
{
                struct csfifo_sched_data *q = (void*)sch->data;

375             FCT_IN;
                if (opt == NULL) {
                    return -EINVAL;
                } else {
                        struct tc_csfifo_qopt * ctl = RTA_DATA(opt);
380                     /*
                         * check if we received a valid control packet
                         */
                        if (opt->rta_len < RTA_LENGTH(sizeof(*ctl)))
                                return -EINVAL;
385
                        if ( ctl->limit != 0 )
                        {       /* set limit if given */
                                q->limit = ctl->limit;
                        }
390                     else
                                q->limit = sch->dev->tx_queue_len;

                        if ( ctl->probe_interval != 0 )
                        {       /* set if given */
395                             q->probe_interval = ( ctl->probe_interval*HZ)/1000;
                        }
                        else
                                q->probe_interval = ( DEFAULT_PROBE_INTERVAL*HZ)/1000;

400                     if ( ctl->lookahead_window != 0 )
                        {       /* set if given */
                                q->lookahead_window = ctl->lookahead_window;
                        }
                        else
405                             q->lookahead_window = sch->dev->tx_queue_len;

                        if ( ctl->drop_th != 0 )
                        {
```

```
                        q->drop_th = ctl->drop_th;
410                     }
                        else
                        {
                                q->drop_th = sch->dev->tx_queue_len >>1;
                        }
415
                        if ( ctl->drop_P )
                        {
                                q->drop_P = ctl->drop_P;
                        }
420                     else
                        {
                                q->drop_P = DEFAULT_DROP_P;
                        }

425                     if ( ! ctl->wchmon_id[0] )              /* set monitor if given */
                        {
                                printk (" csfifo : Wireless channel monitor not specified !\n");
                                return -EINVAL;
                        }
430
                        if ( wsched_wchmon_add( sch->dev, ctl->wchmon_id )!= 0)
                        {
                                printk (" csfifo : Unable to add wireless channel monitor to device!\n");
                                return -EINVAL;
435                     }
                }
                MOD_INC_USE_COUNT;
                FCT_OUT;
                return 0;
440 }

    /*
     * remove queuing discipline
     */
445 void csfifo_destroy (struct Qdisc *sch, struct rtattr *opt)
    {
                FCT_IN;
                MOD_DEC_USE_COUNT;
                wsched_wchmon_del( sch->dev );
450             FCT_OUT;
    }

    /*
     * send information about the parameters of the qdisc
455  * via Netlink socket to the user space control program
     */

    #ifdef CONFIG_RTNETLINK
     static int csfifo_dump(struct Qdisc *sch, struct sk_buff *skb)
460  {
                struct csfifo_sched_data *q = (void*)sch->data;
                unsigned char    *b = skb->tail;
                struct tc_csfifo_qopt opt;

465             FCT_IN;
                opt.limit                       = q->limit;
                opt.probe_interval              = (q->probe_interval*1000)/HZ;
                opt.lookahead_window            = q->lookahead_window;

470             memcpy(opt.wchmon_id, sch->dev->wsched_chmon->id, WCHMON_MAX_ID_LEN);

                RTA_PUT(skb, TCA_OPTIONS, sizeof(opt), &opt);
                FCT_OUT;
                return skb->len;
475
     rtattr_failure :
                skb_trim(skb, b - skb->data);
                return -1;
    }
480 #endif

    struct Qdisc_ops csfifo_qdisc_ops =
    {
                NULL,
                NULL,
485             " csfifo ",
                sizeof (struct csfifo_sched_data ),

                csfifo_enqueue ,
                csfifo_dequeue ,
490             csfifo_requeue ,
                csfifo_drop ,
```

```
        csfifo_init ,
        csfifo_reset ,
        csfifo_destroy ,
        csfifo_init ,

#ifdef CONFIG_RTNETLINK
        csfifo_dump,
#endif
};


/*
 * if the qdisc is compiled as a module instead of being statically included
 * in the kernel, it (de-)registers its queuing discipline on (un-)loading
 */

#ifdef MODULE
int init_module(void)
{
        FCT_IN;
        printk("Wireless Channel-State Dependent FIFO (csfifo), LW [" __DATE__ " " __TIME__ "]\n");
        FCT_OUT;
        return register_qdisc(&csfifo_qdisc_ops);
}


void cleanup_module(void)
{
        FCT_IN;
        unregister_qdisc(&csfifo_qdisc_ops);
        FCT_OUT;
}
#endif
```

B.3 Wireless H-FSC Scheduler

The kernel part of the modified H-FSC algorithm is implemented in the qdisc module `sch_hfsc.c`. Parsing of command line options and assembling netlink messages is done in user space in a module for the `tc` program implemented in `q_hfsc.c`.

B.3.1 Kernel Module

Listing B.3: *Declarations for wireless H-FSC (*`sch_hfsc.h`*).*

```
/* $Id: */
/*
 * net/sched/sch_hfsc.h      Declarations for the
 *                           Hierarchical Fair Service Curve Algorithm (H-FSC)
 *
 *
 * Authors:      Lars Wischhof <wischhof@ieee.org>,
 *               Washington University, St. Louis, MO, USA
 *               Technical University of Berlin, Germany
 *
 *               based on the implementation of H-FSC
 *               of Carnegie Mellon University in ALT-Q 3.0
 *               (see copyright notice below)
 *               queuing functions by UC Berkeley
 *               (see second copyright notice below)
 *
 * $Log: sch_hfsc.h,v $
 * Revision 1.1.2.1   2001/11/12 20:37:38   wischhof
 * Monday afternoon version - final
 *
 * Revision 1.2   2001/10/31 21:57:24   wischhof
 * fix: problems when compiling without CONF_NET_WIRELESS_SCHED
 *
 */


/*
 * Copyright (c) 1997-1999 Carnegie Mellon University. All Rights Reserved.
 *
 * Permission to use, copy, modify, and distribute this software and
 * its documentation is hereby granted (including for commercial or
 * for-profit use), provided that both the copyright notice and this
 * permission notice appear in all copies of the software, derivative
 * works, or modified versions, and any portions thereof, and that
 * both notices appear in supporting documentation, and that credit
 * is given to Carnegie Mellon University in all publications reporting
 * on direct or indirect use of this code or its derivatives.
 *
 * THIS SOFTWARE IS EXPERIMENTAL AND IS KNOWN TO HAVE BUGS, SOME OF
 * WHICH MAY HAVE SERIOUS CONSEQUENCES. CARNEGIE MELLON PROVIDES THIS
 * SOFTWARE IN ITS "AS IS" CONDITION, AND ANY EXPRESS OR IMPLIED
 * WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES
 * OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
 * DISCLAIMED. IN NO EVENT SHALL CARNEGIE MELLON UNIVERSITY BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT
 * OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR
 * BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
 * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
 * (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE
 * USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH
 * DAMAGE.
 *
 * Carnegie Mellon encourages (but does not require) users of this
 * software to return any improvements or extensions that they make,
 * and to grant Carnegie Mellon the rights to redistribute these
 * changes without encumbrance.
 *
 */

/*
 * Copyright (c) 1991, 1993
 *      The Regents of the University of California. All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
```

```
 *      notice , this  list  of  conditions  and the  following  disclaimer .
 * 2.  Redistributions  in  binary  form must reproduce  the  above  copyright
 *      notice , this  list  of  conditions  and the  following  disclaimer  in  the
 *      documentation and/or  other  materials  provided with the  distribution .
 * 4.  Neither  the  name of the  University  nor the  names of  its  contributors
 *      may be used  to  endorse  or promote products  derived  from this  software
 *      without  specific  prior  written  permission .
 *
 * THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS “AS IS” AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 *
 */

#ifndef  _NET_SCHED_SCH_HFSC_H_
#define  _NET_SCHED_SCH_HFSC_H_

 /* Unfortunately  in Linux the queue.h  file  is  not  part  of  the  kernel
    include  files , at  least  not  in  2.4.5.  In  order  to  be able  to  use
    those  definitions , we simple include  the  relevant  parts  of  the  file  here */

#ifndef  _SYS_QUEUE_H
#define  _SYS_QUEUE_H 1

/*
 * This  section  defines  three  types  of  data  structures :  lists ,  tail  queues,
 * and  circular  queues.
 *
 * A  list  is  headed by a  single  forward  pointer  ( or an  array  of  forward
 * pointers  for  a hash  table  header) . The elements  are  doubly  linked
 * so  that  an  arbitrary  element can be removed without a need  to
 * traverse  the  list . New elements can be added to  the  list  after
 * an  existing  element or at  the  head of  the  list . A  list  may only be
 * traversed  in  the  forward  direction .
 *
 * A  tail  queue  is  headed by a  pair  of  pointers , one  to  the  head of  the
 * list  and the  other  to  the  tail  of  the  list . The elements  are  doubly
 * linked  so  that  an  arbitrary  element can be removed without a need  to
 * traverse  the  list . New elements can be added to  the  list  after
 * an  existing  element , at  the  head of  the  list , or at  the  end of  the
 * list . A  tail  queue may only be  traversed  in  the  forward  direction .
 *
 * A  circle  queue  is  headed by a  pair  of  pointers , one  to  the  head of  the
 * list  and the  other  to  the  tail  of  the  list . The elements  are  doubly
 * linked  so  that  an  arbitrary  element can be removed without a need  to
 * traverse  the  list . New elements can be added to  the  list  before  or  after
 * an  existing  element , at  the  head of  the  list , or at  the  end of  the  list .
 * A  circle  queue may be  traversed  in  either  direction , but has a more
 * complex end of  list  detection .
 *
 * For  details  on the  use  of  these  macros , see  the  queue(3)  manual page.
 */

/*
 * List  definitions .
 */
#ifdef  LIST_HEAD
  #undef LIST_HEAD
#endif
#define  LIST_HEAD(name, type)                                          \
struct  name {                                                          \
        struct  type  *  lh_first ;    /*  first  element */            \
}

#define  LIST_ENTRY(type)                                               \
struct  {                                                               \
        struct  type  *  le_next ;     /*  next  element */             \
        struct  type  **  le_prev ;    /*  address  of  previous  next  element */  \
}

/*
 * List  functions .
 */
#define  LIST_INIT(head) {                                              \
        (head)->lh_first  =  NULL;                                      \
}
```

```
#define LIST_INSERT_AFTER(listelm, elm, field) {                        \
        if (((elm)->field.le_next = (listelm)->field.le_next) != NULL) \
                (listelm)->field.le_next->field.le_prev =               \
                    &(elm)->field.le_next;                              \
        (listelm)->field.le_next = (elm);                               \
        (elm)->field.le_prev = &(listelm)->field.le_next;               \
}

#define LIST_INSERT_HEAD(head, elm, field) {                            \
        if (((elm)->field.le_next = (head)->lh_first) != NULL)          \
                (head)->lh_first->field.le_prev = &(elm)->field.le_next;\
        (head)->lh_first = (elm);                                       \
        (elm)->field.le_prev = &(head)->lh_first;                       \
}

#define LIST_REMOVE(elm, field) {                                       \
        if ((elm)->field.le_next != NULL)                               \
                (elm)->field.le_next->field.le_prev =                   \
                    (elm)->field.le_prev;                               \
        *(elm)->field.le_prev = (elm)->field.le_next;                   \
}

/*
 * Tail queue definitions.
 */
#define TAILQ_HEAD(name, type)                                          \
struct name {                                                           \
        struct type *tqh_first; /* first element */                     \
        struct type **tqh_last; /* addr of last next element */         \
}

#define TAILQ_ENTRY(type)                                               \
struct {                                                                \
        struct type *tqe_next;  /* next element */                      \
        struct type **tqe_prev; /* address of previous next element */  \
}

/*
 * Tail queue functions.
 */
#define TAILQ_INIT(head) {                                              \
        (head)->tqh_first = NULL;                                       \
        (head)->tqh_last = &(head)->tqh_first;                          \
}

#define TAILQ_INSERT_HEAD(head, elm, field) {                           \
        if (((elm)->field.tqe_next = (head)->tqh_first) != NULL)        \
                (elm)->field.tqe_next->field.tqe_prev =                 \
                    &(elm)->field.tqe_next;                             \
        else                                                            \
                (head)->tqh_last = &(elm)->field.tqe_next;              \
        (head)->tqh_first = (elm);                                      \
        (elm)->field.tqe_prev = &(head)->tqh_first;                     \
}

#define TAILQ_INSERT_TAIL(head, elm, field) {                           \
        (elm)->field.tqe_next = NULL;                                   \
        (elm)->field.tqe_prev = (head)->tqh_last;                       \
        *(head)->tqh_last = (elm);                                      \
        (head)->tqh_last = &(elm)->field.tqe_next;                      \
}

#define TAILQ_INSERT_AFTER(head, listelm, elm, field) {                 \
        if (((elm)->field.tqe_next = (listelm)->field.tqe_next) != NULL)\
                (elm)->field.tqe_next->field.tqe_prev =                 \
                    &(elm)->field.tqe_next;                             \
        else                                                            \
                (head)->tqh_last = &(elm)->field.tqe_next;              \
        (listelm)->field.tqe_next = (elm);                              \
        (elm)->field.tqe_prev = &(listelm)->field.tqe_next;             \
}

#define TAILQ_REMOVE(head, elm, field) {                                \
        if (((elm)->field.tqe_next) != NULL)                            \
                (elm)->field.tqe_next->field.tqe_prev =                 \
                    (elm)->field.tqe_prev;                              \
        else                                                            \
                (head)->tqh_last = (elm)->field.tqe_prev;               \
        *(elm)->field.tqe_prev = (elm)->field.tqe_next;                 \
}

/*
 * Circular queue definitions.
```

```
 */
#define CIRCLEQ_HEAD(name, type)                                        \
struct name {                                                           \
        struct type * cqh_first ;               /* first element */     \
        struct type * cqh_last ;                /* last element */      \
}

#define CIRCLEQ_ENTRY(type)                                             \
struct {                                                                \
        struct type *cqe_next;                  /* next element */      \
        struct type *cqe_prev;                  /* previous element */  \
}

/*
 * Circular queue functions .
 */
#define CIRCLEQ_INIT(head) {                                            \
        (head)->cqh_first = (void *)(head);                             \
        (head)->cqh_last = (void *)(head);                              \
}

#define CIRCLEQ_INSERT_AFTER(head, listelm, elm, field) {               \
        (elm)->field.cqe_next = (listelm)->field.cqe_next;              \
        (elm)->field.cqe_prev = (listelm);                              \
        if ((listelm)->field.cqe_next == (void *)(head))                \
                (head)->cqh_last = (elm);                               \
        else                                                            \
                (listelm)->field.cqe_next->field.cqe_prev = (elm);      \
        (listelm)->field.cqe_next = (elm);                              \
}

#define CIRCLEQ_INSERT_BEFORE(head, listelm, elm, field) {              \
        (elm)->field.cqe_next = (listelm);                              \
        (elm)->field.cqe_prev = (listelm)->field.cqe_prev;              \
        if ((listelm)->field.cqe_prev == (void *)(head))                \
                (head)->cqh_first = (elm);                              \
        else                                                            \
                (listelm)->field.cqe_prev->field.cqe_next = (elm);      \
        (listelm)->field.cqe_prev = (elm);                              \
}

#define CIRCLEQ_INSERT_HEAD(head, elm, field) {                         \
        (elm)->field.cqe_next = (head)->cqh_first;                      \
        (elm)->field.cqe_prev = (void *)(head);                         \
        if ((head)->cqh_last == (void *)(head))                         \
                (head)->cqh_last = (elm);                               \
        else                                                            \
                (head)->cqh_first->field.cqe_prev = (elm);              \
        (head)->cqh_first = (elm);                                      \
}

#define CIRCLEQ_INSERT_TAIL(head, elm, field) {                         \
        (elm)->field.cqe_next = (void *)(head);                         \
        (elm)->field.cqe_prev = (head)->cqh_last;                       \
        if ((head)->cqh_first == (void *)(head))                        \
                (head)->cqh_first = (elm);                              \
        else                                                            \
                (head)->cqh_last->field.cqe_next = (elm);               \
        (head)->cqh_last = (elm);                                       \
}

#define CIRCLEQ_REMOVE(head, elm, field) {                              \
        if ((elm)->field.cqe_next == (void *)(head))                    \
                (head)->cqh_last = (elm)->field.cqe_prev;               \
        else                                                            \
                (elm)->field.cqe_next->field.cqe_prev =                 \
                    (elm)->field.cqe_prev;                              \
        if ((elm)->field.cqe_prev == (void *)(head))                    \
                (head)->cqh_first = (elm)->field.cqe_next;              \
        else                                                            \
                (elm)->field.cqe_prev->field.cqe_next =                 \
                    (elm)->field.cqe_next;                              \
}
#endif  /* queuing functions */

/*
 * we need some more operations , here is the rest :
 */

#define TAILQ_LAST(head, headname)                                      \
        (*(( struct headname *)((head)->tqh_last))->tqh_last))

#define TAILQ_FOREACH(var, head, field)                                 \
        for ((var) = TAILQ_FIRST((head));                               \
```

```
        (var);                                                  \
        (var) = TAILQ_NEXT((var), field))

#define TAILQ_FIRST(head)       ((head)->tqh_first)

#define TAILQ_NEXT(elm, field) ((elm)->field.tqe_next)

#define TAILQ_INSERT_BEFORE(listelm, elm, field) do {           \
        (elm)->field.tqe_prev = (listelm)->field.tqe_prev;      \
        TAILQ_NEXT((elm), field) = (listelm);                   \
        *(listelm)->field.tqe_prev = (elm);                     \
        (listelm)->field.tqe_prev = &TAILQ_NEXT((elm), field);  \
} while (0)

#define LIST_EMPTY(head)        ((head)->lh_first == NULL)
#define LIST_FOREACH(var, head, field)                          \
        for ((var) = LIST_FIRST((head));                        \
            (var);                                              \
            (var) = LIST_NEXT((var), field))
#define LIST_FIRST(head)        ((head)->lh_first)
#define LIST_NEXT(elm, field)   ((elm)->field.le_next)


/*
 * H-FSC data structures
 */

/*
 * kernel internal service curve representation
 *      coordinates are given by 64 bit unsigned integers.
 *      x-axis: unit is clock count.  for the intel x86 architecture,
 *              the raw Pentium TSC (Timestamp Counter) value is used.
 *              virtual time is also calculated in this time scale.
 *      y-axis: unit is byte.
 *
 *      the service curve parameters are converted to the internal
 *      representation.
 *      the slope values are scaled to avoid overflow.
 *      the inverse slope values as well as the y-projection of the 1st
 *      segment are kept in order to to avoid 64-bit divide operations
 *      that are expensive on 32-bit architectures.
 *
 *  note: Intel Pentium TSC never wraps around in several thousands of years.
 *      x-axis doesn't wrap around for 1089 years with 1GHz clock.
 *      y-axis doesn't wrap around for 4358 years with 1Gbps bandwidth.
 */

/* kernel internal representation of a service curve */
struct internal_sc {
        u64     sm1;    /* scaled slope of the 1st segment */
        u64     ism1;   /* scaled inverse-slope of the 1st segment */
        u64     dx;     /* the x-projection of the 1st segment */
        u64     dy;     /* the y-projection of the 1st segment */
        u64     sm2;    /* scaled slope of the 2nd segment */
        u64     ism2;   /* scaled inverse-slope of the 2nd segment */
};

/* runtime service curve */
struct runtime_sc {
        u64     x;      /* current starting position on x-axis */
        u64     y;      /* current starting position on y-axis */
        u64     sm1;    /* scaled slope of the 1st segment */
        u64     ism1;   /* scaled inverse-slope of the 1st segment */
        u64     dx;     /* the x-projection of the 1st segment */
        u64     dy;     /* the y-projection of the 1st segment */
        u64     sm2;    /* scaled slope of the 2nd segment */
        u64     ism2;   /* scaled inverse-slope of the 2nd segment */
};

/* for TAILQ based ellist and actlist implementation */
struct hfsc_class;
typedef TAILQ_HEAD(_eligible, hfsc_class)   ellist_t;
typedef TAILQ_ENTRY(hfsc_class) elentry_t;
typedef TAILQ_HEAD(_active, hfsc_class)   actlist_t;
typedef TAILQ_ENTRY(hfsc_class) actentry_t;
#define   ellist_first  (s)       TAILQ_FIRST(s)
#define   actlist_first (s)       TAILQ_FIRST(s)
#define   actlist_last  (s)       TAILQ_LAST(s, _active)

/*
 * generalized service curve used for admission control
 */
struct segment {
        LIST_ENTRY(segment)   _next;
```

```
405         u64       x, y, d, m;
};

typedef LIST_HEAD(gen_sc, segment) gsc_head_t;

410 /*
 * hfsc private classinfo structure
 */
struct hfsc_classinfo {
        struct service_curve rsc;       /* real-time service curve */
415     struct service_curve fsc;       /* fair service curve */
        gsc_head_t      gen_rsc;        /* generalized real-time sc */
        gsc_head_t      gen_fsc;        /* generalized fsc */
        int             qlimit;
};
420


struct hfsc_class {
        u32                     cl_id;          /* class id */
425     unsigned long           cl_handle;      /* class handle */
        struct Qdisc            *cl_sch;        /* back pointer to struct Qdisc */
        int                     cl_flags;       /* misc flags */

        struct hfsc_class       *cl_parent;     /* parent class */
430     struct hfsc_class       *cl_siblings;   /* sibling classes */
        struct hfsc_class       *cl_children;   /* child classes */

        struct hfsc_classinfo   *cl_clinfo;     /* class info, for admission control */
        struct Qdisc            *cl_q;          /* qdisc for storing packets of this class */
435     struct sk_buff          *cl_peeked;     /* next skb to be dequeued */
        unsigned int            cl_limit;       /* limit of class queue */
        unsigned int            cl_qlen;        /* number of stored packets */

        u64                     cl_total;       /* total work in bytes */
440     u64                     cl_cumul;       /* cumulative work in bytes
                                                   done by real-time criteria */
        u64                     cl_drop_cumul;  /* cumulative amount of service
                                                   for drop curve (in bytes) */
        u64                     cl_d;           /* deadline */
445     u64                     cl_e;           /* eligible time */
        u64                     cl_vt;          /* virtual time */
        u64                     cl_k;           /* packet drop time */
        u64                     cl_delta_t_drop; /* difference between real time and
                                                    time of packet drop service curve */
450
        struct internal_sc      *cl_rsc;        /* internal real-time service curve */
        struct internal_sc      *cl_fsc;        /* internal fair service curve */
        struct internal_sc      *cl_dsc;        /* packet drop service curve */
        struct runtime_sc       cl_deadline;    /* deadline curve */
455     struct runtime_sc       cl_eligible;    /* eligible curve */
        struct runtime_sc       cl_virtual;     /* virtual curve */
        struct runtime_sc       cl_dropcurve;   /* packet drop curve */

#ifdef CONFIG_NET_WIRELESS_SCHED
460     /*
         * declarations for wireless synchronization classes
         *
         * These classes are used in order to be able to reduce
         * realtime service curves in a subtree in cases when not
465      * enough bandwidth is available.
         */
        struct hfsc_class       *cl_sync_class; /* parent sync class */
        struct runtime_sc       cl_sync_el_sum; /* internal representation of sum of all
                                                   eligible curves - if this is a sync class */
470     u64                     cl_sync_required_cap;   /* avg. required capacity in byte/s */
#endif

        unsigned int            cl_vtperiod;    /* vt period sequence no */
        unsigned int            cl_parentperiod; /* parent's vt period seqno */
475     int                     cl_nactive;     /* number of active children */
        actlist_t               *cl_actc;       /* active children list */

        actentry_t              cl_actlist;     /* active children list entry */
        elentry_t               cl_ellist;      /* eligible list entry */
480
        struct {
                unsigned long   xmit_cnt;
                unsigned long   drop_cnt;
                unsigned int    period;
485     } cl_stats;
        unsigned int            cl_refcnt;      /* counts references to this class */
        unsigned int            cl_filters;     /* counts number of attached filters / classifies */
};
```

```
/*
 * constants for scheduler state
 */

#define HFSC_SCHED_EST_NOT_ACTIVE 0
#define HFSC_SCHED_EST_ACTIVE     1

/*
 * definitions for rate adaptation mechanism
 *
 * Ideally we should never fall behind the current time but since the calls
 * to the scheduler are rather coarse, we need the following definitions.
 * The values should be chosen as low as possible but high enough to guarantee
 * that are rate adaption is only performed if we have a long-term tendency to
 * fall behind the current time.
 *
 * We chose different values if compiled for real kernel since the hardware is not
 * as responsive as the (idealized, descrete-event driven) simulation.
 */

#ifdef TCSIM_COMPATIBLE
 #define HFSC_MAX_BACKLOG_TIME 1000      /* max. allowed backlog of a class [in us] */
 #define HFSC_REDUCE_BACKLOG_TIME 500    /* amount by which backlog is reduced
                                            if it is larger than the maximum backlog */
#else
 #define HFSC_MAX_BACKLOG_TIME 200000    /* max. allowed backlog of a class [in us] */
 #define HFSC_REDUCE_BACKLOG_TIME 100000 /* amount by which backlog is reduced
                                            if it is larger than the maximum backlog */
#endif /* TCSIM_COMPATIBLE */

/*
 * hfsc scheduler data (per interface)
 */
struct hfsc_sched_data {
        struct hfsc_class       * sched_rootclass;      /* root class */
        struct hfsc_class       * sched_defaultclass;   /* default class */
        struct Qdisc            *sched_requeued_skbs;   /* a requeued skb which was already
                                                           completely processed */

        unsigned int            sched_classes;          /* # of classes in the tree */
        unsigned int            sched_packets;          /* # of packets in the tree */
        unsigned int            sched_classid;          /* class id sequence number */

        ellist_t                * sched_eligible;       /* eligible list */

        struct tcf_proto        * sched_filter_list;    /* list of attached filters */

        struct timer_list       wd_timer;               /* Watchdog timer,
                                                           started when the qdisc has
                                                           packets, but no eligible packets
                                                           and no link-sharing packets just now */

        unsigned char           sched_est_active;       /* currently estimating root class rate? */
        u64                     sched_est_time;         /* time of last degrade factor estimation */
        u64                     sched_est_length;       /* length of last dequeued packet */
        u64                     sched_est_dfactor;      /* current estimation of degrade factor */
        unsigned long           sched_est_interval_log; /* log2 of length of estimation interval (bytes) */
        struct runtime_sc       sched_est_el_sum;       /* sum of all active eligible curves */

        unsigned char           sched_flags;            /* scheduler wide flags */
        u64                     sched_est_required_cap; /* total required scheduler capacity needed to fullfil real-
                time requests */
#ifdef CONFIG_NET_WIRELESS_SCHED
        struct wsched_dest      cur_dst;                /* destination of packet currently scheduled */
        unsigned int            sched_est_reducebad;    /* percentage at which goodput/throughput
                                                           ratio is taken into account */
#endif /* CONFIG_NET_WIRELESS_SCHED */

};

/*
 * functions handling general service curves (i.e. for admission control)
 */
static void gsc_add_sc(struct gen_sc *gsc, struct service_curve *sc);
static void gsc_sub_sc(struct gen_sc *gsc, struct service_curve *sc);
static int is_gsc_under_sc(struct gen_sc *gsc, struct service_curve *sc);
static void gsc_destroy(struct gen_sc *gsc);
static struct segment *gsc_getentry(struct gen_sc *gsc, u64 x);
static int gsc_add_seg(struct gen_sc *gsc,
                       u64 x, u64 y, u64 d, u64 m);
static int gsc_sub_seg(struct gen_sc *gsc,
                       u64 x, u64 y, u64 d, u64 m);
static void gsc_compress(struct gen_sc *gsc);
```

```
static u64 sc_x2y(struct service_curve *sc, u64 x);

#endif /* _NET_SCHED_SCH_HFSC_H_ */
```

Listing B.4: *The wireless H-FSC kernel module (*`sch_hfsc.c`*).*

```
/* $Id: */
/*
 * net/sched/sch_hfsc.c      Implementation of the
 *                           Hierarchical Fair Service Curve Algorithm (H-FSC)
 *
 *                           with modifications for the support of a
 *                           hierarchical wireless link sharing model
 *
 * Authors:      Lars Wischhof <wischhof@ieee.org>,
 *               Washington University, St. Louis, MO, USA
 *               Technical University of Berlin, Germany
 *
 *               based on the implementation of H-FSC
 *               of Carnegie Mellon University (as of ALT-Q 3.0)
 *               (see copyright notice below)
 *               and the implementation of the CBQ scheduler
 *               by Alexey Kuznetsov, <kuznet@ms2.inr.ac.ru>
 *
 * Note: Qdisc does not work properly if PSCHED_CPU is not available!
 *       In the TCSIM environment always define the flag
 *       TCSIM_COMPATIBLE or else it will be extremely inaccurate.
 *
 * $Log: sch_hfsc.c,v $
 * Revision 1.1.2.1    2001/11/12 20:37:37    wischhof
 * Monday afternoon version - final
 *
 * Revision 1.8    2001/11/02 19:49:18    wischhof
 * release 0.5.1
 *
 */

/*
 * Copyright (c) 1997-1999 Carnegie Mellon University. All Rights Reserved.
 *
 * Permission to use, copy, modify, and distribute this software and
 * its documentation is hereby granted (including for commercial or
 * for-profit use), provided that both the copyright notice and this
 * permission notice appear in all copies of the software, derivative
 * works, or modified versions, and any portions thereof, and that
 * both notices appear in supporting documentation, and that credit
 * is given to Carnegie Mellon University in all publications reporting
 * on direct or indirect use of this code or its derivatives.
 *
 * THIS SOFTWARE IS EXPERIMENTAL AND IS KNOWN TO HAVE BUGS, SOME OF
 * WHICH MAY HAVE SERIOUS CONSEQUENCES. CARNEGIE MELLON PROVIDES THIS
 * SOFTWARE IN ITS "AS IS" CONDITION, AND ANY EXPRESS OR IMPLIED
 * WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES
 * OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
 * DISCLAIMED. IN NO EVENT SHALL CARNEGIE MELLON UNIVERSITY BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT
 * OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR
 * BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
 * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
 * (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE
 * USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH
 * DAMAGE.
 *
 * Carnegie Mellon encourages (but does not require) users of this
 * software to return any improvements or extensions that they make,
 * and to grant Carnegie Mellon the rights to redistribute these
 * changes without encumbrance.
 *
 */

#include <linux/config.h>
#include <linux/module.h>
#include <asm/uaccess.h>
#include <asm/system.h>
#include <asm/bitops.h>
#include <linux/types.h>
#include <linux/kernel.h>
#include <linux/sched.h>
#include <linux/string.h>
#include <linux/mm.h>
#include <linux/socket.h>
#include <linux/sockios.h>
#include <linux/in.h>
#include <linux/errno.h>
#include <linux/interrupt.h>
#include <linux/if_ether.h>
#include <linux/inet.h>
```

```
#include <linux/netdevice.h>
#include <linux/etherdevice.h>
#include <linux/notifier.h>
#include <net/ip.h>
#include <net/route.h>
#include <linux/skbuff.h>
#include <net/sock.h>
#include <net/pkt_sched.h>

/*
 * compile time switches/definitions
 */
//# define TCSIM_COMPATIBLE        /* define this if you are compiling for TCSIM! */

//# define USE_64BIT_DIVISIONS     /* define this in order to allow the usage of 64 bit divisions.
//                                    this increases accuracy but is not allowed for kernel compilation
//                                    on most platforms (e.g. I386) */

#define DEBUG_ENABLED             /* define this to compile a debugging/development version,
                                     undefine this in order to compile a release version
                                     without debugging overhead */

#define DEBUG_TIME            100
#define DEBUG_FCT_TRACE        10
#define DEBUG_INFO              5
#define DEBUG_WARN              1

#include "sch_hfsc.h"

MODULE_PARM(hfsc_debug, "i");
#ifdef TCSIM_COMPATIBLE
#include "../jiffies.h"                   /* hires-timer in simulation env. */
static unsigned int hfsc_debug = 100;     /* default for TCSIM is debugging */
#else
static unsigned int hfsc_debug = 0;
#endif

#ifdef DEBUG_ENABLED
#define DEBUG(n, args...) if ( hfsc_debug >=(n)) printk(KERN_DEBUG args)
#ifdef ASSERT
#undef ASSERT
#endif
#define ASSERT(x) if (!(x)) { printk("KERNEL: assertion (" #x ") failed at " __FILE__ "(%d):" __FUNCTION__ "\n",
    __LINE__); }
#define FCT_IN if( hfsc_debug >= DEBUG_FCT_TRACE )printk(KERN_DEBUG "->" __FUNCTION__ " (" __FILE__ ":line
    %d)\n", __LINE__);
#define FCT_OUT if( hfsc_debug >= DEBUG_FCT_TRACE )printk(KERN_DEBUG "<-" __FUNCTION__ " (" __FILE__ ":
    line %d)\n", __LINE__);
#else
#define FCT_IN
#define FCT_OUT
#define DEBUG(n, args...)
#define ASSERT(x)
#endif

/*
 * check that clock source is CPU timestamp
 */
#ifndef TCSIM_COMPATIBLE
 #if (PSCHED_CLOCK_SOURCE != PSCHED_CPU)
  #error This qdisc needs PSCHED_CLOCK_SOURCE CPU, define it in "pkt_sched.h"!
 #endif
#endif /* TCSIM_COMPATIBLE */

/*
 * function prototypes
 */
static int hfsc_init (struct Qdisc *sch, struct rtattr *opt);
static void hfsc_destroy (struct Qdisc *sch);
static void hfsc_reset (struct Qdisc *);
static void hfsc_purge (struct Qdisc *);
static struct hfsc_class * hfsc_class_create (struct Qdisc *,
                    struct service_curve *, struct hfsc_class *, int, int, u32);
static int hfsc_class_destroy (struct Qdisc * sch, struct hfsc_class *cl);
static int hfsc_class_modify (struct Qdisc *, struct hfsc_class *,
                              struct service_curve *, struct service_curve *,
                              struct service_curve *);
static struct hfsc_class * hfsc_nextclass (struct hfsc_class *);

static int hfsc_enqueue(struct sk_buff *skb, struct Qdisc *sch);
static struct sk_buff *hfsc_dequeue (struct Qdisc *);

static int hfsc_addq (struct hfsc_class *, struct sk_buff *);
```

```
static struct sk_buff *hfsc_getq ( struct hfsc_class *);
static struct sk_buff *hfsc_pollq ( struct hfsc_class *);
static void hfsc_purgeq ( struct hfsc_class *);

static void set_active ( struct hfsc_class *, int);
static void set_passive ( struct hfsc_class *);

static void init_ed ( struct hfsc_class *, int);
static void update_ed ( struct hfsc_class *, int);
static void update_d ( struct hfsc_class *, int);
static void update_k( struct hfsc_class *cl , int next_len);
static void init_v ( struct hfsc_class *, int);
static void update_v ( struct hfsc_class *, int , int);
static ellist_t * ellist_alloc (void);
static void ellist_destroy ( ellist_t *);
static void ellist_insert ( struct hfsc_class *);
static void ellist_remove ( struct hfsc_class *);
static void ellist_update ( struct hfsc_class *);
struct hfsc_class * ellist_get_mindl ( ellist_t *, u64 );
static actlist_t * actlist_alloc (void);
static void actlist_destroy ( actlist_t *);
static void actlist_insert ( struct hfsc_class *);
static void actlist_remove ( struct hfsc_class *);
static void actlist_update ( struct hfsc_class *);

static __inline u64 seg_x2y (u64, u64);
static __inline u64 seg_y2x (u64, u64);
static __inline u64 m2sm (unsigned int);
static __inline u64 m2ism (unsigned int);
static __inline u64 d2dx (unsigned int);
static unsigned int sm2m (u64);
static unsigned int dx2d (u64);

static void sc2isc ( struct service_curve *, struct internal_sc *);
static void rtsc_init ( struct runtime_sc *, struct internal_sc *,
                        u64, u64);
static u64 rtsc_y2x ( struct runtime_sc *, u64);
static u64 rtsc_x2y ( struct runtime_sc *, u64);
static void rtsc_min ( struct runtime_sc *, struct internal_sc *,
                       u64, u64);
static inline u64 read_machclk(void);

static inline struct hfsc_class *hfsc_class_lookup(struct hfsc_sched_data *q, u32 classid );
static inline unsigned int service_packet_drop ( struct hfsc_class *cl , u64 time );
static void init_k ( struct hfsc_class *cl , int next_len);

static void dfactor_start_estimate ( struct hfsc_sched_data *q );
static void dfactor_increase ( struct hfsc_sched_data *q );
static void dfactor_update ( struct hfsc_sched_data *q, u64 current_time , unsigned long length );
static unsigned long dfactor_rt_service_estimate ( struct hfsc_sched_data *q,
       struct hfsc_class *cl , struct sk_buff * skb );
#ifdef CONFIG_NET_WIRELESS_SCHED
static unsigned long dfactor_vt_service_estimate ( struct hfsc_sched_data *q,
       struct hfsc_class *cl , struct sk_buff * skb );
#endif /* CONFIG_NET_WIRELESS_SCHED */
static void dfactor_stop_estimate ( struct hfsc_sched_data *q );

static inline void hfsc_start_watchdog_timer ( struct hfsc_sched_data *q );
static void hfsc_watchdog(unsigned long arg);


static inline void dfactor_add_curve ( struct hfsc_class * cl , struct runtime_sc *c);
static inline void dfactor_remove_curve ( struct hfsc_class * cl , struct runtime_sc *c);


/*
 * BSDlike clock functions
 */
#ifdef TCSIM_COMPATIBLE
        static inline u64 read_machclk(void)
        { u64 __result ;
          __result = now. jiffies *10000 + now. ujiffies /100;
          return __result ;
        }
#else
        static inline u64 read_machclk(void)
        { u64 __result ;
          // rdtscll ( __result );
          PSCHED_GET_TIME(__result);
          DEBUG(DEBUG_TIME, "time: %Lu\n", __result);
          return __result ;
        }
#endif
```

```
#define machclk_freq 1000000
#define machclk_freq_log 20 /* 2^20 = 1048576 */

/*
 * macros
 */
#define DFACTOR_ESTIMATION_ACTIVE(q) (q->sched_est_active == HFSC_SCHED_EST_ACTIVE)

#define is_a_parent_class (cl) ((cl)->cl_children != NULL)

#define PKTCNTR_ADD(cnt, len) *cnt+=len;

/*
 * initialize a new scheduler with parameters received via rtnetlink
 */

static int
hfsc_init (struct Qdisc *sch, struct rtattr *opt)
{
        unsigned int bandwidth=0;
        struct hfsc_sched_data *q = (struct hfsc_sched_data *) sch->data;
        struct service_curve root_sc;
        struct tc_hfsc_qopt * ctl = RTA_DATA(opt);

        FCT_IN;

        if (opt == NULL)
            return -EINVAL;

        /*
         * check if we received a valid control packet
         */
        if (opt->rta_len < RTA_LENGTH(sizeof(*ctl)))
                return -EINVAL;

        if (!(bandwidth = ctl->bandwidth))
                return -EINVAL;

        q->sched_eligible = ellist_alloc();

        /*
         * assign scheduler flags
         */
        if (ctl->flags & HFSC_SCHED_VAR_RATE)
                q->sched_flags |= HFSC_SCHED_VAR_RATE;

#ifdef CONFIG_NET_WIRELESS_SCHED
        if (ctl->flags & HFSC_SCHED_WIRELESS_MON)
        {
                if (!ctl->wchmon_id[0]) /* set monitor if given */
                {
                        printk("hfsc: Wireless channel monitor not specified!\n");
                        return -EINVAL;
                }

                if (wsched_wchmon_add(sch->dev, ctl->wchmon_id)!= 0)
                {
                        printk("hfsc: Unable to add wireless channel monitor to device!\n");
                        return -EINVAL;
                }

                q->sched_flags |= HFSC_SCHED_WIRELESS_MON;

                if (ctl->reducebad_valid)
                        q->sched_est_reducebad = ctl->reducebad;
                else
                        q->sched_est_reducebad = 100;
        }
#endif /* CONFIG_NET_WIRELESS_SCHED */

        if (ctl->est_interval_log) {
                q->sched_est_interval_log = ctl->est_interval_log;
        } else {
                q->sched_est_interval_log = 13;
        }

        if (q->sched_eligible == NULL) {
                return -ENOMEM;
        }

        /*
         * create root class (@@@ as default class?)
         * (root class is also always a synchronization class)
```

```
330             */

                root_sc.m1 = bandwidth;
                root_sc.d = 0;
                root_sc.m2 = bandwidth;
335             if ((q->sched_rootclass =
                        hfsc_class_create(sch, &root_sc, NULL, 0, HFCF_DEFAULTCLASS | HFCF_SYNCCLASS, sch->handle)) ==
                            NULL) {
                    return -ENOMEM;
                }

340             /*
                 * init watchdog timer
                 */
                init_timer(&q->wd_timer);
                q->wd_timer.data = (unsigned long)sch;
345             q->wd_timer.function = hfsc_watchdog;

                /*
                 * init requeued packet handling
                 */
350             if ((q->sched_requeued_skbs = qdisc_create_dflt(sch->dev, &pfifo_qdisc_ops)) == NULL)
                        return -ENOMEM;

                MOD_INC_USE_COUNT;
                FCT_OUT;
355             return(0);
    }

    /*
     * destructor
360  */
    static void
    hfsc_destroy(struct Qdisc* sch)
    {
                struct hfsc_sched_data *q = (struct hfsc_sched_data *) sch->data;
365
                FCT_IN;

                /*
                 * @@@ virtual time lists, destroy .... @@@
370              */
                ellist_destroy(q->sched_eligible);
                /*
                 * @@@ destroy all filters and all classes @@@
                 */
375             (void) hfsc_reset(sch);
                (void) hfsc_class_destroy(sch, q->sched_rootclass);

    #ifdef CONFIG_NET_WIRELESS_SCHED
                if ( q->sched_flags & HFSC_SCHED_WIRELESS_MON )
380                     wsched_wchmon_del( sch->dev );
    #endif /* CONFIG_NET_WIRELESS_SCHED */

                del_timer(&q->wd_timer);
                MOD_DEC_USE_COUNT;
385             FCT_OUT;
                return;
    }

    /*
390  * bring the qdisc back to the initial state by discarding
     * all the filters and classes except the root class.
     */
    static void
    hfsc_reset(struct Qdisc *sch)
395 {
                struct hfsc_sched_data *q = (struct hfsc_sched_data *) sch->data;
                struct hfsc_class      *cl;

                FCT_IN;
400             /* purge all stored packets */
                hfsc_purge(sch);
                /* clear out the classes */
                while ((cl = q->sched_rootclass->cl_children) != NULL) {
                        /*
405                      * remove the first leaf class found in the hierarchy
                         * then start over
                         */
                        for (; cl != NULL; cl = hfsc_nextclass(cl)) {
                                if (!is_a_parent_class(cl)) {
410                                     (void) hfsc_class_destroy(sch, cl);
                                        break;
                                }
```

```
                }
        }
        FCT_OUT;
        return;
}

/*
 * discard all the queued packets of this scheduler
 */
static void
hfsc_purge(struct Qdisc *sch)
{
        struct hfsc_sched_data *q = (struct hfsc_sched_data *) sch->data;
        struct hfsc_class *cl;

        FCT_IN;
        for (cl = q->sched_rootclass; cl != NULL; cl = hfsc_nextclass(cl))
                hfsc_purgeq(cl);
        FCT_OUT;
}

/*
 * create a new class
 */
struct hfsc_class *
hfsc_class_create(struct Qdisc *sch, struct service_curve *sc, struct hfsc_class *parent,
                  int qlimit, int flags, u32 classid)
{
        struct hfsc_classinfo *parent_clinfo =NULL; /* admission ctl info of parent */
        struct hfsc_classinfo *hfsc_clinfo =NULL;  /* admission ctl info of new class */
        struct hfsc_class *cl, *p;
        struct hfsc_sched_data *q = (struct hfsc_sched_data *) sch->data;

        FCT_IN;

        /* check that parent class has not filters attached */
        if (parent != NULL)
            if (parent-> cl_filters ) {
                printk(" hfsc_class_create : cannot create child class of a parent with filters !\n");
                return NULL;
            }
        /* admission control */
        if (parent != NULL && !is_sc_null(sc)) {
                parent_clinfo = parent->cl_clinfo;
                gsc_add_sc(&parent_clinfo->gen_rsc, sc);
                gsc_add_sc(&parent_clinfo->gen_fsc, sc);
                if (!is_gsc_under_sc(&parent_clinfo->gen_rsc,
                                     &parent_clinfo->rsc) ||
                    !is_gsc_under_sc(&parent_clinfo->gen_fsc,
                                     &parent_clinfo->fsc)) {
                        /* admission control failure */
                        printk(" hfsc_class_create : admission control failed !\n");
                        gsc_sub_sc(&parent_clinfo->gen_rsc, sc);
                        gsc_sub_sc(&parent_clinfo->gen_fsc, sc);
                        return NULL;
                }
        }

        if ((hfsc_clinfo = kmalloc(sizeof(*hfsc_clinfo), GFP_KERNEL)) == NULL) {
                printk(" hfsc_class_create : out of mem\n");
                return NULL;
        }
        memset(hfsc_clinfo, 0, sizeof(*hfsc_clinfo));
        hfsc_clinfo->rsc = *sc;
        hfsc_clinfo->fsc = *sc;
        LIST_INIT(&hfsc_clinfo->gen_rsc);
        LIST_INIT(&hfsc_clinfo->gen_fsc);
        hfsc_clinfo->qlimit = qlimit;

        if ((parent == NULL) && !(q->sched_flags & HFSC_SCHED_WIRELESS_MON)) {
                /*
                 * if this is a root class, reserve 20% of the real-time
                 * bandwidth for safety.
                 * many network cards are not able to saturate the wire,
                 * and if we allocate real-time traffic more than the
                 * maximum sending rate of the card, hfsc is no longer
                 * able to meet the delay bound requirements.
                 *
                 * In the wireless case we assume that the "bandwidth"
                 * specified by the user is already a valid estimate.
                 */
                hfsc_clinfo->rsc.m1 = hfsc_clinfo->rsc.m1 / 10 * 8;
                hfsc_clinfo->rsc.m2 = hfsc_clinfo->rsc.m2 / 10 * 8;
        }
```

```
        cl = kmalloc (sizeof(struct hfsc_class), GFP_KERNEL);
        if (cl == NULL)
500     {
                DEBUG( DEBUG_WARN, "hfsc_class_create: out of mem\n" );
                return (NULL);
        }
        memset(cl, 0, sizeof(struct hfsc_class));
505
        cl->cl_clinfo = hfsc_clinfo;      /* assign structure for admission ctrl */

        if ((cl->cl_q = qdisc_create_dflt (sch->dev, &pfifo_qdisc_ops)) == NULL)
                goto err_ret;
510
        cl->cl_actc = actlist_alloc ();
        if (cl->cl_actc == NULL)
                goto err_ret;

515     if (qlimit == 0)
                qlimit = 50;    /* use default */
        cl->cl_limit = qlimit;
        cl->cl_flags = flags;

520     /*
         * convert service curve to internal representation and
         * initialize link-sharing and real-time curves to sc
         */
        if (sc != NULL && (sc->m1 != 0 || sc->m2 != 0)) {
525             cl->cl_rsc = kmalloc(sizeof(struct internal_sc), GFP_KERNEL);
                if (cl->cl_rsc == NULL)
                        goto err_ret;
                memset(cl->cl_rsc, 0, sizeof(struct internal_sc));
                sc2isc(sc, cl->cl_rsc);
530
                rtsc_init (&cl->cl_deadline, cl->cl_rsc, 0, 0);
                rtsc_init (&cl->cl_eligible, cl->cl_rsc, 0, 0);

                cl->cl_fsc=kmalloc(sizeof(struct internal_sc), GFP_KERNEL);
535             if (cl->cl_fsc == NULL)
                        goto err_ret;
                memset(cl->cl_fsc, 0, sizeof(struct internal_sc));
                sc2isc(sc, cl->cl_fsc);
                rtsc_init (&cl->cl_virtual, cl->cl_fsc, 0, 0);
540     }

        cl->cl_id = classid;
        cl->cl_handle = (unsigned long)cl;    /* XXX: just a pointer to this class */
        cl->cl_sch = sch;
545     cl->cl_parent = parent;

        sch_tree_lock(sch);

        q->sched_classes++;
550
        if (flags & HFCF_DEFAULTCLASS) {
                if (q->sched_defaultclass) {
                        /*
                         * remove default class flag from old default class
555                      */
                        q->sched_defaultclass->cl_flags ^= HFCF_DEFAULTCLASS;
                }
                q->sched_defaultclass = cl;
        }
560     else if (parent && (parent == q->sched_defaultclass))
        {
                printk(" hfsc_class_create: default class cannot be parent! new default is child!\n");
                q->sched_defaultclass = cl;
                parent->cl_flags ^= HFCF_DEFAULTCLASS;
565             cl->cl_flags = cl->cl_flags | HFCF_DEFAULTCLASS;
        }

        /* add this class to the children list of the parent */
        if (parent == NULL) {
570             /* this is root class */
        }
        else if ((p = parent->cl_children) == NULL)
                parent->cl_children = cl;
        else {
575             while (p->cl_siblings != NULL)
                        p = p->cl_siblings;
                p->cl_siblings = cl;
        }

580 #ifdef CONFIG_NET_WIRELESS_SCHED
```

```
        /*
         * locate and set parent (real-time) synchronization class
         *
         * (if this class is a synchronization class itself,
         *  the class is its own synchronization class)
         *
         * The user is warned if the parent synchronization class has no valid
         * real-time sc, since this is probably a configuration error.
         * However, there are situations in which it makes sense to have different
         * synchronization classes for real-time and link-sharing criteria, which
         * is while we still allow to continue.
         */
        p = cl;
        while ((p != NULL) && (cl->cl_sync_class == NULL)){
                if (p->cl_flags & HFCF_SYNCCLASS){
                        if (p->cl_rsc != NULL){
                                cl->cl_sync_class = p;
                        } else {
                                printk("hfsc_class_create: Warning, parent sync class has no realtime sc!\n");
                                printk("hfsc_class_create: Real-time sync class is therefore not equal to link-sharing
                                    sync!\n");
                                printk("hfsc_class_create: Is this really what you want?\n");
                                /* do nothing, continue search for realtime sc */
                        }
                }
                p = p->cl_parent;
        }
#endif /* CONFIG_NET_WIRELESS_SCHED */

        sch_tree_unlock(sch);

        FCT_OUT;
        return (cl);

err_ret:
        /* cancel admission control */
        if (parent != NULL && !is_sc_null(sc)) {
                gsc_sub_sc(&parent_clinfo->gen_rsc, sc);
                gsc_sub_sc(&parent_clinfo->gen_fsc, sc);
        }

        /* destroy allocated lists */
        if (cl->cl_actc != NULL)
                actlist_destroy (cl->cl_actc);
        if (cl->cl_fsc != NULL)
                kfree(cl->cl_fsc);
        if (cl->cl_rsc != NULL)
                kfree(cl->cl_rsc);
        kfree(cl);
        return (NULL);
}

/*
 * class destructor
 */
static int
hfsc_class_destroy(struct Qdisc *sch, struct hfsc_class *cl)
{
        struct hfsc_sched_data *q = (struct hfsc_sched_data *) sch->data;
        struct tcf_proto *this_filter = NULL;
        struct tcf_proto *next_filter = NULL;
        struct tcf_proto **prev_filter_ptr = NULL;

        FCT_IN;
        ASSERT(cl != NULL);
        if (is_a_parent_class(cl))
        {
                printk("hfsc: cannot destroy a parent class!\n");
                cl->cl_refcnt++;
                return (-EINVAL);
        }

        if (cl->cl_flags & HFCF_DEFAULTCLASS)
        {
                DEBUG(DEBUG_INFO, "hfsc: default class deleted\n");
                q->sched_defaultclass = NULL;
        }
        /* delete filters referencing to this class */
         this_filter = q-> sched_filter_list;
         prev_filter_ptr = &(q-> sched_filter_list);
        while (this_filter) {
                if (this_filter->classid == cl->cl_id) {
                        next_filter = this_filter->next;
```

```
                        * prev_filter_ptr = this_filter ->next;

                        this_filter ->ops->destroy( this_filter );
                        this_filter = next_filter ;
                }
                else
                {
                        prev_filter_ptr = &((* prev_filter_ptr )->next);
                        this_filter = this_filter ->next;
                }
        }

        hfsc_purgeq( cl );

        if ( cl->cl_parent == NULL) {
                /* this is root class */
        } else {
                struct hfsc_class *p = cl->cl_parent->cl_children;

                if ( p == cl )
                        cl->cl_parent->cl_children = cl->cl_siblings;
                else do {
                        if ( p->cl_siblings == cl ) {
                                p->cl_siblings = cl->cl_siblings;
                                break;
                        }
                } while (( p = p->cl_siblings) != NULL);
                ASSERT(p != NULL);
        }
        (( struct hfsc_sched_data *) cl->cl_sch->data)->sched_classes--;

        actlist_destroy (cl->cl_actc);

        if ( cl->cl_fsc != NULL)
                kfree( cl->cl_fsc);
        if ( cl->cl_rsc != NULL)
                kfree( cl->cl_rsc);
        if ( cl->cl_clinfo)
        {
                gsc_destroy(&cl->cl_clinfo->gen_rsc);
                gsc_destroy(&cl->cl_clinfo->gen_fsc);
        }

        kfree( cl );
        FCT_OUT;
        return (0);
}

/*
 * modify the service cuves associated with a class
 */
static int
hfsc_class_modify( struct Qdisc *sch, struct hfsc_class *cl,
                   struct service_curve *rsc, struct service_curve *fsc, struct service_curve *dsc)
{
        struct hfsc_classinfo * parent_clinfo =NULL; /* admission ctl info of parent */
        struct hfsc_classinfo * hfsc_clinfo =NULL; /* admission ctl info of new class */
        struct internal_sc *tmp;
        struct service_curve old_rsc , old_fsc;

        FCT_IN;

        if ( ! cl->cl_parent )
        {
            DEBUG( DEBUG_WARN, "cannot modify class without a parent!\n" );
            return(-EINVAL);
        }

        hfsc_clinfo     = cl->cl_clinfo;
        parent_clinfo = cl->cl_parent->cl_clinfo;

        /* save old service curves */
        old_rsc = hfsc_clinfo ->rsc;
        old_fsc = hfsc_clinfo ->fsc;

        /* admission control */
        if ( rsc) {
                if (! is_gsc_under_sc(&hfsc_clinfo ->gen_rsc, rsc)) {
                        /* admission control failure */
                        printk ("class modify: admission control failure \n");
                        return (-EINVAL);
                }

                gsc_sub_sc(& parent_clinfo ->gen_rsc, &hfsc_clinfo ->rsc);
```

```
                gsc_add_sc(&parent_clinfo->gen_rsc, rsc);
                if (! is_gsc_under_sc(&parent_clinfo->gen_rsc,
                                      &parent_clinfo->rsc)) {
                        /* admission control failure */
                        gsc_sub_sc(&parent_clinfo->gen_rsc, rsc);
                        gsc_add_sc(&parent_clinfo->gen_rsc, &hfsc_clinfo->rsc);
                        printk("class modify: admission control failure\n");
                        return (-EINVAL);
                }
                hfsc_clinfo->rsc = *rsc;
        }
        if (fsc) {
                if (! is_gsc_under_sc(&hfsc_clinfo->gen_fsc, fsc)) {
                        /* admission control failure */
                        printk("class modify: admission control failure\n");
                        return (-EINVAL);
                }

                gsc_sub_sc(&parent_clinfo->gen_fsc, &hfsc_clinfo->fsc);
                gsc_add_sc(&parent_clinfo->gen_fsc, fsc);
                if (! is_gsc_under_sc(&parent_clinfo->gen_fsc,
                                      &parent_clinfo->fsc)) {
                        /* admission control failure */
                        gsc_sub_sc(&parent_clinfo->gen_fsc, fsc);
                        gsc_add_sc(&parent_clinfo->gen_fsc, &hfsc_clinfo->fsc);
                        printk("class modify: admission control failure\n");
                        return (-EINVAL);
                }
                hfsc_clinfo->fsc = *fsc;
        }

        hfsc_purgeq(cl);

        sch_tree_lock(sch);

        if (rsc != NULL) {
                if (rsc->m1 == 0 && rsc->m2 == 0) {
                        if (cl->cl_rsc != NULL) {
                                kfree(cl->cl_rsc);
                                cl->cl_rsc = NULL;
                        }
                } else {
                        if (cl->cl_rsc == NULL) {
                                tmp=kmalloc(sizeof(struct internal_sc), GFP_KERNEL);
                                if (tmp == NULL) {
                                        sch_tree_unlock(sch);
                                        /* modify failed!, restore the old service curves */
                                        if (rsc) {
                                                gsc_sub_sc(&parent_clinfo->gen_rsc, rsc);
                                                gsc_add_sc(&parent_clinfo->gen_rsc, &old_rsc);
                                                hfsc_clinfo->rsc = old_rsc;
                                        }
                                        if (fsc) {
                                                gsc_sub_sc(&parent_clinfo->gen_fsc, fsc);
                                                gsc_add_sc(&parent_clinfo->gen_fsc, &old_fsc);
                                                hfsc_clinfo->fsc = old_fsc;
                                        }
                                        return (-ENOMEM);
                                }
                                cl->cl_rsc = tmp;
                        }
                        memset(cl->cl_rsc, 0, sizeof(struct internal_sc));
                        sc2isc(rsc, cl->cl_rsc);
                        rtsc_init(&cl->cl_deadline, cl->cl_rsc, 0, 0);
                        rtsc_init(&cl->cl_eligible, cl->cl_rsc, 0, 0);
                }
        }

        if (dsc != NULL) {
                if (dsc->m1 == 0 && dsc->m2 == 0) {
                        if (cl->cl_dsc != NULL) {
                                kfree(cl->cl_dsc);
                                cl->cl_dsc = NULL;
                        }
                } else {
                        if (cl->cl_dsc == NULL) {
                                tmp=kmalloc(sizeof(struct internal_sc), GFP_KERNEL);
                                if (tmp == NULL) {
                                        sch_tree_unlock(sch);
                                        /* modify failed!, restore the old service curves */
                                        if (rsc) {
                                                gsc_sub_sc(&parent_clinfo->gen_rsc, rsc);
                                                gsc_add_sc(&parent_clinfo->gen_rsc, &old_rsc);
                                                hfsc_clinfo->rsc = old_rsc;
```

```
                                        }
                                        if ( fsc ) {
                                                gsc_sub_sc(&parent_clinfo->gen_fsc, fsc);
                                                gsc_add_sc(&parent_clinfo->gen_fsc, &old_fsc);
                                                hfsc_clinfo->fsc = old_fsc;
                                        }
                                        return (-ENOMEM);
                                }
                                cl->cl_dsc = tmp;
                        }
                        memset(cl->cl_dsc, 0, sizeof(struct internal_sc));
                        sc2isc(dsc, cl->cl_dsc);
                        rtsc_init(&cl->cl_dropcurve, cl->cl_dsc, 0, 0);
                }
        }

        if ( fsc != NULL) {
                if ( fsc->m1 == 0 && fsc->m2 == 0) {
                        if ( cl->cl_fsc != NULL) {
                                kfree(cl->cl_fsc);
                                cl->cl_fsc = NULL;
                        }
                } else {
                        if ( cl->cl_fsc == NULL) {
                                tmp=kmalloc(sizeof(struct internal_sc), GFP_KERNEL);
                                if (tmp == NULL) {
                                        sch_tree_unlock(sch);
                                        /* modify failed !, restore the old service curves */
                                        if ( rsc ) {
                                                gsc_sub_sc(&parent_clinfo->gen_rsc, rsc);
                                                gsc_add_sc(&parent_clinfo->gen_rsc, &old_rsc);
                                                hfsc_clinfo->rsc = old_rsc;
                                        }
                                        if ( fsc ) {
                                                gsc_sub_sc(&parent_clinfo->gen_fsc, fsc);
                                                gsc_add_sc(&parent_clinfo->gen_fsc, &old_fsc);
                                                hfsc_clinfo->fsc = old_fsc;
                                        }
                                        return (-ENOMEM);
                                }
                                cl->cl_fsc = tmp;
                        }
                        memset(cl->cl_fsc, 0, sizeof(struct internal_sc));
                        sc2isc(fsc, cl->cl_fsc);
                        rtsc_init(&cl->cl_virtual, cl->cl_fsc, 0, 0);
                }
        }
        sch_tree_unlock(sch);;
        FCT_OUT;
        return (0);
}

/*
 * hfsc_nextclass returns the next class in the tree.
 *   usage:
 *      for ( cl = hif->hif_rootclass; cl != NULL; cl = hfsc_nextclass(cl))
 *              do_something;
 */
static struct hfsc_class *
hfsc_nextclass(struct hfsc_class *cl)
{
        if ( cl->cl_children != NULL)
                cl = cl->cl_children;
        else if ( cl->cl_siblings != NULL)
                cl = cl->cl_siblings;
        else {
                while (( cl = cl->cl_parent) != NULL)
                        if ( cl->cl_siblings) {
                                cl = cl->cl_siblings;
                                break;
                        }
        }

        return ( cl);
}

/*
 * hfsc_enqueue is the enqueue function of the qdisc
 * interface
 */
static int
hfsc_enqueue(struct sk_buff *skb, struct Qdisc *sch)
{
        struct hfsc_sched_data *q = (struct hfsc_sched_data *) sch->data;
```

```
        struct hfsc_class *cl=NULL;
        struct tcf_result res;
        int result;
        unsigned int len = 0;

        FCT_IN;
        /* classify packet */
        if (!q->sched_filter_list ||
            (tc_classify(skb, q->sched_filter_list, &res)) < 0) {
                DEBUG(DEBUG_INFO, "hfsc_enqueue: packet for default class\n");
                cl = q->sched_defaultclass;
        }
        else {
        cl = (struct hfsc_class *) res.class;
        }

        if (!cl && TC_H_MAJ(res.classid))
        {
          cl = hfsc_class_lookup(q, res.classid);
        }

        if (!cl)
        {
         /*
          * packet was apparently classified by filter but we
          * are unable to locate the class - bug in qdisc or filter ?
          */
         DEBUG(DEBUG_WARN, "packet classified but unable to locate class\n");
         cl = q->sched_defaultclass;
        }

        if (!cl)
        {
         /*
          * no valid default class - this can happen if the user
          * deleted the default class and did not provide a new one yet
          */
         DEBUG(DEBUG_WARN, "no default class - packet dropped!\n");
         PKTCNTR_ADD(&cl->cl_stats.drop_cnt, len);
         sch->stats.drops++;
         kfree_skb(skb);
         return(NET_XMIT_DROP);
        }


        len = skb->len;
        if ((result = hfsc_addq(cl, skb)) != 0) {
                /* drop occurred. sk_buff was freed in internal qdisc */
                PKTCNTR_ADD(&cl->cl_stats.drop_cnt, len);
                sch->stats.drops++;
                return (result);
        }
        else {
                /* packet successfully enqueued, update stats */
                sch->q.qlen++;
                sch->stats.packets++;
                sch->stats.bytes+=len;
        }

        q->sched_packets++;

        /* successfully queued. */
        if (cl->cl_qlen == 1)
                set_active(cl, skb->len);

#ifdef HFSC_PKTLOG
        /* put the logging_hook here */
#endif
        FCT_OUT;
        return (0);
}

/*
 * hfsc_dequeue is a dequeue function to be registered to
 * the qdisc interface
 */
static struct sk_buff *
hfsc_dequeue(struct Qdisc *sch)
{
        struct hfsc_sched_data *q = (struct hfsc_sched_data *) sch->data;
        struct hfsc_class *cl;
        struct sk_buff *m = NULL;
        unsigned int len, service_len = 0;
        unsigned int realtime = 0;
```

```
1000        unsigned int dropped = 0;
            u64 current_time = 0;

    #ifdef CONFIG_NET_WIRELESS_SCHED
            struct wsched_channel_monitor *mon=sch->dev->wsched_chmon;
1005    #endif

            FCT_IN;

            /*
             * check if there are any pending transmissions
             * (requeued skbs) which were already processed
             */
            if ( q->sched_requeued_skbs->q.qlen > 0 )
            {
1015                m = q->sched_requeued_skbs->dequeue( q->sched_requeued_skbs );
                    ASSERT( m != NULL );
                    sch->q.qlen--;
                    q->sched_packets--;
                    return m;
1020        }

            /*
             * notify channel monitor of end of previous transmission:
             */
1025
    #ifdef CONFIG_NET_WIRELESS_SCHED
            if ( ( q->sched_flags & HFSC_SCHED_WIRELESS_MON) && (mon) )
                    mon->end_transmit(mon, NULL);
    #endif
1030        /*
             * no pending transmissions, dequeue a new skb:
             */

            current_time = read_machclk();
1035
            do {
                    if (q->sched_packets == 0) {
                            /* no packet in the tree */
                            if ( DFACTOR_ESTIMATION_ACTIVE(q) )
1040                            dfactor_stop_estimate (q);
                            FCT_OUT;
                            return (NULL);
                    }
                    /* selection of next class allowed to tranmit:
1045                 *
                     * if there are eligible classes, use real-time criteria.
                     * find the class with the minimum deadline among
                     * the eligible classes.
                     */
1050                if ((cl = ellist_get_mindl (q->sched_eligible, current_time)) != NULL) {
                            realtime = 1;
                    } else {
                            /*
                             * use link-sharing criteria
1055                         * get the class with the minimum vt in the hierarchy
                             */
                            cl = q->sched_rootclass;
                            while ( is_a_parent_class (cl)) {
                                    cl = actlist_first (cl->cl_actc);
1060                                if ( cl == NULL) {
                                            hfsc_start_watchdog_timer (q);
                                            FCT_OUT;
                                            return (NULL);
                                    }
1065                        }
                    }

                    /*
                     * packet drop scheduling: check if drop deadline has been reached
1070                 * and drop all packets which are over their deadline then start again
                     */
                    if ( ( cl->cl_k != 0) && (cl->cl_k < (cl->cl_delta_t_drop + current_time)))
                            dropped = service_packet_drop (cl, cl->cl_delta_t_drop + current_time);

1075                m = hfsc_pollq(cl);

                    if (m == NULL)
                    {
                     /*
1080                  * class has no valid packets to send
                      * (can happen i.e. if all were dropped by the packet drop server)
                      */
                     ASSERT(cl->cl_qlen == 0);
```

```
                set_passive (cl);
            }
        } while (dropped);

        m = hfsc_getq(cl);

        if (m == NULL){
            DEBUG(DEBUG_WARN, "hfsc_dequeue: bug! no skb although nothing dropped!\n");
            return NULL;
        }

        len = m->len;

        ((struct hfsc_sched_data *)cl->cl_sch->data)->sched_packets--;

        /*
         * update virtual time, work received under real-time criterion
         * and the time-shift between drop curve and real-time curve
         */
#ifdef CONFIG_NET_WIRELESS_SCHED
        if (q->sched_flags & HFSC_SCHED_WIRELESS_MON) {
            update_v(cl, len, dfactor_vt_service_estimate (q, cl, m));
        }
        else
#endif /* CONFIG_NET_WIRELESS_SCHED */
        {
            update_v(cl, len, len);
        }

        /* update delta t drop (difference between time of drop curve/real-time curve) */
        if (cl->cl_dsc != NULL)
        {
            cl->cl_drop_cumul += len;
            cl->cl_delta_t_drop = rtsc_y2x(&cl->cl_dropcurve, cl->cl_drop_cumul) - current_time;
        }
        if (realtime) {
            cl->cl_cumul += dfactor_rt_service_estimate (q, cl, m);

            /*
             * variable rate handling:
             *
             * check if system is in an overload state,
             * advance real-time service counter and
             * start monitoring current rate if necessary
             */
            if ((q->sched_flags & HFSC_SCHED_VAR_RATE) &&
                ((cl->cl_d + (HFSC_MAX_BACKLOG_TIME/2)) < current_time))
            {
                long int advance_cumul;

                if (!DFACTOR_ESTIMATION_ACTIVE(q))
                    dfactor_start_estimate (q);
                advance_cumul = (rtsc_x2y(&cl->cl_deadline, current_time) -
                                 rtsc_x2y(&cl->cl_deadline, cl->cl_d));

                if (((current_time - cl->cl_d) > HFSC_MAX_BACKLOG_TIME) &&
                    (advance_cumul > 0))
                {
                    /*
                     * The session has accumulated more than the max. valid
                     * backlog. This should not happen and is a sign that we
                     * do not reserve the necessary bandwidth which can be
                     * because of inaccurate information from bandwidth estimation
                     * in the channel monitor/the scheduler.
                     * We increase the required bandwidth and correct the backlog.
                     */
                    dfactor_increase (q);
                    cl->cl_cumul += (advance_cumul/(HFSC_MAX_BACKLOG_TIME/
                        HFSC_REDUCE_BACKLOG_TIME));
                    cl->cl_total += (advance_cumul/(HFSC_MAX_BACKLOG_TIME/
                        HFSC_REDUCE_BACKLOG_TIME));
                    DEBUG(DEBUG_INFO, "overload: class %x accumulated backlog of %li time units, "
                        lost %li units\n",
                          cl->cl_id, (long int) (current_time - cl->cl_d), (advance_cumul/(
                              HFSC_MAX_BACKLOG_TIME/HFSC_REDUCE_BACKLOG_TIME)));
                }
            } /* backlog */
        } /* realtime */

        if (DFACTOR_ESTIMATION_ACTIVE(q))
            dfactor_update (q, current_time, len);

        /*
         * update deadline, drop time and eligibile time if necessary
```

```
         */
1165     if (cl->cl_qlen > 0) {

             if ((cl->cl_rsc != NULL) || (cl->cl_dsc != NULL)) {
                 service_len = (hfsc_pollq(cl))->len;
                 len = service_len;
1170             if (DFACTOR_ESTIMATION_ACTIVE(q)){
                     /*
                      * system is in an overload state, reduce service of class
                      * to new_class_service = normal_service *
                      *     (estimated_sync_class_rate / specified_sync_class_rate)
1175                  */
                     service_len = dfactor_rt_service_estimate(q, cl, hfsc_pollq(cl));

                     DEBUG(DEBUG_INFO,
                       "overload: need %i service for %i packet length\n",
1180                     service_len, len);

                 }

                 if (cl->cl_dsc != NULL) {
1185             /* update drop time with original (unstretched) length */
                     update_k(cl, len);
                 }
             }

1190         if (cl->cl_rsc != NULL) {
                 /* update ed */
                 if (realtime)
                     update_ed(cl, service_len);
                 else
1195                 update_d(cl, service_len);
             }

         } else {
             /* the class becomes passive */
1200         set_passive(cl);
         }

#ifdef HFSC_PKTLOG
         /* put the logging_hook here */
1205 #endif

#ifdef CONFIG_NET_WIRELESS_SCHED
         if ((q->sched_flags & HFSC_SCHED_WIRELESS_MON) && (mon)){
             if (m) {
1210             mon->skb2dst(mon, m, &q->cur_dst);
                 mon->start_transmit(mon, &q->cur_dst, m->len);
             }
             else
                 mon->channels_idle(mon);
1215     }
#endif /* CONFIG_NET_WIRELESS_SCHED */

         /*
          * update statistics and remember which
1220      * class is sending in case a requeue event
          * occurs
          */

         PKTCNTR_ADD(&cl->cl_stats.xmit_cnt, m->len);
1225     sch->q.qlen--;

         FCT_OUT;
         return (m);
}
1230
/*
 * add a packet to the internal qdisc of a class
 */
static int
1235 hfsc_addq(struct hfsc_class *cl, struct sk_buff *m)
{
         int result;

         if (cl->cl_q)
1240     {
             result = cl->cl_q->enqueue(m, cl->cl_q);
             if (result == 0)
                 cl->cl_qlen++;
             return(result);
1245     }
         else
             DEBUG(DEBUG_WARN, "cannot equeue since no qdisc for class exists\n");
```

```
        return (0);
}

/*
 * get the next packet from the internal qdisc of a class
 */
static struct sk_buff *
hfsc_getq(struct hfsc_class *cl)
{
struct sk_buff *skb=NULL;

        if ( cl->cl_peeked != NULL ){
          /*
           * There is a skb left from the last peek operation.
           */
          skb = cl->cl_peeked;
          cl->cl_peeked = NULL;
        }
        else {
          if ( cl->cl_q != NULL)
            skb = cl->cl_q->dequeue(cl->cl_q);
        }
        if ( skb != NULL )
        {
          ASSERT(cl->cl_qlen > 0);
          cl->cl_qlen--;
        }
        return skb;
}

/*
 * The pollq function returns the next skb to be dequeued.
 *
 * Since unfortunately the Linux qdisc_ops interface has no
 * way to get information about the next skb without dequeuing it
 * it is dequeued and stored in a temporary pointer until the
 * actual dequeue operation takes place. This is not a good
 * behavior since the interior qdisc sees the dequeue operation
 * too early but there seems to be no other way.
 */
static struct sk_buff *
hfsc_pollq ( struct hfsc_class *cl)
{
        if ( ! cl->cl_peeked )
          if ( cl->cl_q )
            cl->cl_peeked = cl->cl_q->dequeue(cl->cl_q);
        return cl->cl_peeked;
}

/*
 * delete all packets currently stored in a class
 */
static void
hfsc_purgeq(cl)
        struct hfsc_class *cl;
{
        if ( cl->cl_q != NULL )
          qdisc_reset (cl->cl_q);
        if ( cl->cl_qlen > 0 )
        {
          set_passive (cl);
          cl->cl_qlen = 0;
        }
        cl->cl_peeked = NULL;
}

/*
 * change the status of a class to active and initialize
 * the necessary deadline and/or virtual time lists
 */
static void
set_active (struct hfsc_class *cl, int len)
{
        FCT_IN;
        if ( !( cl->cl_flags & HFCF_CLASS_ACTIVE))
        {
                struct hfsc_sched_data *q = (struct hfsc_sched_data *) cl->cl_sch->data;

#ifdef CONFIG_NET_WIRELESS_SCHED
                /*
                 * In the wireless case, update curves taking information
                 * about long-term channel state (g_i) into account
                 */
```

```
                if ( cl->cl_rsc != NULL)
                        init_ed(cl, dfactor_rt_service_estimate(q, cl, hfsc_pollq(cl)));
                if ( cl->cl_fsc != NULL)
                        init_v(cl, dfactor_vt_service_estimate(q, cl, hfsc_pollq(cl)));
#else
                /*
                 * In the wired case the service is equal to the length of the packet
                 * unless the scheduler is in an overload situation. Virtual time is
                 * always based on length.
                 */
                if ( cl->cl_rsc != NULL) {
                        if ( DFACTOR_ESTIMATION_ACTIVE(q) ){
                                init_ed(cl, dfactor_rt_service_estimate(q, cl, hfsc_pollq(cl)));
                        }
                        else {
                                init_ed(cl, len);
                        }
                }
                if ( cl->cl_fsc != NULL)
                        init_v(cl, len);
#endif /* CONFIG_NET_WIRELESS_SCHED */

                /*
                 * deadline curve is updated (regardless of channel state)
                 */
                if ( cl->cl_dsc != NULL)
                        init_k(cl, len);

                /*
                 * add requirements for variable rate over-limit scheduling
                 */
                if ( cl->cl_rsc != NULL)
#ifdef CONFIG_NET_WIRELESS_SCHED
                        dfactor_add_curve( cl->cl_sync_class, &cl->cl_eligible);
#else
                        dfactor_add_curve( cl, &cl->cl_eligible);
#endif /* CONFIG_NET_WIRELESS_SCHED */

                cl->cl_flags |= HFCF_CLASS_ACTIVE;

                cl->cl_stats.period++;
        }
        FCT_OUT;
}

/*
 * change the status of a class to passive and
 * destroy the deadline and/or virtual time lists
 */
static void
set_passive(struct hfsc_class *cl)
{
        FCT_IN;
        if ( cl->cl_flags & HFCF_CLASS_ACTIVE )
        {
                cl->cl_flags ^= HFCF_CLASS_ACTIVE;

                if ( cl->cl_rsc != NULL)
#ifdef CONFIG_NET_WIRELESS_SCHED
                        dfactor_remove_curve( cl->cl_sync_class, &cl->cl_eligible);
#else
                        dfactor_remove_curve( cl, &cl->cl_eligible);
#endif /* CONFIG_NET_WIRELESS_SCHED */

                if ( cl->cl_rsc != NULL)
                        ellist_remove(cl);

                if ( cl->cl_fsc != NULL) {
                        while ( cl->cl_parent != NULL) {
                                if (--cl->cl_nactive == 0) {
                                        /* remove this class from the vt list */
                                        actlist_remove(cl);
                                } else
                                        /* still has active children */
                                        break;

                                /* go up to the parent class */
                                cl = cl->cl_parent;
                        }
                }
        }
        FCT_OUT;
}
```

```
/*
 * initialize eligible list
 */
static void
init_ed(struct hfsc_class *cl, int next_len)
{
        u64 cur_time;

        FCT_IN;
        cur_time = read_machclk();

        /* update the deadline curve */
        rtsc_min(&cl->cl_deadline, cl->cl_rsc, cur_time, cl->cl_cumul);

        /*
         * update the eligible curve.
         * for concave, it is equal to the deadline curve.
         * for convex, it is a linear curve with slope m2.
         */
        cl->cl_eligible = cl->cl_deadline;
        if (cl->cl_rsc->sm1 <= cl->cl_rsc->sm2) {
                cl->cl_eligible.dx = 0;
                cl->cl_eligible.dy = 0;
        }

        /* compute e and d */
        cl->cl_e = rtsc_y2x(&cl->cl_eligible, cl->cl_cumul);
        cl->cl_d = rtsc_y2x(&cl->cl_deadline, cl->cl_cumul + next_len);

        ellist_insert(cl);
        FCT_OUT;
}

/*
 * initialize the packet drop curve
 *
 * It's x-scale is based on the total amount of all processed data
 * for this class (including all dropped data).
 */
static void
init_k(struct hfsc_class *cl, int next_len)
{
        u64 cur_time;

        FCT_IN;
        cur_time = read_machclk();

        /* update the packet drop curve */
        rtsc_min(&cl->cl_dropcurve, cl->cl_dsc, cur_time + cl->cl_delta_t_drop, cl->cl_drop_cumul);

        /* compute drop time */
        cl->cl_k = rtsc_y2x(&cl->cl_dropcurve, cl->cl_drop_cumul + next_len);

        FCT_OUT;
}


/*
 * update eligibile time and deadline
 */
static void
update_ed(struct hfsc_class *cl, int next_len)
{
        cl->cl_e = rtsc_y2x(&cl->cl_eligible, cl->cl_cumul);
        cl->cl_d = rtsc_y2x(&cl->cl_deadline, cl->cl_cumul + next_len);

        ellist_update(cl);
}

/*
 * update deadline
 */
static void
update_d(struct hfsc_class *cl, int next_len)
{
        cl->cl_d = rtsc_y2x(&cl->cl_deadline, cl->cl_cumul + next_len);
}

/*
 * update packet drop time
 */
static void
update_k(struct hfsc_class *cl, int next_len)
{
```

```
1500            cl->cl_k = rtsc_y2x(&cl->cl_dropcurve, cl->cl_drop_cumul + next_len);
        }

        /*
         * initialze virtual time
1505     */
        static void
        init_v(struct hfsc_class *cl, int len)
        {
                struct hfsc_class *min_cl, *max_cl;
1510
                FCT_IN;
                while (cl->cl_parent != NULL) {

                        if (cl->cl_nactive++ > 0)
1515                            /* already active */
                                break;

                        min_cl = actlist_first(cl->cl_parent->cl_actc);
                        if (min_cl != NULL) {
1520                            u64 vt;

                                /*
                                 * set vt to the average of the min and max classes.
                                 * if the parent's period didn't change,
1525                             * don't decrease vt of the class.
                                 */
                                max_cl = actlist_last(cl->cl_parent->cl_actc);
                                vt = (min_cl->cl_vt + max_cl->cl_vt) / 2;
                                if (cl->cl_parent->cl_vtperiod == cl->cl_parentperiod)
1530                                    vt = max(cl->cl_vt, vt);
                                cl->cl_vt = vt;
                        } else {
                                /* no packet is backlogged. set vt to 0 */
                                cl->cl_vt = 0;
1535                    }

                        /* update the virtual curve */
                        rtsc_min(&cl->cl_virtual, cl->cl_fsc,
                                 cl->cl_vt, cl->cl_total);
1540
                        cl->cl_vtperiod++;  /* increment vt period */
                        cl->cl_parentperiod = cl->cl_parent->cl_vtperiod;
                        if (cl->cl_parent->cl_nactive == 0)
                                cl->cl_parentperiod++;
1545
                        actlist_insert(cl);

                        /* go up to the parent class */
                        cl = cl->cl_parent;
1550            }
                FCT_OUT;
        }

        /*
1555     * update virtual time of class
         */
        static void
        update_v(struct hfsc_class *cl, int len, int service)
        {
1560            int increase_vt = len;

                FCT_IN;

                while (cl->cl_parent != NULL) {
1565    #ifdef CONFIG_NET_WIRELESS_SCHED
                        if ((cl->cl_flags & HFCF_SYNCCLASS) && (len<service))
                        {
                                /*
                                 * This is a wireless synchronization class and
1570                             * the virtual time in the subtree below is in goodput
                                 * and the virtual time of the parent/this class is
                                 * in units of raw throughput (resource consumption)
                                 * on the wireless link -- therefore we switch to
                                 * handing up the needed raw service amount.
1575                             */
                                increase_vt = service;
                        }
        #endif
                        cl->cl_total += increase_vt;
1580
                        if (cl->cl_fsc != NULL) {
                                cl->cl_vt = rtsc_y2x(&cl->cl_virtual, cl->cl_total);
```

```
                        /* update the vt list */
                        actlist_update (cl);
                }

                /* go up to the parent class */
                cl = cl->cl_parent;
        }
        FCT_OUT;
}


/*
 * TAILQ based ellist and actlist implementation
 * (ion wanted to make a calendar queue based implementation)
 */
/*
 * eligible list holds backlogged classes being sorted by their eligible times.
 * there is one eligible list per interface.
 */

static ellist_t *
ellist_alloc ()
{
        ellist_t *head;

        FCT_IN;

        head=kmalloc(sizeof( ellist_t ), GFP_KERNEL);
        if (head)
                memset(head, 0, sizeof( ellist_t ));
        TAILQ_INIT(head);

        FCT_OUT;

        return (head);
}

static void
ellist_destroy ( ellist_t *head)
{
        /* @@@ destroy whole list or this entry only? check ... @@@ */
        kfree(head);
}

static void
ellist_insert (struct hfsc_class *cl)
{
        struct hfsc_class *p;
        struct hfsc_sched_data *q = (struct hfsc_sched_data *) cl->cl_sch->data;

        /* check the last entry first */
        if ((p = TAILQ_LAST(q->sched_eligible, _eligible)) == NULL ||
            p->cl_e <= cl->cl_e) {
                TAILQ_INSERT_TAIL(q->sched_eligible, cl, cl_ellist);
                return;
        }

        TAILQ_FOREACH(p, q->sched_eligible, cl_ellist) {
                if (cl->cl_e < p->cl_e) {
                        TAILQ_INSERT_BEFORE(p, cl, cl_ellist);
                        return;
                }
        }
        ASSERT(0); /* should not reach here */
}

static void
ellist_remove (struct hfsc_class *cl)
{
        struct hfsc_sched_data *q = (struct hfsc_sched_data *) cl->cl_sch->data;

        TAILQ_REMOVE(q->sched_eligible, cl, cl_ellist);
}

static void
ellist_update (struct hfsc_class *cl)
{
        struct hfsc_sched_data *q = (struct hfsc_sched_data *) cl->cl_sch->data;
        struct hfsc_class *p, *last;

        /*
         * the eligible time of a class increases monotonically.
         * if the next entry has a larger eligible time, nothing to do.
```

```
                 */
                p = TAILQ_NEXT(cl, cl_ellist);
1670            if (p == NULL || cl->cl_e <= p->cl_e)
                        return;

                /* check the last entry */
                last = TAILQ_LAST(q->sched_eligible, _eligible);
1675            ASSERT(last != NULL);
                if (last->cl_e <= cl->cl_e) {
                        TAILQ_REMOVE(q->sched_eligible, cl, cl_ellist);
                        TAILQ_INSERT_TAIL(q->sched_eligible, cl, cl_ellist);
                        return;
1680            }

                /*
                 * the new position must be between the next entry
                 * and the last entry
1685             */
                while ((p = TAILQ_NEXT(p, cl_ellist)) != NULL) {
                        if (cl->cl_e < p->cl_e) {
                                TAILQ_REMOVE(q->sched_eligible, cl, cl_ellist);
                                TAILQ_INSERT_BEFORE(p, cl, cl_ellist);
1690                            return;
                        }
                }
                ASSERT(0); /* should not reach here */
        }
1695
        /*
         * find the class with the minimum deadline among the eligible classes
         */
        struct hfsc_class *
1700    ellist_get_mindl(ellist_t *head, u64 cur_time)
        {
                struct hfsc_class *p, *cl = NULL;

                TAILQ_FOREACH(p, head, cl_ellist) {
1705                    if (p->cl_e > cur_time)
                                break;
                        if (cl == NULL || p->cl_d < cl->cl_d)
                                cl = p;
                }
1710            return (cl);
        }

        /*
         * active children list holds backlogged child classes being sorted
1715     * by their virtual time.
         * each intermediate class has one active children list.
         */
        static actlist_t *
        actlist_alloc()
1720    {
                actlist_t *head;

                head=kmalloc(sizeof(actlist_t), GFP_KERNEL);
                TAILQ_INIT(head);
1725            return (head);
        }

        static void
        actlist_destroy(head)
1730            actlist_t *head;
        {
                kfree(head);
        }

1735    static void
        actlist_insert(struct hfsc_class *cl)
        {
                struct hfsc_class *p;

1740            /* check the last entry first */
                if ((p = TAILQ_LAST(cl->cl_parent->cl_actc, _active)) == NULL
                    || p->cl_vt <= cl->cl_vt) {
                        TAILQ_INSERT_TAIL(cl->cl_parent->cl_actc, cl, cl_actlist);
                        return;
1745            }

                TAILQ_FOREACH(p, cl->cl_parent->cl_actc, cl_actlist) {
                        if (cl->cl_vt < p->cl_vt) {
                                TAILQ_INSERT_BEFORE(p, cl, cl_actlist);
1750                            return;
                        }
```

```
                }
                ASSERT(0); /* should not reach here */
        }

        static void
        actlist_remove (struct hfsc_class *cl)
        {
                TAILQ_REMOVE(cl->cl_parent->cl_actc, cl, cl_actlist);
        }

        static void
        actlist_update (struct hfsc_class *cl)
        {
                struct hfsc_class *p, *last;

                /*
                 * the virtual time of a class increases monotonically during its
                 * backlogged period.
                 * if the next entry has a larger virtual time, nothing to do.
                 */
                p = TAILQ_NEXT(cl, cl_actlist);
                if (p == NULL || cl->cl_vt <= p->cl_vt)
                        return;

                /* check the last entry */
                last = TAILQ_LAST(cl->cl_parent->cl_actc, _active);
                ASSERT(last != NULL);
                if (last->cl_vt <= cl->cl_vt) {
                        TAILQ_REMOVE(cl->cl_parent->cl_actc, cl, cl_actlist);
                        TAILQ_INSERT_TAIL(cl->cl_parent->cl_actc, cl, cl_actlist);
                        return;
                }

                /*
                 * the new position must be between the next entry
                 * and the last entry
                 */
                while ((p = TAILQ_NEXT(p, cl_actlist)) != NULL) {
                        if (cl->cl_vt < p->cl_vt) {
                                TAILQ_REMOVE(cl->cl_parent->cl_actc, cl, cl_actlist);
                                TAILQ_INSERT_BEFORE(p, cl, cl_actlist);
                                return;
                        }
                }
                ASSERT(0); /* should not reach here */
        }

        /*
         * service curve support functions
         *
         *  external service curve parameters
         *      m: bits/sec
         *      d: msec
         *  internal service curve parameters
         *      sm: (bytes/tsc_interval) << SM_SHIFT
         *      ism: (tsc_count/byte) << ISM_SHIFT
         *      dx: tsc_count
         *
         * SM_SHIFT and ISM_SHIFT are scaled in order to keep effective digits.
         * we should be able to handle 100K-1Gbps linkspeed with 200Hz-1GHz CPU
         * speed.  SM_SHIFT and ISM_SHIFT are selected to have at least 3 effective
         * digits in decimal using the following table.
         *
         *  bits/set       100Kbps      1Mbps      10Mbps     100Mbps    1Gbps
         *  ----------+-------------------------------------------------------------
         *  bytes/nsec   12.5e-6    125e-6     1250e-6    12500e-6   125000e-6
         *  sm(500MHz) 25.0e-6    250e-6     2500e-6    25000e-6   250000e-6
         *  sm(200MHz) 62.5e-6    625e-6     6250e-6    62500e-6   625000e-6
         *
         *  nsec/byte    80000       8000       800        80         8
         *  ism(500MHz) 40000       4000       400        40         4
         *  ism(200MHz) 16000       1600       160        16         1.6
         */
        #define SM_SHIFT        24
        #define ISM_SHIFT       10

        #define SC_LARGEVAL (1LL << 32)
        #define SC_INFINITY     0   xffffffffffffffffLL

        static inline u64
        seg_x2y(x, sm)
                u64 x;
                u64 sm;
        {
```

```
        u64 y;

        if (x < SC_LARGEVAL)
                y = x * sm >> SM_SHIFT;
        else
                y = (x >> SM_SHIFT) * sm;
        return (y);
}

static inline u64
seg_y2x(y, ism)
        u64 y;
        u64 ism;
{
        u64 x;

        if (y == 0)
                x = 0;
        else if (ism == SC_INFINITY)
                x = SC_INFINITY;
        else if (y < SC_LARGEVAL)
                x = y * ism >> ISM_SHIFT;
        else
                x = (y >> ISM_SHIFT) * ism;
        return (x);
}

static inline u64
m2sm(m)
        unsigned int m;
{
        u64 sm;

        /*
         * approximate
         * m << SM_SHIFT) / 8 / machclk_freq
         * by using shift operations only (IA-32 does not have 64-bit division)
         */
        sm = ((u64)m + ((u64) m>>5)) << (SM_SHIFT - (machclk_freq_log+3));
        return (sm);
}

static inline u64
m2ism(m)
        unsigned int m;
{
        u64 ism;

        if (m == 0)
                ism = SC_INFINITY;
        else {
                /* calculate ism = ((u64)machclk_freq << ISM_SHIFT) * 8 / m;
                 *
                 * this is very problematic since we have no u64 division on IA-32 unless
                 * we link the code with the gcc lib. In order to avoid this, we do the
                 * division in 32-bit - this should be ok for up to 100MBit
                 * (another solution might be to include the 64 bit division code)
                 *
                 * machclk_freq << 10 = 1024000000      less than 2^32 = 4294967296
                 */
                ism = ((u64) (((u32)(machclk_freq << ISM_SHIFT)) / (u32) m)) << 3;
        }
        return (ism);
}

static inline u64
d2dx(d)
        unsigned int      d;
{
        u64 dx;

        dx = ((u32)d * (machclk_freq / 1000));
        return (dx);
}

static unsigned int
sm2m(sm)
        u64 sm;
{
        u64 m;

        m = (sm * 8 * machclk_freq) >> SM_SHIFT;
        return ((unsigned int)m);
}
```

```
1920
 static unsigned int
dx2d(dx)
        u64 dx;
{
1925    u64 d;

        d = (u32) (dx * 1000) / (u32) machclk_freq;
        return ((unsigned int)d);
}
1930
 static void
sc2isc(sc, isc)
        struct service_curve    *sc;
        struct internal_sc      *isc;
1935 {
        isc->sm1 = m2sm(sc->m1);
        isc->ism1 = m2ism(sc->m1);
        isc->dx = d2dx(sc->d);
        isc->dy = seg_x2y(isc->dx, isc->sm1);
1940    isc->sm2 = m2sm(sc->m2);
        isc->ism2 = m2ism(sc->m2);
}


1945 /*
 * initialize the runtime service curve with the given internal
 * service curve starting at (x, y).
 */
 static void
1950 rtsc_init(rtsc, isc, x, y)
        struct runtime_sc       *rtsc;
        struct internal_sc      *isc;
        u64                     x, y;
{
1955    rtsc->x =       x;
        rtsc->y =       y;
        rtsc->sm1 =     isc->sm1;
        rtsc->ism1 =    isc->ism1;
        rtsc->dx =      isc->dx;
1960    rtsc->dy =      isc->dy;
        rtsc->sm2 =     isc->sm2;
        rtsc->ism2 =    isc->ism2;
}

1965 /*
 * calculate the y-projection of the runtime service curve by the
 * given x-projection value
 */
 static u64
1970 rtsc_y2x(rtsc, y)
        struct runtime_sc       *rtsc;
        u64                     y;
{
        u64     x;
1975
        if (y < rtsc->y)
                x = rtsc->x;
        else if (y <= rtsc->y + rtsc->dy) {
                /* x belongs to the 1st segment */
1980            if (rtsc->dy == 0)
                        x = rtsc->x + rtsc->dx;
                else
                        x = rtsc->x + seg_y2x(y - rtsc->y, rtsc->ism1);
        } else {
1985            /* x belongs to the 2nd segment */
                x = rtsc->x + rtsc->dx
                      + seg_y2x(y - rtsc->y - rtsc->dy, rtsc->ism2);
        }
        return (x);
1990 }

 static u64
rtsc_x2y(rtsc, x)
        struct runtime_sc       *rtsc;
1995    u64                     x;
{
        u64     y;

        if (x <= rtsc->x)
2000            y = rtsc->y;
        else if (x <= rtsc->x + rtsc->dx)
                /* y belongs to the 1st segment */
                y = rtsc->y + seg_x2y(x - rtsc->x, rtsc->sm1);
```

```
        else
                /* y belongs to the 2nd segment */
                y = rtsc->y + rtsc->dy
                    + seg_x2y(x - rtsc->x - rtsc->dx, rtsc->sm2);
        return (y);
}

/*
 * update the runtime service curve by taking the minimum of the current
 * runtime service curve and the service curve starting at (x, y).
 */
static void
rtsc_min(rtsc, isc, x, y)
        struct runtime_sc       *rtsc;
        struct internal_sc      *isc;
        u64                     x, y;
{
        u64     y1, y2, dx, dy;

        if (isc->sm1 <= isc->sm2) {
                /* service curve is convex */
                y1 = rtsc_x2y(rtsc, x);
                if (y1 < y)
                        /* the current rtsc is smaller */
                        return;
                rtsc->x = x;
                rtsc->y = y;
                return;
        }

        /*
         * service curve is concave
         * compute the two y values of the current rtsc
         *      y1: at x
         *      y2: at (x + dx)
         */
        y1 = rtsc_x2y(rtsc, x);
        if (y1 <= y) {
                /* rtsc is below isc, no change to rtsc */
                return;
        }

        y2 = rtsc_x2y(rtsc, x + isc->dx);
        if (y2 >= y + isc->dy) {
                /* rtsc is above isc, replace rtsc by isc */
                rtsc->x = x;
                rtsc->y = y;
                rtsc->dx = isc->dx;
                rtsc->dy = isc->dy;
                return;
        }

        /*
         * the two curves intersect
         * compute the offsets (dx, dy) using the reverse
         * function of seg_x2y()
         *      seg_x2y(dx, sm1) == seg_x2y(dx, sm2) + (y1 - y)
         */
        dx = ((u32)(((u32)(y1 - y)) << (SM_SHIFT/2))) / ((u32)(isc->sm1 - isc->sm2));
        dx = dx << (SM_SHIFT/2);
        /*
         * check if (x, y1) belongs to the 1st segment of rtsc.
         * if so, add the offset.
         */
        if (rtsc->x + rtsc->dx > x)
                dx += rtsc->x + rtsc->dx - x;
        dy = seg_x2y(dx, isc->sm1);

        rtsc->x = x;
        rtsc->y = y;
        rtsc->dx = dx;
        rtsc->dy = dy;
        return;
}

/*
 * root service degrade factor calculation
 */

#define DFACTOR_SHIFT 15

/*
 * for variable rate links: estimate degrade factor of sc of root class
 *
```

```
 * (over estimation interval of 2^( sched_est_interval_log ) transmitted bytes)
 */
static void dfactor_update ( struct hfsc_sched_data *q, u64 current_time , unsigned long length )
{
        u32         delta_t =0;
        u32         delta_s =0;
        u32         new_dfactor;

        FCT_IN;
        if ( q->sched_est_length == 0 ) {
                /*
                 * new estimation started , initially we assume
                 * factor 1, if we have been estimating before
                 * and this is not too long ago, reuse old estimate
                 */
                if ( current_time - q->sched_est_time > (100000))
                        q->sched_est_dfactor = (1 << DFACTOR_SHIFT);

        } else {
                /*
                 * update degrade factor estimate:
                 *
                 * We estimate the current service degrade factor by
                 * calculating the slope of the piece of the current
                 * service curve which the system experiences.
                 * The the amount of service provided is divided by
                 * the amount of service needed to satisfy the sum
                 * of all eligibility curves. The inverse of this
                 * ratio is the degrade factor, which is kept as a
                 * running average.
                 *                              service required in delta_t
                 * degrade factor =   ------------------------------------
                 *                              service received in delta_t
                 *
                 * Later all service curves are reduced by the
                 * degrade factor, by increasing packet deadlines.
                 *
                 * Since no floating point calculations should be done
                 * in the kernel, the factor is shifted and stored as
                 * integer. A degrade factor lower than one is an
                 * indication that we receive more service than needed
                 * and can stop monitoring.
                 */
                delta_t = current_time - q->sched_est_time;
                delta_s = rtsc_x2y(&q->sched_est_el_sum, current_time)
                        - rtsc_x2y(&q->sched_est_el_sum, q->sched_est_time);

                new_dfactor = (( u32)( delta_s << DFACTOR_SHIFT)) / (u32)q->sched_est_length;

                if ( delta_s < (1 << q->sched_est_interval_log )) {
                        q->sched_est_dfactor = q->sched_est_dfactor -
                        ((q->sched_est_length * q->sched_est_dfactor ) >> q->sched_est_interval_log );
                        q->sched_est_dfactor = q->sched_est_dfactor +
                        ((q->sched_est_length * new_dfactor) >> q->sched_est_interval_log );
                } else {
                        q->sched_est_dfactor = new_dfactor;
                }

                if ( q->sched_est_dfactor < (1 << DFACTOR_SHIFT))
                {
                        DEBUG( DEBUG_INFO, "dfactor less than 1 - no overload ?\n" );
                        /*
                         * we have left the overload state , stop estimation
                         */
                        // @@@ dfactor_stop_estimate(q);
                        // currently we only stop estimation if scheduler becomes idle
                        // -> might steal some performace, but placing the stop call
                        // here still needs some testing !
                }
        }

        q->sched_est_time = current_time;
        q->sched_est_length = length;

        FCT_OUT;
}

/*
 * estimate the real-time service needed to send the packet
 * on the desired channel
 *
 * the estimate depends on:
 *
 *          a) current rate of total reduced service
```

```
 *          b) in the wireless case: the current goodput rate on this channel
 */

static unsigned long dfactor_rt_service_estimate ( struct hfsc_sched_data *q,
        struct hfsc_class *cl, struct sk_buff * skb )
{
        unsigned long service=0;
        unsigned long length = skb->len;

#ifdef CONFIG_NET_WIRELESS_SCHED
        u32 capacity = 0;
        struct wsched_channel_monitor *mon=cl->cl_sch->dev->wsched_chmon;

        FCT_IN;
        if ( ( q->sched_flags & HFSC_SCHED_WIRELESS_MON) && (mon) ){
                mon->skb2dst(mon, skb, &q->cur_dst);

                capacity = mon->get_capacity(mon, &q->cur_dst);

                DEBUG( DEBUG_INFO, "wchmon returned channel capacity of %u byte/s\n", capacity );

                /*
                 * estimate required service taking into account:
                 * a) relative channel quality
                 * b) reduction of rate by decrease factor
                 */
                if ( capacity ) {
                        service = (( unsigned long)( length * cl->cl_sync_class->cl_sync_required_cap))/( capacity );
                        service = (( unsigned long)( service * q->sched_est_reducebad))/100;
                }
                else
                        service = (( unsigned long)( length * q->sched_est_reducebad))/100;

                if ( q->sched_est_reducebad < 100 )
                        service += (( u32)(( length * q->sched_est_dfactor *
                                (100 - q->sched_est_reducebad))>> DFACTOR_SHIFT))/(u32)100;
        }
        else
#else
        FCT_IN;
#endif /* CONFIG_NET_WIRELESS_SCHED */

        {

                /*
                 * calculate and return currently needed service to process packet
                 */
                service = ( length * q->sched_est_dfactor) >> DFACTOR_SHIFT;
        }

        if ( service < length )
                service = length;

        DEBUG( DEBUG_INFO, "RT Service %lu for packet length %lu.\n", service, length );
        FCT_OUT;
        return service;
}

#ifdef CONFIG_NET_WIRELESS_SCHED
/*
 * estimate the virtual-time service needed to send the packet
 * on the desired channel
 *
 */

static unsigned long dfactor_vt_service_estimate ( struct hfsc_sched_data *q,
        struct hfsc_class *cl, struct sk_buff * skb )
{
        unsigned long service=0;
        unsigned long length = skb->len;
        u32 capacity=0;
        struct wsched_channel_monitor *mon=cl->cl_sch->dev->wsched_chmon;

        FCT_IN;
        if ( ( q->sched_flags & HFSC_SCHED_WIRELESS_MON) && (mon) ){
                mon->skb2dst(mon, skb, &q->cur_dst);

                capacity = mon->get_capacity(mon, &q->cur_dst);

                DEBUG( DEBUG_INFO, "wchmon returned channel capacity of %u byte/s\n", capacity );

                /*
                 * raw bandwidth vt service is length/g_i = length * ( raw_bandwidth_of_root/capacity_of_channel )
                 * The raw bandwidth service is weighted according to the "reduce bad channels" configuration
```

```
                 * parameter of the scheduler.
                 */

                if ( capacity ) {
                        service = (( unsigned long)( length * (( q->sched_rootclass->cl_clinfo->rsc.m2)»3)) )/( capacity );
                        service = (( unsigned long)( service * q->sched_est_reducebad))/100;
                        if ( q->sched_est_reducebad < 100 )
                                service += (( u32)( length * (100 - q->sched_est_reducebad)))/(u32)100;
                }
                else
                        service = length;
        }

        if ( service < length )
                service = length;

        DEBUG( DEBUG_INFO, "VT Service %lu for packet length %lu.\n", service, length );
        FCT_OUT;
        return service;
}
#endif /* CONFIG_NET_WIRELESS_SCHED */

/*
 * start estimation of rootclass service
 */
static void dfactor_start_estimate ( struct hfsc_sched_data *q )
{
        FCT_IN;
        q->sched_est_dfactor      = (1 « DFACTOR_SHIFT);

        q->sched_est_length       = 0;
        q->sched_est_active       = HFSC_SCHED_EST_ACTIVE;
        FCT_OUT;
}

/*
 * stop estimation of rootclass service
 */
static void dfactor_stop_estimate ( struct hfsc_sched_data *q )
{
        FCT_IN;
        q->sched_est_dfactor      = (1 « DFACTOR_SHIFT);

        q->sched_est_length       = 0;
        q->sched_est_active       = HFSC_SCHED_EST_NOT_ACTIVE;
        FCT_OUT;
}

/*
 * adaptive increase of service degrading
 *
 * This function is called whenever the scheduler recognizes it it
 * is unable to keep the deadlines for packets. This can be caused by
 * inaccurate estimates of the total bandwidth or - in the wireless case -
 * partial bandwidth towards a destination.
 *
 * We increase the required bandwidth by about 1% ( arbitrary number)
 * in order to compensate the too high estimates of the available bandwidth.
 */
static void dfactor_increase ( struct hfsc_sched_data *q )
{
        FCT_IN;

        q->sched_est_dfactor         += q->sched_est_dfactor »7;
        q->sched_est_required_cap += q->sched_est_required_cap »7;
        /*
         * in order to protect us from to optimistic estimates
         * of the channel monitor, the required capacity of the
         * synchronization class can also be increased here
         *
         * But, of course, it is better to fix the channel monitor,
         * therefore the code is not active right now.
         *
         sync_cl->cl_sync_required_cap  += sync_cl->cl_sync_required_cap »7;

         * @@@ make estimation information permanent???
           q->sched_est_el_sum.sm2          += q->sched_est_el_sum.sm2»4;
           sync_cl->cl_sync_el_sum.sm2    += sync_cl->cl_sync_el_sum.sm2»4; */
        FCT_OUT;
}


/*
 * add an eligible curve to a wireless synchronization class
```

```
2340  */
      static inline void dfactor_add_curve ( struct hfsc_class *cl , struct runtime_sc *curve )
      {
              struct hfsc_sched_data *q = ( struct hfsc_sched_data *) (cl->cl_sch->data);

2345  #ifdef CONFIG_NET_WIRELESS_SCHED
              #ifndef USE_64BIT_DIVISIONS
              u64 share = 0;
              #endif
              FCT_IN;
2350          ASSERT( cl->cl_flags & HFCF_SYNCCLASS );
              /*
               * since we only perform x2y translations on this curve and
               * it goes through the origin , we only add the neccesary data
               * fields in order to save some time, all the others are always zero
2355           */
              if ( curve->sm1 > curve->sm2 ){
                      cl->cl_sync_el_sum.sm2 += curve->sm1;
              } else {
                      cl->cl_sync_el_sum.sm2 += curve->sm2;
2360          }
              /*
               * update estimate of avg. needed capacity (in byte/s)
               */
              cl->cl_sync_required_cap = rtsc_x2y(&cl->cl_sync_el_sum, 1000000)
2365                                           - rtsc_x2y(&cl->cl_sync_el_sum, 0);

      #ifdef USE_64BIT_DIVISIONS
              /*
               * in simulation environment we can use 64 bit divisions :
2370           */

              cl->cl_sync_required_cap *= ((q->sched_rootclass->cl_clinfo->rsc.m2)»3);

              ASSERT(cl->cl_clinfo->rsc.m2 || cl->cl_clinfo->rsc.m1);
2375
              if (cl->cl_clinfo->rsc.m2 > cl->cl_clinfo->rsc.m1) {
                      cl->cl_sync_required_cap /= (cl->cl_clinfo->rsc.m2)»3;
              }
              else
2380                  cl->cl_sync_required_cap /= (cl->cl_clinfo->rsc.m1)»3;
        #else
              /* avoid u64 division if not in simulation environment:
               *
               * since required capacity is measured in byte/s
2385           * the following work-around is ok for up to 268 Mbit/s wireless links
               *
               *    required rate is
               *
               *    total rate « 7
2390           * ( ---------------------- * sync_el_sum(0,1000) )» 7
               *    sync class rate
               *
               * note: since rsc is in bit/s, it has to be divided by 8
               */
2395          share = (( q->sched_rootclass->cl_clinfo->rsc.m2) «4 ) ;

              if (cl->cl_clinfo->rsc.m2 > cl->cl_clinfo->rsc.m1) {
                      share = (unsigned long) share /
                              (((unsigned long int)(cl->cl_clinfo->rsc.m2)»3)+1);
2400          } else {
                      share = (unsigned long) share /
                              (((unsigned long int)(cl->cl_clinfo->rsc.m1)»3)+1);
              }

2405          cl->cl_sync_required_cap    = share * ( rtsc_x2y(&cl->cl_sync_el_sum, 1000000)
                                                  - rtsc_x2y(&cl->cl_sync_el_sum, 0));

              /*
               * undo shifting
2410           */
              cl->cl_sync_required_cap    = cl->cl_sync_required_cap»7;

        #endif /* USE_64BIT_DIVISIONS */
      #else /* CONFIG_NET_WIRELESS_SCHED */
2415          FCT_IN;
      #endif /* CONFIG_NET_WIRELESS_SCHED */

              if ( curve->sm1 > curve->sm2 ){
                      q->sched_est_el_sum.sm2 += curve->sm1;
2420          } else {
                      q->sched_est_el_sum.sm2 += curve->sm2;
              }
```

```
        q->sched_est_required_cap = rtsc_x2y(&q->sched_est_el_sum, 1000000)
2425                                  - rtsc_x2y(&q->sched_est_el_sum, 0);

        FCT_OUT;
}

2430 static inline void dfactor_remove_curve( struct hfsc_class *cl, struct runtime_sc *curve )
{
        struct hfsc_sched_data *q = (struct hfsc_sched_data *)(cl->cl_sch->data);

#ifdef CONFIG_NET_WIRELESS_SCHED
2435    #ifndef USE_64BIT_DIVISIONS
        u64 share = 0;
        #endif
        FCT_IN;
        ASSERT( (curve->sm1 <= cl->cl_sync_el_sum.sm2) && (curve->sm2 <= cl->cl_sync_el_sum.sm2) );
2440    if ( curve->sm1 > curve->sm2 ){
                cl->cl_sync_el_sum.sm2 -= curve->sm1;
        } else {
                cl->cl_sync_el_sum.sm2 -= curve->sm2;
        }
2445
        /*
         * update estimate of avg. needed capacity (in byte/s)
         */
        cl->cl_sync_required_cap = rtsc_x2y(&cl->cl_sync_el_sum, 1000000)
2450                                  - rtsc_x2y(&cl->cl_sync_el_sum, 0);

#ifdef USE_64BIT_DIVISIONS
        /*
         * in simulation environment we can use 64 bit divisions :
2455     */

        cl->cl_sync_required_cap *= ((q->sched_rootclass->cl_clinfo->rsc.m2)»3);

        ASSERT(cl->cl_clinfo->rsc.m2 || cl->cl_clinfo->rsc.m1);
2460
        if (cl->cl_clinfo->rsc.m2 > cl->cl_clinfo->rsc.m1) {
                cl->cl_sync_required_cap /= (cl->cl_clinfo->rsc.m2)»3;
        }
        else
2465            cl->cl_sync_required_cap /= (cl->cl_clinfo->rsc.m1)»3;
  #else
        /* avoid u64 division if not in simulation environment:
         *
         * since required capacity is measured in byte/s
2470     * the following work-around is ok for up to 268 Mbit/s wireless links
         *
         *    required rate is
         *
         *    total rate « 7
2475     * ( ---------------- * sync_el_sum(0,1000) )» 7
         *    sync class rate
         *
         * note: since rsc is in bit/s, it has to be divided by 8
         */
2480    share = (( q->sched_rootclass->cl_clinfo->rsc.m2)«4 );

        if (cl->cl_clinfo->rsc.m2 > cl->cl_clinfo->rsc.m1) {
                share = (unsigned long) share /
                        (((unsigned long int)(cl->cl_clinfo->rsc.m2)»3)+1);
2485    } else {
                share = (unsigned long) share /
                        (((unsigned long int)(cl->cl_clinfo->rsc.m1)»3)+1);
        }

2490    cl->cl_sync_required_cap    = share * ( rtsc_x2y(&cl->cl_sync_el_sum, 1000000)
                                      - rtsc_x2y(&cl->cl_sync_el_sum, 0));

        /*
         * undo shifting
2495     */
        cl->cl_sync_required_cap    = cl->cl_sync_required_cap»7;

  #endif /* USE_64BIT_DIVISIONS */
#else /* CONFIG_NET_WIRELESS_SCHED */
2500    FCT_IN;
#endif /* CONFIG_NET_WIRELESS_SCHED */

        if ( curve->sm1 > curve->sm2 ){
                q->sched_est_el_sum.sm2 -= curve->sm1;
2505    } else {
                q->sched_est_el_sum.sm2 -= curve->sm2;
        }
```

```
        q->sched_est_required_cap = rtsc_x2y(&q->sched_est_el_sum, 1000000)
2510                                  - rtsc_x2y(&q->sched_est_el_sum, 0);

        FCT_OUT;
}

2515 /* hfsc_start_watchdog_timer
 *
 * setting up a watchdog timer in order to wake up the qdisc when the
 * next packet is elligible to be sent
 *
2520 */

static inline void hfsc_start_watchdog_timer ( struct hfsc_sched_data *q )

{
2525        struct hfsc_class *cl;
        u64 current_time;

        FCT_IN;
        current_time = read_machclk();
2530
        cl = TAILQ_FIRST(q->sched_eligible);

        if ( cl && !netif_queue_stopped(q->sched_rootclass->cl_sch->dev)) {
                long delay;
2535
                if ( cl->cl_d < current_time ) {
                        DEBUG(DEBUG_WARN, "hfsc_start_watchdog_timer: eligible packet but nothing sent!\n");
                        delay = 1;
                } else {
2540                    delay = PSCHED_US2JIFFIE((psched_tdiff_t)(cl->cl_d - current_time));
                }
                del_timer(&q->wd_timer);
                if ( delay <= 0)
                        delay = 1;
2545            q->wd_timer.expires = jiffies + delay;
                DEBUG(DEBUG_INFO, "timer in %li jiffies\n", delay);
                add_timer(&q->wd_timer);
        }

2550    FCT_OUT;
}

/*
 * watchdog function - schedules call of dequeue if possible
2555 */

static void hfsc_watchdog(unsigned long arg)
{
        struct Qdisc *sch = ( struct Qdisc*)arg;
2560
        FCT_IN;
        netif_schedule (sch->dev);
        FCT_OUT;
}
2565


/*
 * implementation of functions exported via qdisc class interface
 */
2570
/*
 * create/modify a H-FSC class
 */
static int
2575 hfsc_change_class(struct Qdisc *sch, u32 classid, u32 parentid, struct rtattr **tca,
                    unsigned long *arg)
{
        struct hfsc_class *cl = ( struct hfsc_class *)*arg;
        struct service_curve *rsc = NULL;       /* real-time service curve      */
2580    struct service_curve *fsc = NULL;       /* link-sharing service curve */
        struct service_curve *dsc = NULL;       /* packet drop service curve  */
        struct hfsc_sched_data *q = ( struct hfsc_sched_data *) sch->data;
        struct rtattr *opt = tca[TCA_OPTIONS-1];
        struct rtattr *tb[TCA_HFSC_MAX];
2585    struct hfsc_class *parent;
        struct tc_hfsc_modify_class *class_mod; /* class modification data      */

        FCT_IN;
        /*
2590     * read traffic control attributes into table
         */
```

```
    if (( opt == NULL) ||
        rtattr_parse (tb, TCA_HFSC_MAX, RTA_DATA(opt), RTA_PAYLOAD(opt)))
          return -EINVAL;

    /*
     * check received info
     */
    if ( tb[TCA_HFSC_MODCLASS-1] &&
         RTA_PAYLOAD(tb[TCA_HFSC_MODCLASS-1]) < sizeof( struct tc_hfsc_modify_class ))
          return -EINVAL;
    class_mod = ( struct tc_hfsc_modify_class *) RTA_DATA(tb[TCA_HFSC_MODCLASS-1]);

        rsc = &(class_mod->rt_curve);
        fsc = &(class_mod->ls_curve);
    if ( class_mod->type & HFSC_DROPSC )
        dsc = &(class_mod->pd_curve);       /* packet drop curve */

    /*
     * now we have three possibilities :
     * a) the class exists , it is to be modified
     * b) the class does not exist and has to be created
     * c) the user specified senseless parameters
     */

    if ( cl )
    {
      /*
       * check parentid if available
       */
      if ( parentid ) {
            if (cl->cl_parent && (cl->cl_parent->cl_id != parentid))
                return -EINVAL;
            if (!( cl->cl_parent) && (parentid != TC_H_ROOT))
                return -EINVAL;
      }
      /*
       * modify class
       */
      return(hfsc_class_modify(sch, cl, rsc, fsc, dsc));
    }

    /*
     * handle possibility b) and other part of c)
     */
    if ( classid )
    {
      /*
       * user has specified a class id, check if it is valid
       */
      if (TC_H_MAJ(classid^sch->handle) || hfsc_class_lookup(q, classid))
      {
          DEBUG( DEBUG_WARN, "class exists or does not fit to scheduler handle\n" );
          return -EINVAL;
      }
    }
    else
    {
      /*
       * user has not specified an id, try to create one
       */
      int i;
      classid = TC_H_MAKE(sch->handle, 0x8000);
      for (i=0; i<0x8000; i++)
      {
          if ( ++q->sched_classid >= 0x8000 )
              q->sched_classid = 1;
          if ( hfsc_class_lookup(q, classid|q->sched_classid) == NULL)
              break;
      }
      if ( i>= 0x8000 )
      {
          DEBUG( DEBUG_WARN, "out of class handles\n" );
          return -ENOSR;
      }
      classid = classid | q->sched_classid;
    }

    /*
     * by now there is a class handle assigned, get parent
     */
    if (! parentid )
    {
        DEBUG( DEBUG_WARN, "no parent id specified\n" );
        return -EINVAL;
```

```
        }
        if (!( parent = hfsc_class_lookup ( q , parentid ) ))
        {
            DEBUG( DEBUG_WARN, "invalid␣parent␣id␣specified\n" );
            return -EINVAL;
        }

        if ( ( rsc && fsc) || ( rsc && dsc) || ( fsc && dsc) )
        {
                /*
                 * if the link-sharing service curve is diffrent from
                 * the real-time service curve , we first create a class with the
                 * smaller service curve and then modify the other service curve.
                 */
                if ( rsc->m2 <= fsc->m2) {
                        /*
                         * init with real-time sc , modify with link-sharing sc
                         */
                        cl= hfsc_class_create (sch , rsc , parent , 0, class_mod->flags, classid );
                        if ( cl )
                            return(hfsc_class_modify(sch , cl , NULL, fsc, dsc));
                } else {
                        /*
                         * init with link sc , modify mit real-time sc
                         */
                        cl= hfsc_class_create (sch , fsc , parent , 0, class_mod->flags, classid );
                        if ( cl )
                            return(hfsc_class_modify(sch , cl , rsc , NULL, dsc));
                }
        }
        else
        {
           if ( ( rsc ) ) {
               return ( hfsc_class_create (sch , rsc , parent , 0, class_mod->flags, classid ) ? 0 : - EINVAL); }
           if ( ( fsc ) ) {
               return ( hfsc_class_create (sch , fsc , parent , 0, class_mod->flags, classid ) ? 0 : - EINVAL); }
        }

    printk ("unable␣to␣create␣class\n");
    return -EINVAL;
}

/*
 * bind a filter to a class with a specific class id
 *
 *   In H-FSC filters can only be attached to leaf classes
 *   since interior classes are used to set-up link-sharing rules
 *   only and do not contain any packets . This behavior is different
 *   i.e. to CBQ.
 */
static unsigned long
hfsc_bind_filter (struct Qdisc *sch , unsigned long parent ,
                                            u32 classid )
{
        struct hfsc_sched_data *q = ( struct hfsc_sched_data *) sch->data;
        /* struct hfsc_class *p = ( struct hfsc_class *) parent ; */
        struct hfsc_class *cl = hfsc_class_lookup (q , classid );

        FCT_IN;
        if ( cl )
        {
                if ( is_a_parent_class (cl)) {
                        printk (" hfsccmd_add_filter :␣not␣a␣leaf␣class !\n");
                        return (0) ;
                }
                cl-> cl_filters ++;
                FCT_OUT;
                return (unsigned long) cl ;
        }
        FCT_OUT;
        return(0) ;
}

/*
 * unbind a filter from a leaf class
 * (pre-condition is that the class exists )
 */
static void
hfsc_unbind_filter (struct Qdisc *sch , unsigned long arg)
{
        struct hfsc_class *cl = ( struct hfsc_class *) arg;

        FCT_IN;
        ASSERT(cl->cl_filters > 0) ;
```

```
2760            ASSERT(!is_a_parent_class(cl));
                cl->cl_filters--;
                FCT_OUT;
                return;
        }
2765
        #ifdef CONFIG_RTNETLINK
        static int
        hfsc_dump_class(struct Qdisc *sch, unsigned long arg,
                        struct sk_buff *skb, struct tcmsg *tcm)
2770    {
                struct hfsc_class *cl = (struct hfsc_class *)arg;
                struct tc_hfsc_modify_class c_opt;          /* basic class opts */
                struct tc_hfsc_xstats c_xstats;             /* class statistics */
                unsigned char *b        = skb->tail;
2775
                FCT_IN;
                if (!cl)
                  return -1;

2780            if (cl->cl_parent) {
                  tcm->tcm_parent = cl->cl_parent->cl_id;
                } else {
                  tcm->tcm_parent = TC_H_ROOT;
                }
2785            tcm->tcm_handle = cl->cl_id;

                if (cl->cl_q)
                  tcm->tcm_info = cl->cl_q->handle;

2790            /* clear structs */
                memset(&c_opt, 0, sizeof(c_opt));
                memset(&c_xstats, 0, sizeof(c_xstats));

                /* copy basic class options */
2795            if(cl->cl_rsc != NULL)
                {
                        c_opt.rt_curve.m1 = sm2m(cl->cl_rsc->sm1);
                        c_opt.rt_curve.d  = dx2d(cl->cl_rsc->dx);
                        c_opt.rt_curve.m2 = sm2m(cl->cl_rsc->sm2);
2800                    c_opt.type |= HFSC_REALTIMESC;
                }
                if(cl->cl_fsc != NULL)
                {
                        c_opt.ls_curve.m1 = sm2m(cl->cl_fsc->sm1);
2805                    c_opt.ls_curve.d  = dx2d(cl->cl_fsc->dx);
                        c_opt.ls_curve.m2 = sm2m(cl->cl_fsc->sm2);
                        c_opt.type |= HFSC_LINKSHARINGSC;
                }
                c_opt.flags = cl->cl_flags;
2810
                /* put it in netlink socket */
                RTA_PUT(skb, TCA_OPTIONS, sizeof(c_opt), &c_opt);

                /* copy extended class statistics */
2815            c_xstats.total          = cl->cl_total;
                c_xstats.cumul          = cl->cl_cumul;
                c_xstats.d              = cl->cl_d;
                c_xstats.e              = cl->cl_e;
                c_xstats.k              = cl->cl_k;
2820            c_xstats.vt             = cl->cl_vt;
                c_xstats.qlen           = cl->cl_qlen;
                c_xstats.xmit_cnt       = cl->cl_stats.xmit_cnt;
                c_xstats.drop_cnt       = cl->cl_stats.drop_cnt;
                c_xstats.period         = cl->cl_stats.period;
2825
                /* put it in netlink socket */
                RTA_PUT(skb, TCA_XSTATS, sizeof(c_xstats), &c_xstats);

                FCT_OUT;
2830            return skb->len;

        rtattr_failure:
                skb_trim(skb, b - skb->data);
                return -1;
2835    }
        #endif /* CONFIG_RTNETLINK */

        /*
         * admission control using generalized service curve
2840     *
         * (Although it would be nicer to handle this in user space,
         *  we currently handle this here in the kernel since do
         *  only have access to the necessary information here.
```

```
 * A future improvement could be to read the necessary info
 * via rtnetlink socket and evaluate it in user space.)
 */
#define INFINITY        (~(u64)0)       /* positive infinity */

/* add a new service curve to a generilized service curve */
static void
gsc_add_sc(struct gen_sc *gsc, struct service_curve *sc)
{
        if (is_sc_null(sc))
                return;
        if (sc->d != 0)
                gsc_add_seg(gsc, 0, 0, sc->d, sc->m1);
        gsc_add_seg(gsc, sc->d, 0, INFINITY, sc->m2);
}

/* subtract a service curve from a generilized service curve */
static void
gsc_sub_sc(struct gen_sc *gsc, struct service_curve *sc)
{
        if (is_sc_null(sc))
                return;
        if (sc->d != 0)
                gsc_sub_seg(gsc, 0, 0, sc->d, sc->m1);
        gsc_sub_seg(gsc, sc->d, 0, INFINITY, sc->m2);
}

/*
 * check whether all points of a generalized service curve have
 * their y-coordinates no larger than a given two-piece linear
 * service curve.
 */
static int
is_gsc_under_sc(struct gen_sc *gsc, struct service_curve *sc)
{
        struct segment *s, *last, *end;
        u64 y;

        if (is_sc_null(sc)) {
                if (LIST_EMPTY(gsc))
                        return (1);
                LIST_FOREACH(s, gsc, _next) {
                        if (s->m != 0)
                                return (0);
                }
                return (1);
        }
        /*
         * gsc has a dummy entry at the end with x = INFINITY.
         * loop through up to this dummy entry.
         */
        end = gsc_getentry(gsc, INFINITY);
        if (end == NULL)
                return (1);
        last = NULL;
        for (s = LIST_FIRST(gsc); s != end; s = LIST_NEXT(s, _next)) {
                if (s->y > sc_x2y(sc, s->x))
                        return (0);
                last = s;
        }
        /* last now holds the real last segment */
        if (last == NULL)
                return (1);
        if (last->m > sc->m2)
                return (0);
        if (last->x < sc->d && last->m > sc->m1) {
                y = last->y + (sc->d - last->x) * last->m;
                if (y > sc_x2y(sc, sc->d))
                        return (0);
        }
        return (1);
}

/*
 * destroy a generalized service curve
 */
static void
gsc_destroy(struct gen_sc *gsc)
{
        struct segment *s;

        while ((s = LIST_FIRST(gsc)) != NULL) {
                LIST_REMOVE(s, _next);
                kfree(s);
```

```
        }
}

/*
 * return a segment entry starting at x.
 * if gsc has no entry starting at x, a new entry is created at x.
 */
static struct segment *
gsc_getentry(struct gen_sc *gsc, u64 x)
{
        struct segment *new, *prev, *s;

        prev = NULL;
        LIST_FOREACH(s, gsc, _next) {
                if (s->x == x)
                        return (s);     /* matching entry found */
                else if (s->x < x)
                        prev = s;
                else
                        break;
        }

        /* we have to create a new entry */
        if ((new = kmalloc(sizeof(struct segment), GFP_KERNEL)) == NULL)
                return (NULL);

        new->x = x;
        if (x == INFINITY || s == NULL)
                new->d = 0;
        else if (s->x == INFINITY)
                new->d = INFINITY;
        else
                new->d = s->x - x;
        if (prev == NULL) {
                /* insert the new entry at the head of the list */
                new->y = 0;
                new->m = 0;
                LIST_INSERT_HEAD(gsc, new, _next);
        } else {
                /*
                 * the start point intersects with the segment pointed by
                 * prev. divide prev into 2 segments
                 */
                if (x == INFINITY) {
                        prev->d = INFINITY;
                        if (prev->m == 0)
                                new->y = prev->y;
                        else
                                new->y = INFINITY;
                } else {
                        prev->d = x - prev->x;
                        new->y = prev->d * prev->m + prev->y;
                }
                new->m = prev->m;
                LIST_INSERT_AFTER(prev, new, _next);
        }
        return (new);
}

/* add a segment to a generalized service curve */
static int
gsc_add_seg(struct gen_sc *gsc, u64 x, u64 y, u64 d, u64 m)
{
        struct segment * start, * end, * s;
        u64 x2;

        if (d == INFINITY)
                x2 = INFINITY;
        else
                x2 = x + d;
        start = gsc_getentry(gsc, x);
        end = gsc_getentry(gsc, x2);
        if (start == NULL || end == NULL)
                return (-1);

        for (s = start; s != end; s = LIST_NEXT(s, _next)) {
                s->m += m;
                s->y += y + (s->x - x) * m;
        }

        end = gsc_getentry(gsc, INFINITY);
        for (; s != end; s = LIST_NEXT(s, _next)) {
                s->y += m * d;
        }
```

```
        return (0);
}

/* subtract a segment from a generalized service curve */
static int
gsc_sub_seg(struct gen_sc *gsc, u64 x, u64 y, u64 d, u64 m)
{
        if (gsc_add_seg(gsc, x, y, d, -m) < 0)
                return (-1);
        gsc_compress(gsc);
        return (0);
}

/*
 * collapse adjacent segments with the same slope
 */
static void
gsc_compress(struct gen_sc *gsc)
{
        struct segment *s, *next;

 again:
        LIST_FOREACH(s, gsc, _next) {

                if ((next = LIST_NEXT(s, _next)) == NULL) {
                        if (LIST_FIRST(gsc) == s && s->m == 0) {
                                /*
                                 * if this is the only entry and its
                                 * slope is 0, it's a remaining dummy
                                 * entry. we can discard it.
                                 */
                                LIST_REMOVE(s, _next);
                                kfree(s);
                        }
                        break;
                }

                if (s->x == next->x) {
                        /* discard this entry */
                        LIST_REMOVE(s, _next);
                        kfree(s);
                        goto again;
                } else if (s->m == next->m) {
                        /* join the two entries */
                        if (s->d != INFINITY && next->d != INFINITY)
                                s->d += next->d;
                        LIST_REMOVE(next, _next);
                        kfree(next);
                        goto again;
                }
        }
}

/* get y-projection of a service curve */
static u64
sc_x2y(struct service_curve *sc, u64 x)
{
        u64 y;

        if (x <= sc->d)
                /* y belongs to the 1st segment */
                y = x * sc->m1;
        else
                /* y belongs to the 2nd segment */
                y = sc->d * sc->m1
                        + (x - sc->d) * sc->m2;
        return (y);
}

static __inline__ struct hfsc_class *
hfsc_class_lookup(struct hfsc_sched_data *q, u32 classid)
{
        struct hfsc_class *cl;

        for (cl = q->sched_rootclass; cl; cl = hfsc_nextclass(cl))
                if (cl->cl_id == classid)
                        return cl;
        return NULL;
}

/*
 * drop packets according to packet drop service curve
 */
```

```
static inline unsigned int service_packet_drop ( struct hfsc_class *cl , u64 time )
{
  struct sk_buff *skb;
3100  unsigned int dropped=0;

  FCT_IN;
  while( cl->cl_k < time ) {
        /*
3105     * at least one skb has finished processing
         * in the "drop server" first - dequeue and
         * drop one packet
         */
        skb = hfsc_getq(cl);
3110    if ( !skb )
        {
            DEBUG( DEBUG_WARN, "service_packet_drop: no skb\n");
            return 0;
        }
3115    PKTCNTR_ADD(&cl->cl_stats.drop_cnt, skb->len);
        DEBUG( DEBUG_INFO, "packet served by drop server: drop time %lld\n",
                           cl->cl_k);
        cl->cl_sch->stats.drops++;
        cl->cl_drop_cumul += skb->len;
3120
        kfree_skb(skb);
        dropped++;
        ((struct hfsc_sched_data *) cl->cl_sch->data)->sched_packets--;
        cl->cl_sch->q.qlen--;
3125
        /*
         * update drop time
         */
        skb = hfsc_pollq(cl);
3130    if ( skb )
            update_k(cl , skb->len);
        else
        {
            set_passive (cl);
3135        return dropped;
        }
  }
  FCT_OUT;
  return dropped;
3140 }

/*
 * additional functions for Linux scheduling framework
 * (qdisc interface )
3145 */

/*
 * the "graft" operation is used to attach a new qeuing discipline
 * to this class which is used for storing packets of this class
3150 */
static int hfsc_graft (struct Qdisc *sch , unsigned long arg , struct Qdisc *new,
                       struct Qdisc **old)
{
        struct hfsc_class *cl = (struct hfsc_class *)arg;
3155
        FCT_IN;
        if (cl) {
                if (new == NULL) {
                        if ((new = qdisc_create_dflt (sch->dev, &pfifo_qdisc_ops)) == NULL)
3160                            return -ENOBUFS;
                }
                sch_tree_lock(sch);
                /* return old qdisc and set new one */
                *old = cl->cl_q;
3165            cl->cl_q = new;
                sch_tree_unlock(sch);
                qdisc_reset(*old);
                FCT_OUT;
                return 0;
3170    }
        return -ENOENT;
}

static struct Qdisc *
3175 hfsc_leaf (struct Qdisc *sch , unsigned long arg)
{
        /* @@@ fix this for variable qdisc @@@ struct hfsc_class *cl = (struct hfsc_class *)arg ; */
        FCT_IN;
        // return cl ? cl->cl_q : NULL;
```

```
3180            FCT_OUT;
                return NULL;
        }

        static unsigned long hfsc_get(struct Qdisc *sch, u32 classid)
3185    {
                struct hfsc_sched_data *q = (struct hfsc_sched_data *)sch->data;
                struct hfsc_class *cl = hfsc_class_lookup(q, classid);

                FCT_IN;
3190            if (cl) {
                        cl->cl_refcnt++;
                        FCT_OUT;
                        return (unsigned long)cl;
                }
3195            DEBUG( DEBUG_INFO, "hfsc_get: did not find class with id: %d\n", classid);
                return 0;
        }

        static void hfsc_put(struct Qdisc *sch, unsigned long arg)
3200    {
                struct hfsc_class *cl = (struct hfsc_class *)arg;

                FCT_IN;
                if (--cl->cl_refcnt == 0) {
3205                    /* hfsc_class_destroy (sch, cl); */
                }
                FCT_OUT;
        }

3210    /*
         * "walk over" (traverse) all classes and perform an
         * arbitrary function on all of them starting after number arg->skip
         */
        static void hfsc_walk(struct Qdisc *sch, struct qdisc_walker *arg)
3215    {
                struct hfsc_sched_data *q = (struct hfsc_sched_data *)sch->data;
                struct hfsc_class *cl;

                FCT_IN;
3220            if (arg->stop)
                        return;

                for (cl = q->sched_rootclass; cl; cl = hfsc_nextclass(cl)) {
                        if (arg->count < arg->skip) {
3225                            arg->count++;
                                continue;
                        }
                        if (arg->fn(sch, (unsigned long)cl, arg) < 0) {
                                arg->stop = 1;
3230                            return;
                        }
                        arg->count++;
                }
                FCT_OUT;
3235    }

        /*
         * This is the tcf_chain operation of the class interface
         * which is used to find the address of the classifier chain.
3240     * With this information the scheduling API is able to put a new
         * filter in the list or remove an existing one.
         */
        static struct tcf_proto ** hfsc_find_tcf(struct Qdisc *sch, unsigned long arg)
        {
3245            struct hfsc_sched_data *q = (struct hfsc_sched_data *)sch->data;
                /* struct hfsc_class *cl = (struct hfsc_class *)arg; */

                FCT_IN;
                FCT_OUT;
3250            return &q-> sched_filter_list;
        }

        /*
         * the requeue function is called when the lower layer(s) were
3255     * not able to transmit an already dequeued packet
         * (which happens quite often with some drivers, especially
         * in case of sending large packets)
         */

3260    static int
        hfsc_requeue(struct sk_buff *skb, struct Qdisc *sch)
        {
                struct hfsc_sched_data *q = (struct hfsc_sched_data *)sch->data;
```

```
        int ret = NET_XMIT_CN;

        FCT_IN;

        if (( ret = q->sched_requeued_skbs->enqueue(skb, q->sched_requeued_skbs)) == 0)
        {
                /*
                 * packet succesfully requeued
                 *
                 * (A superior approach would be to queue it in the real
                 *  qdisc we dequeued it from but then we have to change
                 *  all service counters - since this packet was already
                 *  completely processed and the interior qdisc could hand
                 *  us a different packet at the next class dequeue event.
                 *  This way it is much simpler - but less flexible .)
                 */
                sch->q.qlen++;
                q->sched_packets++;
                FCT_OUT;
                return 0;
        }
        else
        {
                /*
                 * requeuing failed
                 * (skb was freed in internel qdisc)
                 */
                // cl->cl_stats.drop_cnt++;
                sch->stats.drops++;
                DEBUG( DEBUG_WARN, "hfsc_requeue: unable to put skb in requeued_skb queue!\n" );
                return ret;
        }

        ASSERT(0);
        return NET_XMIT_CN;
}

/*
 * dump global scheduler parameters via rtnetlink socket to
 * userspace control program (usually "tc")
 */
static int hfsc_dump(struct Qdisc *sch, struct sk_buff *skb)
{
        struct hfsc_sched_data *q = (struct hfsc_sched_data*)sch->data;
        struct tc_hfsc_qopt     opt;
        unsigned char *b        = skb->tail;

        FCT_IN;
        opt.bandwidth = sm2m(q->sched_rootclass->cl_rsc->sm2);
        RTA_PUT(skb, TCA_OPTIONS, sizeof(opt), &opt);
        FCT_OUT;
        return skb->len;

 rtattr_failure :
        skb_trim(skb, b - skb->data);
        return -1;
}

/*
 * drop function: drops one packet
 *
 * Since we do not use any static priorities, we simply try to
 * drop a packet of the default class.
 */

static int hfsc_drop(struct Qdisc* sch)
{
        struct hfsc_sched_data *q = (struct hfsc_sched_data *)sch->data;
        struct hfsc_class *cl;

        FCT_IN;

        if (!(cl = q->sched_defaultclass))
                return(0);
        if (cl->cl_q->ops->drop && cl->cl_q->ops->drop(cl->cl_q))
                return(1);

        FCT_OUT;
        return 0;
}

static int hfsc_delete(struct Qdisc *sch, unsigned long arg)
{
        struct hfsc_class *cl = (struct hfsc_class *)arg;
```

```
        FCT_IN;

        if (--cl->cl_refcnt == 0)
                hfsc_class_destroy (sch, cl);

        FCT_OUT;
        return 0;
}


/*
 * interface for class operations
 */
static struct Qdisc_class_ops hfsc_class_ops =
{
        hfsc_graft ,
        hfsc_leaf ,
        hfsc_get ,
        hfsc_put ,
        hfsc_change_class ,
        hfsc_delete ,
        hfsc_walk,

        hfsc_find_tcf ,
        hfsc_bind_filter ,
        hfsc_unbind_filter ,

#ifdef CONFIG_RTNETLINK
        hfsc_dump_class,
#endif
};

/*
 * interface for qdisc operations
 */
struct Qdisc_ops hfsc_qdisc_ops =
{
        NULL,
        &hfsc_class_ops,
        "hfsc",
        sizeof (struct hfsc_sched_data),

        hfsc_enqueue,
        hfsc_dequeue,
        hfsc_requeue,
        hfsc_drop,

        hfsc_init ,
        hfsc_reset ,
        hfsc_destroy ,
        NULL /* hfsc_change */,

#ifdef CONFIG_RTNETLINK
        hfsc_dump,
#endif
};


/*
 * interface for loading/unloading module
 */
#ifdef MODULE
int init_module(void)
{
        FCT_IN;
        printk (" Hierarchical Fair Service Curve Scheduler (H-FSC) [" __DATE__ " " __TIME__ "]\n");

#ifdef TCSIM_COMPATIBLE
        printk ("H-FSC: -- version for TCSIM simulation environment ---\n");
#endif
#ifdef CONFIG_NET_WIRELESS_SCHED
        printk ("H-FSC: -- compiled with wireless scheduling extensions --\n");
#endif
#ifdef DEBUG_ENABLED
        printk ("H-FSC: -- compiled with debugging support -------\n");
#endif
        FCT_OUT;
        return register_qdisc (&hfsc_qdisc_ops);
}

void cleanup_module(void)
{
        FCT_IN;
```

```
        unregister_qdisc (&hfsc_qdisc_ops);
        FCT_OUT;
}
#endif /* MODULE */

/* end of file : sch_hfsc.c */
```

B.3.2 User Space TC Module

Listing B.5: *The H-FSC* `tc` *module (*`q_hfsc.c`*).*

```
/* $Id: q_hfsc.c,v 1.1.2.1 2001/11/12 20:37:37 wischhof Exp $ */
/*
 * q_hfsc.c      modified hierarchical fair service curve algorithm,
 *               userspace tc module.
 *
 *
 *               This program is free software; you can redistribute it and/or
 *               modify it under the terms of the GNU General Public License
 *               as published by the Free Software Foundation; either version
 *               2 of the License, or (at your option) any later version.
 *
 * Authors:      Lars Wischhof <wischhof@ieee.org>
 *               Washington University, St. Louis, MO
 *               Technical University of Berlin, Germany
 *
 *               based on the original implementation (NetBSD/ALT-Q 3.0)
 *               by Sony Labs (core scheduler by Carnegie Mellon University)
 *               and the iproute2 package by Alexey Kuznetsov, <kuznet@ms2.inr.ac.ru>
 *
 * $Log: q_hfsc.c,v $
 * Revision 1.1.2.1  2001/11/12 20:37:37  wischhof
 * Monday afternoon version - final
 *
 * Revision 1.4  2001/10/31 22:08:07  wischhof
 * fix: compile warning if compiled without CONFIG_NET_WIRELESS_SCHED
 *
 */

// # define CONFIG_NET_WIRELESS_SCHED // define if compiling for wireless scheduling simulations

#include <sys/param.h>
#include <sys/socket.h>
#include <sys/ioctl.h>
#include <sys/fcntl.h>
#include <net/if.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <stddef.h>
#include <string.h>
#include <ctype.h>
#include <errno.h>
#include <syslog.h>
#include <netdb.h>
#include <math.h>

#ifdef CONFIG_NET_WIRELESS_SCHED
#include "wsched_util.h"
#endif /* CONFIG_NET_WIRELESS_SCHED */

#include "utils.h"
#include "tc_util.h"


static int read_sc(int *argcp, char ***argvp,
                   int *type, unsigned int *m1, unsigned int *d, unsigned int *m2);
static int validate_sc(struct service_curve *sc);
static int get_bitrate(unsigned *rate, char *str);

#define EQUAL(s1, s2)   (strcmp((s1), (s2)) == 0)

static void explain_class(void)
{
        fprintf(stderr, "Hiererchical Fair Service Curve Scheduler (H-FSC)\n\n");
        fprintf(stderr, "Class parameters: [sc M1 D M2] [rt M1 D M2] [ls M1 D M2] [dc M1 D M2]\n");
        fprintf(stderr, "                  [pshare PERCENT] [grate BPS] [default] [help]");
#ifdef CONFIG_NET_WIRELESS_SCHED
        fprintf(stderr, " [sync]");
#endif /* CONFIG_NET_WIRELESS_SCHED */
        fprintf(stderr, "\n\n");
        fprintf(stderr, "   sc              defines a service curve which consists of a two linear\n");
        fprintf(stderr, "                   pieces: up to delay D (in ms) with slope M1 and with\n");
        fprintf(stderr, "                   slope M2 afterwards (equal to defining a real-time and\n");
        fprintf(stderr, "                   a link-sharing curve with the same parameters)\n");
        fprintf(stderr, "   rt              defines a real-time service curve with parameters M1, D, M2\n");
        fprintf(stderr, "   ls              defines a link-sharing service curve\n");
```

```
80          fprintf ( stderr , "                    (NULL link-sharing curve can be used to shape bandwidth)\n");
            fprintf ( stderr , "  dc                defines a packet drop curve\n");
            fprintf ( stderr , "  pshare            defines a linear link-sharing curve with PERCENT of the total\n");
            fprintf ( stderr , "                    bandwidth (equal to a linear [ls 0 0 M2] curve)\n");
            fprintf ( stderr , "  grate             defines a linear real-time service curve with a rate of BPS\n");
85          fprintf ( stderr , "                    (equivalent to a linear [rt 0 0 M2] real-time curve\n");
            fprintf ( stderr , "  default           specify this class as the (new) default class\n");
#ifdef CONFIG_NET_WIRELESS_SCHED
            fprintf ( stderr , "  sync              this class is a wireless synchronization class\n");
#endif /* CONFIG_NET_WIRELESS_SCHED */
90          fprintf ( stderr , "\n");
}

static void explain(void)
{
95          fprintf ( stderr , " Hierarchical  Fair Service Curve Scheduler (H-FSC)\n\n");
            fprintf ( stderr , "Usage: ... hfsc bandwidth BPS [estint SIZE] [varrate] \n");

#ifdef CONFIG_NET_WIRELESS_SCHED
            fprintf ( stderr , "                    [wireless] [wchmon NAME] [reducebad PERCENT]\n\n");
100 #endif /* CONFIG_NET_WIRELESS_SCHED */

            fprintf ( stderr , "  bandwidth   total bandwidth of the interface\n");

#ifdef CONFIG_NET_WIRELESS_SCHED
105         fprintf ( stderr , "                    (for wireless LAN: max. expected total link throughput\n");
            fprintf ( stderr , "                    e.g. about 5.0 MBit/s for 11 Mbit/s card,\n");
            fprintf ( stderr , "                    about 1.4 MBit/s for 2 Mbit/s card)\n");
#endif /* CONFIG_NET_WIRELESS_SCHED */

110         fprintf ( stderr , "  varrate     use variable bandwidth adaption\n");
            fprintf ( stderr , "  estint      size of interval for bandwidth estimation/averaging\n");

#ifdef CONFIG_NET_WIRELESS_SCHED
            fprintf ( stderr , "  wireless    enable wireless channel monitoring/error rate estimation\n");
115         fprintf ( stderr , "  wchmon      install wireless channel monitor NAME on the target device\n");
            fprintf ( stderr , "  reducebad   percentage of the additional bandwidth which a class needs\n");
            fprintf ( stderr , "                    because of retransmissions/rate adaption taken into account\n");
            fprintf ( stderr , "                    when scheduling in an overload state\n");
#endif /* CONFIG_NET_WIRELESS_SCHED */
120 }

static void explain1(char *arg)
{
            fprintf ( stderr , " Illegal  \"%s\"\n", arg);
125 }

#define usage() return(-1)


130 /*
 * parser interface
 */
static int
hfsc_parse_opt(struct qdisc_util *qu, int argc, char **argv, struct nlmsghdr *n)
135 {
        unsigned int    bandwidth = 0;
        unsigned int    estint    = 0;
        struct tc_hfsc_qopt qopt;
#ifdef CONFIG_NET_WIRELESS_SCHED
140     unsigned int    wchmon_specified = 0;
#endif /* CONFIG_NET_WIRELESS_SCHED */

        memset(&qopt, 0, sizeof(qopt));

145     /*
         * process options
         */
        while (argc > 0) {
                if (EQUAL(*argv, "bandwidth")) {
150                     argc--; argv++;
                        if ((argc > 0) && (get_bitrate (&bandwidth, *argv) == -1) )
                          {
                            explain1("bandwidth");
                            return -1;
155                       }
                } else if (EQUAL(*argv, "varrate")) {
                        qopt.flags |= HFSC_SCHED_VAR_RATE;
                } else if (EQUAL(*argv, "estint")) {
                        argc--; argv++;
160                     if ((argc > 0) && (get_size(&estint, *argv) == -1))
                          {
                            explain1(" estint ");
                            return -1;
```

```
                        }
165
#ifdef CONFIG_NET_WIRELESS_SCHED
                } else if (EQUAL(*argv, "reducebad")) {
                        argc--; argv++;
                        if ((argc > 0) && (get_integer(&qopt.reducebad, *argv, 10) == -1))
170                     {
                                explain1("reducebad");
                                return -1;
                        }
                        qopt.reducebad_valid++;
175             } else if (EQUAL(*argv, "wireless")) {
                        qopt.flags |= HFSC_SCHED_WIRELESS_MON;
                        /* default: reduce using wireless info */
                        qopt.reducebad = 100;
                } else if (EQUAL(*argv, "wchmon")) {
180                     argc--; argv++;
                        if (strlen(*argv)>WCHMON_MAX_ID_LEN) {
                                fprintf(stderr, "Id of \"wchmon\" too long.");
                                return -1;
                        }
185                     else
                        {
                                strncpy(qopt.wchmon_id, *argv, WCHMON_MAX_ID_LEN);
                                wchmon_specified = 1;
                        }
190 #endif /* CONFIG_NET_WIRELESS_SCHED */

                } else if (EQUAL(*argv, "help")) {
                        explain();
                } else if (EQUAL(*argv, "hfsc")) {
195                     /* just skip */
                } else {
                        fprintf(stderr, "Unknown keyword '%s'\n", *argv);
                        return (-1);
                }
200             argc--; argv++;
        }

        /*
         * options are processed, check & prepare for rtnetlink socket
205      */
        if (bandwidth == 0) {
                fprintf(stderr, "hfsc: bandwidth is required parameter.\n");
                return -1;
        }
210
        if (estint > 0) {
                /* calculate log2 of averaging interval */
                qopt.est_interval_log = abs(log10((double) estint)/log10(2));
                /* enable variable rate extensions */
215             qopt.flags |= HFSC_SCHED_VAR_RATE;
        }

#ifdef CONFIG_NET_WIRELESS_SCHED
        if ((qopt.flags & HFSC_SCHED_WIRELESS_MON) && !(wchmon_specified)){
220             fprintf(stderr, "hfsc: wireless channel monitor not specified\n");
                return -1;
        }
        if (!(qopt.flags & HFSC_SCHED_WIRELESS_MON) && wchmon_specified){
                fprintf(stderr, "hfsc: wireless channel monitor specified but option \"wireless\" missing!\n");
225             return -1;
        }
        if ((qopt.reducebad != 0) && !(qopt.flags & HFSC_SCHED_WIRELESS_MON)) {
                fprintf(stderr, "hfsc: the option \"reducebad\" requires wireless scheduling.\n");
                return -1;
230     }
#endif /* CONFIG_NET_WIRELESS_SCHED */

        qopt.bandwidth = bandwidth;

235     addattr_l(n, 1024, TCA_OPTIONS, &qopt, sizeof(qopt));
        return (0);
}

static int
240 hfsc_parse_class_opt(struct qdisc_util *qu, int argc, char **argv, struct nlmsghdr *n)
{
        u_int   m1, d, m2, rm1, rd, rm2, fm1, fd, fm2, dm1, dm2, dd;
        int     qlimit = 50;
        int     flags = 0, admission = 0;
245     int     type = 0, received_types = 0;
                                                /* struct holding the info to be sent */
        struct  tc_hfsc_modify_class mod_class; /* via rtnetlink socket to kernel     */
```

```
        struct  rtattr  * tail ;

250
        rm1 = rd  = rm2 = fm1 = fd  = fm2 = dm1 = dm2 = dd = 0;
        memset (&mod_class, 0,  sizeof (mod_class));

        while ( argc  > 0)  {
255             if  (* argv [0] ==  ' [ ' ) {
                        if  ( read_sc(&argc, &argv, &type, &m1, &d, &m2) != 0) {
                                fprintf ( stderr ,  "Bad service curve!\n");
                                return  (0) ;
                        }
260                     received_types  |=  type;
                        if  ( type & HFSC_REALTIMESC) {
                                rm1 = m1; rd  = d ;  rm2 = m2;
                        }
                        if  ( type & HFSC_LINKSHARINGSC) {
265                             fm1 = m1; fd  = d ;  fm2 = m2;
                        }
                        if  ( type & HFSC_DROPSC) {
                                dm1 = m1; dm2 = m2; dd = d;
                        }
270             } else  if  ( EQUAL(*argv, "pshare")) {
                        argc--; argv++;
                        if  ( argc  > 0)  {
                                /* struct   ifinfo  * ifinfo ;
                                unsigned  int  pshare;
275
                                pshare  = ( u_int ) strtoul (*argv ,  NULL, 0);
                                if  (( ifinfo  =  ifname2ifinfo (ifname))  !=  NULL) {
                                        fm2 =  ifinfo ->bandwidth / 100 * pshare;
                                        type  |=  HFSC_LINKSHARINGSC;
280                             } */
                                fprintf ( stderr ,  "option \"pshare\" currently  not supported!\n");
                                return  -1;
                        }
                } else  if  ( EQUAL(*argv, "grate")) {
285                     argc--; argv++;
                        if  ( argc  > 0)  {
                                if  (   get_bitrate (&rm2, *argv) == -1)
                                {
                                    explain1 (*argv);
290                                 return  -1;
                                }

                                type  |=  HFSC_REALTIMESC;
                        }
295             } else  if  ( EQUAL(*argv, "qlimit")) {
                        argc--; argv++;
                        if  ( argc  > 0)
                                qlimit  =  strtoul (*argv ,  NULL, 0);
                } else  if  ( EQUAL(*argv, "default"))  {
300                     flags  |=  HFCF_DEFAULTCLASS;
#ifdef  CONFIG_NET_WIRELESS_SCHED
                } else  if  ( EQUAL(*argv, "sync")) {
                        flags  |=  HFCF_SYNCCLASS;
#endif  /* CONFIG_NET_WIRELESS_SCHED */
305             } else  if  ( EQUAL(*argv, "admission")) {
                        argc--; argv++;
                        if  ( argc  > 0)  {
                                if  ( EQUAL(*argv, "guaranteed")
                                      ||  EQUAL(*argv, "cntlload"))
310                                     admission  = 1;
                                else  if  ( EQUAL(*argv, "none")) {
                                        /* nothing */
                                } else  {
                                        explain_class () ;
315                                     fprintf ( stderr ,
                                        "unknown admission type!\n");
                                        return  (0) ;
                                }
                        }
320             } else  if  ( EQUAL(*argv, "red")) {
                        flags  |=  HFCF_RED;
                } else  if  ( EQUAL(*argv, "ecn")) {
                        flags  |=  HFCF_ECN;
                } else  if  ( EQUAL(*argv, "rio")) {
325                     flags  |=  HFCF_RIO;
                } else  if  ( EQUAL(*argv, "cleardscp")) {
                        flags  |=  HFCF_CLEARDSCP;
                } else  if  ( EQUAL(*argv, "help")) {
                        explain_class () ;
330                     return  -1;
                } else  {
```

```
                        explain_class () ;
                        fprintf ( stderr ,
                            "Unknown␣keyword␣'%s'␣\n", *argv);
                        return (-1);
                }

                argc--; argv++;
        }

        if (( received_types == 0) || ( received_types == HFSC_DROPSC)) {
                explain_class () ;
                fprintf ( stderr ,
                    "hfsc:␣service␣curve␣not␣specified !\n") ;
                return (-1);
        }

        if (( flags & HFCF_ECN) && (flags & (HFCF_RED|HFCF_RIO)) == 0)
                flags |= HFCF_RED;

        /*
         * prepare netlink info
         */
        if ( received_types & HFSC_REALTIMESC ){
                mod_class.rt_curve.m1 = rm1;
                mod_class.rt_curve.m2 = rm2;
                mod_class.rt_curve.d = rd;
        }

        if ( received_types & HFSC_LINKSHARINGSC) {
                mod_class.ls_curve.m1 = fm1;
                mod_class.ls_curve.m2 = fm2;
                mod_class.ls_curve.d = fd;
        }

        if ( received_types & HFSC_DROPSC) {
                mod_class.pd_curve.m1 = dm1;
                mod_class.pd_curve.m2 = dm2;
                mod_class.pd_curve.d = dd;
        }

        mod_class.type = type;
        mod_class.flags = flags;

        if (( validate_sc (&mod_class.rt_curve) == -1) ||
            ( validate_sc (&mod_class.ls_curve) == -1) )
        {
            explain_class () ;
            return -1;
        }

        /*
         * send data via rtnetlink
         */
        tail = ( struct rtattr *) (((void*)n)+NLMSG_ALIGN(n->nlmsg_len));
        addattr_l (n, 1024, TCA_OPTIONS, NULL, 0);
        addattr_l (n, 1024, TCA_HFSC_MODCLASS, &mod_class, sizeof(mod_class));
        tail->rta_len = (((void*)n)+NLMSG_ALIGN(n->nlmsg_len)) - (void*)tail;
        return 0;
}

/*
 * read service curve parameters
 * '[' < type> <m1> <d> <m2> ']'
 *  type := " sc ", " rt ", or " ls "
 */
static int
read_sc(int *argcp, char ***argvp, int *type, unsigned int *m1, unsigned int *d, unsigned int *m2)
{
        int argc = *argcp;
        char **argv = *argvp;
        char *cp;

        cp = *argv;
        if (*cp++ != '[')
                return (-1);
        if (*cp == '\0') {
                cp = *++argv; --argc;
        }
        if (*cp == 's' || *cp == 'S')
                *type = HFSC_DEFAULTSC;
        else if (*cp == 'r' || *cp == 'R')
                *type = HFSC_REALTIMESC;
        else if (*cp == 'l' || *cp == 'L')
                *type = HFSC_LINKSHARINGSC;
```

```
        else if (*cp == 'd' || *cp == 'D')
                *type = HFSC_DROPSC;
        else
                return (-1);
        cp = *++argv; --argc;
        if ( get_bitrate (m1, cp) == -1)
        {
          explain1 ( cp ) ;
          return -1;
        }
        cp = *++argv; --argc;
        *d = ( u_int ) strtoul (cp, NULL, 0);
        cp = *++argv; --argc;
        if ( get_bitrate (m2, cp) == -1)
        {
          explain1 ( cp ) ;
        }
        *argcp = argc;
        *argvp = argv;
        return (0) ;
}


/*
 * read a bitrate ( tolerating closing bracket )
 */
static int get_bitrate (unsigned *rate , char *str )
{
        char *p;
        double bitps = strtod ( str , &p);

        if ( p == str )
                return -1;

        if (*p) {
                if (( strcasecmp(p, "kbps") == 0) || ( strcasecmp(p, "kbps]") == 0) )
                        bitps *= 1024*8;
                else if (( strcasecmp(p, "mbps") == 0) || ( strcasecmp(p, "mbps]") == 0) )
                        bitps *= 1024*1024*8;
                else if (( strcasecmp(p, "mbit") == 0) || ( strcasecmp(p, "mbit]") == 0) )
                        bitps *= 1024*1024;
                else if (( strcasecmp(p, "kbit") == 0) || ( strcasecmp(p, "kbit ]") == 0) )
                        bitps *= 1024;
                else if (( strcasecmp(p, "bps") == 0) || ( strcasecmp(p, "bps]") == 0) )
                        bitps *=8;
                else return -1;
        }

        *rate = bitps ;
        return 0;
}

/*
 * simple service curve validation
 */
static int
validate_sc ( struct service_curve *sc)
{
        /* the 1 st segment of a concave curve must be zero */
        if ( sc->m1 < sc->m2 && sc->m1 != 0) {
                fprintf ( stderr , "m1 must be 0 for convex!\n");
                return (-1);
        }
        return (0) ;
}

/*
 * print scheduler parameters
 */
static int hfsc_print_opt ( struct qdisc_util *qu, FILE *f, struct rtattr *opt)
{
        struct tc_hfsc_qopt *qopt;

        if ( opt == NULL)
                return 0;

        if ( RTA_PAYLOAD(opt) < sizeof(*qopt))
                return -1;
        qopt = RTA_DATA(opt);
        if ( strcmp(qu->id, "hfsc") == 0) {
                SPRINT_BUF(b1);
                fprintf (f , " bandwidth %s ", sprint_rate(qopt->bandwidth/8, b1));
        }
        else
```

```
500             {       fprintf(f, "␣invalid␣id␣of␣queuing␣discipline\n");
                        return -1;
                }
                return 0;
};
505
/*
 * print class parameters
 */
static int hfsc_print_class_opt(struct qdisc_util *qu, FILE *f, struct rtattr *opt)
510 {
        struct tc_hfsc_modify_class *mod_opt;
        SPRINT_BUF(b1);
        SPRINT_BUF(b2);

515     if (opt == NULL)
                return 0;

        if (RTA_PAYLOAD(opt) < sizeof(*mod_opt))
                return -1;
520     mod_opt = RTA_DATA(opt);

        if (mod_opt->type & HFSC_REALTIMESC)
                fprintf(f, "␣[rt␣␣%s␣%ims␣%s]␣",
                   sprint_rate(mod_opt->rt_curve.m1/8, b1),
525                mod_opt->rt_curve.d,
                   sprint_rate(mod_opt->rt_curve.m2/8, b2));
        if (mod_opt->type & HFSC_LINKSHARINGSC)
                fprintf(f, "␣[ls␣␣%s␣%ims␣%s]␣",
                   sprint_rate(mod_opt->ls_curve.m1/8, b1),
530                mod_opt->ls_curve.d,
                   sprint_rate(mod_opt->ls_curve.m2/8, b2));
        if (mod_opt->type & HFSC_DROPSC)
                fprintf(f, "␣[dc␣␣%s␣%ims␣%s]␣",
                   sprint_rate(mod_opt->pd_curve.m1/8, b1),
535                mod_opt->pd_curve.d,
                   sprint_rate(mod_opt->pd_curve.m2/8, b2));
        if (mod_opt->flags & HFCF_DEFAULTCLASS)
                fprintf(f, "␣default␣");
        if (mod_opt->flags & HFCF_SYNCCLASS)
540             fprintf(f, "␣sync␣");
        if (mod_opt->flags & HFCF_CLASS_ACTIVE)
                fprintf(f, "␣active␣");
        return 0;
};
545
static int hfsc_print_xstats(struct qdisc_util *qu, FILE *f, struct rtattr * xstats)
{
        struct tc_hfsc_xstats * st;

550     if (xstats == NULL)
                return 0;

        if (RTA_PAYLOAD(xstats) < sizeof(*st))
                return -1;
555
        st = RTA_DATA(xstats);
        fprintf(f, "total␣service␣received:␣%Lu␣␣realtime-service␣received:␣%Lu␣\n",
                st->total, st->cumul);

560     fprintf(f, "transmitted:␣%u␣␣dropped:␣%u␣␣queue␣length:␣%u␣␣period:␣%u\n",
                st->xmit_cnt, st->drop_cnt, st->qlen, st->period);

        fprintf(f, "internal␣state␣-␣d:␣%Lu␣␣e:␣%Lu␣␣vt:␣%Lu␣␣k:␣%Lu\n",
                st->d, st->e, st->vt, st->k);
565
 return 0;
};

/*
570 * define interface for tc program:
 */

struct qdisc_util hfsc_util = {
        NULL,
575     "hfsc",
        hfsc_parse_opt,
        hfsc_print_opt,
        hfsc_print_xstats,

580     hfsc_parse_class_opt,
        hfsc_print_class_opt,
};
```

References

[1] Werner Almesberger. Linux traffic control - implementation overview. Technical report, EPFL, Nov 1998. `http://tcng.sourceforge.net`.

[2] Werner Almesberger, Jamal Hadi Salim, and Alexey Kuznetsov. Differentiated services on linux, June 1999. `http://diffserv.sourceforge.net`.

[3] J. C. R. Bennett and H Zhang. Wf2q : Worst-case fair weighted fair queueing. In *Proceedings of INFOCOM '96*, San Francisco, CA, Mar. 1996.

[4] Jon C. R. Bennett and Hui Zhang. Hierarchical packet fair queueing algorithms. *IEEE/ACM Transactions on Networking*, 5(5):675–689, 1997.

[5] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. Rfc2475 - an architecture for differentiated services. Informational Category, Dec. 1998.

[6] R. Braden, D. Clark, and S. Shenker. Rfc1633 - integrated services in the internet architecture: an overview. Informal Category, Jul. 1994.

[7] R. Braden, L. Zhang, S. Bersion, S. Herzog, and S. Jamin. Rfc2205 - resource reservation protocol (rsvp) - version 1 functional specification. Standards Track, Sept. 1997.

[8] Stefan Bucheli, Jay R. Moorman, John W. Lockwood, and Sung-Mo Kang. Compensation modeling for qos support on a wireless network. Technical report, Coordinated Science Laboratory, University of Illinois at Urbana-Champaign, Nov. 2000.

[9] Kenjiro Cho. *Alternate Queuing for BSD Unix (ALTQ) Version 3.0*. Sony Computer Science Laboratories. `http://www.csl.sony.co.jp/ kjc/software.html`.

[10] Sunghyun Choi. *QoS Guarantees in Wireless/Mobile Networks*. PhD thesis, University of Michigan, 1999.

[11] Alan Demers, Srinivasan Keshav, and Scott Shenker. Analysis and simulation of a fair queueing algorithm. In *Proceedings of SIGCOMM '89*, pages 3–12, 1989.

[12] Saman Desilva and Samir R. Das. Experimental evaluation of channel state dependent scheduling in an in-building wireless lan. In *Proceedings of the 7th International Conference on Computer Communications and Networks (IC3N)*, pages 414–421, Lafayette, LA, October 1998.

[13] Jean-Pierre Ebert and Andreas Willig. A gilbert-elliot bit error model and the efficient use in packet level simulation. Technical report, Department of Electrical Engineering, Technical University of Berlin, 1999.

[14] David Eckhardt and Peter Steenkiste. Measurement and analysis of the error characteristics of an in-building wireless network. In *Proceedings of SIGCOMM'96*, pages 243–254, 1996.

[15] Sally Floyd. Notes on cbq and guaranteed service, 1995.

[16] Sally Floyd and Van Jacobsen. Link and resource management models for packet networks. *IEEE/ACM Transactions on Networking*, 3(4), 1995.

[17] Sally Floyd and Van Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, August 1993.

[18] Pawan Goyal, Harrick M. Vin, and Haichen Cheng. Start-time fair queuing: A scheduling algorithm for integrated servicespacket switching networks. Technical Report CS-TR-96-02, Department of Computer Science, The University of Texas at Austin, 1, 1996.

[19] J. Heinanen, F. Baker, W. Weiss, and J. Wroclawski. Rfc2597 - assured forwarding phb group. Standards Track, Jun. 1999.

[20] Bert Hubert, Gregory Maxwell, Remco van Mook, Martijn van Oosterhout, Paul B. Schroeder, and Jasper Spaans. Linux 2.4 advanced routing howto, Apr 2001. `http://www.ds9a.nl/2.4Routing`.

[21] Raylink Inc. Raylink user manual. `http://www.raylink.com/pdf/pccard.pdf`.

[22] V. Jacobson, K. Nichols, and K. Poduri. Rfc2598 - an expedited forwarding phb. Standards Track, Jun. 1999.

[23] Linux sources and gnu general public license. `http://www.kernelnotes.org`.

[24] S. Keshav. On the efficient implementation of fair queueing. *Journal of Internetworking Research and Experience*, 2(3), 1991.

[25] Alexey Kuznetsov. iproute2 package. `ftp://ftp.inr.ac.ru/ip-routing/`.

[26] P. Lin, B. Bensaou, Q.L. Ding, and K.C. Chua. A wireless fair scheduling algorithm for error-prone wireless channels. *Proceedings of ACM WoWMOM 2000*, 2000.

[27] S. Lu, T. Nandagopal, and V. Bharghavan. Design and analysis of an algorithm for fair service in error-prone wireless channels. *Wireless Networks Journal*, Feb. 1999.

[28] Songwu Lu, Vaduvur Bharghavan, and R. Srikant. Fair scheduling in wireless packet networks. *Proceedings of ACM SIGCOMM 1997*, 1997.

[29] Paul E. McKenney. Stochastic fairness queueing. *Journal of Internetworking Research and Experience*, 2:113–131, 1991.

[30] Jay R. Moorman. Supporting quality of service on a wireless channel for atm wireless networks. Master's thesis, University of Illinois at Urbana-Champaign, 1999.

[31] Jay R. Moorman. *Quality of Service Support For Heterogeneous Traffic Across Hybrid Wired and Wireless Networks*. PhD thesis, University of Illinois at Urbana-Champaign, 2001.

[32] Jay R. Moorman, John Lockwood, and Sung-Mo Kang. Wireless quality of service using multiclass priority fair queuing. *IEEE Journal on Selected Areas in Communications*, Aug. 2000.

[33] Jay R. Moorman and John W. Lockwood. Multiclass priority fair queuing for hybrid wired/wireless quality of service support. *WoWMOM 99 - Proceedings of The Second ACM International Workshop on Wireless Mobile Multimedia*, pages 43–50, Aug. 1999.

[34] Christian Worm Mortensen. An implementation of a weighted round robin scheduler for linux traffic control. `http://wipl-wrr.sourceforge.net`.

[35] Thyagarajan Nandagopal, Songwu Lu, and Vaduvur Bharghavan. A unified architecture for the design and evaluation of wireless fair queueing algorithms. *MobiCom'99 - Proceedings of The Fifth Annual ACM/IEEE International Conference on Mobile Computing and Networking*, pages 132–142, Aug. 1999.

[36] T. S. Eugene Ng, Donpaul C. Stephens, Ion Stoica, and Hui Zhang. Supporting best-effort traffic with fair service curve. In *Measurement and Modeling of Computer Systems*, pages 218–219, 1999.

[37] T. S. Eugene Ng, Ion Stoica, and Hui Zhang. Packet fair queueing algorithms for wireless networks with location-dependent errors. *Proceedings of IEEE INFOCOMM*, page 1103, 1998.

[38] T. S. Eugene Ng, Ion Stoica, and Hui Zhang. Packet fair queueing algorithms for wireless networks with location-dependent errors. Technical Report CMU-CS-00-112, School of Computer Science, Carnegie Mellon University, 2000.

[39] David Olshefski. *TC API Beta 1.0*. IBM T.J. Watson Research. `http://oss.software.ibm.com/developerworks/projects/tcapi/`.

[40] Abhay K. Parekh and Robert G. Gallage. A generalized processor sharing approach to flow control in integrated services networks: The single-node case. *IEEE/ACM Transactions on Networking*, 1(3), Jun. 1993.

[41] Pcmcia card services for linux. `http://pcmcia-cs.sourcforge.net`.

[42] Saravanan Radhakrishnan. Linux - advanced networking overview - version 1. Technical report, Information and Telecommunications Technology Center, Department of Electrical Engineering and Computer Science, The University of Kensas, Lawrence, KS 66045-2228, Aug 1999.

[43] Parameswaran Ramanathan and Prathima Agrawal. Adapting packet fair queueing algorithms to wireless networks. In *Mobile Computing and Networking*, pages 1–9, 1998.

[44] E. Rosen, A. Viswanathan, and R. Callon. Rfc3031 - multiprotocol label switching architecture. Standards Track, Jan. 2001.

[45] Rusty Russell. Linux 2.4 packet filtering howto. `http://netfilter.samba.org`.

[46] S. Shenker, C. Partridge, and R. Guerin. Rfc2212 - specification of guaranteed quality of service. Standards Track, Sept. 1997.

[47] Scott Shenker, David D. Clark, and Lixia Zhang. A scheduling service model and a scheduling architecture for an integrated services packet network preprint. http://citeseer.nj.nec.com/shenker93scheduling.html, 1993.

[48] M. Shreedhar and George Varghese. Efficient fair queueing using deficit round robin. In *Proceedings of SIGCOMM '95*, pages 231–242, 1995.

[49] M. Srivastava, C. Fragouli, and V. Sivaraman. Controlled multimedia wireless link sharing via enhanced class-based queueing with channel-state-dependent packet scheduling. In *Proceedings of the Conference on Computer Communications (IEEE Infocom)*, pages 572–, April 1998.

[50] W. Richard Stevens. *TCP/IP Illustrated: The Protocols*. Addison-Wesley Publishing Company, 1994. ISBN 0-201-63346-9 (v.1).

[51] Ion Stoica, Hui Zhang, and T.S. Eugene Ng. A hierarchical fair service curve algorithm for link-sharing, real-time and priority service. In *Proceedings of SIGCOMM '97*, 1997. `http://www.cs.cmu.edu/~zhang/HFSC/`.

[52] Sun java 2 platform version 1.3 (standard edition). `http://java.sun.com/j2se/1.3/`.

[53] Agere Systems. User's guide for orinoco pc card, Sept. 2000. `ftp://ftp.orinocowireless.com/pub/docs/ORINOCO/MANUALS/ug_pc.pdf` .

[54] Jean Tourrilhes. *Linux Wireless Extension/Wireless Tools*. Hewlett Packard. `http://www.hpl.hp.com/personal/Jean_Tourrilhes/Linux/Tools.html`.

[55] Zheng Wang. *Internet QoS: Architectures and Mechanisms for Quality of Service*. Morgan Kaufmann Publishers, 2001.

[56] A. Willig, M. Kubisch, and A. Wolisz. Bit error rate measurements – second campaign, longterm1 measurement. TKN Technical Report Series TKN-00-009, Telecommunication Networks Group, Technical University Berlin, November 2000.

[57] A. Willig, M. Kubisch, and A. Wolisz. Bit error rate measurements – second campaign, longterm2 measurement. TKN Technical Report Series TKN-00-010, Telecommunication Networks Group, Technical University Berlin, November 2000.

[58] Lars Wischhof. Modified h-fsc scheduler and linux wireless scheduling patch. `http://wsched.sourceforge.net`.

[59] J. Wroclawski. Rfc2211 - specification of the controlled-load network element service. Standards Track, Sept. 1997.

Index