# Packet Scheduling for Bandwidth Sharing and Quality of Service Support in Wireless Local Area Networks

Diplomarbeit
am Lehrstuhl für Telekommunikationsnetze
Prof. Dr.-Ing. Adam Wolisz
Institut für Telekommunikationssysteme
Fakultät IV (Elektrotechnik und Informatik)
Technische Universität Berlin

von

**Lars Wischhof**
Matrikelnummer: 176815

Betreuer:

Prof. John W. Lockwood
Applied Research Laboratory
Department of Computer Science
Washington University in St. Louis, USA

Dipl.-Ing. Ana Cristina Costa Aguiar
Lehrstuhl für Telekommunikationsnetze
Institut für Telekommunikationssysteme
Technische Universität Berlin

# 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 thesis, 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-known 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 Medium Access Control (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.

## Zusammenfassung

Durch die wachsende Verbreitung von drahtlosen lokalen Netzwerken, die steigende Zahl von Anbietern drahtloser Internetzugänge sowie die Forderung, unterschiedliche, datentypspezifische Dienstqualitäten zu unterstützen, entsteht ein Bedarf nach Verfahren, welche das kontrollierte, gemeinsame Nutzen eines drahtlosen Kanals durch verschiedene Parteien und/oder Datenklassen erlauben. Innerhalb dieser Arbeit wird ein Ansatz entwickelt, der über einen längeren Zeitraum gesammelte Informationen über den Zustand des drahtlosen Kanals zu einer bestimmten Mobilstation nutzt, um die Verteilung der verfügbaren Bandbreite zu optimieren.

Dieser Algorithmus erlaubt es dem Betreiber eines drahtlosen Netzwerkes, die gewünschte Bandbreitenverteilung mit Hilfe von nutzer- und betreiberorientierten Kriterien zu spezifizieren und die für einen Nutzer aufgewendeten Ressourcen den von ihm getragenen Kosten anzupassen. Um die Anwendung dieses Ansatzes zu untersuchen, wird er in einen der bekannten Algorithmen, den Hierarchical Fair Service Curve Algorithm (H-FSC), integriert. Simulationen zeigen, dass der modifizierte H-FSC Algorithmus die kontrollierte Nutzung des drahtlosen Kanals verbessert, ohne Änderungen der Medienzugriffskontrollschicht (MAC Schicht) zu erfordern.

Zudem werden Design und Implementation eines Linux-basierten Prototypen entwickelt. Messungen innerhalb eines entsprechenden Testbetts demonstrieren, dass der vorgeschlagene Ansatz mit auf dem IEEE 802.11 Standard basierender Hardware verwirklicht werden kann.

## Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Diplomarbeit selbstständig und ohne unerlaubte Hilfe angefertigt, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Berlin, den 29. Januar 2002

Lars Wischhof

# Contents

# List of Figures

# List of Tables

# Listings

# 1. Introduction

In the last 10 years the usage of data services has increased tremendously: In 1991 less than 700,000 Internet hosts were registered, in September 2001 this number had increased to 123.3 million. From being a research project and medium for scientists, 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 medium 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), one 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 [54] 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 behavior of the algorithm is verified using simulations and by prototype results.

## 1.2 Organization of this Thesis

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 the simulations are in Chapter 7, and measurements performed using the Linux-based prototype within a small wireless testbed are presented in Chapter 8. The main part concludes with a summary in Chapter 9.

Additional information about implemented testbed tools and testbed configurations are included in Appendix A. The various source code files of the prototype implementation are described in Appendix B.

# 2. QoS Architectures

This chapter gives a brief overview of some 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 detailed overview can be found e.g. in [62].

## 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 [51] provides strict delay bounds and guaranteed bandwidth, whereas the Controlled Load Service [69] 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 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.

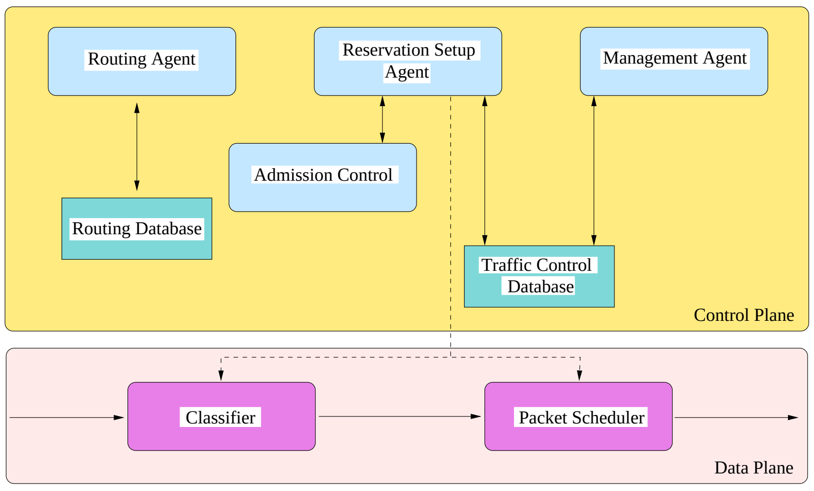


**Figure 2.1:** *Integrated Services reference model. [6], [62]*

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 class of the packet 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 re-marked 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.

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)* [21] and *Expedited Forwarding (EF)* [24]. 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.

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

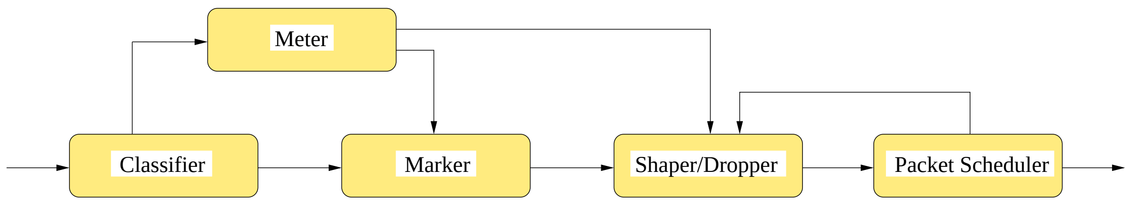


**Figure 2.2:** *Differentiated Services traffic classification and conditioning components.*

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) [49] 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 is that the capacity of the link 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.

[1]Note that we do not consider the position of the QoS scheduler yet – it could e.g. be part of the network protocol stack of the operating system or part of the Link Layer mechanism implemented within the firmware of the network interface itself. This issue will be considered in Chapter 4.

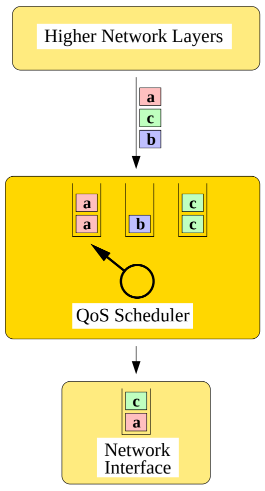


**Figure 3.1:** *Simple downlink scheduling model.*

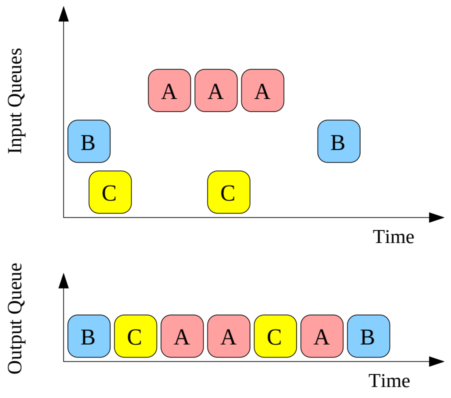


**Figure 3.2:** *Example for FIFO/FCFS scheduling.*

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

A simple scheduling scheme 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 example for scheduling traffic from three different sessions in FIFO/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 real-time connection like voice traffic can be starved by a high-throughput data transfer as shown in Figure 3.3. 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 Served (LCFS) and Strict Priority. In LCFS the packet which arrived last is served first, whereas in Strict Priority it

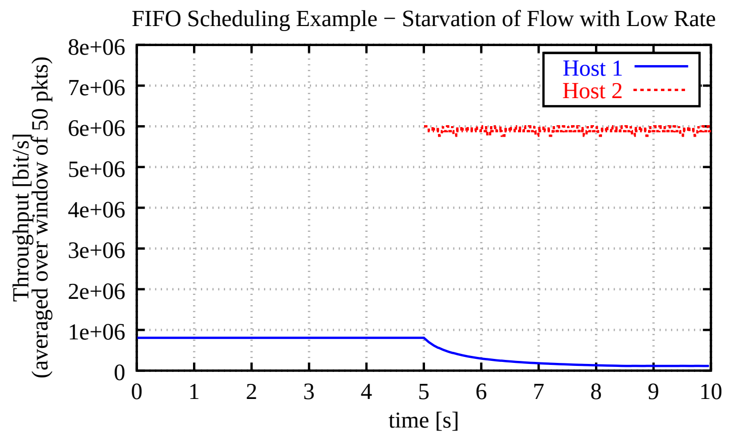


**Figure 3.3:** *A flow of Host 1 which is sent at a low rate is starved by a high-bandwidth flow of Host 2 which starts being transmitted 5 seconds later.*

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) [45] 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 $i$ is assigned a weight $\phi_i$ 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[4]

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

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 $B(\tau_1, \tau_2) \subset \{1, 2, \dots, N\}$ of all flows backlogged, the remaining capacity will be distributed proportional to their weights $\phi_i$, which leads to an increased service share $g_i^{inc}$ for each backlogged flow:

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

[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 $\tau$ if there are packets stored in the queue of the flow.

[4]Note that for Equation 3.2 only flow $i$ is assumed to be continously backlogged within $(\tau_1, \tau_2]$ – flow $j$ is allowed to have periods without demand for service.

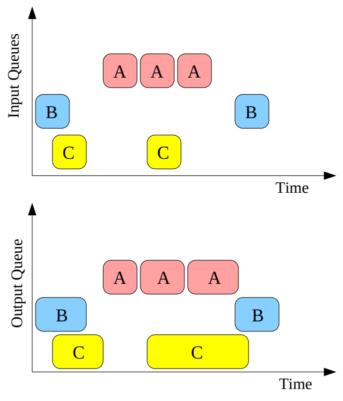


**Figure 3.4:** *Example for GPS scheduling with* $\phi_A = 0.5, \phi_B = 0.25, \phi_C = 0.25$.

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

- It is *work conserving*.[5]
- 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 $\phi_i$ to unimportant background traffic, most of the channel capacity can be used by flows of higher priority (and therefore with a weight $\phi_j \gg \phi_i$) if they are backlogged, while the remaining capacity will be used by the background traffic.
- The maximum delay $D_i^{max}$ a flow $i$ experiences 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 [45], [53] 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

[5]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.

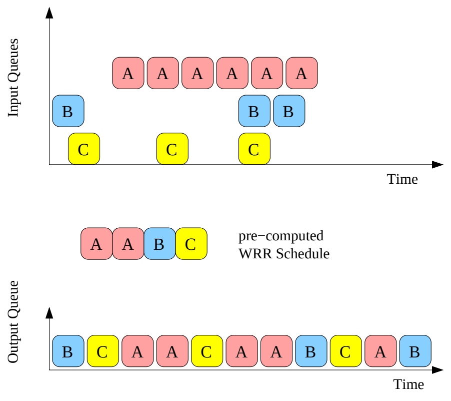


**Figure 3.5:** *Example for WRR scheduling with* $w_A = 2, w_B = 1, w_C = 1$.

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.

### 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) [33] 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 may be treated unfairly – therefore the fairness guarantees of STFQ 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 [53] 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, a variant called DRR+ was introduced [53] which divides the flows into 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 by more than one packet of every other latency critical flow.

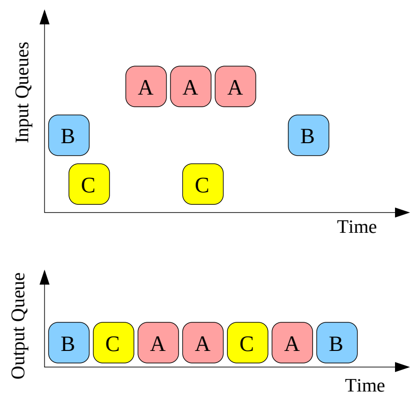


**Figure 3.6:** *Example for WFQ scheduling with* $\phi_A = 0.5, \phi_B = 0.25, \phi_C = 0.25$ *assuming a constant packet length* $L_p$. *Although the packets would have finished in a different sequence (B,C,A,A,A,C,B) under ideal GPS service, no packet is delayed more than* $\frac{L_{max}}{r}$ *time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

### 3.1.6 Weighted Fair Queuing (WFQ)

Another often cited scheduling scheme which approximates GPS on a packet level is known as Weighted Fair Queuing (WFQ) or Packet-by-packet Generalized Processor Sharing (PGPS) [45]. It can also be seen as the weighted version of a similar algorithm developed based on Bit-by-bit Round robin (BR) as Fair Queuing [12], [26].

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.
3. Thus, the server cannot be work conserving and processes 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 $i$:

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

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 $\phi_i$, 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$^2$Q) 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) [19], 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 properties and a 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.

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

$$F(p_j^f) = \begin{cases} S(p_j^f) + \frac{L(p_j^f)}{r_f} & \text{if } j \geq 1 \\ 0 & \text{if } j = 0 \end{cases} \tag{3.8}$$

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) [3], which is neither ahead nor 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 a WF$^2$Q flow is not ahead of the corresponding GPS flow by more than one packet of maximum packet size. 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:

$$V_{WF^2Q+}(t+\tau) = \max(V_{WF^2Q+}(t) + W(t,t+\tau), \min_{i \in \hat{B}(t)} (S_i^{h_i(t)})) \tag{3.9}$$

in which $W(t,t+\tau)$ denotes the total amount of service in the period, $S_i^{h_i(t)}$ is the start tag of the head-of-line packet of queue $i$, and $\hat{B}(t)$ is the set of backlogged flows. Since at least one packet has a virtual start time lower than 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 ($S_i$ and $F_i$) 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.

With these modifications the algorithm is still able to achieve the same worst-case fairness and bounded delay properties as WF$^2$Q [4].

## 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 [63]:

- location dependent errors as illustrated in Figure 3.7 (e.g. because of multipath propagation)
- higher probability of transmission errors/error bursts (caused by noise and interference on the radio channel)
- 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 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 [14] 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*.

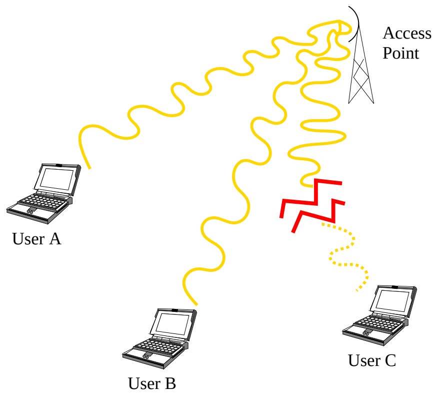


**Figure 3.7:** *Location dependent errors on the wireless channel. While users A and B have the full link capacity available, user C experiences an error burst.*

### 3.2.1 Idealized Wireless Fair Queuing (IWFQ)

One adaptation of WFQ to handle location-dependent error bursts is Idealized Wireless Fair Queuing (IWFQ) [31], [39]. In IWFQ the error-free fluid service (GPS) for the flows is simulated as in WFQ. Each arriving packet is assigned a start tag and a finish tag as in SFQ (Section 3.1.7). The virtual time $v(t)$ is being computed from the set of backlogged flows $B(t)$ of the error-free service in the following way:

$$\frac{dv(t)}{dt} = \frac{C}{\sum_{i \in B(t)} r_i} \tag{3.10}$$

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.)

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.[6]

2. A flow $i$ can lead by a maximum of $l_i$ bits. If the head of line packet (of length $L_{i,hol}$) has a start tag $s_{i,hol}$ more than $\frac{l_i}{r_i}$ in the future, it is adjusted to

$$s_{i,hol} = v(t) + \frac{l_i}{r_i}, \quad f_{i,hol} = s_{i,hol} + \frac{L_{i,hol}}{r_i} \tag{3.11}$$

[6]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.

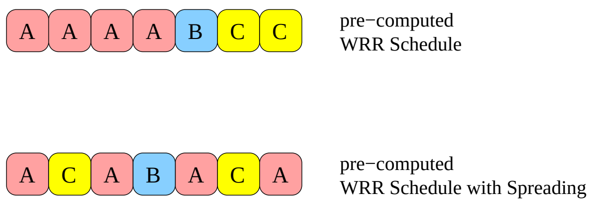

A A A A B C C
pre−computed WRR Schedule
A C A B A C A
pre−computed WRR Schedule with Spreading

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

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

$$d_{max,IWFQ} \leq d_{max,WFQ} + \frac{B}{C} \tag{3.12}$$

### 3.2.2 Wireless Packet Scheduling (WPS)

Wireless Packet Scheduling (WPS) [31] 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:

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/debit 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 that 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 [41], [42].

It was developed to provide

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

- long-time fairness for sessions with bounded channel error
- 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.*[7] So in any case the virtual time of the selected flow is advanced in SFQ manner (Equation 3.8).

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 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. Therefore, the sum over the lag parameters of all flows is alway zero: $\sum_i lag_i = 0$.

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-$\alpha$) 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 the corresponding reference system.
2. Advance the virtual time for flow *i*.
3. If the flow is leading and has given at least a share of $\alpha$ percent for compensation of lagging flows or it is in-sync[8]/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 rates of the flows.
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 $\sum_i lag_i = 0$ 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 experience in an error system.

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

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

### 3.2.4 Wireless Fair Service (WFS)

A different algorithm that also supports the CIF properties is Wireless Fair Service (WFS) [30], 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* 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 $\Phi_i$. 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 $F(p_i^k)$

$$S(p_i^k) = \max(V(A(p_i^k)), S(p_i^{k-1}) + \frac{L_i^{k-1}}{r_i} \tag{3.13}$$

$$F(p_i^k) = S(p_i^k) + \frac{L_i^k}{\Phi_i} \tag{3.14}$$

where $V(A(p_i^k))$ is the virtual time when packet $p_i^k$ arrives and $L_i^{k-1}$ is the length of the $(k-1)$th packet of flow $i$. For the virtual time the same notion as in IWFQ (Section 3.2.1, Equation 3.10) is used.

Of all slots whose start tag is not ahead of the current virtual time by more than the schedulers lookahead parameter[9] 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 $\Phi_i$.

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 flows 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.[10] 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 an 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.
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.

[9]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 $\infty$ then the selection mechanism is equal to Earliest Deadline First (EDF).

[10]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 [30].

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$^2$F$^2$Q)

As the name suggests, the W$^2$F$^2$Q algorithm is an adaptation of Worst-case Fair Weighted Fair Queuing plus (W$^2$FQ+, see Section 3.1.9) for the wireless environment. It uses the same notion of virtual time as WF$^2$Q+ (Equation 3.9), which avoids the need to simulate a corresponding GPS reference system. In addition to the compensation inherited from WF$^2$Q+, which prefers flows recovering from an error-period since they have lower start tags, W$^2$F$^2$Q uses a wireless error compensation consisting of the following components:

1. **Readjusting:** If $L$ denotes the length of a packet and $\phi_i$ the weight of flow $i$, each packet with a start tag more than $\frac{L}{\phi_i} + \delta_{i,max}$ ahead of the current virtual time $V(t)$ is labeled with a start tag of $V(t) + \frac{L}{\phi_i} + \delta_{i,max}$ (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 $V(t) - \frac{L}{\phi_i} + \gamma_{i,max}$ (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$^2$F$^2$Q uses an exponential decrease. A leading flow which is more than one packet transmission length $\frac{L}{\phi_i}$ ahead will yield $(1-\alpha)$ of its service share to non leading flows. The parameter $\alpha$ depends on the number of lagging flows:

$$\alpha = \frac{1-\alpha_{min}}{e-1} e^{1-x} + \frac{e\alpha_{min}-1}{e-1} \tag{3.15}$$

   where $x$ is the relation of combined weights of the flows in-sync to that of the flows not in-sync: $x = \frac{\sum_{\text{j in-sync}} \phi_j}{\sum_{\text{j not in-sync}} \phi_j}$.

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. This is done in proportion to their weight as in CIF-Q (Section 3.2.3), with the exception that the start tag for each packet is decreased, instead of recalculating a *lag* variable.

### 3.2.6 Server Based Fairness Approach (SBFA)

A generalized approach for adapting wireline packet fair queuing algorithms to the wireless domain was developed as Server Based Fairness Approach (SBFA) [48]. 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 a 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 [29]. Whereas the schemes presented before use a two-state one-step-prediction for the channel-state and try to maximize throughput for users experiencing a “good” channel,[11] 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 $\phi_i$ as in most wireline schedulers 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) as long as the channel quality results in a $C_i(t)$ better or equal to $C_i^{th}$. The time-varying service share $\psi_i$ is computed as

$$\psi_i = \begin{cases} \frac{\phi_i}{C_i(t)} & \text{if } C_i(t) \geq C_i^{th} \\ \frac{\phi_i}{C_i^{th}} & \text{otherwise} \end{cases} \tag{3.16}$$

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 [34], [36], [37] and later improved with a wireless compensation mechanism in [8] and [35], 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.

[11]This is referred to as *totalitarian situation* in [29] in contrast to an “egalitarian system” which tries to provide equal goodput for each user.

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

**Table 3.1:** *MPFQ mapping of ATM classes to priorities/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 $maxCTD$ of the flow.[12]

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 the delay requirements are not that tight for these classes. The WRR algorithm serves the flows according to their Minimum Cell Rate (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 reinserted 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.

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

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 reinsertion of the slot which could not be transmitted in the recirculating 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$^2$Q | GPS ref. | yes | - | - | - | - | high | high | - |
| WF$^2$Q+ | 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$^2$F$^2$Q | 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 |

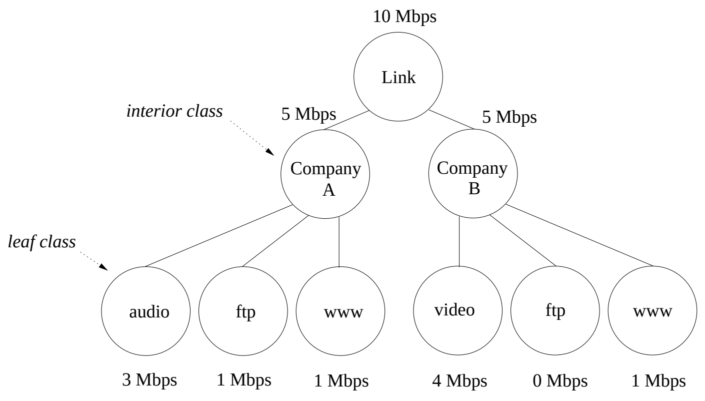


**Figure 3.9:** *Link-sharing between two companies and their traffic classes.*

## 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[13] – 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) [17] 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. This is done by calculating an exponentially 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 specified rate of the class.

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 company's share the bandwidth is distributed to

[13]In the following, the term *class* is used for a set of (micro-)flows sharing a specific property. A link-sharing class does not necessarily correspond to a specific QoS class, e.g. in Figure 3.9 the "Company A" class contains flows with various different QoS requirements but all share the property that they are to/from hosts of Company A.

| 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 shown in Table 3.3.

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*[14] 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*.

   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

[14]Actually the listed guidelines are called *alternate link-sharing guidelines* in [17] since they are a variant of the formal link-sharing guidelines which avoids oscillation of a class between the regulated and unregulated state.

- 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 [16], [17]. 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 (CBQ+CSDPS)

In [56] 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 (CSDPS). 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 modification of the CBQ algorithm is that a restricted class is allowed to transmit even if unsatisfied classes exist, in case that 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.

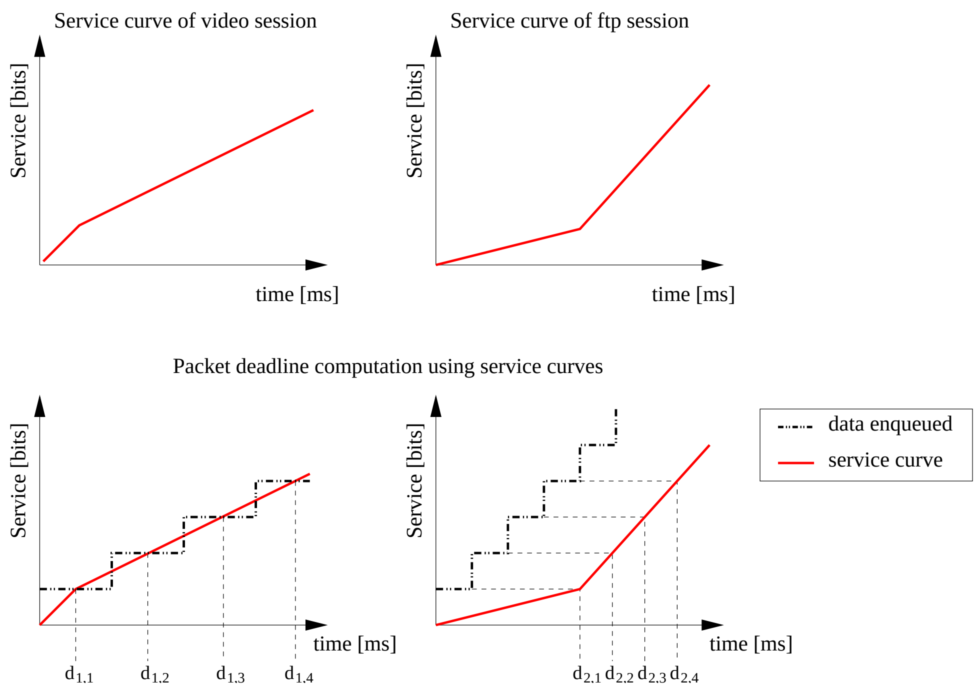


**Figure 3.10:** *An example for a service curve of a video and a FTP session ($d_{ij}$ denotes the deadline of the $j$-th packet of flow $i$). The (interactive) video session requires a low delay at a moderate rate, whereas the FTP session is guaranteed a high rate but with a higher delay. [58]*

## 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 [58]. H-FSC[15] 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

$$S_i(t_2 - t_1) \leq w_i(t_1, t_2); \;\; t_1 < t_2 \tag{3.17}$$

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[16] 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:

$$S(t) \geq \sum_i S_i(t) \tag{3.18}$$

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 class which received excess

[15]This algorithm is presented in more detail because it serves as basis for the wireless algorithm developed in Chapter 5.

[16]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.

service will not be punished later (fairness properties). However, in their paper [58] 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, 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 $c_i$, which is 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 3.19, which by taking into account only the service received under real-time criterion does not punish a session for using excess service.

$$D_i(a_i^k;t) = \min(D_i(a_i^{k-1};t), S_i(t - a_i^k) + c_i(a_i^k)); \quad t \geq a_i^k \tag{3.19}$$

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[17]

$$E_i(a_i;t) = D_i(a_i;t) + [\max_{\acute{t}>t}(D_i(a_i;\acute{t}) - D_i(a_i;t) - S_i(\acute{t} - t))]^+; \quad t \geq a_i \tag{3.20}$$

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

$$V_i(a_i^k;v) = \min(V_i(a_i^{k-1};v), S_i(v - v_{p(i)}^s(a_i^k)) + w_i(a_i^k)); \quad v \geq v_{p(i)}^s(a_i^k) \tag{3.21}$$

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.

[17]... where $[x]^+$ denotes $\max(x,0)$.

In [40] 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 (e.g. 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. 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 quality of the wireless link is time varying and the error probability 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 capacity of the wireless link. While this 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.[2]

## 4.1 Problem Example

In order to illustrate the problem, this section presents a simplified example scenario, which will also be used later on to show the benefits of our approach: An Internet Service

[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] 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 [64], [65] and are best handled by MAC layer retransmissions.

Provider (ISP) offers services using currently available (e.g. IEEE 802.11) technology on a wireless link with a total capacity[3] of 6 Mbit/s. The ISP has installed an access point (AP) in order to serve two different companies: Company A has bought 85% of the capacity of the AP 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 (downlink) capacity is to be enforced at the access point using a link-sharing scheduler.

In the following, an arbitrary variable $g_i$ is assumed which is proportional to the quality of the wireless link to a mobile station $i$ (a formal definition follows in the next section). The maximal value of $g_i$ is 1, which corresponds to a perfect link, and its minimal value is 0, meaning that the correct transmission of information is impossible on this link with the used wireless technology. In general, a decreasing link quality means that the rate at which goodput can be sent to a mobile station is reduced. This can, for example, be caused by a higher probability for corrupted packets or by a physical layer which chooses a more robust modulation in order to adapt to the link quality.

For a link-sharing scheduler implemented above the MAC layer, the only visible effect, if $g_i$ for a specific mobile station $i$ decreases, is a reduction of the *total* system goodput. The reason is that it is not aware of details as the modulation technique or average number of retransmissions for a mobile station. Therefore the total system goodput will be shared by the mobile stations in a fixed ratio, which is independent of the link qualities $g_i$ for the different mobile stations. In the example scenario 85% of the total system goodput would be scheduled for MS 1 and 15% for MS 2.

If the total amount of resources per time unit available at the access point is assumed to be a constant value (e.g. equal to the capacity of the AP), a decrease of $g_i$ causes an increase of resources needed to transmit an unit of goodput. This leads to unfairness in terms of costs per used resource as soon as one assumes different values of $g_i$ for each mobile station $i$.

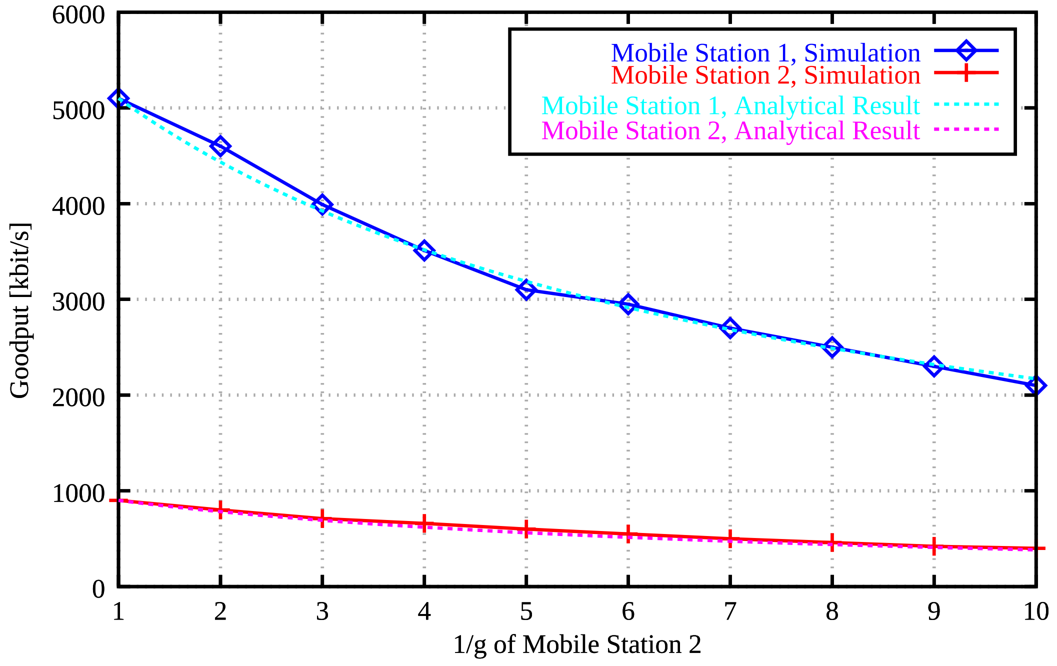
Figure 4.1 shows the results of a simulation[4] of the example scenario, where the quality of the wireless link to MS 2 $g_{ms2}$ varies while MS 1 has a constant value of $g_{ms1} = 1$. As shown in Figure 4.1(b), the relative share of MS 2 of the resources increases and the goodput for *both* mobile 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.

## 4.2 Ratio of Goodput to Consumed Resources

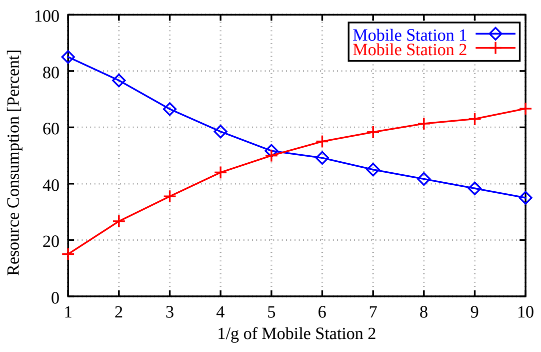
In order to be able to describe the effects of varying resource consumption in wireless LANs, the terms used in this context will be defined in the following. Only the downlink scheduling within an AP to $N$ mobile stations is considered. (Usually a user works on exactly one mobile station, therefore the terms *user* and *mobile station* are used synonymously.)

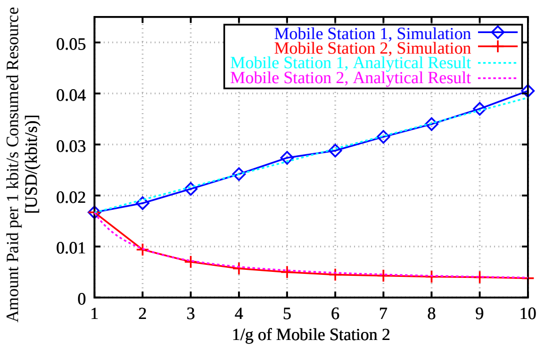
[3]Although this is an arbitrary value, it is close to the maximal amount of goodput one usually achieves using 11 MBit/s 802.11b WLAN cards.

[4]The results were obtained using a simulation of Class-Based Queuing[17] with a separate class for each company. For more details on the simulation environment used refer to Chapter 6. Furthermore, an analytical description of the observed behavior will be given in Section 4.3.



(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.*

Since a wireless link layer correcting the majority of the errors is assumed, the term *goodput* is used for all data seen above the link layer. The rate of goodput to a mobile station $i$ at time $t$ is denoted by $b_{good,i}(t)$, and the total goodput rate $B_{good}(t)$ for an AP is:

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

As mentioned, the amount of resources available at the access point is assumed to be constant, e.g. determined by the width of the frequency band reserved. Depending on the used wireless technology, a certain maximum rate at which the AP can send data to the mobile stations (given a perfect wireless channel) corresponds to this amount of resources. This maximum rate is called *raw throughput* $B_{raw}$ of the AP. Each mobile station has a time varying share $b_{raw,i}(t)$ of the constant total raw throughput:

$$B_{raw} \geq \sum_{1 \leq i \leq N} b_{raw,i}(t) \tag{4.2}$$

In an overload condition all resources are consumed and therefore the sum of the individual shares is equal to $B_{raw}$.

One can now define a *goodput to raw throughput ratio (GTR)* for an active mobile station $i$ as the amount of goodput achieved at time $t$ with its share of raw throughput:

$$g_i(t) = \frac{b_{good,i}(t)}{b_{raw,i}(t)} \tag{4.3}$$

This ratio enables a scheduler to combine resource-based and goodput-based scheduling: The idea is that an entity, the *Wireless Channel Monitor*, which is either part of the data link layer, the lower OS layers or even the scheduler itself (a possible implementation is presented in Chapter 8), keeps a windowed average of the raw throughput needed per byte of goodput for each mobile station. The GTR is then used by the scheduler in order to estimate the needed resources when handing data for a specific destination to the lower layers.

The advantage of this approach[5] is that it allows an unified consideration of the effects of various data link and physical layer techniques as:

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

## 4.3 Purely Goodput-Based Wireless Link Sharing

This section presents a brief mathematical analysis of the observed behavior in the example scenario (Section 4.1). Most implementations of a link-sharing scheduler for CBQ

[5]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.

are based on the DRR algorithm (Sections 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 $w_i$ for the DRR 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 downlink bandwidth (in terms of goodput) and the resources necessary to transmit to different mobile stations vary. 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. Then Equation 4.4 must hold at any point in time where active classes exist, since the CBQ scheduler is work conserving and distributes the available bandwidth completely to the backlogged classes.

$$\sum_{1\leq i\leq N} \frac{b_{good,i}(t)}{g_i(t)} = B_{raw} \tag{4.4}$$

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[6] $w_i$ of the $N$ classes. Thus, if a class $i$ is in the set of active classes $\mathcal{A}(\mathrm{t})$, its share of goodput at time $t$ is:

$$b_{good,i}(t) = \frac{w_i}{\sum_{j\in\mathcal{A}(\mathrm{t})} w_j} \cdot B_{good}(t); \quad i \in \mathcal{A}(\mathrm{t}) \tag{4.5}$$

Since a passive class does not consume any resources, one can use Equation 4.5 to substitute $b_{good,i}(t)$ in Equation 4.4:

$$B_{good}(t) \cdot \sum_{i\in\mathcal{A}(\mathrm{t})} \frac{w_i}{g_i(t)\cdot\sum_{j\in\mathcal{A}(\mathrm{t})} w_j} = B_{raw} \tag{4.6}$$

The total goodput rate can therefore be computed if $B_{raw}$ and the individual GTRs $g_i(t)$ of the mobile stations are known:

$$B_{good}(t) = \frac{B_{raw}}{\sum_{i\in\mathcal{A}(\mathrm{t})} \frac{w_i}{g_i(t)\cdot\sum_{j\in\mathcal{A}(\mathrm{t})} w_j}} \tag{4.7}$$

With Equation 4.5, the goodput scheduled for a single, active mobile station $i$ is:

$$\begin{aligned} b_{good,i}(t) &= \frac{w_i}{\sum_{j\in\mathcal{A}(\mathrm{t})} w_j} \cdot \frac{B_{raw}}{\sum_{k\in\mathcal{A}(\mathrm{t})} \frac{w_k}{g_k(t)\cdot\sum_{j\in\mathcal{A}(\mathrm{t})} w_j}}; \quad i \in \mathcal{A}(\mathrm{t}) \\ &= w_i \cdot \frac{B_{raw}}{\sum_{k\in\mathcal{A}(\mathrm{t})} \frac{w_k}{g_k(t)}} \end{aligned} \tag{4.8}$$

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 at time $t$ 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.

[6]As mentioned in Section 3.1.3, each class has an integer weight $w_i$ and is guaranteed a share of $\frac{w_i}{\sum_j w_j}$ of the total rate.

| 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.*

**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 capacity, 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. an user will be guaranteed a rate of 64 kbit/s as long as his signal to noise ratio (SNR) is above a specified value. After this point, the scheduled goodput rate for him will be reduced in order to limit his consumption of the network resources.

## 4.4 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.

**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 station using a specific service. Table 4.1 lists the service curve parameters for the leaf classes. Two piece-wise linear service curves [58] 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.

### 4.4.1 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

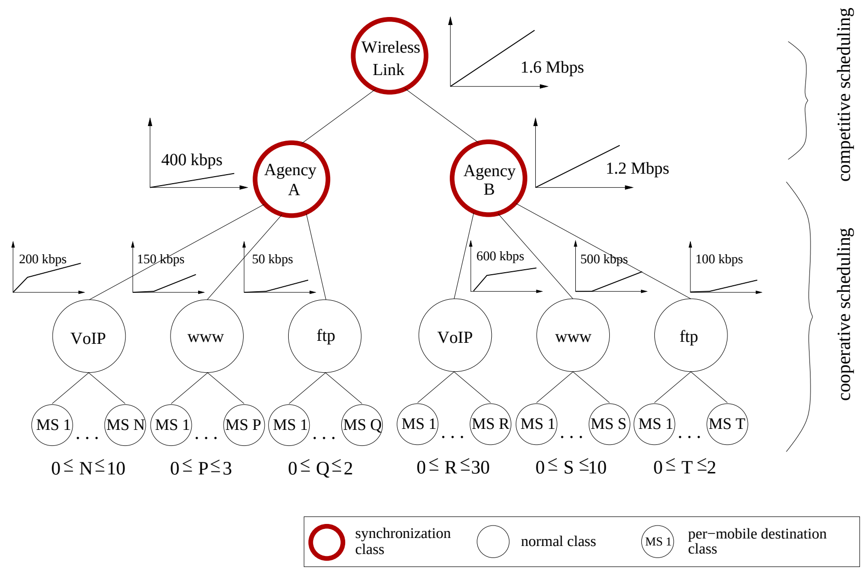


**Figure 4.2:** *Example for a wireless link-sharing hierarchy.*

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. If the queues of both stations are constantly backlogged, this leads 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 any 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 independently of the GTR of a class.

### 4.4.2 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 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 to the Hierarchical Fair Service Curve Algorithm (see Section 3.4.3).

Although the proposed approach of combining resource- and goodput-based scheduling within one wireless scheduling hierarchy using *synchronization classes* could be applied to many scheduling schemes (e.g. CBQ or H-PFQ), H-FSC was chosen as base algorithm for the following reasons:

- It is designed using a formal approach (e.g. in contrast to CBQ).
- It offers better and stronger real-time guarantees and more accurate link-sharing than any other hierarchical scheduler [58].
- The service curve approach decouples the allocation of delay and bandwidth resources in a more elegant way than a static priority scheme.
- Various other projects (e.g. DARWIN [28], Router Plugins [11], ALT-Q [9]) have successfully tested H-FSC schedulers in wired networks.

In the following, 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.[1] Finally, a simple way to support dropping of outdated packets for real-time classes is shown.

## 5.1 Cooperative Scheduling

As defined in Section 4.4.1, 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)

[1]Note that it is not possible to completely avoid overload situations if a high utilization of the medium is demanded because of the random nature 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.

in situations where enough resources are available (Equation 3.18 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.[2] Therefore, compared to an unmodified H-FSC scheduler 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, analogous 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:

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

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.

[2]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 thesis but could be used in order to be able to give tighter delay guarantees.

## 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 subclass 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 $\tau_{max}$, 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 7.3.1.)

## 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 is able to drop 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: First, Section 6.1 briefly gives the necessary background information about 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 wireline 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 6.3 and 6.4.

## 6.1 The Linux Traffic Control Framework (TC)

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

[1]The GNU General Public License is published by the Free Software Foundation. License and Linux sources can be downloaded at [25].

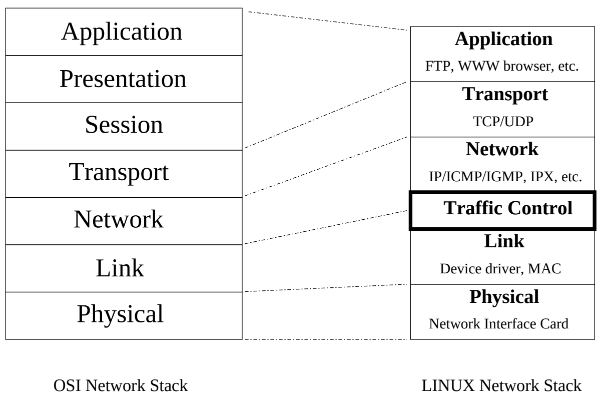


**Figure 6.1:** *Linux networking and the OSI reference model.*

Most parts of the kernel 2.4.13 Traffic Control (including Integrated Services and RSVP) were written by Alexey Kuznetsov [27], 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 [57] 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. This way, 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 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. In addition, a special ingress queuing discipline can be used to limit the maximum rate at which packets of each class are received – this can be used e.g. to protect the host against various types of Denial of Service (DoS) attacks. 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), policed (e.g. it could be assigned a lower priority if a certain rate is exceeded), 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 [47], the DiffServ components are presented in [2].

### 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

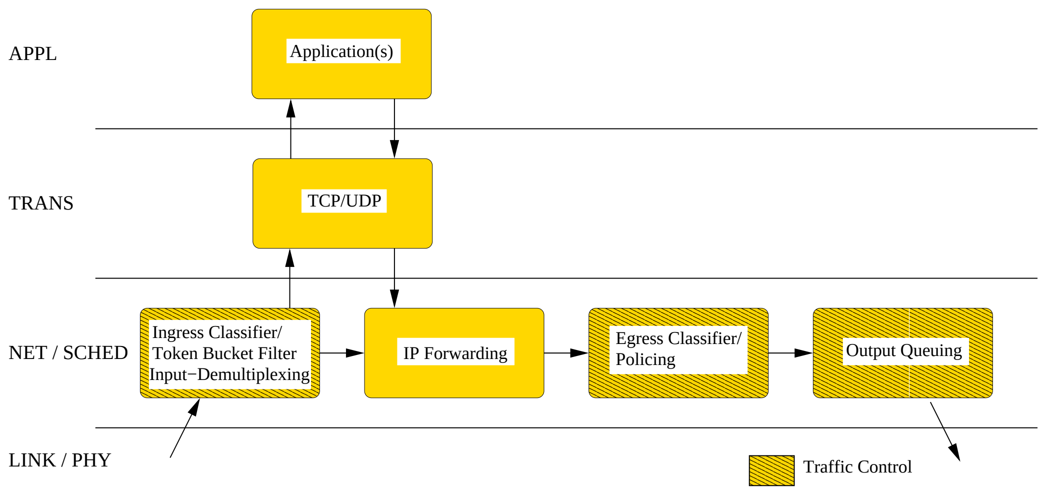


**Figure 6.2:** *Parts in the network stack where traffic control is involved.*

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
- filters

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 Traffic Control program `tc`), which illustrates the ways in which the different components can be combined.

#### 6.1.1.1 Queuing Disciplines

Queuing Disciplines (*qdiscs*) are kernel modules, which 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 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.[2] 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

[2]Therefore, the term *class* in Linux TC is more general than in the context of hierarchical link-sharing (Chapter 3). Basically, the only precondition is that it is possible to implement a filter module, which identifies the packets of a class.

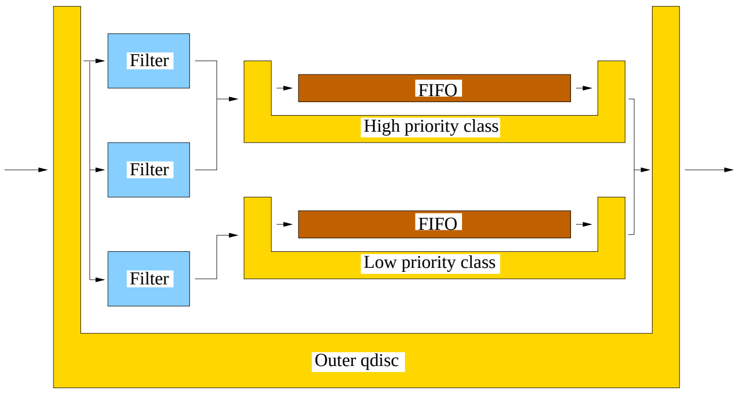


**Figure 6.3:** *Simple example of a qdisc which has inner classes, filters and qdiscs*

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.[3]

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 Section 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.[4]
- **red:** Implements the Random Early Detection (RED) [18] 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.
- **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.

[3]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.

[4]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.

- **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 [52] 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[5] 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.[6] All methods which are sending/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 *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.

[5]This queuing discipline is not part of the standard kernel but can be downloaded at [38].

[6]This is a simple way to approximate an object oriented design using standard C.

| 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.*

| 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.*

#### 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:

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 [50].
   - **cls_route:** The packet is classified using routing table information [22].
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. These functions are listed in Table 6.3. 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 to the structure with filter operations.

### 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.1.)

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.

| Name | Description |
|---|---|
| `*next` | Pointer to the next filter in the list. |
| `*id` | 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.*

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 [27]. Because of space limitations, the detailed usage of `tc` will not be explained as part of this thesis (an interested reader will find more information in [22]), 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 Wireline 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 ISP of the company 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):

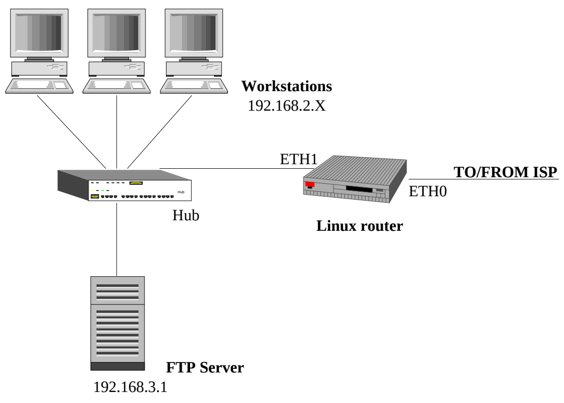


**Figure 6.4:** *Example network for wireline bandwidth sharing.*

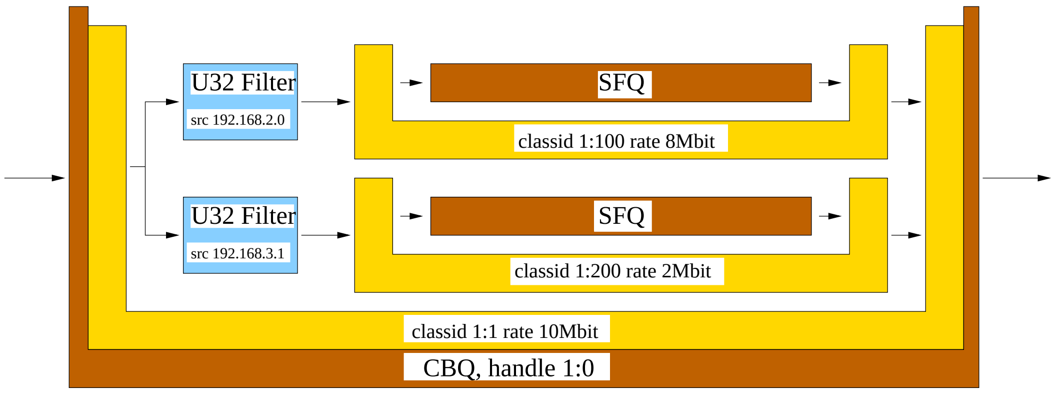


**Figure 6.5:** *The configuration of network interface eth0 in the example network.*

```
# 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 an 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 very 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* [44].[7] It provides a library which enables an 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), IBM hopes 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 wireline scheduling scenarios (e.g. IntServ, DiffServ, MPLS, etc.), and many wireline 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 about 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.

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 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 are:

- Support for the implementation of a wide range of wireless scheduling algorithms.

[7]Version Beta 1.0 of the API is available since Oct. 2001 at [44].

- Integration in the current framework:
  - Re-use existing parts/interfaces whenever possible.
  - Usage of similar structures for new (internal) interfaces.
  - Configuration of wireless scheduling components with the same mechanisms as used in wireline setups (`tc`, netlink socket).
- 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, maximal one wireless channel monitor is active per wireless device at any point in time.[8] 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 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 6.4).

[8]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.

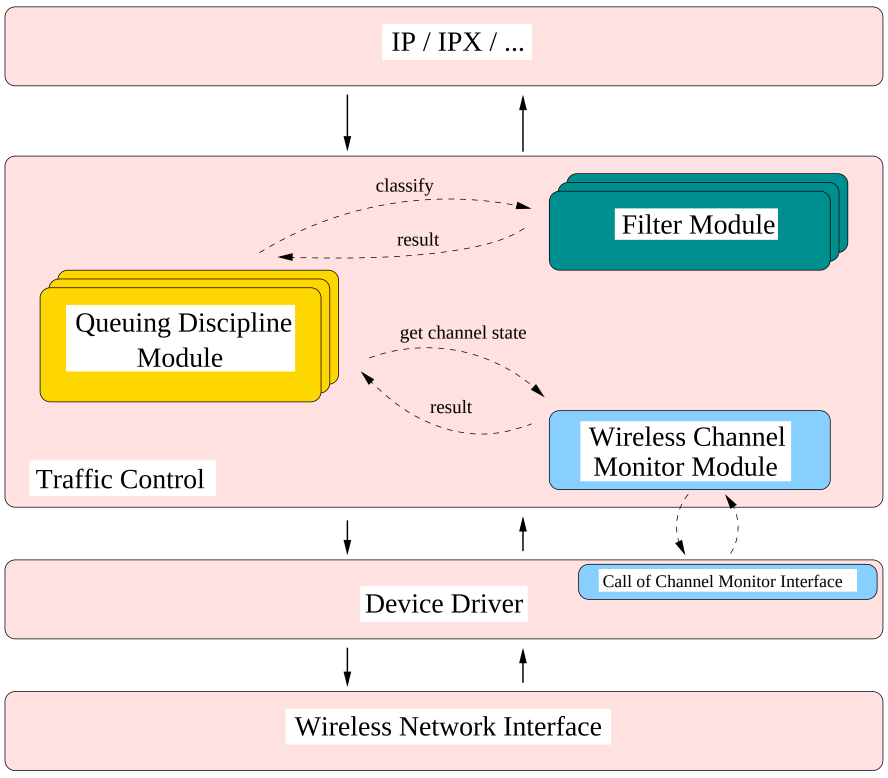

IP / IPX / ...
classify
result
Filter Module
Queuing Discipline Module
get channel state
result
Wireless Channel Monitor Module
Traffic Control
Call of Channel Monitor Interface
Device Driver
Wireless Network Interface

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

### 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 channel 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`,[9] 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 of the wireless channel/estimated goodput rate currently available to a specific mobile station. 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 de-

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

| 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 mobile station (dst). |
| `reset_channel_state (*mon, *dst)` | Resets all channel-state information for the specified mobile station. |
| `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.*

vice.[10] When a wireless channel monitor module is loaded (e.g. by issuing the `insmod` command), it registers 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 signals this fact 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 kernel's 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

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

| 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.*

packet). Unfortunately – since the WLAN 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.

### 6.2.2 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 Section 6.1.1.1, also has to be implemented by a wireless queuing discipline. Thus, it is 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 wireline 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 point 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

**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);
```

such a queuing discipline will be briefly presented, afterwards it is shown how such a discipline was implemented within the framework.[11]

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 than 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 wireline 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 before, e.g. by a token bucket filter), the number 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

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

**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) ;
```

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 – because 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.

Note that before the `start_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, the destination 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 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 about 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

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

```
skb = skb_peek_tail ( queue ) ;   /* look at last enqueued packet     */
min_level = $\infty$ ;
drop_skb  = $\emptyset$ ;
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 is only dropped if the channel was recently probed       */
   /* Note : ` jiffies ` is a system variable which indicates the     */
   /*          current time .                                          */
   if ( jiffies - state ->last_probed < probe_interval $\wedge$
        state ->level < min_level )
   {
         drop_skb  = skb;
         min_level = state ->level;
   }
   skb = skb->prev;         /* advance to previously enqueued packet */
}

if ( skb $\neq \emptyset$ )
    drop_skb(skb) ;         /* unlink packet from queue and drop it   */
```

$g_i$ 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:

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

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:

- 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 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 [58] distributed as part of Sony's ALTQ package for BSD UNIX [9]. Therefore, the task consisted of the following steps:

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

The scheduler itself consists of two parts: a (user space) configuration module for the traffic control program `tc`[12] and a kernel module, which implements the scheduling algorithm[13].

### 6.3.1 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 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 wireline 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.

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

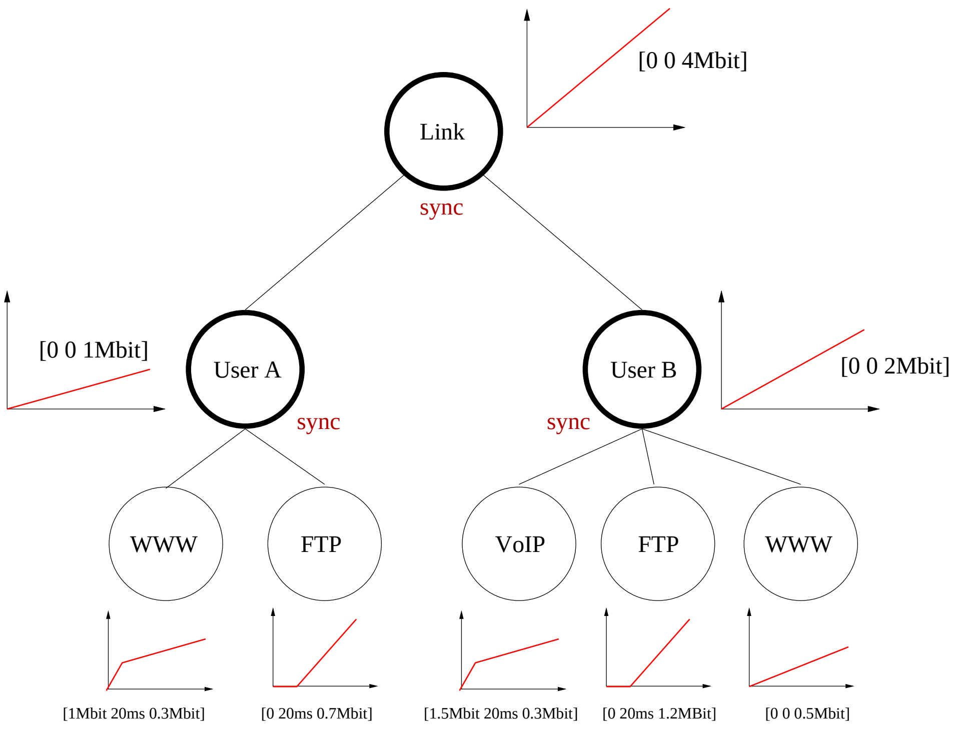


**Figure 6.7:** *Wireless link-sharing for two users with the modified H-FSC scheduler. In the [$m_1$ d $m_2$] triplets of a H-FSC scheduler configuration, the slopes $m_1$ and $m_2$ are specified by the amount of bits, which have to be transmitted in a second.*

### 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 rt [0 0 *BPS*]).
- **default**: Specify this class as the (new) default class.
- **sync**: This class is a wireless synchronization class. [+]

### An Example Setup

Listings 6.4, 6.5 and 6.6 show how the modified H-FSC scheduler would be configured for the wireless link-sharing structure shown in Figure 6.7.

**Listing 6.4:** *Example Setup Part 1 of 3: Installing the channel monitor and qdisc setup.*

```
#!/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 6.5:** *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
```

The first step is to load a wireless channel monitor – in this case the type “`driver`” is used – and to add the qdisc (Listing 6.4).

Then the classes for both users are configured (Listing 6.5), 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 6.6). Within the subtree of each user the scheduling is cooperative.

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

## 6.3.2 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 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

**Listing 6.6:** *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 1.5Mbit 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 ...
```

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 parent's service) a generalized service curve, which can consist of multiple pieces in form of a linked list, is used.

### 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

| Name | Description |
|---|---|
| *sched_rootclass | pointer to the root class of the scheduler |
| *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<br>(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<br>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 [byte] [+] |
| 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<br>into account in overload situations [+] |

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

| 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 [byte] |
| cl_drop_cumul | total packet drop service received [byte] |
| 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.*

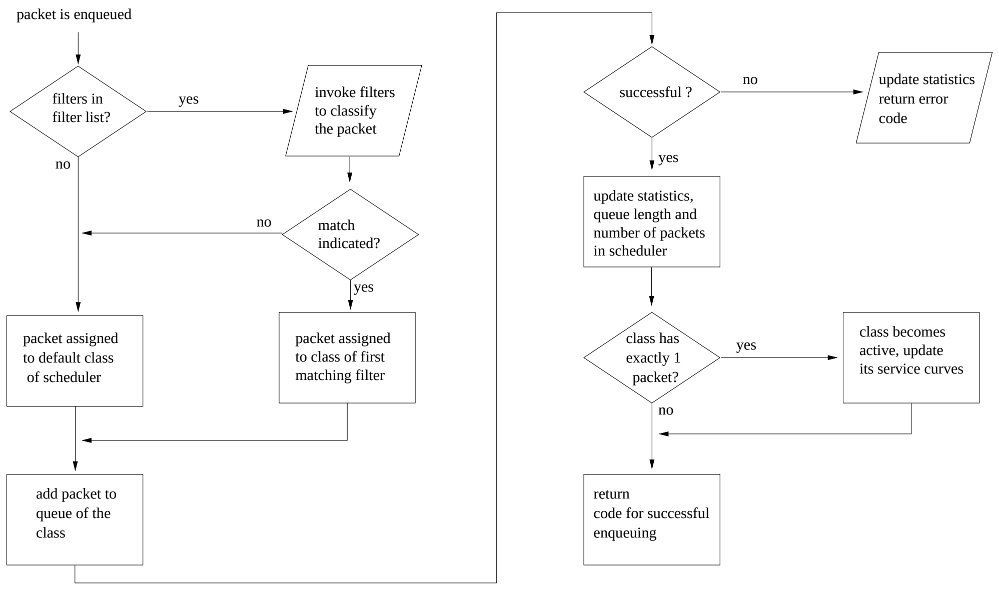

packet is enqueued
filters in filter list?
yes
no
invoke filters to classify the packet
match indicated?
no
yes
packet assigned to default class of scheduler
packet assigned to class of first matching filter
add packet to queue of the class
successful ?
no
update statistics return error code
yes
update statistics, queue length and number of packets in scheduler
class has exactly 1 packet?
yes
class becomes active, update its service curves
no
return code for successful enqueuing

**Figure 6.8:** *Flow chart of packet enqueuing in the modified H-FSC implementation.*

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 of the subtree. In an overload situation, the scheduler 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 specifies, which is its root. 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 the scheduling is based on goodput, service curves need to be converted to the corresponding resource-based amounts 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 the goodput-based *and* the resource-based update values as parameters. It then updates the virtual times of each node corresponding to its scheduling mode.

### 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 6.8.

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.[14]
- Estimating the available bandwidth and reacting to overload situations.
- Updating the service curves.

Figure 6.9 shows the reaction to a dequeue event within the `dequeue` function and its subfunctions as a process flow chart.

## 6.4 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 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/noise information reported by the MAC [13], 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.[15] (Some other designs, e.g. the one of the *Raylink* 2 MBit/s cards, 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 [13] [15] [20] 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 [23] and a *Lucent* 11 MBit/s DSSS card [60], both conforming to the IEEE 802.11 standard, confirmed this assumption (Figure 6.10). (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.)

[14]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.

[15]The signal quality is the ratio of signal level to noise level.

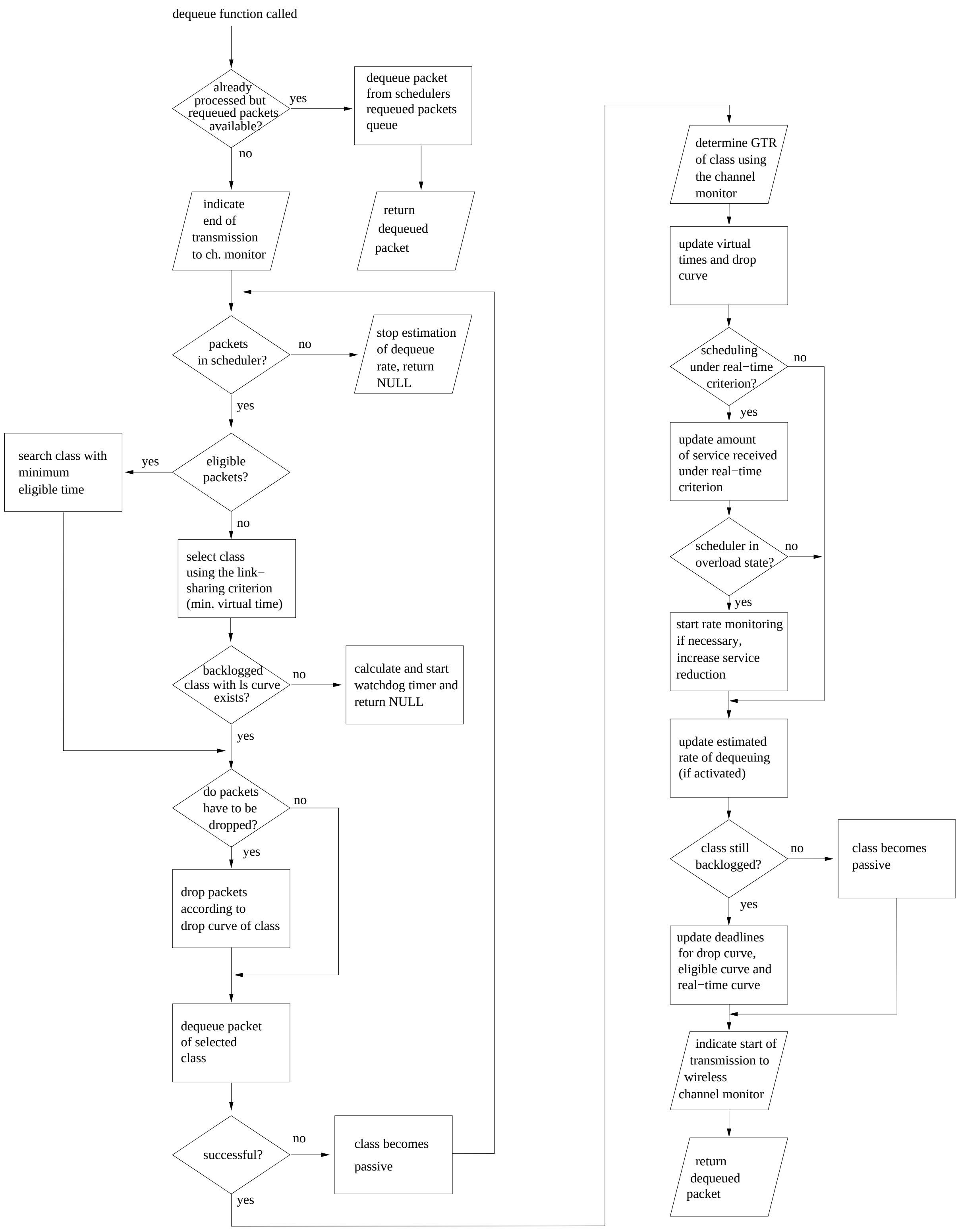
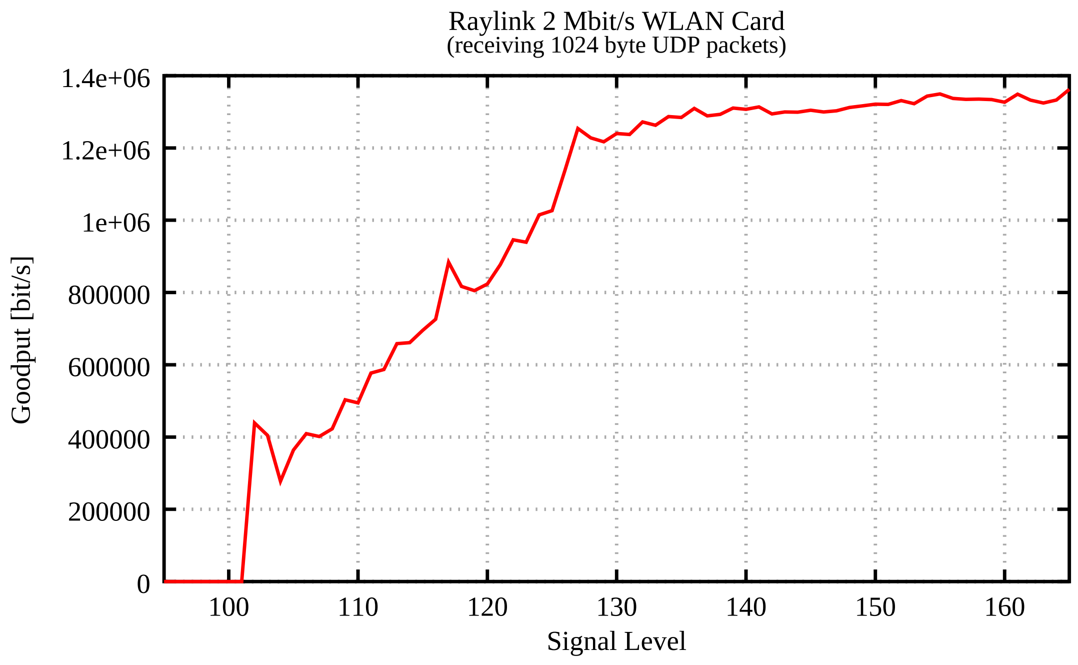
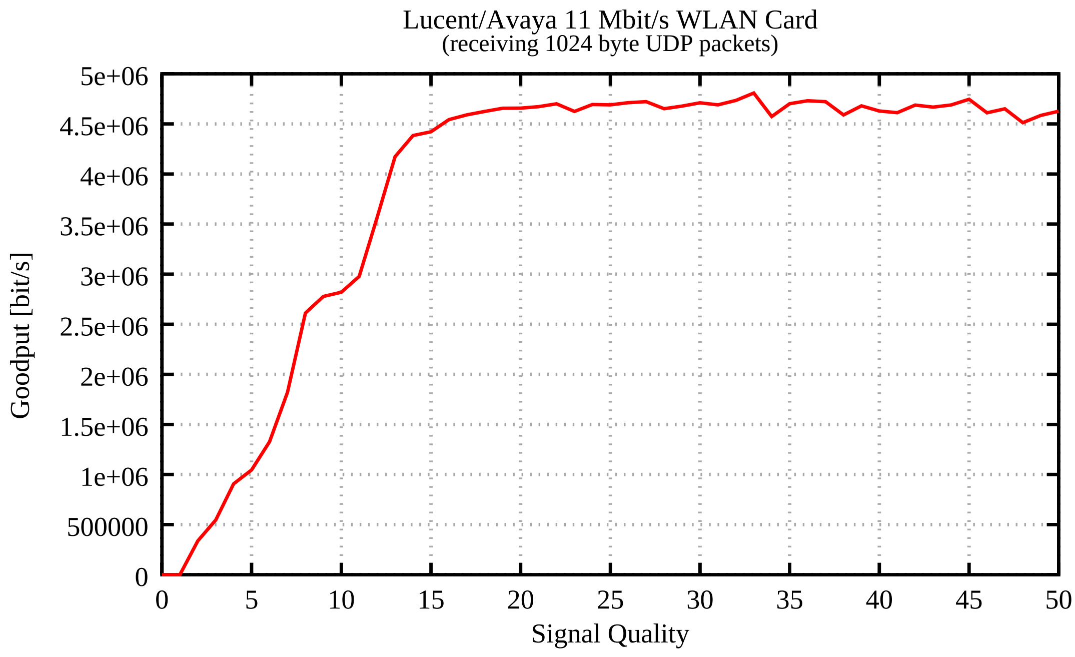

dequeue function called
already processed but requeued packets available?
yes
no
dequeue packet from schedulers requeued packets queue
indicate end of transmission to ch. monitor
return dequeued packet
packets in scheduler?
no
stop estimation of dequeue rate, return NULL
yes
eligible packets?
yes
search class with minimum eligible time
no
select class using the link−sharing criterion (min. virtual time)
backlogged class with ls curve exists?
no
calculate and start watchdog timer and return NULL
yes
do packets have to be dropped?
no
yes
drop packets according to drop curve of class
dequeue packet of selected class
successful?
no
class becomes passive
yes
determine GTR of class using the channel monitor
update virtual times and drop curve
scheduling under real−time criterion?
no
yes
update amount of service received under real−time criterion
scheduler in overload state?
no
yes
start rate monitoring if necessary, increase service reduction
update estimated rate of dequeuing (if activated)
class still backlogged?
no
class becomes passive
yes
update deadlines for drop curve, eligible curve and real−time curve
indicate start of transmission to wireless channel monitor
return dequeued packet

**Figure 6.9:** *Flow chart of packet dequeuing in the modified H-FSC implementation.*



(a) *Raylink* 2 Mbit/s FHSS



(b) *Lucent* 11 Mbit/s DSSS

**Figure 6.10:** *Correlation between signal level/quality and goodput-rate. Since the Raylink card does not report a signal quality value, the signal level was used instead. Signal level and signal quality values are vendor specific and not comparable.*

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* [61] 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 signal level 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 signal 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 *data* packet[16], the current system forces the mobile stations to send packets in regular intervals by sending an *ICMP echo request* packet to each mobile station every second. This is done by starting a "ping" for each client on the access point, 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 is likely to 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 6.10. 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 (or a connected station in the wireline network) 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

[16]Meaning that the reception of an IEEE 802.11 management/control frame or a data frame containing only a CF-ACK and no payload does usually not cause an update of this information.

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 simply 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. Simulation

This chapter describes the used simulation environment and presents the results of the conducted simulations.

## 7.1 Simulation Model

For simulation purposes, a simple model was developed in order to evaluate the behavior of the wireless H-FSC scheduler (Figure 7.1) in various situations. The basic assumptions are the following: The scheduler is implemented at the access point of the wireless LAN. Only downlink traffic (from the AP to the various mobile stations) is simulated. The scheduler receives packets from the IP layer, rearranges their order[1] and hands them down to the device driver.

The properties of the wireless channel towards each mobile station depend on the position and speed of the MS. Therefore, a two-state Markov model for the wireless channel, with separate parameters for each station, is specified. Note that these are not physically different channels but descriptions for the behavior of the downlink channel to each mobile station.

Autonomous retransmissions and rate adaptation of the wireless network adapter are simulated by increasing the time needed to transmit a packet. For simplicity, the additional overhead/delay for acknowledgements and rate signaling is assumed to be small and not taken into account. Therefore, the time needed to transmit a packet including one retransmission is exactly twice the time to send it on an ideal channel. In scenarios where the effects of adaptive modulation are studied retransmissions are disabled, in all other cases the maximum number of retransmissions is specified as part of the description of the scenario.

The wireless channel monitor used for the simulations estimates the quality of the wireless channel towards a mobile station based on the time needed to transmit a packet. For example, if the raw throughput for a wireless device is $B_{raw}$ and the monitor observed that a packet of length $L_p$ was sent by the wireless NIC in $\delta_t$ to mobile station $i$, it would estimate the GTR as

[1] Of course, the order of packets for a specific micro-flow is preserved.

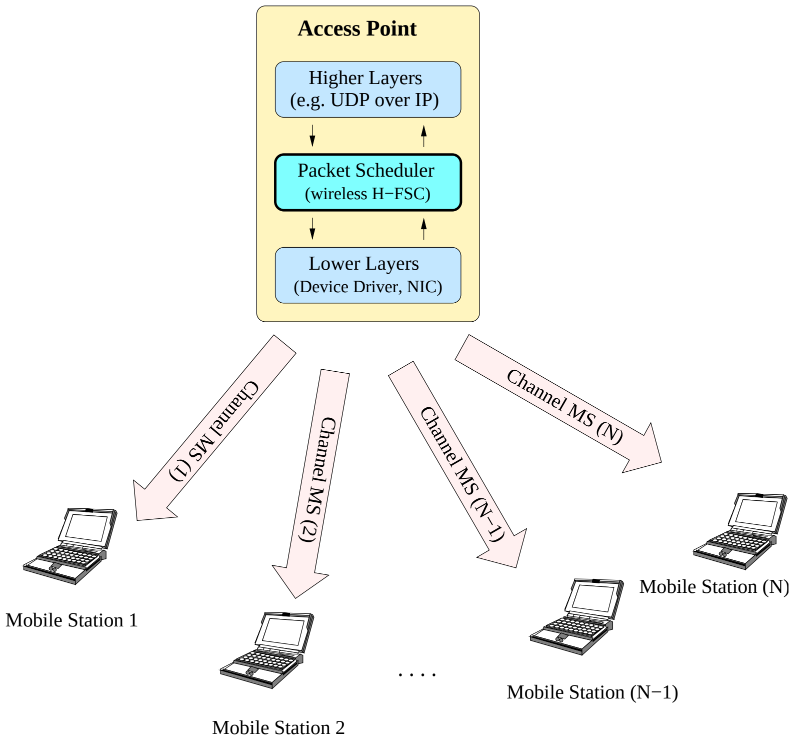


**Figure 7.1:** *Illustration of simulation model.*

$$g_i = \frac{L_p}{\delta_t \cdot B_{raw}} \tag{7.1}$$

Thus, the simulated channel monitor performs an accurate estimation of the wireless channel quality with a delay of the time needed to transmit one packet. (This is caused by the fact that it can only update its estimation *after* the packet is transmitted.)

For simplicity, the wireless NIC is assumed to have an internal buffer for storing exactly one packet (independent of its size). Although this is not the case for most of the equipment in use today, a larger buffer would only influence the accuracy of the wireless channel monitor, which is not the focus of these simulations.

In this way, a level of detail was chosen for the simulations which allows us to study the effects of relevant parameters, such as the number of retransmissions or adaptive modulation, on the behavior of the scheduler. On the other hand, we abstract from details as e.g. the wireless MAC protocol used, RTS/CTS mechanism or physical layer parameters, which would influence the quantitative results observed but not the general behavior of the scheduler.

## 7.2 Simulation Environment

All simulations were done using the TCSIM environment (see Section 6.1.3), which allows the event-driven simulation of packet scheduling. Since TCSIM was developed for wireline scheduling, several modifications and extensions were necessary in order to evaluate the wireless components by using simulations. Basically, three parts were added/extended:

- simulation of wireless network interfaces

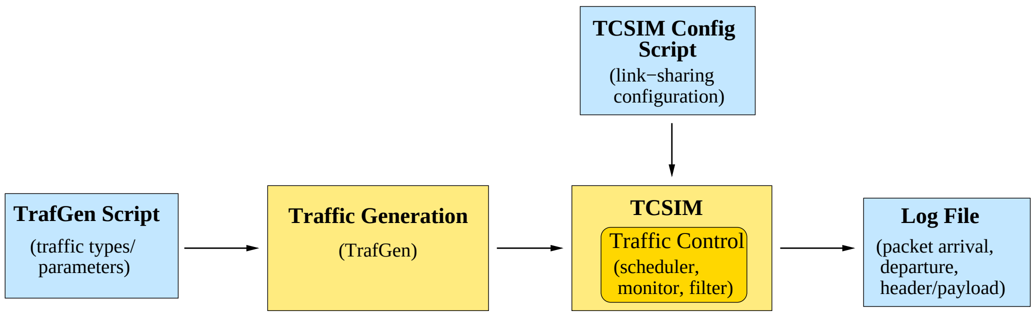

TCSIM Config Script
(link−sharing configuration)
TrafGen Script
(traffic types/ parameters)
Traffic Generation
(TrafGen)
TCSIM
Traffic Control
(scheduler, monitor, filter)
Log File
(packet arrival, departure, header/payload)

**Figure 7.2:** *Structure of simulation environment.*

- traffic generation
- trace analysis

Figure 7.2 shows the structure of the simulation environment: A traffic configuration file describes the different types of data flows (e.g. constant bit-rate or Poisson distributed arrival, rates, start and stop times, etc.) for a simulated scenario. Based on this information, a trace of packet arrivals is generated. The link-sharing hierarchy is specified for the TCSIM simulation environment within a separate configuration file, which contains the same traffic control commands used for a real Linux based access point. The TCSIM program acts as a wrapper providing the environment necessary to run the scheduler, monitor, and filter kernel modules. Simulation of wireless channel states is performed within a special wireless simulation scheduler module, which is described in more detail in the following section. The result of each simulation run is a log file recording the arrival/departure time and content for each processed packet.

### 7.2.1 Wireless Channel Simulator

Since the chosen simulation environment TCSIM did not have support for simulation of wireless network interfaces and 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 wireline interface simulation: Whereas a wireline 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 7.3) is a two-state Markov model, which simulates the wireless characteristics by assuming that the channel can be in only two states: a good state, in which the probability for a packet error[2] is zero, and a bad state, in which a

[2]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.

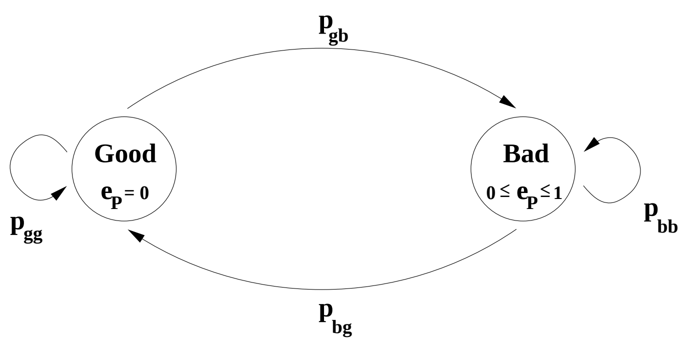


**Figure 7.3:** *Gilbert-Elliot model of a wireless channel.*

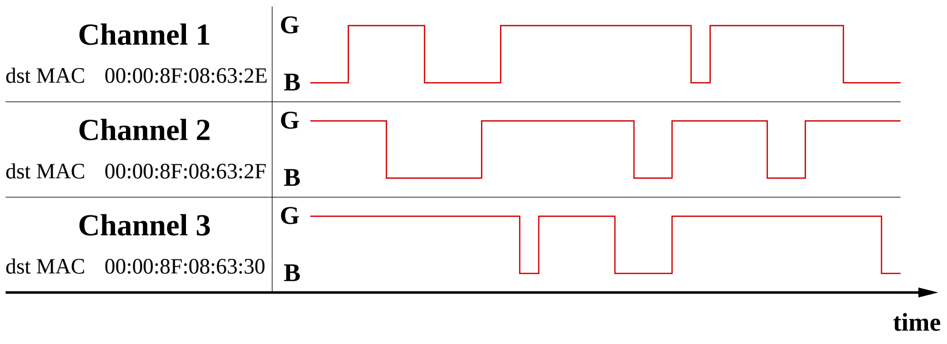


**Figure 7.4:** *State of the wireless link from an access point towards three mobile stations.*

packet is corrupted with a probability $e_P, (0 \le e_P < 1)$. 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 7.4.
- 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.
- 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.

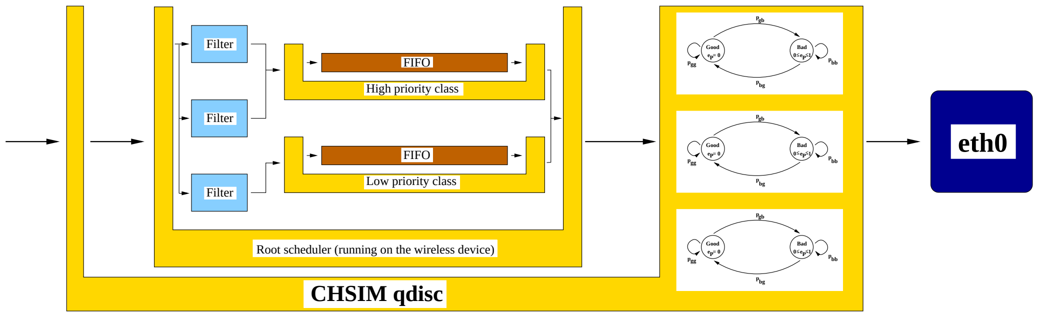


**Figure 7.5:** *Regular scheduler encapsulated in the wireless channel simulation scheduler, which is running on a regular (TCSIM) network interface.*

- A simulated channel corresponds to a scheduling class of the wireless simulation qdisc. 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 the 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 7.5. 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 first 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 simulation qdisc dequeues packets from the inner scheduler until the internal queue of the simulated wireless device 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.

**Listing 7.1:** *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)"
```

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).

### 7.2.2 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.

A configuration example is shown in Listing 7.1. Listing 7.2 is an excerpt of the generated trace. Using TrafGen, constant bit-rate, Poisson, uniform, and bursty flows can be simulated.

### 7.2.3 Additional Trace Analysis Tools

Currently the support for trace analysis in TCSIM is limited. The main tools are the filter script `filter.pl`, which is able to filter traces for source/destination addresses, ports,

**Listing 7.2:** *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)
...
```

and enqueue/dequeue events, and the plot script `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 script called `avg_calc.pl` was implemented to calculate the maximum, minimum, average, and standard deviation values over windows of trace data.

### 7.2.4 Steps of a Simulation Run

A typical TCSIM simulation run within the simulation environment consists of the following steps:

1. Create a *TrafGen* configuration (optional).
2. Generate a traffic trace using *TrafGen* (optional).
3. Create a TCSIM configuration.
4. Run the simulation (with generated traffic trace/CBR flows).
5. Filter the trace data using `filter.pl`.
6. Analysis of the trace log files using `plot.pl` and `avg_calc.pl`.

## 7.3 Simulation Results

The scheduler was tested in three different simulation scenarios: The first one uses a very simple configuration consisting of only two mobile stations and one access point and is similar to the situation in which the prototype was tested. The others study two different constellations using a larger scheduling hierarchy with a larger number of mobile stations. Since only minor changes were made influencing the behavior in a state in which excess bandwidth is available, the simulated scenarios mainly concentrate on the properties of the scheduler in overload situations.

### 7.3.1 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 Section 4.1. A wireless link with a maximum (goodput) rate of 6 MBit/s is shared by

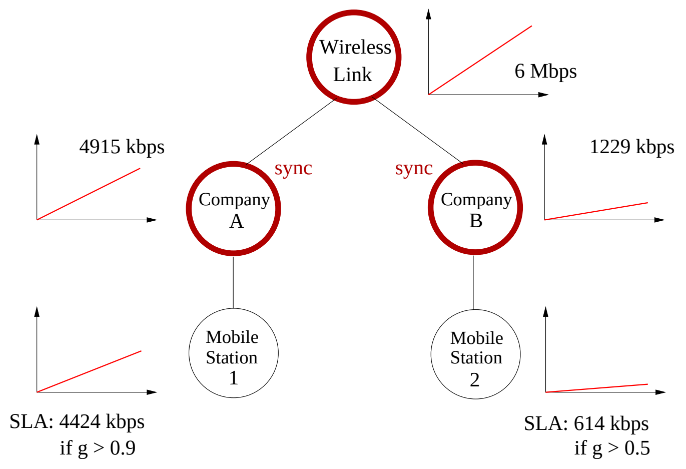


**Figure 7.6:** *Link-sharing hierarchy for Scenario 1.*

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 7.6, the complete configuration script is part of the Appendix (Section A.4.3). 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[3] 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[4] 7.7 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 $\geq 3$, 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! On the other hand, the H-FSC scheduler modified according to the wireless link-sharing model 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

[3]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.

[4]The bounds of the 95% confidence interval for the calculated rates are not plotted in the following graphs because the symbols used for data points are larger than the intervals at the given scale. But the detailed results (including the confidence intervals) are included in Section A.4.3 of the Appendix.

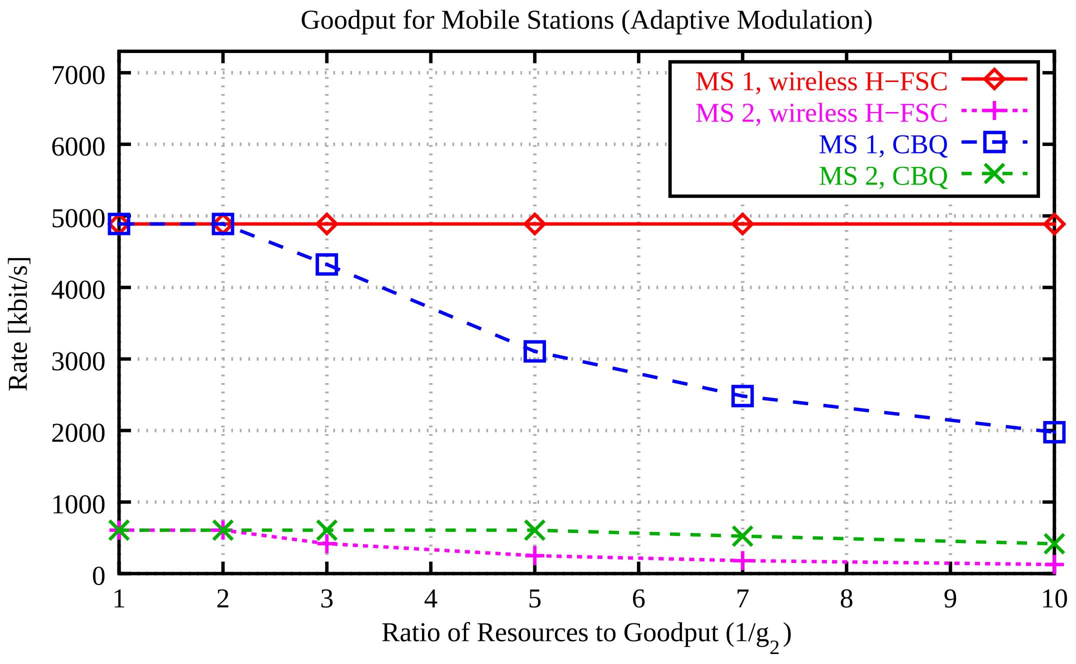


**Figure 7.7:** *Comparison of goodput of mobile stations in Scenario 1 for the modified H-FSC scheduler and a regular CBQ scheduler above the link layer when the modulation used by MS 2 is varied (e.g. because MS 2 adapts to a decreasing link quality).*

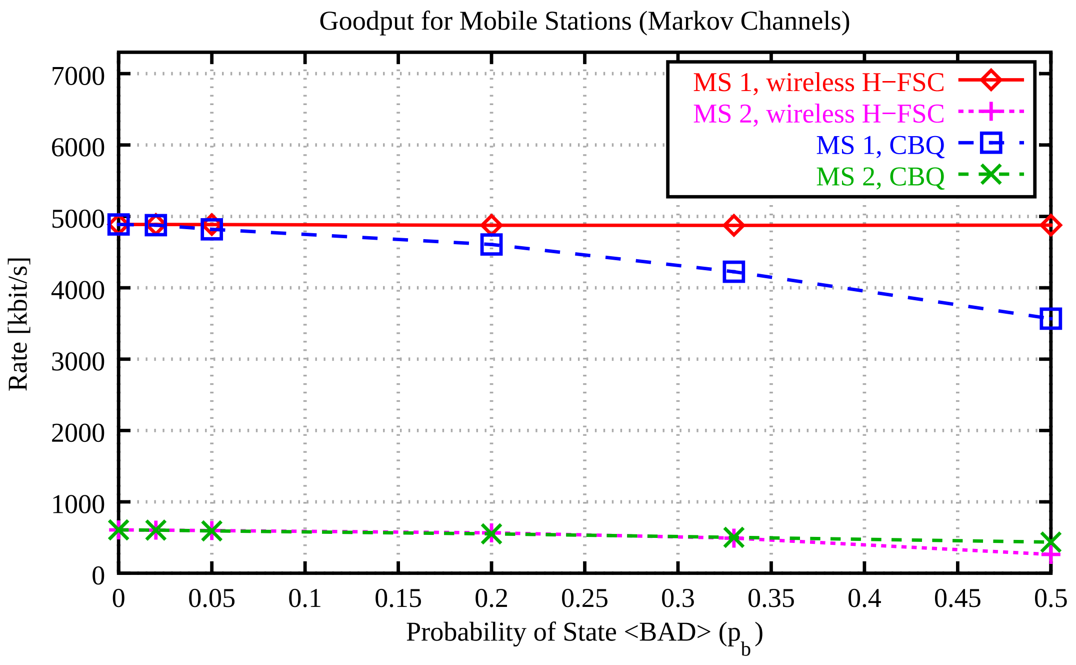


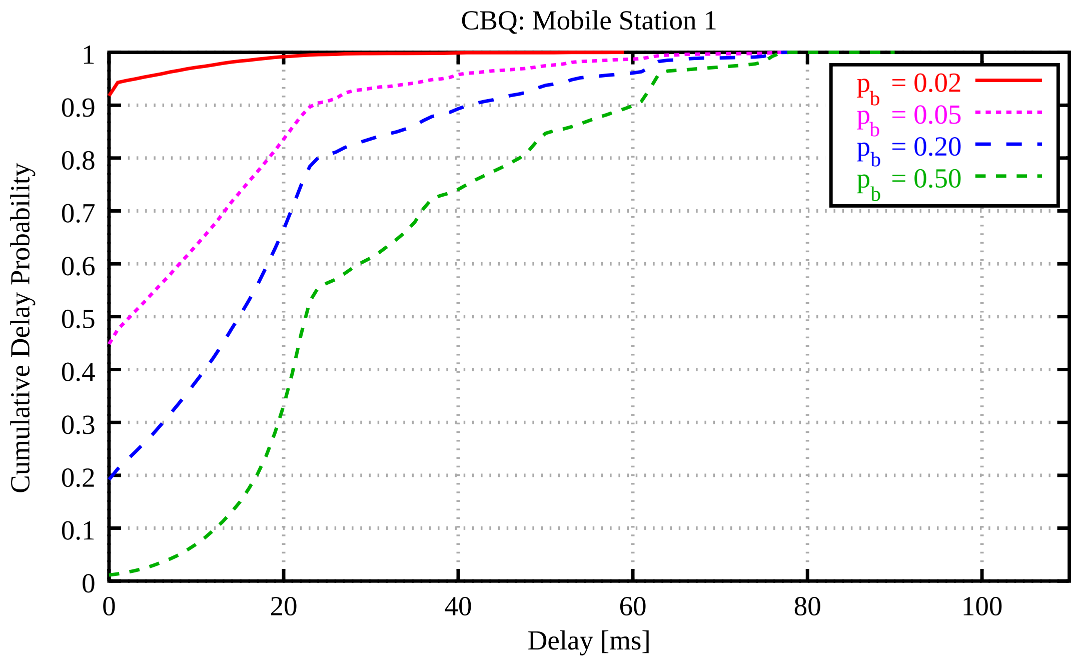
**Figure 7.8:** *Comparison of goodput of mobile stations in Scenario 1 for the modified H-FSC scheduler and a regular CBQ scheduler above the link layer. In this case, a Markov model simulates the wireless channel, and the probability for a bad channel state for MS 2 is varied.*

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 7.8 illustrates that the wireless H-FSC scheduler still is able to perform the resource-based scheduling.

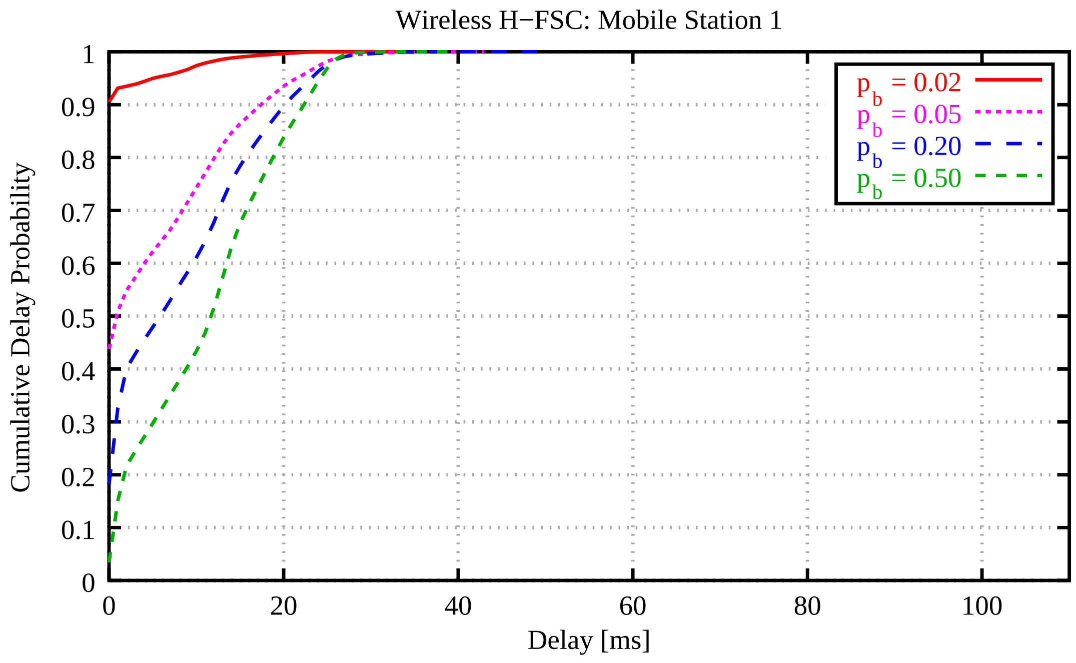
Figure 7.9 shows the influence of the probability for a bad channel state of MS 2 on the cumulative delay probability for packets sent to MS 1. For both schedulers increasing $p_b$ for the wireless channel of MS 2 increases the probability for higher delays for MS 1. However, Figure 7.9(b) indicates 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. Therefore, its delay bound 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 $\tau_{max,g_{min}}$:

$$d_{max,1} = 1.9ms + 2 \cdot 14.3ms = 30.5ms \tag{7.2}$$



(a) CBQ



(b) wireless H-FSC

**Figure 7.9:** *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)*

| 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.*

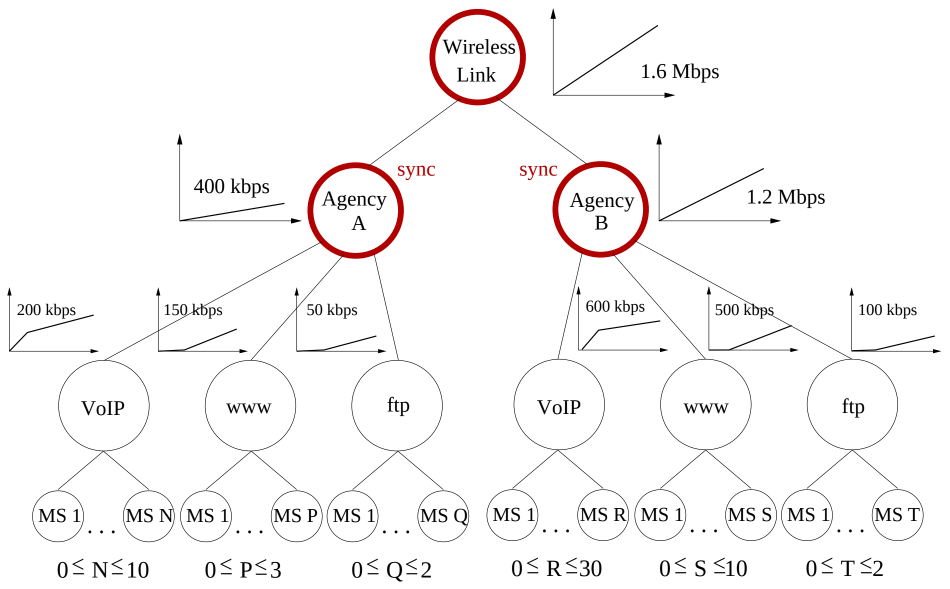


**Figure 7.10:** *Link-sharing hierarchy used in Scenario 2 and 3.*

### 7.3.2 Scenario 2

In the second scenario, scheduling within a more complex hierarchy is investigated. It is based on the example introduced in Chapter 4: The wireless link is shared by two agencies, of which the first has 10 and the second has 30 mobile stations. Furthermore, three different types of traffic exist (WWW, FTP and VoIP), which are to be treated according to their specific QoS requirements. In Scenario 2, all classes present in the hierarchy shown in Figure 7.10 are transmitting at their maximum rates. 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 7.11 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 20 ms. 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.*

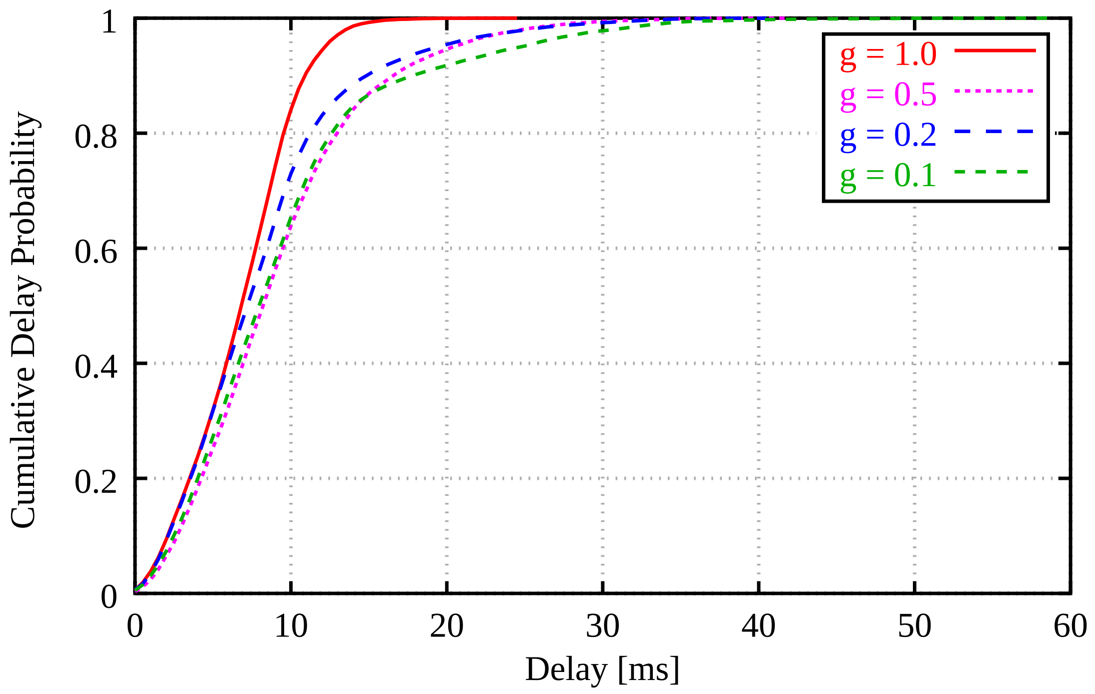


**Figure 7.11:** *Effect of $g_{ms2}$ (MS 2, Agency A) on the packet delay of VoIP traffic of a mobile in Agency B. The delay of VoIP packets of the competing agency remains almost unchanged although the link quality for MS 2 in Agency A decreases dramatically. (For $g_{ms2} = 1.0$, the system is not in an overload condition. Therefore, the VoIP traffic of all mobile stations stations has less than is maximum delay.)*

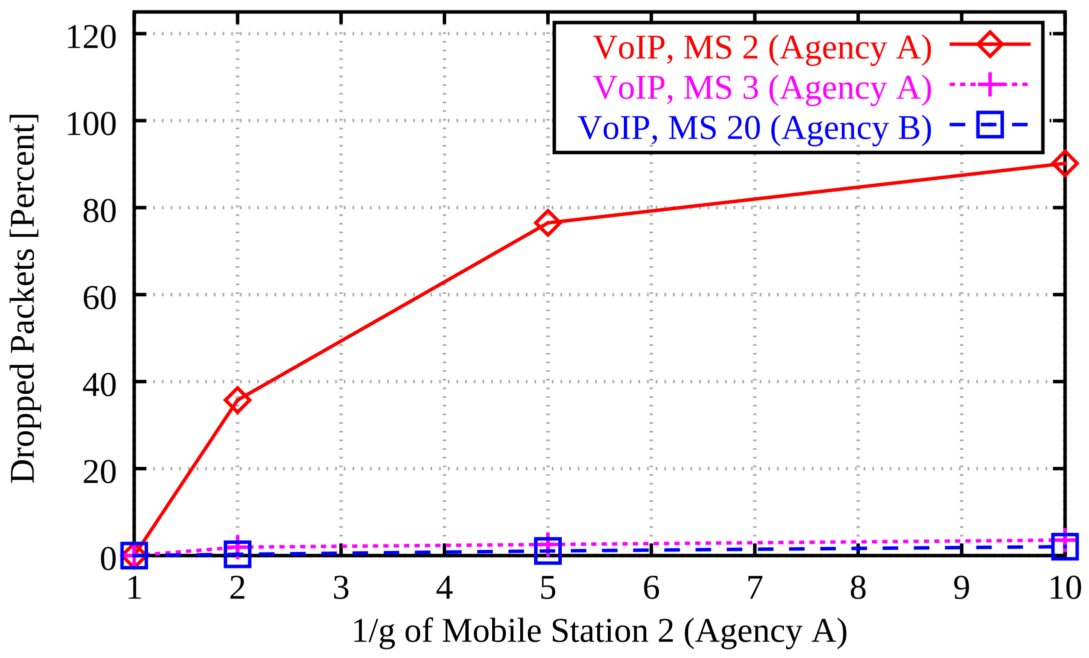


**Figure 7.12:** *Effect of $g_{ms2}$ on the percentage of packets dropped for the mobile experiencing the bad link (MS 2), a mobile of the same Agency (MS 3), and a mobile of Agency B (MS 20) in an overload situation. MS 20 is not affected because of the competitive scheduling among the two agencies, 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 is able to limit the impact on a mobile in the competing agency. Figure 7.12 compares the number of dropped packets. 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. Note that the way in which 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.

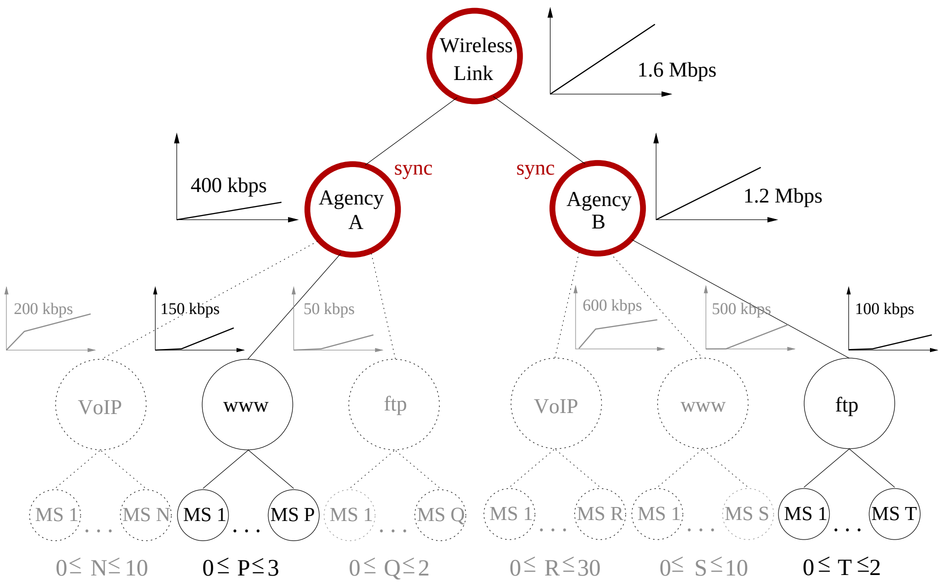


**Figure 7.13:** *Active classes in Scenario 3. Only WWW traffic is sent to Agency A, and only FTP flows are active in Agency B.*

### 7.3.3 Scenario 3

This scenario demonstrates the effects of competitive and cooperative scheduling. For Agency A, only WWW traffic is sent to three mobile stations (Poisson distributed, each station receives 160 kbit/s), Agency B has two customers receiving FTP traffic (Poisson distributed, 650 kbit/s). All other classes are idle (Figure 7.13). 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 7.14, the variation of the bandwidth available to one of the FTP customers and two WWW customers, of which MS 1 is experiencing a constantly 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 kbit/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 kbit/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). Thus, 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.

## 7.4 Simulation Validation

The simulation model which was introduced in Section 7.1 abstracts from various details, e.g. the used MAC technology. In order to validate the results of the simulations, they

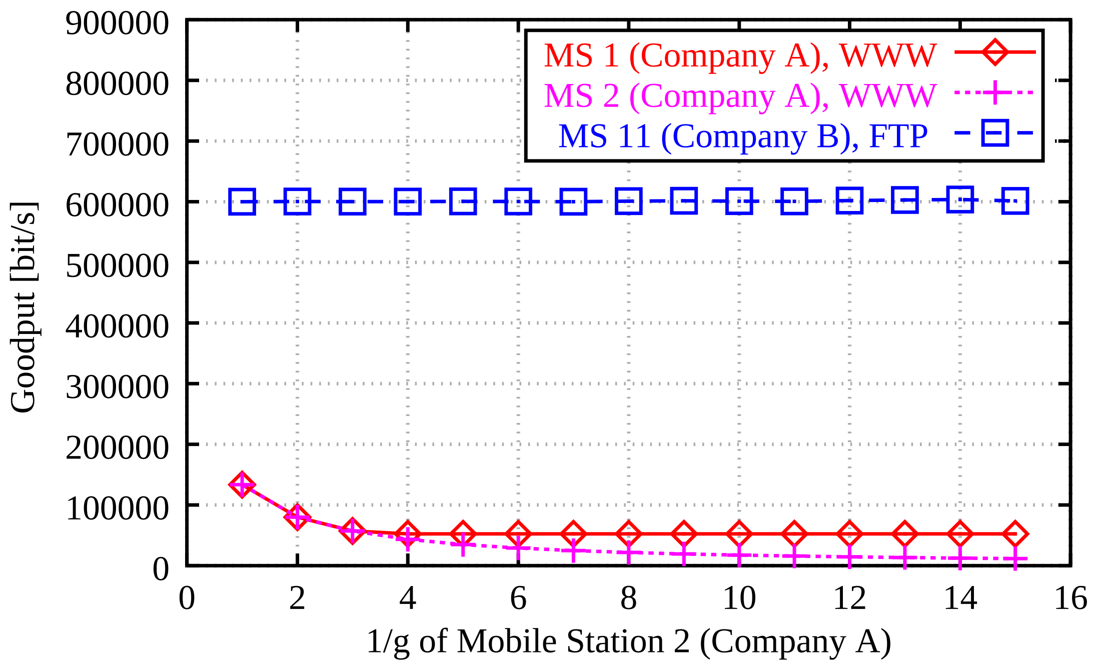


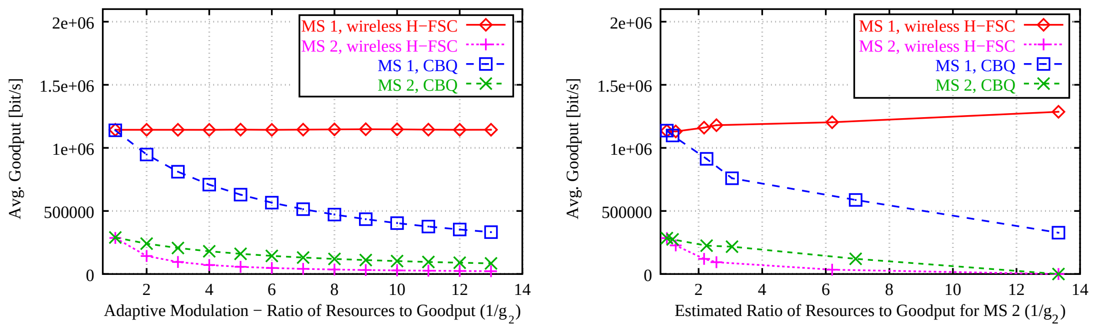
**Figure 7.14:** *Goodput available for selected mobile stations in Scenario 3. As the amount of resources needed to transmit to MS 2 (and MS 3) of Agency A increases, the goodput available for the competing Agency B is not affected. As long as excess bandwidth is available ($\frac{1}{g} < 4$) within Agency A, it is used to compensate the low link quality of MS 1 (and MS 3).*

are compared to those obtained in the prototype measurements with IEEE 802.11 WLAN hardware (see Chapter 8 for the details) in this section. For comparison, we concentrate on the *Raylink* 2 Mbit/s FHSS cards since this was the wireless hardware mainly used in the measurements.

The first scenario of the simulations (Section 7.3.1) is similar to the testbed environment. It is modified in the following ways in order to make the conditions comparable to those of the prototype measurements:

- The maximum goodput rate of the wireless interface is set to 1429 kbit/s, which is the rate achieved by the 2 Mbit/s *Raylink* cards.
- By sending data to MS 1 with a rate of 1200 kbit/s and to MS 2 with 300 kbit/s, it is guaranteed that the queues at the AP are always backlogged. The data generated consists of UDP packets with a payload of 1024 Bytes.
- For simplicity, the simulation uses only adaptive modulation in order to vary the GTR $g_2$ of MS 2.

A problem when comparing prototype measurement and simulation is that the ratio of resources to goodput (inverse GTR, $\frac{1}{g}$) needs to be estimated for the different positions in which measurements were done. If we approximate the GTR of the mobile station in the perfect position (MS 1) to be equal to 1 and the queues for both mobile stations at the AP are always backlogged, the amount of resources available for MS 2 can be computed



(a) **Simulation**

(b) **Prototype measurement** (*Raylink* 2 MBit/s FHSS cards, see page 98)

**Figure 7.15:** *Comparison of simulation results and prototype measurement.*

by subtracting the rate achieved by MS 1 from the maximum total goodput rate $B_{good,max}$. Therefore, the inverse GTR for MS 2 $\frac{1}{g_2}$ is approximated by

$$\frac{1}{g_2} \approx \frac{B_{good,max} - b_{good,1}}{b_{good,2}} \tag{7.3}$$

where $b_{good,i}$ is the goodput rate achieved by MS $i$.

Using Equation 7.3, the GTR was computed from the data of the (calibrated) *Raylink* measurements (Table 8.2). Figure 7.15 illustrates that the behavior of the scheduler in the simulation is in accordance with the prototype measurements.

Furthermore, since the TCSIM simulation environment allowed us to use the almost unmodified kernel scheduler code, we can guarantee that the scheduler used for the simulations is identical to the code implementing the wireless H-FSC algorithm in the Linux prototype.

The prototype measurements, the wireless testbed, and possible causes for the errors observed are presented in more detail in Chapter 8.

# 8. Measurements

This chapter presents results obtained using a prototype implementation of the proposed algorithm within the Linux kernel (V. 2.4.13) as described in Section 6.2.

## 8.1 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:

1. *Raylink* 2 MBit/s FHSS [23]

2. *Lucent/Avaya Gold* 11 MBit/s DSSS [60] (128 bit RC4 WEP encryption)

The *Raylink* cards operate at 1 and 2 MBit/s according to the IEEE 802.11 standard [54], the *Lucent* cards additionally support the 5.5 and 11 Mbit/s modes specified in IEEE 802.11b [55]. 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 measurement 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.

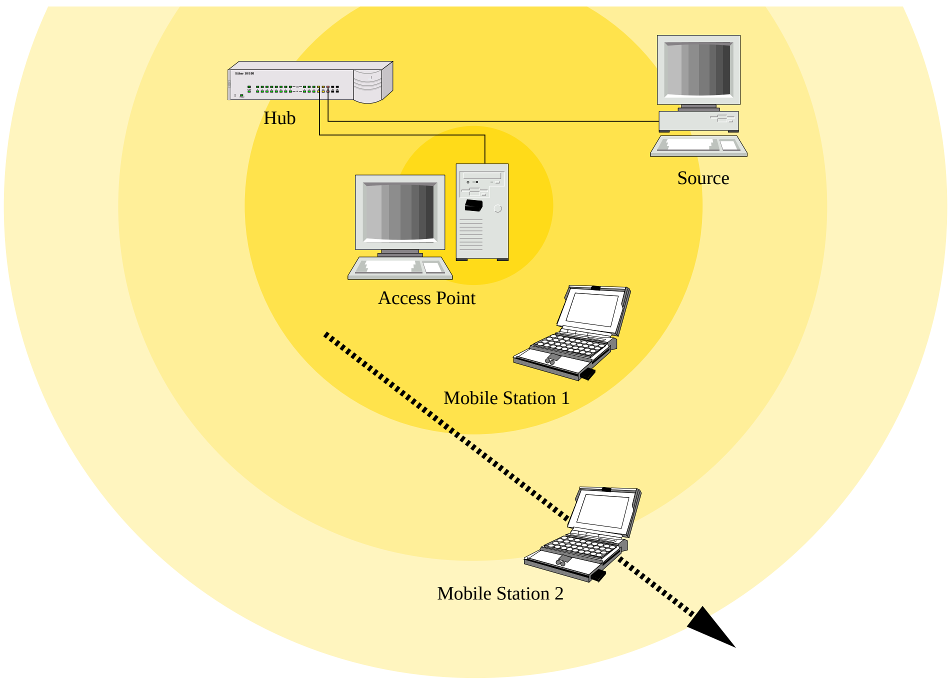


**Figure 8.1:** *The wireless testbed and main test scenario. While Mobile Station 1 (MS 1) was constantly in a perfect position, the position of Mobile Station 2 (MS 2) was varied during a measurement series.*

## 8.2 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 IEEE 802.11 testbed yet.

The main test scenario is rather simple and similar to the one presented in Section 7.3.1: 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[1] to both mobile stations at a high rate (3 Mbit/s for MS 1, 1 MBit/s for MS 2 in *Raylink* tests and 6 Mbit/s for MS 1, 1.5 MBit/s for MS 2 in *Lucent/Avaya* tests) guaranteeing that the classes of both mobile stations were always backlogged. This was also verified using the traffic control statistics.[2] MS 1 was constantly in a good position, and the position of MS 2 was varied. Since the wireless card only reports the link quality

[1]This size was chosen in order to approximate an "average packet size" seen in typical Internet traffic, which usually consists of a large number of small (< 100 bytes) packets and a smaller number of large packets (≈1500 bytes – the maximum IP packet size in an Ethernet), which still are responsible for the majority of the IP traffic (in terms of bytes) [32].

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

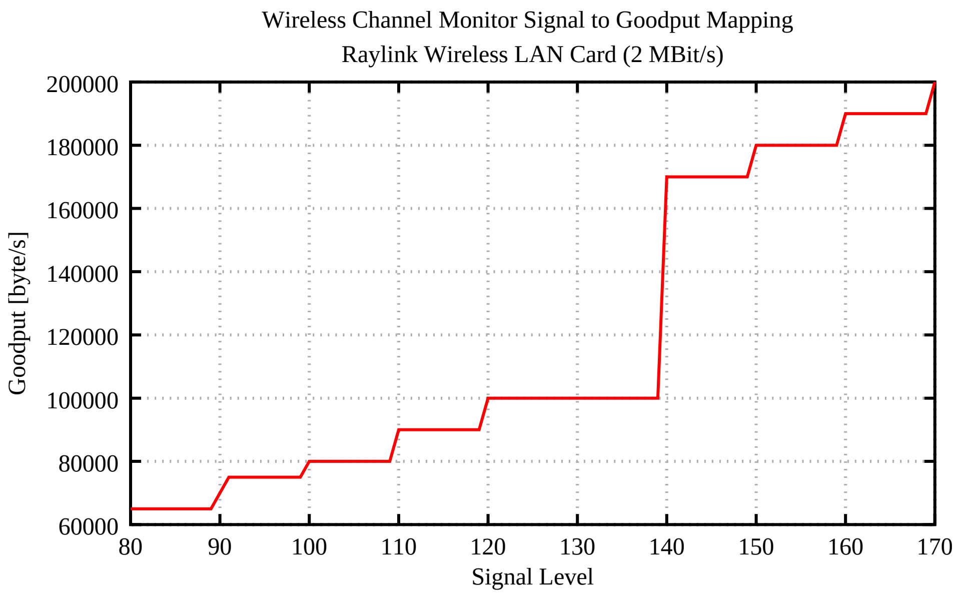


**Figure 8.2:** *Estimated signal level to goodput mapping for Raylink card (2 Mbit/s, FHSS) used in an early stage of the prototype.*

to a mobile station when a packet is received, the mobile clients were forced to send an *ICMP echo reply* every second.[3] All packets correctly received at the mobile stations were recorded with timestamp and delay information. The setup is illustrated in Figure 8.1.

### 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: Because 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.

## 8.3 *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 [68], this was based on a few single goodput measurements. Figure 8.2 illustrates this estimation.

[3]Because of the small size of the ICMP packets (56 Bytes) and the extremely low rate of one packet per station per second, its influence on the goodput measurements can be neglected in our test scenario.

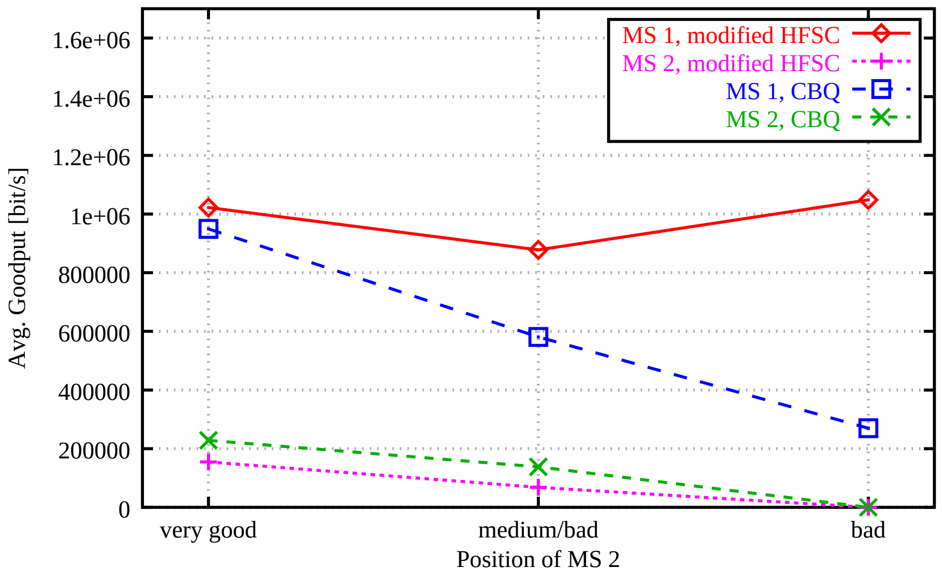


**Figure 8.3:** *Early prototype results without calibration. The resource-based scheduling decreases the influence of channel quality of MS 2 on the amount of resources available for MS 1.*

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 8.3. (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 can be compared.) Table 8.1 lists the average bit-rates, the bounds of the 95% confidence interval, the minimum bit-rates, and the standard deviation $s$.

## 8.4 *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 6.4). Figure 8.4 shows the calibration profile obtained for the *Raylink* cards. Note that although – as shown in Chapter 6, Figure 6.10(a) – the goodput drops to $\approx 0$ for levels below 100, it is mapped to a value of 5 kbyte/s in order to avoid completely blocking the queue for MS 2.[4] 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 8.5. The small but continuous increase of

[4]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 $\frac{k}{m}$.

| Position MS 2 | Mobile Station | Scheduler | Average [kbit/s] | 95% CI [kbit/s] | Min. [kbit/s] | Max. [kbit/s] | Std. Dev. [kbit/s] |
|---|---|---|---|---|---|---|---|
| good | 1 | mHFSC | 1022 | ± 5 | 849 | 1212 | 42 |
| | | CBQ | 949 | ± 2 | 870 | 1048 | 18 |
| | 2 | mHFSC | 154 | ± 3 | 80 | 228 | 30 |
| | | CBQ | 228 | ± 1 | 198 | 257 | 10 |
| med./bad | 1 | mHFSC | 877 | ± 9 | 634 | 1233 | 80 |
| | | CBQ | 580 | ± 7 | 384 | 727 | 58 |
| | 2 | mHFSC | 68 | ± 2 | 30 | 114 | 14 |
| | | CBQ | 138 | ± 2 | 46 | 232 | 20 |
| bad | 1 | mHFSC | 1048 | ± 12 | 329 | 1203 | 108 |
| | | CBQ | 270 | ± 2 | 215 | 321 | 19 |
| | 2 | mHFSC | 0 | - | 0 | 0 | - |
| | | CBQ | 0 | - | 0 | 0 | - |

**Table 8.1:** *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.*

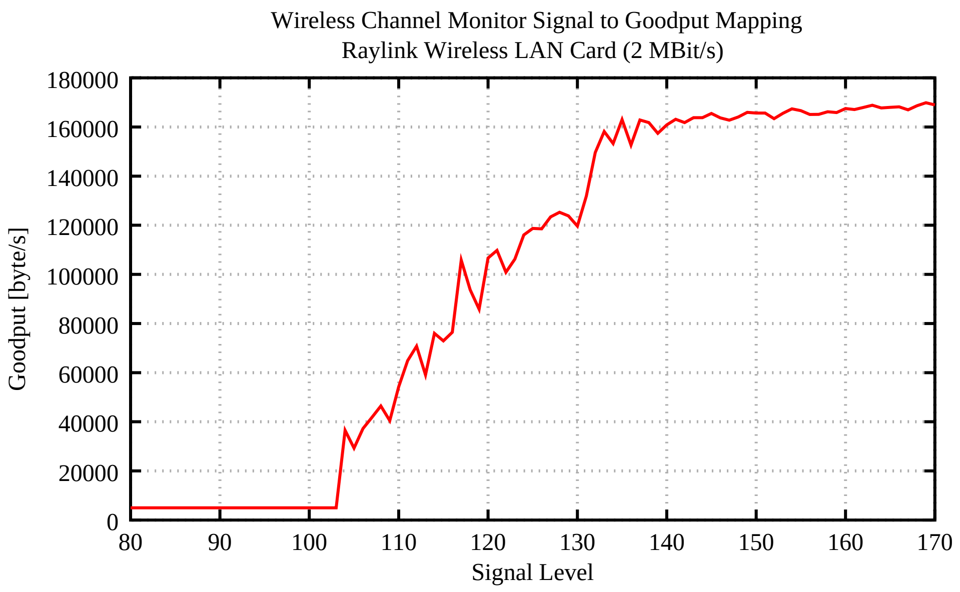


**Figure 8.4:** *Signal level to goodput mapping for Raylink card (2 Mbit/s, FHSS) obtained using the* WchmonSigMap *calibration tool described in Section 6.4.*

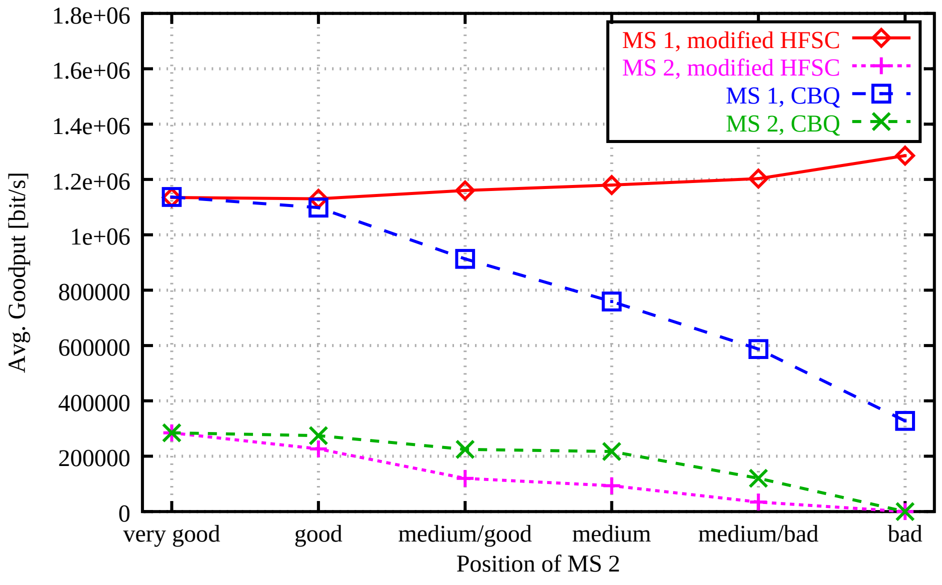


**Figure 8.5:** *Prototype results for Raylink with calibration. For the modified H-FSC algorithm the bandwidth available for MS 1 remains almost constant when the position of MS 2 is varied.*

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 maximum error occurs when MS 2 is in the "medium/bad" position. Here, MS 1 receives 68 kbit/s more goodput than it should (Table 8.2), which is an error of less than 6%.

Since in this case MS 2 is in a position where its GTR[5] is less than 0.17, this means that MS 2 receives about 11 kbit/s less goodput than it should. (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 in order to correct the estimations of the channel monitor.)

If a regular 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 8.2.

## 8.5 *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[6] and support higher data rates. Therefore, the

[5]Since the scheduler is work-conserving, this can be estimated by dividing the goodput achieved in this position by the goodput seen in the good position, where GTR is assumed to be approximately one. Of course, the observed error of scheduling 68 kbit/s more for MS 1 has to be taken into account, which means an estimation of a GTR for MS 2 of $g_{MS2} \approx \frac{35}{285-68} = 0.16$.

[6]Direct-Sequence Spread Spectrum (DSSS) instead of Frequency-Hopping Spread Spectrum (FHSS)

| Position MS 2 | Mobile Station | Scheduler | Average [kbit/s] | 95% CI [kbit/s] | Min. [kbit/s] | Max. [kbit/s] | Std. Dev. [kbit/s] |
|---|---|---|---|---|---|---|---|
| very good | 1 | mHFSC | 1135 | ± 4 | 757 | 1372 | 35 |
| | | CBQ | 1137 | ± 3 | 1035 | 1203 | 23 |
| | 2 | mHFSC | 285 | ± 2 | 260 | 496 | 14 |
| | | CBQ | 285 | ± 2 | 252 | 480 | 17 |
| good | 1 | mHFSC | 1130 | ± 10 | 370 | 1363 | 89 |
| | | CBQ | 1098 | ± 3 | 1002 | 1178 | 28 |
| | 2 | mHFSC | 227 | ± 8 | 42 | 934 | 68 |
| | | CBQ | 274 | ± 2 | 244 | 421 | 15 |
| med./good | 1 | mHFSC | 1160 | ± 11 | 917 | 1380 | 96 |
| | | CBQ | 913 | ± 15 | 151 | 1439 | 129 |
| | 2 | mHFSC | 120 | ± 5 | 17 | 362 | 47 |
| | | CBQ | 225 | ± 7 | 126 | 67 | 63 |
| medium | 1 | mHFSC | 1180 | ± 15 | 564 | 1439 | 131 |
| | | CBQ | 759 | ± 13 | 412 | 98 | 113 |
| | 2 | mHFSC | 94 | ± 6 | 8 | 404 | 55 |
| | | CBQ | 217 | ± 4 | 110 | 547 | 36 |
| med./bad | 1 | mHFSC | 1203 | ± 20 | 732 | 1439 | 175 |
| | | CBQ | 587 | ± 14 | 269 | 901 | 120 |
| | 2 | mHFSC | 35 | ± 4 | 8 | 328 | 33 |
| | | CBQ | 120 | ± 5 | 8 | 244 | 42 |
| bad | 1 | mHFSC | 1286 | ± 5 | 1060 | 1439 | 48 |
| | | CBQ | 328 | ± 8 | 17 | 766 | 71 |
| | 2 | mHFSC | 0 | - | 0 | 0 | - |
| | | CBQ | 0 | - | 0 | 0 | - |

**Table 8.2:** *Detailed results for the 2 MBit/s Raylink cards with the calibrated channel monitor.*

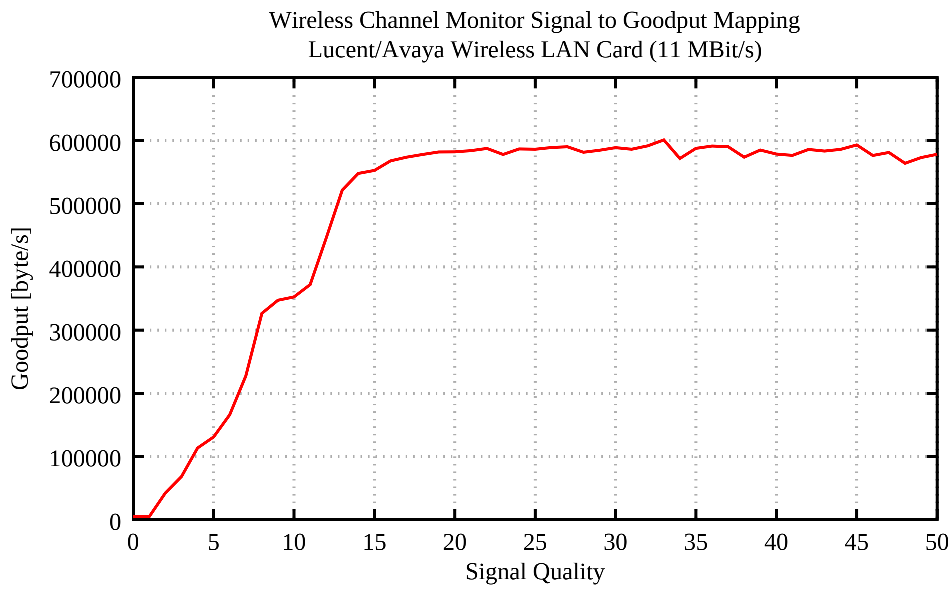


**Figure 8.6:** *Link quality level to goodput mapping for Lucent card (11 Mbit/s, DSSS) obtained using the* WchmonSigMap *calibration tool described in Section 6.4.*

signal to goodput mapping shown in Figure 8.6 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 8.7 and Table 8.3 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 were placed in the "medium/good" position.

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 both stations have the same GTR. (As it was the case in the previous scenarios, when both stations were in the "good" position.)

## 8.6 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

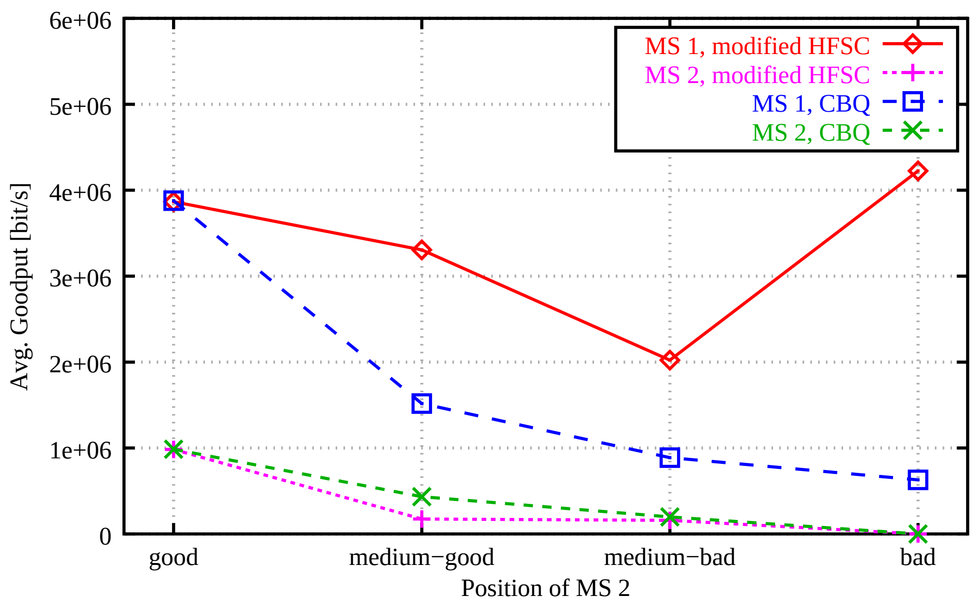


**Figure 8.7:** *Results for Lucent WLAN cards (11 Mbit/s, DSSS) with calibration.*

| Position MS 2 | Mobile Station | Scheduler | Average [kbit/s] | 95% CI [kbit/s] | Min. [kbit/s] | Max. [kbit/s] | Std. Dev. [kbit/s] |
|---|---|---|---|---|---|---|---|
| good | 1 | mHFSC | 3866 | ± 33 | 2608 | 4721 | 288 |
| | | CBQ | 3878 | ± 33 | 2474 | 4780 | 288 |
| | 2 | mHFSC | 983 | ± 3 | 833 | 1018 | 27 |
| | | CBQ | 986 | ± 3 | 825 | 1018 | 29 |
| med./good | 1 | mHFSC | 3306 | ± 65 | 1060 | 4914 | 578 |
| | | CBQ | 1518 | ± 28 | 740 | 2449 | 245 |
| | 2 | mHFSC | 174 | ± 8 | 8 | 446 | 74 |
| | | CBQ | 433 | ± 8 | 151 | 706 | 71 |
| med./bad | 1 | mHFSC | 2022 | ± 96 | 67 | 5310 | 848 |
| | | CBQ | 888 | ± 20 | 513 | 1380 | 180 |
| | 2 | mHFSC | 158 | ± 7 | 8 | 337 | 66 |
| | | CBQ | 197 | ± 10 | 8 | 412 | 87 |
| bad | 1 | mHFSC | 4225 | ± 60 | 437 | 5377 | 527 |
| | | CBQ | 630 | ± 4 | 497 | 825 | 37 |
| | 2 | mHFSC | 0 | - | 0 | 0 | - |
| | | CBQ | 0 | - | 0 | 0 | - |

**Table 8.3:** *Detailed results for the Lucent/Avaya 11 MBit/s wireless cards.*

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 estimations of the wireless channel monitor and therefore the accuracy of the scheduler.

# 9. 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 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 thesis presented a new approach to use *long-term* information about 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 quality for 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 sending 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) [66]
   - Linux 2.4.X kernel sources [25]
     (patch was developed for 2.4.13 but should work with all 2.4.X kernels)
   - iproute2 sources (Version 001007 or newer) [27]
   - PCMCIA card services (Version 3.1.29 or newer) [46]
     (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:
   - 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).

[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.

- 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 6.3).

## A.2 Kernel Configuration

Table A.1 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 only necessary 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 by 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 6.4, 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 the Sun Java 2 runtime environment [59]. 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 [61] via the Java Native Interface (JNI). This library is copied to `/usr/lib` as part of the `make` process. The main application is started with `java WchmonSigMap.WchmonSigMap` in the directory of the application, the simple UDP packet generation tool is executed with `java WchmonPackGen`.

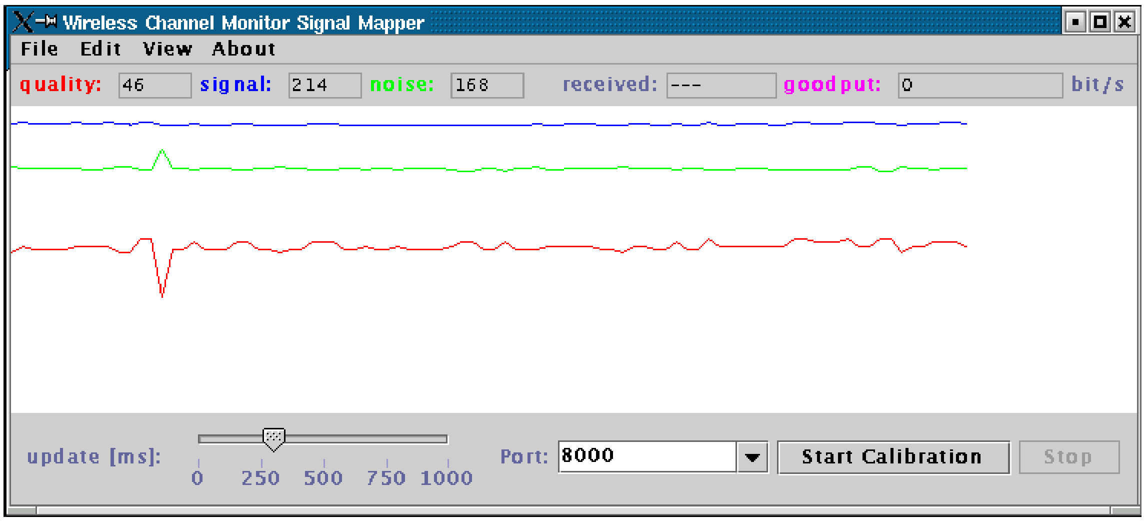
#### Configuration

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, 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), it should be set to “signal”. *This must be set to the same property which is sent by the modified device driver (see Section 6.4) 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. if the test mobile is moving). (default: `100`)

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.

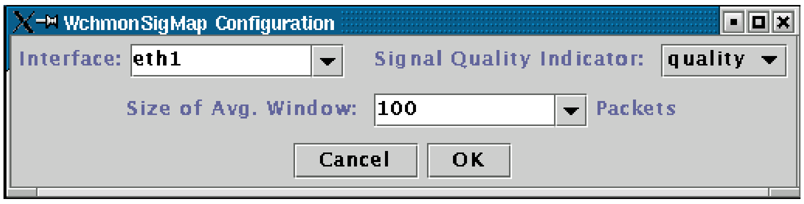


quality: 46 signal: 214 noise: 168 received: --- goodput: 0 bit/s

update [ms]: 0 250 500 750 1000

Port: 8000

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



Interface: eth1 Signal Quality Indicator: quality

Size of Avg. Window: 100 Packets

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

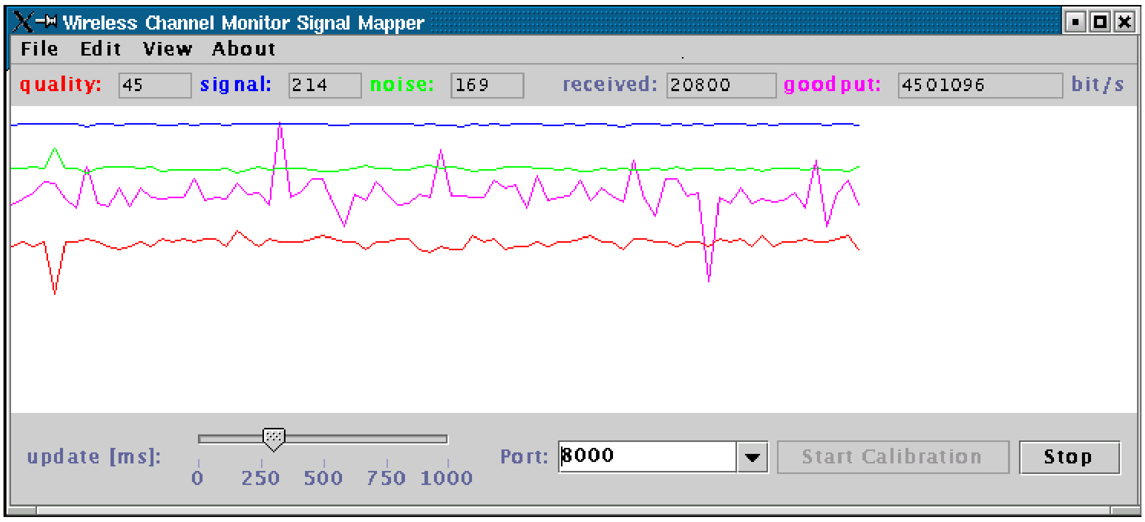
### Note: Signal Levels in Ad-Hoc Mode

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[2] 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 that of 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.

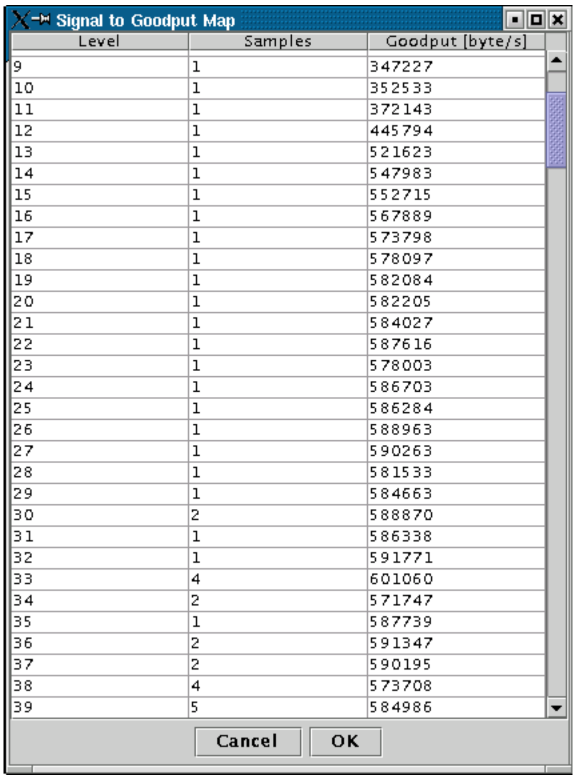
[2]For calibrations done as part of this thesis, UDP packet size was 1024 Bytes (since this was also the size of data packets generated in the prototype measurements) and rates were calculated over a window of 100 packets.



quality: 45 signal: 214 noise: 169 received: 20800 goodput: 4501096 bit/s

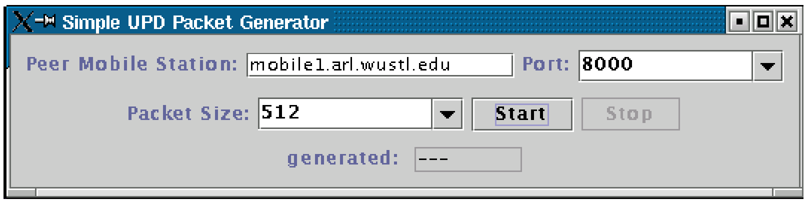
Port: 8000

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



| Level | Samples | Goodput [byte/s] |
|---|---|---|
| 9 | 1 | 347227 |
| 10 | 1 | 352533 |
| 11 | 1 | 372143 |
| 12 | 1 | 445794 |
| 13 | 1 | 521623 |
| 14 | 1 | 547983 |
| 15 | 1 | 552715 |
| 16 | 1 | 567889 |
| 17 | 1 | 573798 |
| 18 | 1 | 578097 |
| 19 | 1 | 582084 |
| 20 | 1 | 582205 |
| 21 | 1 | 584027 |
| 22 | 1 | 587616 |
| 23 | 1 | 578003 |
| 24 | 1 | 586703 |
| 25 | 1 | 586284 |
| 26 | 1 | 588963 |
| 27 | 1 | 590263 |
| 28 | 1 | 581533 |
| 29 | 1 | 584663 |
| 30 | 2 | 588870 |
| 31 | 1 | 586338 |
| 32 | 1 | 591771 |
| 33 | 4 | 601060 |
| 34 | 2 | 571747 |
| 35 | 1 | 587739 |
| 36 | 2 | 591347 |
| 37 | 2 | 590195 |
| 38 | 4 | 573708 |
| 39 | 5 | 584986 |

**Figure A.4:** *Editing the results.*



Peer Mobile Station: mobile1.arl.wustl.edu Port: 8000
Packet Size: 512
generated: ---

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

### 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 6.4). 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.

## 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

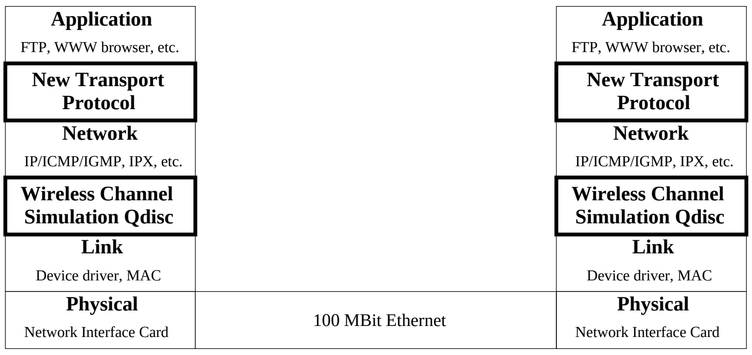


**Figure A.6:** *Example for the performance evaluation of a new/improved transport protocol using the wireless simulation queuing discipline.*

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 7.2.1 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 wireline network interface, whose bandwidth is considerably larger than that of the simulated wireless device, the additional effects of the wireline 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.

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 reordering of packets.

A similar system was proposed by Noble et. al. in [43], where a trace collected in a simulation environment is used to emulate the behavior of a wireless LAN within a wired network. In contrast to their approach, our “simulation qdisc” performs the wireless simulation within the kernel avoiding the need to generate simulation traces beforehand.

## 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 - CSFIFO
 *
 * goodput optimizing wireless fifo qdisc using the "ratio"
 * wireless channel monitor and running on the wireless
 * channel simulation qdisc
 */

#include "ip.def"

/*
 * define packets
 */

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

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


dev eth0 1400 /* 1.4 Mbit/s */

/* Installing the channel monitor module can only be done after
   the "dev" command! We are using the " ratio " monitor. */

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

/* Setup TC architecture */
/* (This is for testing only ... just arbitrary values.) */

/* 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 \
   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  \
   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

/* 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
   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 .                                                   */

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

Additional results to supplement the material previously presented in Chapter 7 for Scenario 1 of the simulations are provided in this subsection.

#### Additional Simulation Results

Tables A.2 and A.3 list the average goodput rates and 95% confidence intervals for the two mobile stations in Scenario 1 presented in Section 7.3.1.

| 1/GTR of MS 2 | Mobile Station | Scheduler | Avg. [kbit/s] | 95% CI [kbit/s] |
|---|---|---|---|---|
| 1 | 1 | mHFSC | 4887 | ± 1.1 |
| | | CBQ | 4887 | ± 1.1 |
| | 2 | mHFSC | 607 | ± 1.0 |
| | | CBQ | 607 | ± 1.0 |
| 2 | 1 | mHFSC | 4887 | ± 1.9 |
| | | CBQ | 4887 | ± 1.9 |
| | 2 | mHFSC | 607 | ± 1.0 |
| | | CBQ | 607 | ± 1.0 |
| 3 | 1 | mHFSC | 4887 | ± 3.4 |
| | | CBQ | 4321 | ± 2.0 |
| | 2 | mHFSC | 419 | ± 1.3 |
| | | CBQ | 607 | ± 1.0 |
| 5 | 1 | mHFSC | 4887 | ± 4.0 |
| | | CBQ | 3106 | ± 2.7 |
| | 2 | mHFSC | 251 | ± 0.9 |
| | | CBQ | 607 | ± 1.0 |
| 7 | 1 | mHFSC | 4886 | ± 6.3 |
| | | CBQ | 2481 | ± 1.8 |
| | 2 | mHFSC | 180 | ± 1.0 |
| | | CBQ | 523 | ± 0.9 |
| 10 | 1 | mHFSC | 4885 | ± 11.0 |
| | | CBQ | 1976 | ± 3.4 |
| | 2 | mHFSC | 126 | ± 1.2 |
| | | CBQ | 417 | ± 1.0 |

**Table A.2:** *Average goodput rates of the two mobile stations in Scenario 1 of the simulations (Section 7.3.1, Figure 7.7) for varying adaptive modulation of MS 2.*

| $p_b$ of MS 2 | Mobile Station | Scheduler | Avg. [kbit/s] | 95% CI [kbit/s] |
|---|---|---|---|---|
| 0 | 1 | mHFSC | 4887 | ± 1.1 |
| | | CBQ | 4887 | ± 1.1 |
| | 2 | mHFSC | 607 | ± 1.0 |
| | | CBQ | 607 | ± 1.0 |
| 0.02 | 1 | mHFSC | 4887 | ± 5.3 |
| | | CBQ | 4878 | ± 11.0 |
| | 2 | mHFSC | 603 | ± 3.1 |
| | | CBQ | 603 | ± 2.2 |
| 0.05 | 1 | mHFSC | 4885 | ± 8.4 |
| | | CBQ | 4818 | ± 32.1 |
| | 2 | mHFSC | 598 | ± 5.6 |
| | | CBQ | 592 | ± 5.7 |
| 0.20 | 1 | mHFSC | 4875 | ± 16.4 |
| | | CBQ | 4606 | ± 58.3 |
| | 2 | mHFSC | 565 | ± 17.2 |
| | | CBQ | 551 | ± 8.3 |
| 0.33 | 1 | mHFSC | 4874 | ± 15.1 |
| | | CBQ | 4224 | ± 95.0 |
| | 2 | mHFSC | 491 | ± 29.9 |
| | | CBQ | 503 | ± 11.8 |
| 0.50 | 1 | mHFSC | 4878 | ± 15.5 |
| | | CBQ | 3567 | ± 93.9 |
| | 2 | mHFSC | 263 | ± 28.2 |
| | | CBQ | 435 | ± 15.0 |

**Table A.3:** *Average goodput rates of the two mobile stations in Scenario 1 of the simulations (Section 7.3.1, Figure 7.8) for a varying probability of a bad channel-state of MS 2 in the Markov model.*

#### Configuration of Link-Sharing Hierarchy

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 6144kbit \
     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 7.10 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 2 & 3
 * ---------------------
 *
 *  1 access point , 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          /* +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 1600kbit \
     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 20 ms .050Mbit]
tc class add dev eth1 parent 10:1200 classid 10:1202 hfsc [ sc 0 20 ms .050Mbit]
tc class add dev eth1 parent 10:1200 classid 10:1203 hfsc [ sc 0 20 ms .050Mbit]

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



/* customers of second agency, VoIP, 30 users */
tc class add dev eth1 parent 10:2100 classid 10:2101 hfsc [ sc 0.03 Mbit 20ms 0.02Mbit ] \
/* [ dc 0 0 .015 Mbit ] */
tc class add dev eth1 parent 10:2100 classid 10:2102 hfsc [ sc 0.03 Mbit 20ms 0.02Mbit ] \
...until
/* [ dc 0 0 .015 Mbit ] */
tc class add dev eth1 parent 10:2100 classid 10:2130 hfsc [ sc 0.03 Mbit 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 0.03s send PACKET_VoIP(1) PAYLOAD_VoIP(1)
every 0.03s send PACKET_VoIP(2) PAYLOAD_VoIP(2)
...until
every 0.03s send PACKET_VoIP(40) PAYLOAD_VoIP(40)


/* (The remaining traffic is generated using TrafGen.) */
```

# B. 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 drivers 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.)

Because of space limitations, the listings of the various source code files are not included in this thesis. The complete sources, including the patch for the Linux 2.4.X kernel, are distributed under the GNU Public License and can be downloaded at [66]. Furthermore, the most important source code files of the prototype implementation are documented in a technical report [67].

| Path | File | Changed /New | Description |
|---|---|---|---|
| `/net` | `netsyms.c` | C | Added export of channel monitor interface. |
| `/net/sched` | `Config.in` | C | Added configuration options. |
| `/net/sched` | `sch_chsim.c` | N | Wireless channel simulation qdisc. |
| `/net/sched` | `sch_csfifo.c` | N | Example qdisc: wireless FIFO. |
| `/net/sched` | `sch_hfsc.c` | N | Wireless H-FSC qdisc. |
| `/net/sched` | `sch_hfsc.h` | N | Data structures for wireless H-FSC. |
| `/net/sched` | `wchmon_api.c` | N | Wireless channel monitor interface. |
| `/net/sched` | `wchmon_driver.c` | N | NIC driver based channel monitor. |
| `/net/sched` | `wchmon_dummy.c` | N | Dummy channel monitor. |
| `/net/sched` | `wchmon_ratio.c` | N | Dequeuing delay based channel monitor. |
| `/net/sched` | `Makefile` | C | Added new modules. |
| `/include/net` | `pkt_sched.h` | C | Clock source changed. (see Section A.2) |
| `/include/net` | `pkt_wsched.h` | N | Declarations for wireless scheduling. |
| `/include/linux` | `pkt_sched.h` | C | Added netlink structures for H-FSC. |
| `/include/linux` | `pkt_wsched.h` | N | Wireless specific netlink structures. |
| `/include/linux` | `netdevice.h` | C | Channel monitor added to network device structure. |

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

| Path | File | Changed /New | Description |
|---|---|---|---|
| `/iproute2/tc` | `q_chsim.c` | N | Parsing options for `chsim` qdisc. |
| `/iproute2/tc` | `q_csfifo.c` | N | Parsing options for `csfifo` qdisc. |
| `/iproute2/tc` | `q_hfsc.c` | N | Parsing options for `hfsc` qdisc. |
| `/iproute2/tc` | `wsched_util.c` | N | Common functions for parsing of wireless scheduler options. |
| `/iproute2/tc` | `wsched_util.h` | N | Declarations for parsing of wireless scheduler options. |

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

| Path | File | Changed /New | Description |
|---|---|---|---|
| `/pcmcia-cs/wireless` | `ray_cs.c` | C | Raylink driver - added calls to channel monitor interface. |
| `/pcmcia-cs/wireless` | `wvlan_cs.c` | C | Lucent/Avaya driver - added calls to channel monitor interface. |
| `/pcmcia-cs` | `Configure` | C | Added wireless configuration. |

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

# Acknowledgments

This work would not have been possible without the support of many others. Most of the research was done at the Applied Research Laboratory, Washington University in St. Louis, and I am very thankful that I had the exciting opportunity to work there. I would like to thank Professor John Lockwood, Professor Gruia-Catalin Roman, and many others at the computer science department of Washington University for making this stay possible. Professor Lockwood was an excellent advisor and always had time to discuss my questions. I would also like to thank Nicola Aschwanden and Marcel Waldvogel for all their help during my first weeks in St. Louis, Ramaprabhu Janakiraman for many suggestions and even more cups of coffee, which improved early versions of this work, also Sarang Dharmapurikar and Wei Deng for enduring me as office mate and contributing equipment for the wireless measurements.

At TU Berlin, I owe many thanks to Professor Adam Wolisz of the Telecommunication Research Group and to the International Office, especially Jutta Gbur, for their support. In addition, I would like to thank Ana Cristina Costa Aguiar for suggesting many improvements and helping me to get everything into the shape of a thesis. Furthermore, James Gross and Michael Jaeger did put a lot of time and effort in proofreading the final version and made many useful comments – I am very grateful for the interesting time we spent studying together in Berlin.

Finally, I would like to thank my family for their unconditional support and encouragement, especially my fiancée Lena for her patience, help, and always being there.

# Acronyms

| | |
|---|---|
| ABR | Available Bit Rate |
| ACK | ACKnowledgment frame |
| AF | Assured Forwarding |
| AGC | Automatic Gain Control |
| ANSI | American National Standards Institute |
| AP | Access Point |
| API | Application Programmer Interface |
| ARP | Address Resolution Protocol |
| ARQ | Automatic Repeat reQuest |
| ATM | Asynchronous Transfer Mode |
| BR | Bit-by-bit Round robin |
| BSD | Berkeley Software Distribution |
| CBR | Constant Bit Rate |
| CBQ | Class Based Queuing |
| CDMA | Code Division Multiple Access |
| CIF | Channel-condition Independent Fair |
| CIF-Q | Channel-condition Independent Fair Queuing |
| CSDPS | Channel-State Dependent Packet Scheduling |
| CSMA/CA | Carrier Sense Multiple Access/Collision Avoidance |
| CSMA/CD | Carrier Sense Multiple Access/Collision Detection |
| CS-WFQ | Channel-State Independent Wireless Fair Queuing |
| CTD | Cell Transfer Delay (ATM) |
| CTS | Clear To Send |
| DiffServ | Differentiated Services |
| DL | Data Link Layer |
| DoS | Denial of Service |
| DRR | Deficit Round Robin |
| DS | Differentiated Services |
| DSCP | Differentiated Services CodePoint |
| DSSS | Direct Sequence Spread Spectrum |
| EF | Expedited Forwarding |
| FCFS | First Come First Served |
| FDMA | Frequency Division Multiple Access |
| FEC | Forward Error Correction |
| FHSS | Frequency Hopping Spread Spectrum |
| FIFO | First In First Out |
| FQ | Fair Queuing |
| FTP | File Transfer Protocol |

| | |
|---|---|
| GFR | Guaranteed Frame Rate |
| GPS | Generalized Processor Sharing |
| GTR | Goodput to raw Throughput Ratio |
| H-GPS | Hierarchical Generalized Processor Sharing |
| H-FSC | Hierarchical Fair Service Curve algorithm |
| HOL | Head Of Line |
| H-PFQ | Hierarchical Packet Fair Queuing |
| ICMP | Internet Control Message Protocol |
| IEEE | Institute of Electrical and Electronics Engineers |
| IETF | Internet Engineering Task Force |
| IntServ | Integrated Services |
| IO | Input Output |
| IOCTL | Input Output ConTroL |
| IP | Internet Protocol |
| ISP | Internet Service Provider |
| IWFQ | Idealized Wireless Fair Queuing |
| JNI | Java Native Interface |
| LAN | Local Area Network |
| LCFS | Last Come First Out |
| LLC | Logical Link Control |
| LSP | Label Switched Path |
| LSR | Label Switched Router |
| LTFS | Long-Term Fairness Server |
| MAC | Medium Access Control |
| MCR | Minimum Cell Rate |
| MPFQ | wireless Multi-class Priority Fair Queuing |
| MPLS | MultiProtocol Label Switching |
| MS | Mobile Station |
| NAT | Network Address Translation |
| NIC | Network Interface Card |
| nrtVBR | non-real-time Variable Bit Rate |
| OS | Operating System |
| OSI | Open Systems Interconnection |
| PCI | Peripheral Component Interconnect (Intel) |
| PCMCIA | Personal Computer Memory Card International Association |
| PGPS | Packet-by-packet Generalized Processor Sharing |
| PHB | Per-Hop-Behavior |
| PHY | PHYsical layer |
| PSA | Parameter Storage Area |
| QoS | Quality of Service |
| RED | Random Early Detection |
| RF | Radio Frequency |
| RSVP | Resource reSerVation Protocol |
| RTS | Request To Send |
| rtVBR | real-time Variable Bit Rate |
| SBFA | Server Based Fairness Approach |
| SC | Service Curve |
| SFQ | Start-time Fair Queuing |
| SLA | Service Level Agreement |

| | |
|---|---|
| SNR | Signal to Noise Ratio |
| STFQ | Stochastic Fair Queuing |
| TBF | Token Bucket Filter |
| TC | Traffic Control |
| TCC | Traffic Control Compiler |
| TCF | Traffic Control Filter |
| TCNG | Traffic Control Next Generation |
| TCSIM | Traffic Control Simulator |
| TCP | Transmission Control Protocol |
| TDMA | Time Division Multiple Access |
| TOS | Type Of Service |
| UBR | Unspecified Bit Rate |
| UDP | User Datagram Protocol |
| VC | Virtual Channel |
| VoIP | Voice over IP |
| WEP | Wired Equivalent Privacy (IEEE 802.11) |
| $WF^2Q$ | Worst-case Fair Weighted Fair Queuing |
| $WF^2Q+$ | Worst-case Fair Weighted Fair Queuing plus |
| $W^2F^2Q$ | Wireless Worst-case Fair Weighted Fair Queuing |
| WFQ | Weighted Fair Queuing |
| WFS | Wireless Fair Service |
| WLAN | Wireless LAN |
| WPS | Wireless Packet Scheduling |
| WRR | Weighted Round Robin |
| WWW | World Wide Web |
| XML | eXtensible Markup Language |

# Bibliography

[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] Dan Decasper, Zubin Dittia, Guru Parulkar, and Bernhard Plattner. Router plugins: A software architecture for next generation routers. In *Proceedings of ACM SIGCOMM 1998*, Vancouver, British Columbia, Sep. 1998.

[12] Alan Demers, Srinivasan Keshav, and Scott Shenker. Analysis and simulation of a fair queueing algorithm. In *Proceedings of SIGCOMM '89*, pages 3–12, 1989.

[13] 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.

[14] 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.

[15] 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.

[16] Sally Floyd. Notes on CBQ and guaranteed service, 1995.

[17] Sally Floyd and Van Jacobsen. Link and resource management models for packet networks. *IEEE/ACM Transactions on Networking*, 3(4), 1995.

[18] Sally Floyd and Van Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, August 1993.

[19] 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.

[20] J. Gross, M. Jaeger, and A. Willig. Measurements of a wireless link in different RF-isolated environments. Technical Report TKN-01-005, Telecommunication Networks Group, Technische Universität Berlin, June 2001.

[21] J. Heinanen, F. Baker, W. Weiss, and J. Wroclawski. RFC2597 - assured forwarding PHB group. Standards Track, Jun. 1999.

[22] 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`.

[23] Raylink Inc. Raylink user manual. `http://www.raylink.com/pdf/pccard.pdf`.

[24] V. Jacobson, K. Nichols, and K. Poduri. RFC2598 - an expedited forwarding PHB. Standards Track, Jun. 1999.

[25] Linux sources and GNU general public license. `http://www.kernelnotes.org`.

[26] S. Keshav. On the efficient implementation of fair queueing. *Journal of Internetworking Research and Experience*, 2(3), 1991.

[27] Alexey Kuznetsov. iproute2 package. `ftp://ftp.inr.ac.ru/ip-routing/`.

[28] L. Keng Lim, Jun Gao, T.S. Eugene Ng, Prashant Chandra, Peter Steenkiste, and Hui Zhang. Customizable virtual private network service with QoS. *Computer Networks, Elsevier Science, special issue on Overlay Networks*, to appear in 2001.

[29] 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.

[30] 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.

[31] Songwu Lu, Vaduvur Bharghavan, and R. Srikant. Fair scheduling in wireless packet networks. *Proceedings of ACM SIGCOMM 1997*, 1997.

[32] Matthew T. Lucas, Dallas E. Wrege, Bert J. Dempsey, and Alfred C. Weaver. Statistical characterization of wide-area ip traffic. In *Proceedings of Sixth International Conference on Computer Communications and Networks (IC3N'97)*, Las Vegas, NV, USA, Sep. 1997.

[33] Paul E. McKenney. Stochastic fairness queueing. *Journal of Internetworking Research and Experience*, 2:113–131, 1991.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] Christian Worm Mortensen. An implementation of a weighted round robin scheduler for linux traffic control. `http://wipl-wrr.sourceforge.net`.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] Brian Noble, M. Satyanarayanan, Giao Thanh Nguyen, and Randy H. Katz. Trace-based mobile network emulation. In *SIGCOMM*, pages 51–61, 1997.

[44] David Olshefski. *TC API Beta 1.0*. IBM T.J. Watson Research. `http://oss.software.ibm.com/developerworks/projects/tcapi/`.

[45] 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.

[46] PCMCIA card services for linux. `http://pcmcia-cs.sourcforge.net`.

[47] 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.

[48] Parameswaran Ramanathan and Prathima Agrawal. Adapting packet fair queueing algorithms to wireless networks. In *Mobile Computing and Networking*, pages 1–9, 1998.

[49] E. Rosen, A. Viswanathan, and R. Callon. RFC3031 - multiprotocol label switching architecture. Standards Track, Jan. 2001.

[50] Rusty Russell. Linux 2.4 packet filtering howto. `http://netfilter.samba.org`.

[51] S. Shenker, C. Partridge, and R. Guerin. RFC2212 - specification of guaranteed quality of service. Standards Track, Sept. 1997.

[52] 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.

[53] M. Shreedhar and George Varghese. Efficient fair queueing using deficit round robin. In *Proceedings of SIGCOMM '95*, pages 231–242, 1995.

[54] IEEE Computer Society. IEEE std. 802.11: Wireless LAN medium access control (MAC) and physical layer (PHY) specications, Jun. 1997.

[55] IEEE Computer Society. Supplement to IEEE std. 802.11: Wireless LAN medium access control (MAC) and physical layer (PHY) specications: Higher-speed physical layer extensions in the 2.4 GHz band, Sep. 1999.

[56] 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.

[57] W. Richard Stevens. *TCP/IP Illustrated: The Protocols*. Addison-Wesley Publishing Company, 1994. ISBN 0-201-63346-9 (v.1).

[58] 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/`.

[59] SUN Java 2 platform version 1.3 (standard edition). `http://java.sun.com/j2se/1.3/`.

[60] Agere Systems. User's guide for orinoco PC card, Sept. 2000. `ftp://ftp.orinocowireless.com/pub/docs/ORINOCO/MANUALS/ug_pc.pdf`.

[61] Jean Tourrilhes. *Linux Wireless Extension/Wireless Tools*. Hewlett Packard. `http://www.hpl.hp.com/personal/Jean_Tourrilhes/Linux/Tools.html`.

[62] Zheng Wang. *Internet QoS: Architectures and Mechanisms for Quality of Service*. Morgan Kaufmann Publishers, 2001.

[63] Ellen Kayata Wesel. *Wireless Multimedia Communications - Networking Video, Voice, and Data*. Addison-Wesley, 1997.

[64] 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.

[65] 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.

[66] Lars Wischhof. Modified h-fsc scheduler and linux wireless scheduling patch. `http://wsched.sourceforge.net`.

[67] Lars Wischhof and John Lockwood. Packet scheduling for link-sharing and quality of service support in wireless local area networks. Technical Report WUCS-01-35, Applied Research Laboratory, Washington University in St. Louis, November 2001.

[68] Lars Wischhof and John Lockwood. A resource-based approach to MAC layer independent hierarchical link-sharing in wireless local area networks. In *Proceedings of European Wireless 2002*, Florence, Italy, Feb. 2002.

[69] J. Wroclawski. RFC2211 - specification of the controlled-load network element service. Standards Track, Sept. 1997.

# Index