Prüfer:	Prof. Dr.-Ing. Dr. h. c. mult. P.J. Kühn
	Prof. Dr. rer. nat. K. Rothermel
Betreuer:	Dipl.-Ing. M. Lorang
	Dipl.-Ing. L. Burgstahler
Beginn am:	1. Juni 1999
Beendet am:	1. Dezember 1999
CR-Nummer:	C.2.1, C.2.5, C.2.6

Diplomarbeit-Nr. 1767

Implementation of a Traffic-Control Architecture for RSVP over ATM

Vladimir Šironja

Abstract

The IETF´s Integrated Services architecture defines Guaranteed Service and Controlled Load Service as new Service classes for the Internet extending the traditional best-effort service. In this architecture, the signaling protocol RSVP supports reservation of Quality of Service in the routers connecting two hosts over an IP network. This work presents an extension to the Linux kernel for routers at the border between ATM and IP networks, which map the QoS requirements of Integrated Services flows into ATM VCs. The extension enables merging of multiple Guaranteed Service flows into one single VC. The Rotating Prioity Queues (RPQ) algorithm is implemented and used for bounding the delay of the merged flows. Furthermore, this work describes the implementation details of the traffic control and ATM modules in the kernel. Finally, the implementation of a simple traffic generator that allows the testing of signaling and measurement of queueing delay is presented.

Zussamenfassung

Die Integrated Services Internet Architektur definiert verschiedene Echtzeit-Klassen neben der default best-effort Klasse. Das signallisierungsprotokol RSVP ermöglicht eine Reservierung der Dienstgüte in den Routern zwischen zwei Rechner für diese Dienste. Diese Arbeit stellt eine Erweiterung des Linux-Kernels vor, für Router zwischen IP- und ATM-Netzen, die einen Integrated Services Verkehrsstrom in eine ATM - VC abbilden. Die Erweiterung ermöglicht eine Aggregation meherer Verkehrsströme in eine ATM-VC. Der RPQ (*Rotating Prioity Queues*) Algorithmus wird implementiert und benutzt, um die maximale erlaubte Verzögerung der aggregierten Verkehrströme zu begrenzen. Diese Arbeit erläutert die Struktur des Verkehrsmanagements in Linux Kernel, sowie die ATM-Kernel Erweiterung. Weitehin wird die Implementierung eines einfachen Lastgenerators und der Verzögerungsmessung beschrieben.

Contents

1 Introduction **1**

2 Convergence of Integrated Services and ATM **3**
2.1 Traffic shaping . . . 3
2.2 Queueing . . . 4
2.3 Integrated Services . . . 5
2.3.1 Guaranteed Service . . . 5
2.3.2 Controlled load service . . . 7
2.3.3 Architecture . . . 7
2.3.4 Data Link Layer Technology . . . 8
2.4 RSVP . . . 9
2.4.1 Protocol Elements . . . 9
2.4.2 RSVP-Daemon . . . 12
2.5 ATM . . . 13
2.5.1 CLIP . . . 14
2.5.2 ATM - API . . . 15
2.6 RSVP over ATM . . . 15
2.6.1 DIANA Architecture . . . 15
2.6.2 DIANA Atmtcd . . . 17

3 Kernel Modules **19**
3.1 Linux TCP/IP Implementation . . . 19
3.1.1 Socket Layer . . . 20
3.1.2 Socket Buffers . . . 20
3.1.3 Transmitting packets . . . 22
3.1.4 Receiving packets . . . 23
3.2 Traffic Control in Linux-Kernel . . . 23
3.2.1 Queueing disciplines . . . 24
3.2.2 Classful queueing discipline . . . 26
3.2.3 Traffic Filters . . . 28
3.2.4 Code . . . 29
3.3 RTNetlink . . . 30
3.3.1 Netlink Sockets . . . 31
3.3.2 Traffic Control & Routing Netlink . . . 32
3.3.3 `tc` Program . . . 33
3.4 ATM in the Linux Kernel . . . 35
3.4.1 ATM-Sockets . . . 35
3.4.2 CLIP . . . 35

4 DIANA Traffic Control Extension **38**
4.1 Adding new elements to traffic control . . . 38
4.2 Dprio . . . 39
4.3 ATM-Dprio . . . 41
4.4 RPQ . . . 43
4.4.1 Architecture . . . 43
4.4.2 Implementation . . . 45
4.4.3 Timers and Jiffies . . . 47
4.5 SDRSVP-Filter . . . 47
4.6 Parking Queue . . . 51
4.6.1 Architecture . . . 52
4.7 Avoiding Race Conditions . . . 54
4.7.1 Bottom Half Scheduling . . . 54
4.7.2 Race Conditions in Traffic Control . . . 56
4.8 ATMTCD . . . 57
4.8.1 Kernel TC Architecture . . . 57
4.8.2 ATMTCD-Interface to TC . . . 57
4.8.3 Runtime Scenario . . . 59

5 Component-Test and Traffic Generation **61**
5.1 Testing the User-Space Interface . . . 61
5.1.1 DejaGnu . . . 62
5.2 Traffic Generation . . . 63
5.2.1 Requirements . . . 63
5.2.2 Architecture . . . 63
5.2.3 Usage . . . 65

6 Graphical Frontend for Traffic Control **68**
6.1 Tcl/Tk, incr Tcl and TclX . . . 68
6.2 Architecture . . . 68
6.3 Usage . . . 70

7 Summary **72**

Bibliography **74**

A Mapping OO to C **78**

List of Figures

2.1 Token Bucket 4
2.2 Maximum delay with a token bucket model and fixed service rate 6
2.3 RSVP Objects 10
2.4 RSVP Messages 11
2.5 RSVP Daemon code structure 12
2.6 Classical IP over ATM (CLIP) 14
2.7 ATM Traffic Control Daemon (atmtcd) 17

3.1 Socket buffer 21
3.2 Traffic control in the protocol stack 24
3.3 IP output and traffic control 25
3.4 Runtime structure of a classful queueing discipline 26
3.5 UML diagram of traffic control elements 27
3.6 Rtnetlink message format 32
3.7 ATM Sockets internal structure 36
3.8 CLIP internal structure 37

4.1 Dprio classful qdisc 39
4.2 ATMDprio 42
4.3 RPQ Example 44
4.4 RPQ 45
4.5 SDRSVP Filter 48
4.6 SDRSVP Subfilter types and positions in the hash table 49
4.7 Inner queues in the Parking qdisc 53
4.8 Scheduling Example 56
4.9 Runtime structure of the atmdprio qdisc 58
4.10 MSC for scenario 1 60

5.1 Client/Server architecture of traffic generation 64

6.1 UML view of the incr Tcl code 69
6.2 TC-GUI Snapshot 71

A.1 Example class associations 78

List of Tables

2.1 Adspec parameters 11
2.2 Integrated Services to ATM mapping 16

3.1 Skbuff methods (`struct sk_buff`) 22
3.2 Skbuff-list methods (`struct sk_buff_head`) 22
3.3 Netlink header fields 34

4.1 TC_MESSAGE Part of the Rtnetlink message with Dprio 40
4.2 Rt-attributes for message type RTM_NEWTCLASS 43
4.3 Rt-attributes for message type RTM_NEWQDISC 47
4.4 TC_MESSAGE Part of the Rtnetlink message with sdrsvp 51
4.5 Access kinds of traffic control methods 57

Chapter 1

Introduction

The Internet has grown up from the big network supporting non-realtime applications like WWW and email to an even bigger network connecting people with realtime application like Voice over IP and Videoconferencing. This new trend of supporting audio, video, realtime and classical data traffic over packet-switched networks has introduced new Quality of Service requirements, not being served well by a best effort type of service the Internet offers today.
Several different extensions of IPv4 have been proposed by IETF (*Internet Engineering Task Force*) and implemented by diverse companies and universities. The most important extensions are IPv6, Integrated Services and Differentiated Services.

This work concentrates on the Integrated Services extension for the Internet and its reservation protocol RSVP. RSVP is a signaling protocol invoking QoS reservations in routers. Possible solutions for interoperation of Integrated Services IP networks with RSVP at one side and ATM at the other have been investigated and developed in the DIANA Project. The work presented here is an extension of the DIANA Project´s work on a RSVP over ATM control architecture.

The main goals of this work can roughly be described as following:

- To investigate the ability of adding new traffic control elements into the Linux kernel and how these elements can be accessed from applications.
- To design and implement the ATM traffic control architecture that enables merging of RSVP-flows into a single VC.
- Implementation of the RPQ queueing discipline.
- Performing measurements on the ATM traffic control.

The results of the work presented here is an extension of the Linux kernel supporting the RSVP over ATM architecture, a traffic generation program designed and implemented to support specific measurement needs and a graphical user interface for traffic control. Besides designing and implementing software, a fairly large amount of time was spent fixing and adapting traffic control software as well as and configuring the network topology and testing environment.

Survey This thesis is structured as follows:
Chapter 2 examines the requirements for Quality of Service supporting architecture in next generation Internet and clarifies the IETF´s solutions and proposals for achieving guaranteed maximum

delay over IP-Networks. Further, Integrated Services and the reservation protocol RSVP are described, with a short description of the DIANA Project´s architecture.

Chapter 3 explains and documents parts of the Linux kernel relevant to the extension. The basic structure of the Linux network part of the kernel is covered. Traffic control module and Rtnetlink sockets code is described, followed with a look at how those two interoperate. Finally, a short description of the ATM kernel patch is presented.

Chapter 4 shows the developed traffic control extension, describing requirements leading to specific design decisions. The code structure is explained in detail, along with some clarification of how the optimization issues and race conditions in the kernel influence the code.

Chapter 5 describes the the component test and the traffic generation program, which was used for performing measurements. The measurement results are evaluated.

Chapter 6 is a short description of the graphical user interface to traffic control, developed with *incr Tcl*, an object oriented extension to a Tcl scripting language.

Appendix A shows a simple and broadly used way of mapping object oriented design into C programming language.

Style Except for chapter 2, where the theoretical background on the relevant network domain is presented, all the following chapters are written in the programmer´s-point-of-view-style. Although the author tried avoiding presenting too much code, the chapters turned out to be similar to some Linux programmer´s - reading. Chapter 3 covers some undocumented parts of the Linux kernel, while chapter 4 presents the implementation in detail; both containing a fair amount of C code portions. UML has been chosen as an industry-standard description language. The mapping of UML into C is then shown in appendix A.

Versions The working environment were two Red Hat 5.0 Linux PCs connected with an ATM Switch. The kernel version 2.2.9 was used, together with the corresponding ATM patch 0.59.

Chapter 2

Convergence of Integrated Services and ATM

This chapter presents two main mechanisms of Quality of Service enabled networks; traffic shaping and queueing disciplines. Further, the Integrated Services model for the Internet, and its architecture components, including RSVP, are presented. Finally, the theoretical background of mapping integrated services onto ATM and the weakness of the simple CLIP approach, when connecting the RSVP-aware subnets through ATM clouds. We discuss the preliminary work on interconnecting RSVP and ATM at the edge-point routers as a part of the DIANA Project.

2.1 Traffic shaping

Quality of Service has been imposed on IP-networks by emerging delay-sensitive multimedia applications. These applications need guarantees and some control over end-to-end delay and average packet loss. Packets arriving out of time can be considered lost by such applications.
Congestion is the main cause of packet loss. Traffic congestion occurs when the queues fill up to an extent where no packets can be accepted. As a result, newly arrived packets are discarded.
If all the traffic were transmitted at some uniform rate, congestions would not be as common. Instead, most of the traffic is bursty. Decreasing the burstiness is the main objective of algorithms like leaky bucket and token bucket[12][15].

Token bucket model (Figure 2.1) parameters are r, rate at which tokens pour into the bucket, measured in bytes per second and depth b in bytes. For each byte of the packet size, the bucket must take and destroy one byte-token. When the connection is idle, the bucket can accumulate up to b tokens, eventually allowing bursts afterwards. So calculating the length of the maximum burst length of some sender transmitting at peak rate p, we get:

$$pt = b + rt \Rightarrow t = \frac{b}{p - r} \tag{2.1}$$

The left hand side of the equation is the input at the peak rate, and the right hand side is the arrival of the tokens plus the maximum amount of accumulated tokens.
The Token bucket algorithm plays two different roles in traffic control:

1. As a *model*, it quantifies the traffic characteristics. It maps the source attributes in measurable values. Thus, the network knows what kind of traffic is to expect, and *admission*

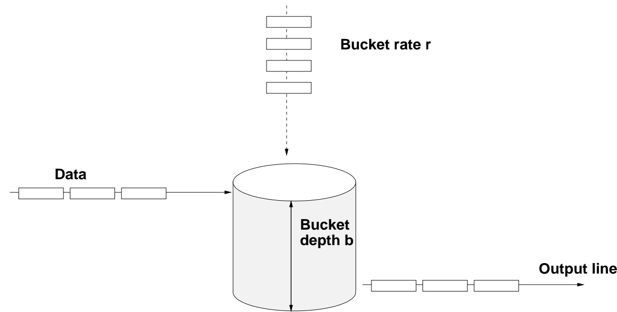

Figure 2.1: Token Bucket

control can decide if the ressources suffice; i.e. if the QoS can be guaranteed without imposing reduced QoS on the previous guarantees. The second function of the admission control is deciding if the user is allowed to send the described type of traffic.

2. As an algorithm or as a filter, token bucket can smoothen bursty traffic and discard *non-conforming* packets; packets arriving at higher rate then the network can handle. Thus, we define a fair *Packet discard policy* and enable flow-control.

2.2 Queueing

The second step in the objective of providing real-time QoS is bounding delay. The packet-delay caused by the transmission from one node to another is composed from: the fixed *transmission delay*, i.e. the time needed for the signal transmission, *queueuing delay*, the time spent in switches or routers, and processing time, the time processor uses to compute protocol relevant data and move the packet through the protocol stack. The time between arrival on some input line and leaving on some output line is spent in router´s queue. The sum of all queue-delays along the routers on the path and the fixed delay is the total end-to-end delay.

The essence of all queueing algorithms is the fact that flows from multiple input queues may share one output queue. Although we mostly multiplex n input lines to more then one output, it doesn´t change the nature of the problem. Now we take a look at some queueing algorithms.
Plain **FIFO queue** works best when connecting one input line to one output. As the packets arrive, they are being enqueued. When the output line is able to handle packets, it dequeues one packet. Having more then one queue introduces a problem of deciding which queue is to be served first.
Weighted fair queueing (WFQ) handles multiple input lines in a fair way, regarding the packet-size. The queues are served in bit-by-bit round-robin algorithm, and thus giving every line exactly the same amount of bandwidth and imposing a *fair queueing policy*.

All of those algorithms work well on fair-queueing and congestion-avoidance, but cannot limit delays on the real-time QoS flows. In order to enable bounded delay, packets have to be classified and thus separated into different priority levels. With other words, we need to define different *packet service policy* for each traffic class.

The Static priority scheduler supports a fixed number of priority levels, each having one FIFO

queue. Incoming packets are classified according to the desired QoS. When the output line becomes idle, the first nonempty queue with the highest priority is selected for transmission. The weakness of this simple algorithm is the lacking of fine differentiation between types of service needed. Another weakness is the unfair treatment of the lower priority classes, which may starve when higher priorities consume all the bandwith.
Earliest deadline first (EDF) scheduler maintains only one queue, and places the arrived packet according in the queue according to its deadline. This almost perfect scheduling mechanism requires searching for a position in the queue for every packet or at least sorting of per flow queues, making it inefficient in the implementation.
In section 4.4, we will present **Rotating Priority Queues** Scheduler (RPQ), that comprises the simplicity of static priority and the effectiveness of the EDF schedulers.

Class based Queuing (CBQ) scheduler has grown up from the need of sharing one link (i.e. leased line) between companies, between different protocol families and between different traffic types. This relatively complex scheduling algorithm enables defining what minimum portion (bandwidth percentage) of the link a company, a protocol and a traffic type (ftp, telnet, etc.) should get. Two mechanisms accompanying the CBQ scheduler are a classifier, determining the belonging class of each packet and an estimator, computing the used bandwith of each class over some period of time.

2.3 Integrated Services

Intergrated Services model [1] [17] defines an extension to the internet architecture *integrating* best-effort service, controlled data-link sharing and real-time services. The work on the IS extension has been motivated by an idea of using the Internet as a multifunctional, IP-based, packet-oriented network, supporting real time services for diverse multimedia applications.
Best effort service is the internet default class-type which this architecture intends to extend.
Data link sharing concept grew up from the need for sharing one dedicated line between different companies, users and traffic classes. The work surrounding this objective overlaps the work on Class Based Queueing (CBQ) and is further described in [18].

Several class types [19] have been proposed by the IETF IS Working Group, including Guaranteed Service, Controlled Load Service, Committed Rate and Protected Best-Effort Service, but only the former two have been formally specified and discussed in detail.

2.3.1 Guaranteed Service

Guaranteed service aims the real-time applications needing fixed end-to-end delay. The parameters define the maximum queueing delay and the bandwidth.
It is defined through following parameters:

- Token bucket model; bucket depth b in bytes, and token arrival rate r in byte/s.
- Peak rate p in byte/s. Peak rate in the maximum rate at which the application will send, measured in *some* time-period.
- Minimum policed unit m in bytes and maximum datagram size M in bytes. While testing for conformance, datagrams less than size m will be counted as of size m. Datagrams bigger than M should be rejected. It is important to note, that M should never exceed the MTU

[1] http://www.ietf.org/html.charters/intserv-charter.html

of the path, because guarantees cannot cope with fragmentation and refragmentation of the packets somewhere on the path.

- Bandwidth R in byte/s defines the virtual dedicated connection (reserved rate) from the sender to the receiver. ($R \geq r$)
- When higher bandwidth is reserved then the traffic source requested, delay is reduced. Slackterm S in microseconds adds an extra delay to the reserved QoS thus loosening the achieved delay. Of course, giving smaller delay and larger bandwidth to the application is preferable, but slackterm gives a possibility of sparing QoS. [27][24]

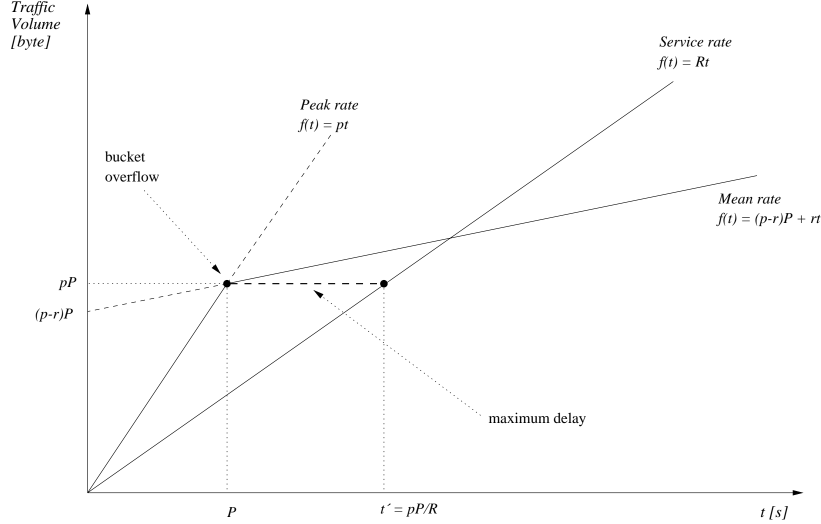

Figure 2.2: Maximum delay with a token bucket model and fixed service rate

None of the above declared parameters defines the amount of acceptable delay directly, however, it can be computed from the token bucket parameters and reserved bandwidth. Figure 2.2 shows the worst case, maximizing delay. The function Rt shows the rate at which the packets are being served – leaving the bucket. If we assume that the sender starts transmitting at peak rate, stops shortly before the bucket overflows, and carries on with the mean rate, we still have token-bucket conform traffic, yet maximizing the time between packet arrival and service. At this point, called *peak end time* P, shortly before the packets are to be discarded, the sender transmits at peak rate p with simultaneous arrival of tokens at rate r. This equation is equivalent to 2.1 defining the maximum burst length.

$$(p-r)P = b \Rightarrow P = \frac{b}{p-r} \tag{2.2}$$

Maximum delay is thereafter

$$d_{max} = t' - P = \frac{pP}{R} - P = \frac{p-R}{R}P$$

After substituting P from formula 2.2, we have:

$$d_{max} = \frac{b}{R}\frac{p-R}{p-r} \tag{2.3}$$

As we can see, in the worst case we have the reserved rate equal to token bucket rate ($R = r$) still obeying $R \geq r$ and maximum delay $d_{max} = \frac{b}{R}$. Increasing the reserved bandwidth R reduces the delay giving the flow better service than needed. This excessive reservation can be corrected with a slack term S (in microseconds), allowing additional delay.
In order to pay regard to differnet hardware architectures, different implementations and fixed delays, GS defines two error-terms that define the deviation from the fluid model. Both apply to per network-element (e.g. router, switch or end-host):

- Rate dependent error term C, measured in bytes, is an extra delay each packet experiences due to the rate parameters of the flow. An example mentioned in [27] is the serialization of IP packets into ATM cells. The rate dependent term depends on the ATM flow rate.
- Rate independent error term D, measured in microseconds, represents the worst case transit time through a service element, independent of the rate.

Having defined the error terms, we can summarize the per-path error term, with C_i and D_i being error terms of the specific network element:

$$C_{tot} = \sum_{i \in Path} C_i \qquad \text{and} \qquad D_{tot} = \sum_{i \in Path} D_i$$

Now we can express the maximum delay along the path, assuming the reserved bandwidth does not exceed the peak rate:

$$d_{max} = \frac{b}{R}\frac{p-R}{p-r} + \frac{C_{tot}}{R} + D_{tot} \qquad \text{for} \qquad r \leq R < p \tag{2.4}$$

We should note that the term b/R, time it takes to drain a full bucket, occurs only once in the formula, independent of the number of routers on the path. This fact is based on the limited size of the bucket b always being serviced by at most R. Thus, only the edge of the QoS network, i.e. first router, or already the sender´s local host, experiences the peak-rate burst and causes the delay. Following routers along the path receive traffic not exceeding their bounds.

Having reserved bandwith larger then the peak rate, no queueing delay occurs, just the delay caused by the nodes in the path, having:

$$d_{max} = \frac{C_{tot}}{R} + D_{tot} \qquad \text{for} \qquad r \leq p \leq R \tag{2.5}$$

2.3.2 Controlled load service

In contrast to Guaranteed service, Controlled Load service class does not define any firm quantitative parameters. Many adaptive real time applications work well on unloaded networks, but their performance degrades quickly under loaded conditions [26]. Therefore, Controlled Load service class provides the client with such QoS under loaded conditions as it would have been in an unloaded net. Although loosely defined, the admission control still needs the traffic specification from the sender. The parameters are: token bucket traffic specification (bucket depth b, token rate r), peak rate p, minimum and maximum policed unit m and M.

2.3.3 Architecture

Routers not capable of handling Integrated Services usually forward packets in the plain FIFO-manner. IS - aware routers should be able to handle single Guaranteed Service and Controlled Load flows.

The idea behind the flow is the definition of virtual paths through a packet network connecting two endpoints. Flows can be identified through their L3 and L4 service access points, i.e. the 4-tuple {*source-address, source port, destination-address, destination-port*}. Each router along the path can identify packets belonging to the flow and determine what QoS class do those packets belong to.
The element separating packets is called a **classifier**. A Classifier maps the incoming packet into one class, depending on the addresses and ports of the packet. This way, not all packets get the same treatment. Here we can differentiate between best-effort, Controlled Load and Guaranteed Service.
After passing the packet to the appropriate class, the **policer** decides whether the packet conforms to the flow-specification. If positive, the **scheduler** accepts the packet, otherwise it is discarded or treated as best-effort traffic. Scheduler manages different classes, queues and timers, and makes sure that each flow becomes the right portion of bandwidth and maximum allowed per-host delay. This is the place where the scheduling algorithms reordering the sequence of queued packets are implemented, e.g. WFQ or CBQ.
Admission Control implements the decision algorithm that a router uses to determine whether a new flow can be granted without effecting earlier guarantees [17]. The second role of admission control is user-accounting, managing the permissions of a specific user to reserve the ressources.
RSVP protocol is a signaling protocol for Integrated Services networks, used to modify the traffic control state in the node.

2.3.4 Data Link Layer Technology

The Scheduler as a central point in the Integrated Services architecture operates in the lower part of the network-layer (L3). Different algorithms can decide how the forwarded or local packets are to be reordered in the queue, in order to minimize the queueing delay of the packets demanding better service then best-effort. With other words, the scheduler passes the packets down to the link-layer and has no further control of when and how are those packets treated.
The Integrated Services over Specific Link Layers (issll) Working Group of IETF [2] studies possible mappings of guaranteed service and controlled load onto different Link Layer technology (L2).

- 802.x protocols over a shared medium like Token Ring and FDDI use tokens for preventing multiple access and collisions and priorities for differentiating class types. These technologies also define maximum access latency for the highest priority, which can be used for Guaranteed Service.
  CSMA/CD protocols with MAC protocol have an unbounded access latency making mapping of Integrated Services a theoretically unsolvable problem. Practically, there are two possible solutions for extending the ethernet based lines, in oder to limit the access latency. First possibility is extending the MAC protocol, which can hardly be realized, since it requires heavy changes in ethernet adapters or device drivers. The second possibility is a technical solution, using *the full duplex switched topology*, described below.

- Having a dedicated point-to-point switched line makes mapping relative straight forward, because there are no collisions and the available bandwidth is known in advance. Such system is called a QoS-passive link layer. Thus, the traffic scheduler is the *only* instance capable of deciding which packets and in which sequence packets are pushed onto the line.
  The so called *full duplex switched topology* for Ethernet based protocols gives exactly one sender the full bandwidth of the line, thus disabling collisions, practically turning off the MAC protocol and simulating a switched circuit.

[2] http://www.ietf.org/html.charters/issll-charter.html

- Network technology with its own traffic control such as ATM enable mapping of flows into VCs, yet requiering transformation of IS into ATM traffic control parameters.

2.4 RSVP

The Resource Reservation Setup Protocol (RSVP) [3] [22][20][19] is the signaling protocol in the Integrated Services architecture.
RSVP is orthogonal to IP, i.e. it extends *and* uses the IP protocol (much like ICMP or IGMP do). RSVP manages and modifies the state of traffic control in the routers along the data-path. It is *not* a data - protocol and not a routing-protocol, although it relies on the routing table. Real-time applications communicate mostly over UDP datagrams. Those datagrams are bundled into a virtual flow, belonging to a *session*.
Let us take a brief look at the RSVP-functionality by describing a short example. The receiver application finds a sender by using some application protocol and informs it that it is interested in receiving data from it. The sender uses RSVP and sends a Path messages carrying its traffic specification (Tspec), which is read by all the routers on the way to the receiver host. The receiver sends a Reservation message back to the sender, which is read by every router on the way. Each router along the way reserves the requested QoS in the scheduler and informs the classifier. Now, the sender can send its packets, which are given the requested QoS at every router.

RSVP has following the characteristics:

- RSVP is *receiver-oriented*, i.e. the receiver initiates the reservation
- RSVP uses *soft-state* for handling crashes and software faults. Each reservation and path message has to be refreshed – resent after a specific period of time. When one of the routers notices a missing message, the reservation is torn-down automatically. So the ressources are released when no new path or reservation message has kept the soft state alive.
- RSVP messages are sent as best-effort traffic, i.e. they can be discarded or lost. The message retransmission mechanism described above can compensate this unreliability, yet causing a time penalty due to a retransmission period.
- A receiver sends a reservation upstream to the sender, and each node in the path makes a reservation or rejects it. Yet, there is no simple and obvious way for a receiver to find out what resulting QoS has been assigned to the flow. The solution is the so called *One Path With Advertising* (OPWA), which sends a data-container as a part of PATH messages, gathering information along the way. This advertisement may then be used from receiver for adapting or sending a new reservation which better fits it needs.
- Reservations may be changed dynamically by simply sending a new reservation message, which automatically obsoletes old reservations.
- RSVP can cope with non IS-routers in the path, which do not process the RSVP messages.

2.4.1 Protocol Elements

An RSVP message consists of a common header, containing message type, flags, checksum, length and its own TTL field, followed by a variable number of objects. Figure 2.3 shows the declaration of RSVP objects, which are used to build RSVP messages, in Backus-Naur Form (BNF). Each

[3] http://www.ietf.org/html.charters/rsvp-charter.html

object is proceeded by an object header, containing the length and two values Class-Num and C-type which together uniquely identify the object type.

```
session = <IP destination address> <port> <IP protocol id>
rsvp_hop = next_hop = previous_hop = <IP address>
style = Wildcard-Filter | Fixed-Filter | Shared-Explicit

flowspec = gs_flowspec | cl_flowspec
cl_flowspec = <r> <b> <p> <m> <M>
gl_flowspec = <r> <b> <p> <m> <M> <R> <S>

filterspec = <IP destination address> <port>

sender-template <IP source address> <port>
sender tspec = <r> <b> <p> <m> <M>

adspec = <non is hop> <is hop count> <path bandwidth> <path latency> <mtu>
(gs_adspec | cl_adspec)

gs_adspec = <c_tot> <d_tot> <c_sum> <d_sum>
cl_adpec = <>
```

Figure 2.3: RSVP Objects

Session identifies the destination of a flow and occurs in every RSVP message. It consists of the destination address and port number of the receiving process. Rsvp-hops store next and previous routers along the path. Managing of predecessors and successors on the path is needed for making sure that all the RSVP and data packets follow exactly the same way through the network. Traffic specification (Tspec) characterizes the sender. CL and GS have already been defined, with CL Flowspec being the subset of the GS Flowspec. Sender template contains the address and the port number of the sender for identification of source packets. Sender traffic specification (Tspec) characterizes the sender´s traffic by means of token bucket parameters, peak-rate, minimum and maximum admission packet size.
Reservation style can be one of the following:

- Wildcard Filter Style creates a single reservation shared by flows from all upstream senders. As new flows arrive, the reserved QoS along the path is extended.
- Fixed Filter Style creates a distinct reservation from the sender to the receiver, not sharing it with other sender´s packets.
- Shared Explicit Style creates a single reservation shared by flows explicitly specified by the receiver.

Adspec Adspec (*Advertisement Specification*) is carried within a path message, and is being updated at every router on the path. This method of information gathering is called *One Pass With Advertisement* (OPWA) [21] and lets the receiver learn the MTU of the path , path latency, available bandwidth and the number of non-IS-capable routers.
The table 4.3 shows the information elements of the Adspec objects. Each router along the path *modifies* the elements according to its own computed values. The composition function is a binary operation taking the present value of the arrived Adspec and the local value as parameters, filling

the Adspec with the computed result. For example, each router adds its own minimum path latency to the field, making the approximation of the fixed delay for the end-host possible. In the same way, each router inserts the MTU in the Adspec, but changing it only if the local MTU is smaller than the previous minimum MTU of the path; thus enabling the end-host define the maximum packet size, in order to avoid fragmentation.
Guaranteed Service uses the same mechanism for computing D_{tot} and C_{tot} respectively. Each router adds (SUMM as a composition function) the local D or C to the appropriate fields in the Adspec-object [4].

Parameter	Description	Composition
NON_IS_HOP	set to 1 if the router doesn´t support QoS	OR ($\vee$)
NUMBER_OF_IS_HOPS	incremented if the router is integrated-services-capable	INCR
AVAILABLE_PATH_BANDW..	maximum available bandwidth for the flow [bytes/sec]	MIN
MINIMUM_PATH_LATENCY	smallest possible packet delay added by the network element [μs]	SUMM
PATH_MTU	maximum transmission unit following this path [byte]	MIN

Table 2.1: Adspec parameters

Messages Table 2.4 shows the composition of RSVP messages out of the previously defined objects. As noted before, each message carries a session object, identifying the flow. Time values specify the time-period, within which the next retransmission is to occur. The periodic transmission of Path and Reservation messages keeps the soft-state and traffic control reservations alive. Path and Resv Teardown messages close the session explicitly. Path error is sent back to the sender, when some router along the path notices an invalid Path message. Reservation error is sent back to the receiver when some router encounters that a reservation can not, or may not be undertaken. [24]

```
path = <session> <previous_hop> <time_values> <sender_template>
<sender_tspec> <adspec>

resv = <session> <next_hop> <time_values> <style> <flowspec> <filterspec>

path_teardown = <session> <prevoius_hop> <sender_template> <sender_tspec>
resv_teardown = <session> <next_hop> <style> <flowspec> <filterspec>

Other:
path_error, resv_error, resv_confirmation
```

Figure 2.4: RSVP Messages

Example A Path message travels from a sender to a receiver, carrying the traffic specification (Tspec). Each router reads the message and adjusts its local tables, and fills the Adspec part as described. If the Path message is invalid, a Path Error message is sent back to the sender. When

[4] C_{sum} and D_{sum} summarize C and D since the last *reshaping* point. Reshaping is an attempt of restoring the shape of traffic, in order to conform to Tspec (see [27] for details).

the Path message has reached the receiver, the Reservation message is sent back to the sender. Each host adjusts its traffic control elements as following; the scheduler is informed what QoS is required, the classifier is infomed which packets belong to a flow. If the router is not able to accept the connection, a Reservation Error is passed back to the receiver. If the receiver requested a confirmation, a Reservation Confirmation is sent back from the sender. Now the sender can transmit packages which are handled accordingly to their QoS reservation. Both the sender and the receiver have to retransmit their Path and Resv messages periodically, in oder to keep the soft states in routers alive. The session is closed either by sending Reservation Teardown message to the sender, or by sending Path Teardown to the receiver, or when some timeout in one of the routers notices a missing Path or Resv messages and closes the session.
Further details about the operation of RSVP can be found in [22][23].

2.4.2 RSVP-Daemon

The process managing the RSVP-protocol messages is called an RSVP-Daemon. The implementation used in this work originates from ISI [5] , and was enhanced for Linux support by A. Kuznetsov. The Linux port uses CBQ [18] as a default scheduler on all links, reserving appropriate shares of the link for GS, CL and best-effort traffic. All interfaces, including ethernet, are handled as QoS-passive link layers not taking the specific technology into account.

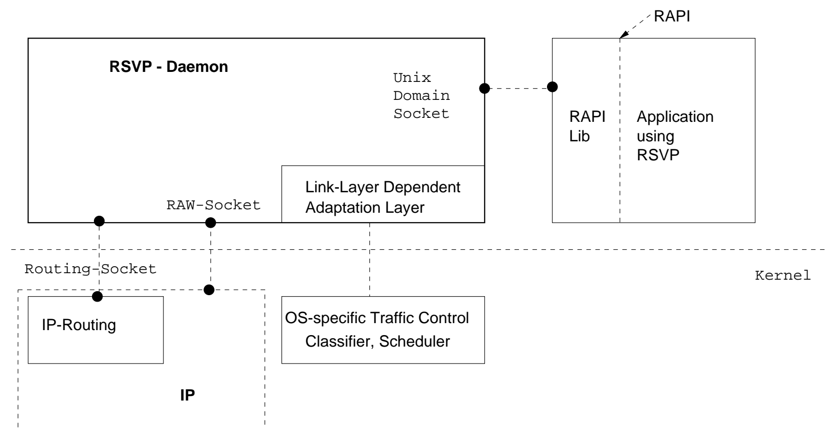

Figure 2.5: RSVP Daemon code structure

As we can see in figure 2.5, the RSVP Damon is closely tied to the operating system beneath. The source consists of two logical parts: the operating system independent [6] part and the link-layer dependent part (LLDAL), differing from system to system, dependent on the traffic control architecture in the kernel. The following protocols and interfaces are used:

- RSVP messages are received and transmitted through Raw-Sockets. Raw sockets are used for all the messages not being processed by the kernel. For example, messages of ICMP and IGMP type are understood (and processed) by the kernel, OSPF and RSVP messages not.

[5] University of Southern California, Information Sciences Institute (ISI), http://www.isi.edu/div7/rsvp/rsvp.html
[6] Almost independent, it works with most modern Unices.

- RSVP needs the information from the routing-tables in the kernel, in order to obtain information about next-hops; this interface uses *routing sockets* [40] for requesting information from the kernel and notification of any route changes. Routing sockets are also used for collecting informations about network interfaces.
- The RSVP Daemon expects the implementation of the following traffic control interface from the LLDAL:
  - `Init, Clear` are used for setting up and closing down the traffic control.
  - `AddFlowspec, DelFlowspec, ModFlowspec` are used for adding a new flow, deleting a flow after the session has been closed, due to a RSVP close message or soft-state path-teardown, and modifying a flow specification.
  - `AddFilter, DelFilter` add and remove filter providing the IP addresses and ports of the sender and receiver.
  - `Adpec` lets the LLDAL modify the OPWA-Information passed within every Path message. The OS-specific implementation should provide the necessary information like maximum path delay, MTU etc.

  On BSD-based systems, LLDAL communicates with the traffic-control in the kernel through diverse `ioctl` system calls. Linux uses the Netlink-Sockets for this purpose (Section 3.3.2).
- Applications that use RSVP-functionality are usually linked with the RAPI-Library[28], which in turn communicates with the daemon over Unix domain sockets. The following calls are defined by RAPI:
  - `rapi_session` initializes the session.
  - `rapi_sender` specifies sender´s parameters and sends a Path message.
  - `rapi_reserve` can make, modify or delete a reservation, depending on the parameters by sending a reservation message.
  - `rapi_release` removes a reservation and closes a session.
  - Upcalls are used for notifying the application of diverse events, such as arrival of path, reservation or error messages or closure of a session.

2.5 ATM

Asynchronous Transfer Mode (ATM) is a broadband, cell-based networking technology. An introduction to ATM is out of the scope of this thesis. Instead, refer to [3][12] for a short description or [1][4] covering the technology in general.
The relevance of ATM is the ability to request a point-to-point Virtual Circuit (VC) with specified QoS. Following classes are defined: Constamt Bit Rate (CBR) for constant rate traffic, typically for *circuit switching emulation*; Variable Bit Rate (VBR) for application with fixed timing relationships between packets, further divided into real-time and non-real-time VBR, Unspecified Bit Rate for best effort traffic and Available Bit Rate for bursty traffic with non-real time characteristics.

The basic of the guaranteed QoS is the *contract* between the user (host or application) and an ATM-Network, agreeing on a service type which is being provided. This agreement is negotiated over the UNI (User Network Interface) signaling, a slightly simplified version of Q.2931. Some relevant QoS and traffic contract parameters are:

- Peak Cell Rate (PCR) is the maximum rate at which the sender is going to send.
- Sustained Cell Rate (SCR) is an average rate at which is being sent for a longer period of time.
- Minimum Cell Rate (MCR) is the lowest rate the sender considers acceptable.
- Cell Delay Variation (CDV) is defined for PCR and SCR. It describes the variation between the perfect cell-interval 1/PCR and the maximum acceptable variation between two cells.
- Cell Loss Ratio (CLR) measures the percantage of cells lost or delivered out of time-bounds.
- Cell Transfer Delay (CTD) is the maximum transmission delay.
- Burst Tolerance (BT) is the maximum burst length in units of time.

2.5.1 CLIP

There are several proposals for interoperability between ATM and IP networks: simple point-to-point permanent VC with LLC encapsulation, Classical ARP and IP over ATM (CLIP), LAN Emulation (LANE), and Multiprotocol over ATM (MPOA) [1]. We take a brief look at one of the simplest mechanisms called Classical IP (over ATM) – CLIP [9].

The main abstract entity of the CLIP approach is the Logical IP Subnet (LIS). All hosts within the ATM-LIS can communicate wit each other over point-to-point VCs. The nodes in the LIS are routers connecting IP subnetworks over an ATM cloud. One of the nodes acts as an ATM-ARP server, managing the transformation of IP addresses to ATM addresses. Figure 2.6 shows an example for CLIP.

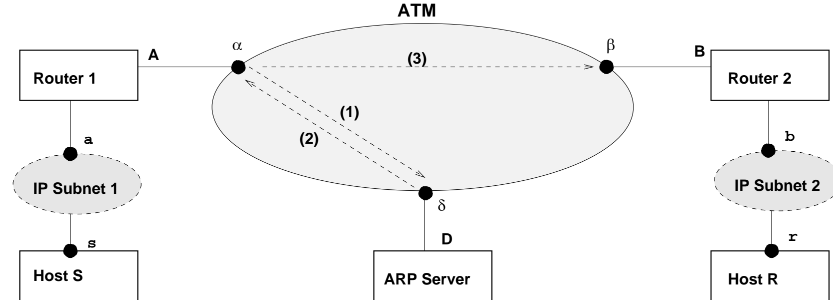

Figure 2.6: Classical IP over ATM (CLIP)

Router_1 has three addresses: the ATM-address α in the E.164 [7] format, the ATM-LIS IP-address A and the $\text{Subnet}_1$1 address a; router_2 having β, B and b. The ATM ARP Server manages the mapping of LIS IP addresses to native ATM-addresses (e.g. $A \rightarrow \alpha$). Every host joining the logical IP subnet registers itself with the ATM-ARP Server.
Now we assume that the Host S sends a packet to host R. The first hop is the Router_1 in the Subnet_1 having the IP-address a. The routing tables of the first router map the final destination to the next-hop Router_2 having the IP-address B. If there is no virtual connection to the ARP-Server, signaling creates a virtual connection between α and γ. Then, an ATMARP request is

[7] ISDN Telephony Numbering Plan

sent, asking for he ATM address ot B (1). ATMARP reply carries the ATM-address β (2). Now, Router_1 establishes a virtual connection to Router 2, forwarding the packet (3). Router_2 forwards the packet to the Host S over the IP-Subnet_2.

In the example above, only IP traffic is passed through the VC between the routers. However, multiple protocols are often multiplexed in one VC, raising the need for some multiplexing information side by side with the data. LLC encapsulation (RFC1483) defines standard header for diverse 802.x protocols being multiplexed over one VC. Therefore, this kind of encapsulation is mostly used in combination with the Classical IP over ATM mechanism.

2.5.2 ATM - API

Accessing the ATM network from host-based software is the primal purpose of the Native ATM Services specification defined in [10]. These services should enable data transfer over various adaptation layers, provisions for setting up SVCs and PVCs and traffic management considerations, and finally, support the development of native ATM applications. However, this specification has not been implemented on many systems [8]. Some companies have developed their own APIs for ATM Devices, and Remote Procedure Calls, diverse Message Passing architectures and Socket-Models have been investigate and partly implemented as parts of research projects [11].
Linux supports the BSD Socket-style ATM API [6] supporting AAL0 and AAL5 layers, provisioning for both PVC and SVC, and support for CBR and UBR services.
The socket setup and termination calls are used for UNI signaling, the send and receive family of commands for data transfer over a dedicated VC. The ATM socket address structure accepts E.164 and private ATM addresses and the definition of various Network Service Access Points.

2.6 RSVP over ATM

Integrated Services introduce two new traffic classes into the Internet architecture, namely Guaranteed Service and Controlled Load, extending the default best-effort service. Both services require traffic control guarantees along the path, which are being set up by the RSVP as a connection-oriented signaling protocol. It is obvious, that Integrated Services Internet has come closer to ATM philosophy and that both present similar packet oriented technologies, whose integration and convergence is being striven to.

2.6.1 DIANA Architecture

DIANA Project [9] has developed a framework for interconnecting IP/RSVP networks over ATM. Some theoretical background on the ATM/IS coupling can be found in [3][36][35]. DIANA specific publications are found in [30][31], which also include further references. Table 2.2 shows parallels between the two architectures.

We will limit our RSVP over ATM discussion to CLIP, as a simple and well understood mechanism of connection IP subnets over ATM clouds. Using CLIP means establishing a point-to-point UBR virtual circuit between two hosts, which carries GS, CL and best-effort traffic. This simple approach limits the control we can take upon traffic classes up to the scheduler operating at L3 and treats ATM as QoS-unaware technology. Better approach is mapping one IS flow into one VC,

[8]Digital Unix and Solaris provide native ATM API, Linux, FreeBSD and WinNT a socket style API, as known to the author.
[9]http://www.telscom.ch/diana

Integrated Services Internet	**ATM**
RSVP	ATM-Signaling
Flow-Spec	Traffic Contract
Flow-Id	VPI/VCI
Packet Scheduling	Traffic-Management
Services Guaranteed-Service Controlled-Load Best-Effort	QoS Classes CBR, rtVBR CBR, nrtVBR, ABR UBR, ABR

Table 2.2: Integrated Services to ATM mapping

with QoS considerations mentioned in [36]. Table 2.2 shows recommended mappings of services into classes. Best effort traffic is thus delivered over the default CLIP VC, and each GS or CL flow needs extra signaling for connection setup. With an increasing number of flows, this simple solution represents a waste of resources, increasing the signaling overhead to an unacceptable extent.

In this work, we show how **merging** (or aggregation) of multiple flows onto one VC can effectively be used for IS traffic over ATM, minimizing the signaling overhead. The main idea behind the concept that each two hosts in the Logical IP Subnet (ATM) are connected by at least 3 VC:

1. One CBR VC used for guaranteed service, through which all the GS flows from one to another host are multiplexed.
2. One UBR VC for controlled load, through which all the CL flows are multiplexed.
3. One default CLIP (UBR) VC for best effort traffic, as before.

The GS and CL use SVC, i.e. the VCs are being established dynamically with the first reservation, and torn down with the last close message.

Using one VC for multiple GS flows with different delay-bounds raises a need for fair scheduling. For example, packets belonging to a flow with a maximum allowed delay of 10 ms should be serviced before the packets that tolerate a delay of 100 ms. The optimal scheduler for this purpose is the Earliest Deadline First (EDF) scheduler always servicing the packet likely to arrive out of time.
Another point to consider is the so called *Dynamic QoS*. Let us assume that one GS VC transmits packets of ten different flows, and the reservation for the eleventh flow arrives. Admission control decides that there are no ressources left for this flow and it is being rejected (or, we establish a new VC as a pool of flows, complicating the design). Second example would be a receiver-application wishing to change (increase) its reservation, but since this would acquire the teardown and establishing of a new VC, the reservation is rejected. Another solution would be the renegotiation of the parameters specified in the traffic contract, broadening the volume of the VC-pipe. The renegotiation protocol has been defined in Q.2963.x and partially implemented from the Linux ATM sockets API.

ATM signaling uses the Signaling AAL Layer for VC setup and termination. SAAL is a reliable and acknowledged protocol. As noted before, RSVP takes the role of the signaling protocol in the IS architecture. However, RSVP is unreliable, since it uses the best effort service class whose packets may be dropped. One possible solution of avoiding CLIP is setting up a distinct PVC between each two hosts in the LIS. This is obviously a very inefficient solution, since for only $n = 10$ hosts,

$n(n-1) = 90$ permanent connections would exist. Using SVC as RSVP - path between hosts is inevitable, adding one more dynamically established connection between each two hosts in the LIS, i.e. 4 at maximum. VCs are set up when RSVP messages arrive and torn down after no RSVP messages have been transmitted for some amount of time.

2.6.2 DIANA Atmtcd

The ATM Traffic Control Daemon (*Atmtcd*) is the key component of the DIANA architecture. It helps the RSVP Daemon to manage admission control on an ATM interface, to establish and tear down SVCs, to transform IS parameters into ATM traffic control options, and to manipulate the kernel traffic control, i.e. the classifier and the scheduler.
Figure 2.7 shows the structure and the connections to other modules and programs. The ATM adaptation module is a patch to the RSVP Daemon, taking control of the ressource reservation when the ATM device is being used as a next hop, and passing relevant LLDAL information to Atmtcd over Unix domain sockets. Arpd is a program belonging to a Linux ATM extension, which uses ATM-ARP protocol for transformation of IP addresses into ATM addresses. ATM sockets API is used for signaling. Kernel sockets use the signaling daemon for UNI signaling.

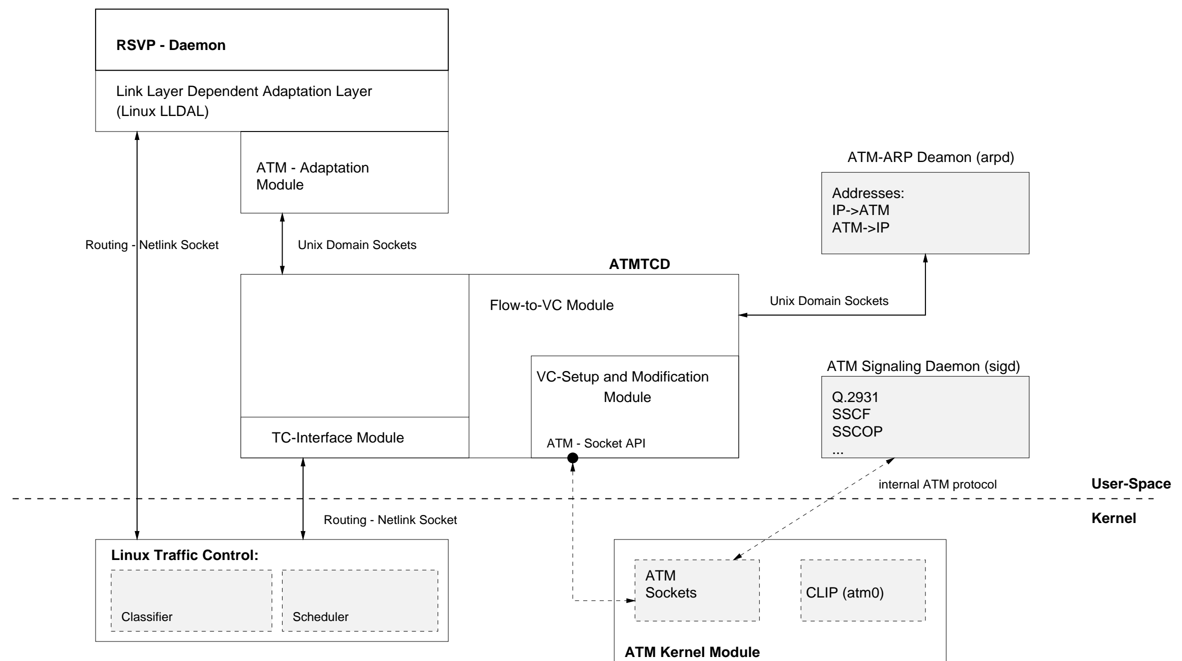

Figure 2.7: ATM Traffic Control Daemon (atmtcd)

- *The Flow-to-VC Mapping* Module is responsible of handling the LLDAL messages from RSVP-Daemon, mapping IS flow parameters into ATM classes and managing runtime flow information. The admission control mentioned as a part of the Integrated Services architecture is also settled here. More details can be found in [29]
- *The VC-Setup and Modification* is responsible for signaling over ATM-Sockets and VC traffic contract renegotiation. See [31] for detailed information.

- *The TC Interface Module* is called from Flow-to-VC Module when traffic control elements have to be changed. It wraps the relatively complex Rtnetlink communication and exports a simple interface to the rest of the Atmtcd.

When the Reservation message reaches the RSVP Daemon and the reservation should be set up for the next router over an ATM link, the ATM Adaptation Module comes into play, and passes the `Add_Flowspec` message to the Atmtcd. The Flow-to-VC Module reads its local information and decides which case applies: new reservation, modification of an existing reservation or repeated reservation message needed for keeping the soft-state alive. If a new reservation is to take place, the following actions are carried out: computing ressources and admission control, contacting the ATM-ARP Daemon and requesting the translation of the IP address into ATM address, instructing the VC-Setup and Modification Module to establish a VC or to renegotiate the old traffic contract, and requesting changes in the Classifier and Scheduler through the TC-Interface Module.

Chapter 3 describes traffic control, Rtnetlink and ATM part of the kernel. In chapter 4, we will take a detailed look at the details hidden behind the Linux traffic control component. We will finally return to Figure 2.7 and the proposed architecture and and describe the functionality of the TC-Interface module within the Atmtcd and the data flow.

Chapter 3

Kernel Modules

This chapter presents the parts of the Linux kernel relevant to the implementation of the DIANA traffic control extension. Since most parts of the Linux kernel are fairly undocumented, the information presented here describes the result of code-reading exercise combined with information obtained from Linux kernel books.

First, we take a look at the structure of the networking kernel part and follow the path of the data coming from and leaving by some hardware adapter. We also describe the operations used for modifying socket buffers - the essential part of the network memory storage management in Linux.
Secondly, we describe the structure of the traffic control in the Linux kernel, implemented by A. Kuznetsov. Traffic control in the kernel follows the OO design implemented in C.
Thirdly, we describe Rtnelink-Sockets, a modified and more powerful Linux version of BSD routing sockets. This part has been coded by A. Cox and A. Kuznetsov.
Finally, we describe the ATM extension to the kernel and ATM sockets, and present how ATM technology fits into the kernel mechanisms. The ATM part is the work of W. Almesberger.

3.1 Linux TCP/IP Implementation

In this section, we will take a rough look at the structure of Linux TCP/IP code [39][45][44], which, although similar, doesn´t inherit any BSD parts. The network code makes excessive use of the object-oriented design, specifically the definition of interfaces. The interfaces are structures whose fields are pointers to functions. These interfaces define a well-known way of using kernel-modules without internal knowledge of the real code [1]. As an example, the structure `device` is an abstract interface to some *real* network hardware device. Depending on the specific hardware, interface-functions are implemented and placed in a variable of type `struct device`. This way, IP-code is hardware independent and the device driver is an entry point to L2.

The core objects of the networking part of the Linux kernel are following [44]:

- Device or interface represents an abstract entity which receives and transmits packets. The structure `device` defines a set of *methods* that any device should support. These methods can further be divided into mandatory methods, which in any case must be implemented,

[1] See Appendix A for discussion about OO and C.

and other methods whose existence is desirable, but not strictly necessary, i.e. the function-pointer is being compared against `NULL` before usage. Some example devices are *eth0* for the first ethernet card, *ppp1* – second PPP channel, *atm0* for CLIP and *lo* – pure software loopback device.

- Protocol provides a uniformed interface to the socket layer. The structure `proto` is thus implemented for the TCP, UDP, Netlink, SVC- and PVC style ATM sockets, to name a few. Some `proto` structures are defined although they are not directly tied to the socket layer, yet define a standard interface and enable forwading of socket operations.
- Sockets are an abstraction on the user space side, enabling Unix file-IO and access through a file descriptor. Each socket exists as a set of structures in the kernel, capable of managing the states of the protocol. Structure `socket` takes care of all the protocols, while the structure `sock` is a specialty of the AF_INET protocol flavor. Invocations of socket system class pass the user space - kernel border at the *socket layer*.
- Socket buffers (skbuff) are the buffers used by the networking layers. It is a more flexible replacement for the BSD-style `mbufs` [39].

3.1.1 Socket Layer

The socket layer is the main entry point for the networking-applications. TCP, UDP, ATM and Routing-Netlink sockets all use the same interface provided by the kernel. Linux socket layer differs slightly from the BSD Net 3 implementation described in [39].

The structure socket contains a pointer to a structure containing pointers to all the socket based operations. Those are [40]:

- setup and termination system calls : `bind, listen, accept, connect` and `shutdown, close`
- input/output system calls: `select, read, readv, recv, recvfrm, recmsg, write, writev, send, sendto, sendmsg`
- administration commands: `fcntl, ioctl, setsockopt, getsockopt, getsockname, getpeername`

The structure containing function prototypes of the above commands is named `proto_ops` and must be implemented for each protocol. The structure `net_proto_family` contains only the family name (e.g. AF_INET) and the pointer to a function capable of determining the protocol (e.g. UDP).
Socket creation through a C library `socket` call is forwarded to a socket system call. The socket system class initializes the `struct socket` structure and looks for the family specified in the parameters. After finding the appropriate `net_proto_family`, the create function is then called which checks for the availability of the specified protocol, and if found, fills the `ops` field of the `socket` structure with the implementation vector of the protocol. From this point, all the socket-based system calls are forwarded to the appropriate functions.

3.1.2 Socket Buffers

The sk_buff structure consists logically of three parts: the data fields in the structure managing data as it passes through the network layers and various pointers to the data-segment, the methods defined for manipulating the skbuffs, and the skbuff list with its own methods used for enqueueing and dequeuing packets.

Skbuff data

The structure `sk_buff`[2] is rather large. We shall mention only some relevant fields and their usage. Skbuffs are kept in doubly-linked lists, therefore there are two pointers `next, prev` and a back-pointer to the list `list`. The field `stamp` contains the timestamp (in `struct timeval`), i.e. arrival time filled in by the `netif_rx` function. The field `dev` contains a pointer to a device structure that the packet arrived on, when moving up the protocol stack, or a device a packet is leaving by, when moving down the protocol stack. The transport layer header is kept in a large union (`h`) containing one of the following header-types: TCP, UDP, ICMP, IGMP or IPIP. Protocols orthogonal to IP are therefore handled as the transport protocols. The network layer header is union (`nh`) containing one of the following header types: IP, IPv6, ARP and IPX. The data link layer header is a union (`mac`) containing either an Ethernet or raw (any other) header. Destination entry field `dst` points to a next-hop address, depending on the layer that manages it, i.e. IP-addresses above the data link layer and ethernet addresses in the data-link layer. Furthermore, there a fields containing the length (`len`), the checksum (`csum`) and queueing-priority (`priority`). The `protocol` indicates the data-link view of the packet protocol being for example ETH_P_IP for IP over ethernet.
Figure 3.1 shows various pointers into the data segment. The shaded area is a Protocol Data Unit at the data-link layer, after all the headers of the layers above have been added. White area is the buffer space allocated but not used. Unused memory in the socket buffer is a tradeoff for exporting a relatively simple interface, compared to high complexity but efficient memory usage of the BSD-mbuf [44].

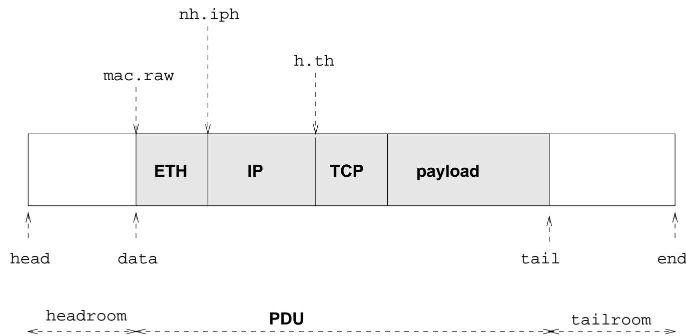

Figure 3.1: Socket buffer

Skbuff methods

If the skbuff is moving up the protocol stack, single pointers are being set to the memory-segment containing the appropriate header, and the PDU-beginning is moved to the start of the next header. The device-driver at the link layer sets the pointer `mac.raw` to the beginning of the PDU (`data`) and removes the ethernet header through `skb_pull`. The IP layer sets the `nh.iph` pointer and removes the IP header. After removing the TCP (or UDP) header and setting the `h.th` pointer, the payload can be accessed.
If the skbuff is moving down the protocol stack, `skb_reserve` is used to leave enough headroom for all the headers to follow and the payload is inserted. TCP adds its header with `skb_push` and marks its beginning with `h.th`. The same push-procedure is repeated at the IP- and the data-link layer.

[2]as in kernel 2.2.9.

As we can see, accessing the port numbers requires moving to the transport-layer header and reading the appropriate fields. Reading the source- and destination-IP address requires information from the network-layer header. This information is not explicitly stored in the skbuff structure.

Method	Description
`alloc_skb`	allocates kernel memory to store a skbuff
`skb_clone`	makes a copy of the an skbuff, not copying the data area
`skb_copy`	makes a deep copy of the skbuff
`kfree_skb`	frees kernel memory containing an skbuff
`skb_put`	adds a data-segment to the end of the PDU (tailroom)
`skb_push`	add a data-segment onto the front of the PDU (headroom)
`skb_pull`	moves the PDU-segment further right, shifting the data-pointer
`skb_reserve`	pre-allocates headroom
`skb_headroom`	return the number of bytes available in the headroom
`skb_tailroom`	return the number of bytes available in the tailroom

Table 3.1: Skbuff methods (`struct sk_buff`)

List-Methods

The data structure defining the skbuff-list is `struct sk_buff_head`. It is a doubly-linked list of skbuffs. The skbuffs can be placed onto the head of the queue with `skb_queue_head` or to the end of the queue with `skb_queue_tail`. Calling `skb_dequeue` removes one skbuff from the head of the queue. Some protocols need more specific placing or reordering capabilities then the plain LIFO and FIFO style. Using `skb_insert` and `skb_append` enables positioning of skbuffs inside the list relative to some known skbuff already in the list. Table 3.2 shows the interface to the list.

Method	Description
`skb_queue_head_init`	initializes the sk_buff_head structure
`skb_queue_head`	adds a given skbuff to the head of the queue
`skb_queue_tail`	adds a given skbuff to the end of the queue
`skb_dequeue`	returns (and removes) a skbuff from the head of the queue
`skb_peek_copy`	returns a skbuff from the head from the queue but doesn´t remove it
`skb_queue_len`	return the number of skbuffs in a given queue
`skb_unlink`	removes the specified skbuff from the list
`skb_insert`	places an skbuff before the specified skbuff
`skb_append`	places an skbuff after the specified skbuff

Table 3.2: Skbuff-list methods (`struct sk_buff_head`)

3.1.3 Transmitting packets

Let us assume that the data originates from an UDP Socket, created by a `socket` system call. After calling `send`, the data is passed through the user-kernel border ending in the `udp_sendmsg`, which allocates an skbuff, fills in the data and adds an UDP - Header (`skb_push`). The skbuff is now passed to the IP-Layer, where the route is computed, and IP-header added. `ip_queue_xmit`

calls finally `dev_queue_xmit` placing the skbuff in an outgoing FIFO queue that is assigned to the device in charge of transmitting packets. The function `dev_hard_xmit` implements the actual transmission of the packet as a part of the device-driver (implementing the device-interface), here, the data-link layer is added and the skbuff is copied into the hardware buffers for transmission. After successful transmission, the hardware adapter raises an interrupt notifying the operating system that another packet can now be sent. Forwarded packets do not go beyond the IP-layer, but are being injected into the data-link layer through the same `dev_queue_xmit` mechanism.

3.1.4 Receiving packets

When a hardware adapter receives a packet from the network (at the physical layer), it copies the arrived packet into the hardware buffer and raises an interrupt. The interrupt handle implemented in the device driver allocates a socket-buffer (`alloc_skb`), fills it with the packet data – reading the range of memory which is used by the adapter and sets the `protocol` (e.g. ETH_P_IP for IP over ethernet) and `dev` fields of the socket-buffer. The `mac.raw` pointer is set to the beginning of the data-segment.
Finally, the `netif_rx` function is invoked which places the packet into the receive-queue and the network bottom half is marked for execution.
Shortly after the the fast interrupt handler has finished its work, the network bottom-half [3] takes one packet from the device´s receive queue and reads the protocol field of the socket-buffer.
For IP, the function `ip_rcv` passes the frame to the IP layer.
IP can now decide wheather the frame is to be forwarded (`ip_forward`), given to some orthogonal protocol like ICMP or passed up to TCP (`tcp_rcv`) or UDP (`udp_rcv`). Passing the frame to the socket layer wakes up the process blocked in a `read` system-call (if any).

3.2 Traffic Control in Linux-Kernel

Traffic control in the Linux kernel covers scheduling mechanisms, but not the link layer capabilities of an underlying network technology.
As the Figure 3.2 shows, the upper part of the data link layer is in the kernel - actually in the device driver and the lower part on the network interface card (e.g Ethernet). Policing and shaping of the Traffic takes place between the IP-output and the input queue of the card.
By means of scheduling, traffic control can prevent the overflow of the buffers in the network interface adapter, and thereby avoid uncontrolled dropping of well-behaved packets belonging to the flow. So by moving the congestion from hardware to software, we can tell which packets are less important, or not conformant and drop those on purpose.
So here are some necessities for controlling traffic:

- being able to differentiate packets, deciding which flow do they belong to
- make buffers available for storing packets until the time they are to be transmitted
- policing traffic and deciding upon the conformance of the packet

The traffic control module in the kernel has relatively clear and thin interfaces to other modules (Fig.3.2). As a part of the protocol stack, traffic control is placed between the IP-layer and the device driver and exports a **network-interface**. Between the control flow `dev_queue_xmit` → `hard_dev_xmit`, we now have the root queueing discipline of an interface, which is accessed by

[3] See 4.7.1

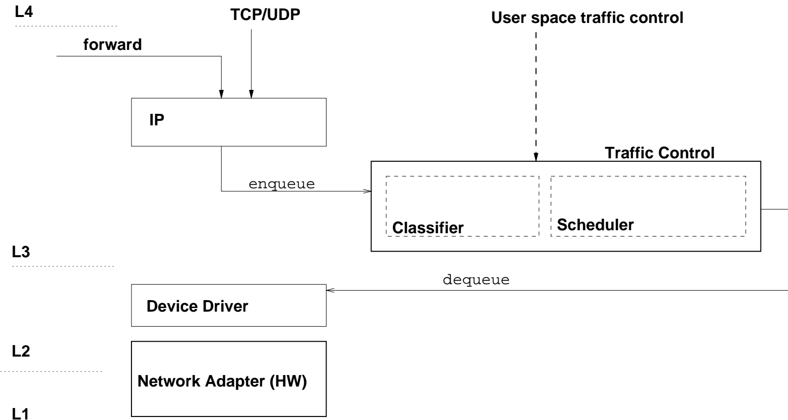

Figure 3.2: Traffic control in the protocol stack

enqueuing one packet, and shortly afterwards *dequeuing* of one packet. These two calls enable the reordering of packets respective to their arrival time. The new control flow is shown in figure 3.3. The second interface enables applications from user space the manipulation of traffic control elements, the so called **user-interface**.
Depending on the role of the specific host, the source of the IP traffic is either the IP Layer or the TCP and UDP Layers. At a router, most of the packets are being forwarded, so the main source is the incoming side of the IP Layer. At an end-host, the traffic is generated from the applications, which use sockets to hand out packets to TCP and UDP layers.

3.2.1 Queueing disciplines

The main conceptual component of the traffic control in the Linux kernel is the *queueing discipline*. The queueing discipline defines the order in which the packets are taken from a queue, with the intention of being served by the transmission mechanism [14]. We can imagine the notion of a queueing discipline as black box, accepting packets at one end, and putting them out at the other end. The simplest and broadly used kind of a qdisc is a FIFO qdisc. The packets are dequeued in exactly the same order as they arrived, i.e. were enqueued. Other possible qdiscs could be LIFO or shortest packet first, both reordering the sequence of arrived packets. Yet another variation of FIFO qdisc is a byte- or packet-limited qdisc; upon arrival of a packet (enqueue), we can determine if there are buffers left, and drop the packet if it threatens to overflow.
The **network interface** of an qdisc consists of the following methods:

- `enqueue` is called from the `dev_queue_xmit` and hands out an skbuff to a qdisc. The qdisc can eather store the packet in the local queue, or drop it.
- `dequeue` is called shortly after `enqueue` or from the net bottom half, when there are outstanding packets in the queue and the device is idle, i.e, being able to transmit. `dequeue` returns a packet for transmission or NULL if there are no packets in the queue, or the qdisc decides that no packets should be sent in order to obey the traffic contract.
- When the device-driver (`dev_hard_xmit`) is not able to transmit the skbuff, `requeue` is invoked to place the skbuff back into the qdisc. Theoretically, the skbuff should take the same place in the queue from where it has been dequeued, practically, the skbuff is placed in front of the queue being the first candidate for transmission.

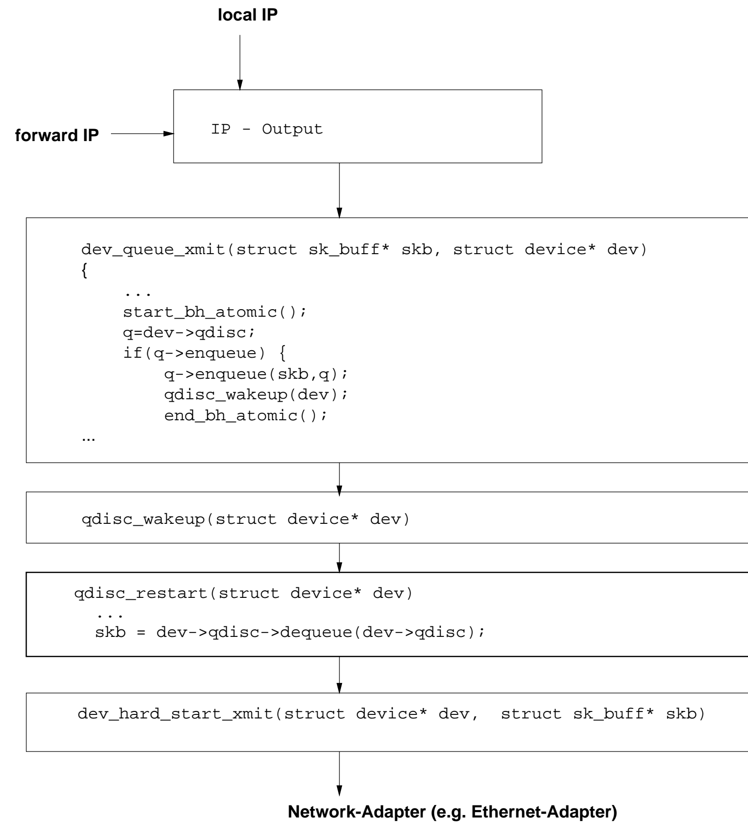

Figure 3.3: IP output and traffic control

- `drop` is not called directly from the protocol stack. The qdisc should discard one packet (using `kfree_skb`) from the queue.

Implementing a simple FIFO qdisc is straight forward. We need a skbuff list (`sk_buff_head`) where the pointers to skbuffs are stored. Enqueuing the skbuff calls `skb_queue_tail` and dequeuing `skb_dequeue`. We can limit the size of the queue by introducing the packet limit. Enqueuing increments and dequeuing decrements the number of queued packets. If the enqueue encounters that the buffers are full, the skbuff is discarded. The limit can also be defined more precisely by counting bytes and the `len` field of an skbuff. These two qdiscs are known as Packet-limited FIFO (*pfifo*) and Byte-limited FIFO (*bfifo*).

Implementing the token bucket qdisc (TBF) is slightly more complicated. The token bucket depth (b) is a byte-limited FIFO queue. Enqueue accepts packets as long as the buffers do not overflow, in which case the packet is dropped. Calling dequeue reads the current time and computes the time elapsed since the last dequeue has been called. The time-difference is multiplied by the rate r, and added to the variable storing available tokens, yet never being larger then bucket depth b. If the length of the skbuff exceeds the number of available tokens, dequeue returns NULL, otherwise, it returns the skbuff from the head of the skbuff-list. There is a danger that after the last dequeue, no further dequeues are called for a relatively long period of time [4]. For this purpose, timers are

[4] Actually, we always get a second chance. Successfull transmission of one packet is reported from the network

used, which are scheduled for execution if the queue contains any skbuffs left over. The timer calls `qdisc_restart`, which then tries dequeuing a packet on its own.

We now describe the second interface of traffic control, the user-interface. Except for fullfilling its duties towards the IP layer, user applications should be capable of inserting, deleting and modifying qdiscs. The **user-interface** has following methods:

- `init` is a qdisc constructor, qdisc specific options are passed to the qdisc which sets up its variables and allocates memory if needed. For example, a FIFO specific option is the maximum length of the packet queue, given in packets for pfifo or bytes for bfifo.
- `destroy` is a qdisc destructor. Allocated memory and any outstanding packets should be freed.
- `change` is used for changing the qdisc specific options at runtime, for example changing the maximum length of the bfifo queue.

3.2.2 Classful queueing discipline

A simple black box qdisc can be helpful when smoothening traffic, but some more demanding scheduling algorithms need differentiating traffic and thus identifying flows, setting priorities, and servicing them respective to their QoS needs.
The abstraction capable of performing these actions is a *classful queueing discipline*. The structure of the classful qdisc has been inspired from and designed for *Class Based Queueing* (CBQ, [18]).

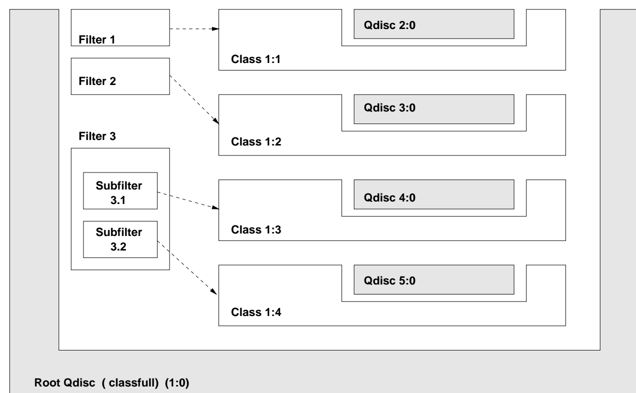

Figure 3.4: Runtime structure of a classful queueing discipline

Figure 3.4 shows the hierarchical runtime structure of traffic control. The outer U-formed object is the main queueing discipline of an interface. As a classfu queueing discipline, it consists of classes and filters differentiating packets for the classes. Classes in turn use their own queueing disciplines for queueing packets.
There are two different enumerations for the traffic control elements; the user-id, human readable form and the kernel-id, used for fast access of kernel structures. The user-id numbering scheme *major:minor* is used the following way:

adapter by raising an interrupt. Shortly afterwards, NET BH is executed which calls dequeue one more time.

- Queuing disciplines have an user id $x:0$ with x being between 0 and $(2^{16}-1)$
- Traffic classes belonging to queuing disciplines are identified through *x:y*, x being the major of a parent queuing discipline and y between 1 and $(2^{16}-1)$.

Filters are identified through their unique *priority* within the parent qdisc, being between 0 and $2^{32}-1$. All the user-ids fit into one 32-bit word.
Kernel-id is of type `unsigned long` and represents the memory address of the traffic control element. For example, casting a kernel-id `k_id` to a qdisc would use following code:

```
struct Qdsic* q = (struct Qdisc*)k_id;
```

Following relations between traffic control elements are assumed (see Fig. 3.4):

- The classful qdisc 1:0 is a *root* (or main) queueing discipline of an interface.
- The qdisc 1:0 is a *parent qdisc* of classes 1:1, 1:2, 1:3 and 1:4
- The qdisc 1:0 is a *parent qdisc* of filters 1,2 and 3
- Filters *point to* classes, more precisely, filters forward packets to classes.
- The class 1:1 is a *parent class* of the qdisc 2:0. Inserting the qdisc 2:0 into the class 1:1 is called *grafting*.

Figure 3.5 shows the Object-oriented Structure of the interfaces in the kernel. Classful qdisc inherits all the fields and methods of a classless qdisc, enabling the network layers to use them interchangeably. On the other hand, classful qdisc extends the user-interface in order to enable manipulation of classes and filters.

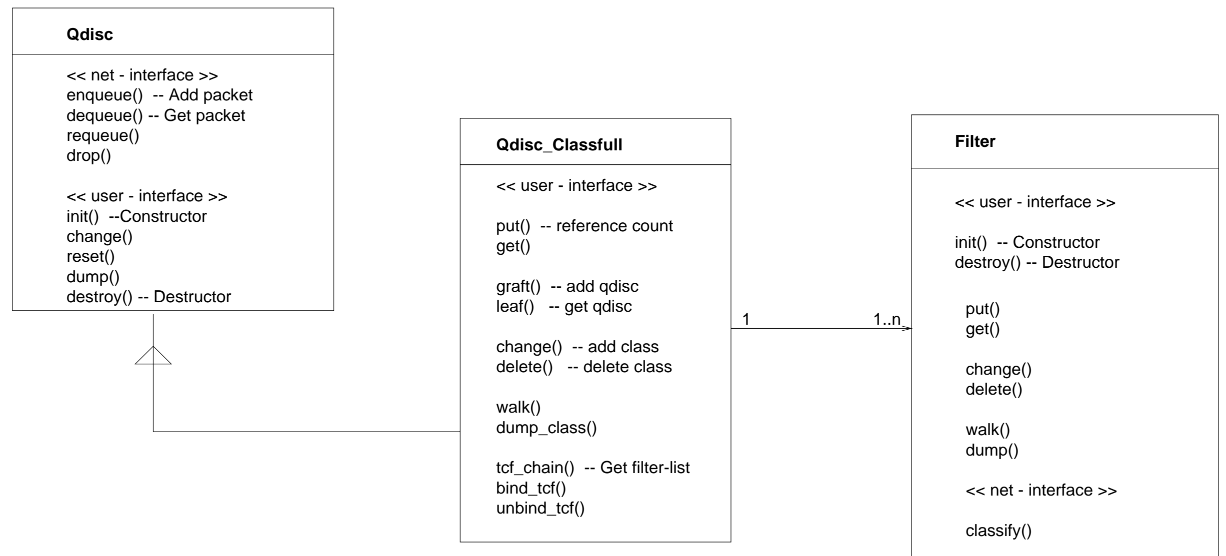

Figure 3.5: UML diagram of traffic control elements

Now we take a look at a data flow within a classful qdisc. After calling `enqueue`, the 1:0 qdisc passes the packet through a linked list of filters calling `classify` on each of them. Filters read the

socket-buffer data and decide if the packet matches, i.e. carries the destination address the filter expects to find. After matching, the filter returns either user-classid or kernel-classid. Having the kernel classid (`unsigned long`), we can cast it to the pointer to the classless queueing discipline. Having only the user-classid means traversing the class-list and searching for a reference.

Calling dequeue invokes the scheduler, which can now decide which class should be given higher priority, thus possibly reordering the queued packets.
It is to note that inner qdisc can themselves be classful, i.e. containing filters, classes and qdiscs thus introducing another hierarchical level to the structure.

The classful **user-interface** exports the following methods:

- `change` adds a new class to the qdisc, the implementation can decide what default qdisc should be *grafted* into the class.
- `delete` removes the class from the qdisc. It is up to the implementation to decide whether the reference count should be taken into account and whether there are filters pointing to this class.
- `get` maps the class user-id into class kernel-id and increments the reference count.
- `put` decrements the reference count.
- `graft` is used to insert a new qdisc into the class, the old qdisc is being removed and remaining packets dropped.
- `leaf` returns the inner (grafted) qdisc of a class.
- `tcf_chain` returns the anchor of the filter list, thus enabling adding new filters to the qdisc.
- `bind_tcf` and `unbind_tcf` are used for maintaining the reference count of a class by storing the number of filters pointing to it. This methods are used in the same context as `get` and `put` and are mostly implemented in the same way.

Mapped onto the Integrated Services architecture from chapter 2.3, the filters take the role of the *classifier*, and the outer qdisc together with its classes the *scheduler*. *Admission control* is implemented in some user-space application or partly in a specific qdisc.

3.2.3 Traffic Filters

Traffic filters are stored in a linked list whose anchor is obtained by `tcf_chain` method of a classful qdisc. Filters are uniquely identified by their priority and their parent qdisc. Having a simple linked list introduces an efficency problem with a large number of filters. This shortcome can be avoided through nesting of subfilters inside a *main* filter. Furtermore, an efficient structure like hash-table can be used for accessing subfilters. Those (inner) subfilters are identified through any 32 bit word (`u32`) at the user-side and by the kernel-id which is being casted into a pointer to a filter. Again, in order to access a subfilter, a three tuple is required: parent qdisc, priority of a main filter and a handle of a subfilter.

The **net interface** of a filter consists of only one method:

- `classify` tries to determine whether the skbuff passed to a filter matches. An skbuff *matches* when it belongs to a traffic class the filter is differentiating. Simple filters may only require reading of a skbuff field, e.g. `priority` field. More complex filters may require accessing an IP header (source, destination) or L4 header (ports). Filters containing subfilters use internal classification.

The **user-interface** of a filter consists of following methods:

- `init` is a filter constructor. Filter specific data is passed through this function, e.g. priority, source-address or destination-port.
- `destroy` is filter-destructor.
- `change` is used for adding a new subfilter.
- `delete` is used for deleting a subfilter.
- `get` is used to map a subfilter user-id to a subfilter kernel-id and for incrementing the reference count.
- `put` is used for decrementing the reference count of a subfilter.

3.2.4 Code

The main data structure of traffic control is `struct Qdisc`, containing Qdisc´s private data. The fields are `dev`, backpointer to a device, `sk_buff_head* q`, a list of skbuffs, `handle` for a qdisc user-id, `refcount` for counting the usage of qdisc – for determining when the qdisc may safely be deleted. The structure `tc_stats` is used for collecting statistics and contains the following relevant fields: `bytes` for counting the volume of the data serviced by a qdisc, `packets` – number of packets passed through a qdisc, `drops` number of packets dropped, and `backlog` – actual number of packets in the queue. These are the standard fields that each qdisc contains. Qdisc-specific data (e.g. limit of a *bfifo*) is anchored in the pointer `data[0]`, which is casted into a qdisc-specific structure before usage. Every class has a pointer to qdisc-methods, namely `struct Qdisc_ops`, where the methods are implemented. Besides pointer functions, this structure stores the unique name of the element (`id`). Classful qdiscs also have a pointer to `struct Qdisc_class_ops`.
The anchor of the possibly large runtime qdisc hierarchy is in the structure `device`. The field `qdisc` points to a root queueing discipline of an interface. The field `qdisc_list` contains a linked list of all the subsequent qdiscs of an interface.

Filters are represented by the structure `tcf_proto`. L3 protocol is stored in the field `protocol`, `root` points to filter-specific data and is being casted appropriately before usage (e.g. hash-table by SDRSVP filter), `struct tcf_proto_ops *ops` are user interface and net interface methods described in the previous section, `q` points to a parent qdisc.

The traffic control implementation is distributed among following files:

- `sch_api.c` and `cls_api` are manager modules accepting Rtnetlink messages at one side and invoking calls on qdiscs and filters respectively. These modules transform the Rtnetlink messages into *user-interface* method invocations of qdiscs and filters. This transformation occurs in three steps: find the parent-qdisc starting at `dev::qdisc_list` or a traffic filter, check for validity of parameters, invoke the method.

- `sch_generic.c` contains three important housekeeping functions. `qdisc_restart` is called from the protocol stack (already seen in Figure 3.3). It calls dequeue and passes the returned skbuff to the device driver. `qdisc_run_queues` is called from the NET BH and calls `qdisc_restart` for all the available devices. `dev_do_watchdog` is a periodic timer, called every 5 seconds. The timer invokes `qdisc_restart` and empties the queues if some skbuffs are still waiting to be served.
- `sch_*.c` are implementations of different classless (e.g. BFIFO) and classful (e.g. CBQ) queueing disciplines.
- `cls_*.c` are implementations of different filters.
- `pkt_sched.h` and `pkt_cls.h` in the `include/net` directory contain internal kernel declarations of structures, interfaces and constants.
- `pkt_sched.h` and `pkt_cls.h` in the `include/linux` directory contain declarations which are possibly shared with user-space programs, e.g. Rtnetlink messages and constants.

The following classless qdiscs are available in the kernel 2.2.9: byte-limited FIFO (bfifo), packet-limited FIFO (pfifo), Random Early Detection Queue (RED) – early detection of congestions, Token Bucket Filter queue (TBF), and Stochastic Fairness qdisc (SFQ) – a version of fair queueing. Following classful qdiscs are implemented: Class Based Queueing (CBQ), Clark-Shenker-Zhang qdisc described in [16], Prio qdisc – a simple priority scheduler, ATM qdisc – an qdisc used for traffic control over ATM, used in the DIANA architecture prior to this work.
Among the filters, there are firewall filters, using the marking functionality of firewall code for filtering skbuffs, Route Filter – route based classifier using the `route` field of a skbuff, RSVP Filter and the so called Universal Filter which can match protocls, IP addresses, etc[5].

3.3 RTNetlink

Routing tables have traditionally been modified and read through diverse `ioctl` system calls. Starting with the BSD 4.3Reno Net implementation [38][39], the so called *routing sockets* are used for the routing table manipulation. An open routing socket doesn´t cross the network border, it is an open connection from user- to kernel-space. The Linux implementation differs from the BSD architecture, following the same idea, but generalizing the concept. Linux *Netlink Socket* is a generic connection from the user space to the data link layer in the kernel. Netlink itself defines the protocol family (PF_NETLINK), but the real functionality is dependent on the protocol type. Some of the defined protocols are NETLINK_ROUTE, used to modify the routing tables, NETLINK_FIREWALL for reception of packages fro the firewall code, NETLINK_ARPD for managing the ARP-table in user space.

NETLINK_ROUTE protocol, further called *Rtnetlink*, originally designed for routing tables, can be used for the following functionality:

- reading and altering the routing tables in kernel
- managing the ip-addresses, link-parameters (interfaces) and neighbors
- managing traffic control elements; queueing disciplines, traffic classes and filters

[5] See code for the details about all mentioned elements.

Using sockets has certain advantages over oneway `ioctls`. Sockets enable the application communicating with the kernel in a well-known and standardized way. The usual file manipulation operators, including asynchronous communication, can be used. Netlink sockets define different multicast groups, which are being notified when any change in the kernel takes place. This way, traffic control sensitive applications use netlink socket-polling for receiving information when some other application adds or deletes a traffic class, a qdisc or a filter. This dedicated connection makes the notion of notifying the listeners about some event more efficient then the use of the broadcasting `/proc` file system. The third advantage is the echo-functionality of the netlink sockets, echoing requests makes the auto-generations of traffic control handles possible, thus letting the kernel generate handles itself.
In the following sections, we will take a look of the general format of netlink messages and further observe how Rtnetlink is used for communication with the traffic control in kernel.

3.3.1 Netlink Sockets

Netlink distinguishes between two main kinds of messages: requests and error messages. Normally, the applications sends a request to the kernel and receives an error message carrying an acknowledgement. The following structure shows the netlink header.

```
struct nlmsghdr
{
        __u32           nlmsg_len;      /* Length of message including header */
        __u16           nlmsg_type;     /* Message content */
        __u16           nlmsg_flags;    /* Additional flags */
        __u32           nlmsg_seq;      /* Sequence number */
        __u32           nlmsg_pid;      /* Sending process PID */
};
```

`nlmsg_len` contains the length of the message, including header and payload. `nlmsg_type` contains the protocol specific type (command). Type NLMSG_ERROR is used in the acknowledgement messages. Following `nlmsg_flags` are defined: NLM_F_REQUEST is set for all the requests, NLM_F_ACK requests an acknowledgement for the request, NLM_F_ECHO echos the request back to the application after being processed by the kernel. `nlmsg_seq` defines the sequence number which is than used to identify the acknowledgement. `nlmsg_pid` contains the PID of the process sending the messsage, being 0 if the sender is the kernel.
As with all socket families, netlink defines its own socket address structure:

```
struct sockaddr_nl
{
        sa_family_t     nl_family;      /* AF_NETLINK   */
        unsigned short  nl_pad;         /* zero         */
        __u32           nl_pid;         /* process pid  */
        __u32           nl_groups;      /* multicast groups mask */
};
```

When connecting to a socket with `bind`, one can specify the multicast group defining what messages are to forwarded to the socket. As an example, RTMGRP_TC (*Routing multicast group traffic control*) receives all the traffic control relevant messages sent to and from the kernel, and RTMGRP_IPV4_ROUTE all the changes in the routing table.

As with all the protocols beneath the socket layer, netlink implements the `proto` structure. When an application sends a traffic control netlink message to the kernel, e.g. requesting adding of a traffic class, this message passes the socket layer and is handed out to netlink, which determines the message type and invokes the appropriate function in the traffic control module. The message is

passed by reference which enables the modification of the message fields. If the operation succeeded, an acknowledgment is sent to the sending process and the message is sent to all the listeners in the multicast group. The sender may itself be a multicats group listener, yet it does not receive the sent message.
In case that the ECHO flag has been set, the original process get the copy of the message as well. It is to note that the echoed message and the multicasted message have been processed by the kernel before sent back.

3.3.2 Traffic Control & Routing Netlink

In this section, we describe the traffic control functionality of the Rtnetlink, which is used for manipulation of the traffic control elements in the kernel. The Rtnetlink message is placed in the payload of the netlink message. The traffic control specific header *TC Message* carries the information on the element, defining what place the element should take in the hierarchy and what network interface is to be modified.

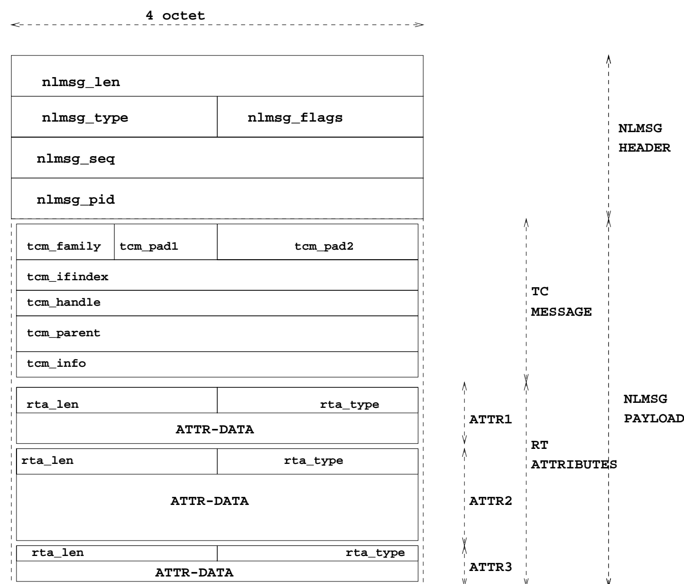

Figure 3.6: Rtnetlink message format

The length field of the netlink header carries the whole length of the message. The message type `nlmsg_type` is used for adding, deleting and requesting information about qdiscs, traffic classes and filters being one of:

```
{ RTM_NEWQDISC, RTM_DELQDISC, RTM_GETQDISC,
  RTM_NEWTCLASS, RTM_DELTCLASS, RTM_GETTCLASS,
  RTM_NEWTFILTER, RTM_DELTFILTER, RTM_GETTFILTER }
```

TC_MESSAGE carries traffic control specific information. `tcm_family` is always AF_UNSPEC padded with 0s. `tcm_ifindex` carries the unique device number. `tcm_parent` is a handle of the parent element, being a class handle when adding qdiscs or qdisc handle when adding classes. `tcm_handle` carries the handle of the element being added or deleted. The `tcm_info` is an extension used for various purposes depending on the qdisc kind.

Third part of the message is one or more **routing - attributes** (rtattr). Routing attributes are used to pass the qdisc, traffic class or filter specific data to the kernel. Each `rtattr` header consists of the 16 bit type field and the 16 bit length field. One standard attribute type is the element kind, being represented in plain ASCII. The rtattr type is TCA_KIND and the length of the element is `strlen(name)+1`. Using the name as identification enables finding the requested `Qdisc` or `tcf_proto` according to the `kind` field.

When passing routing attributes as options, we have two distinctive cases:

- If all the options are passed through one single structure, TCA_OPTIONS attribute type is used. Byte limited fifo qdisc contains the number of bytes in the payload of the attribute, because this is the only option supported.
- If more then one option is used, TCA_OPTIONS carries the length of all the options and other rt-attributes are packed as payload.

Depending on the *message type*, diverse functions in the file `sch_api.c` for qdisc and traffic class manipulation and `cls_api.c` for filter manipulation are called. We roughly describe the control flow with a following example. Let´s assume that a Rtnetlink message arrives, requesting the addition of a new class . Rtnetlink maps the message type RTM_NEWTCLASS into the function `tc_ctl_class`, which is invoked. `dev_base` global variable containing a list of all the devices is traversed, searching for the interface index. If the interface exists, `tc_ctl_class` uses `qdisc_lookup` to map the device-index and parent qdisc handle to a pointer to `struct Qdisc`. If the qdisc has been fond, the field `ops::cl_ops` is compared against NULL to make sure a classful qdisc is accessed. If the user-id has been specified, `get` is used to map the user-id into kernel-id. Kenel-id 0 means that a class with such id does not exist. If the class exists, the handle is in use and an error is returned. Furter, the method `change (class)` adds a new class into a classful qdisc. Finally `put` is invoked to decrement the reference count incremented by `get`.

3.3.3 `tc` Program

TC program is a command line utility used for manipulating the traffic control elements in the kernel. The options are passed to the executable as arguments.
Let us assume that there is a classful qdisc with the handle 1:0 and one class with the handle 1:2 containing the packet FIFO queueing discipline for managing packets. This qdisc carries the handle 3:0. Following command line replaces the 3:0 pfifo qdisc with a bfifo qdisc carrying id 2:0 with the byte limit of 800 bytes.

```
tc qdisc replace dev eth0 parent 1:1 handle 2:0 bfifo limit 800
```

These options are passed in the following order.

1. `tc qdisc` indicates that the command line is about to modify some queueing discipline. Depending on the traffic control element being modified, processing continues in `tc_qdisc.c::do_qdisc`, `tc_class.c::do_class` or `tc_filter.c::do_filter`.
2. `replace` sets type and flags fields of the Netlink header accordingly. Table 3.3 shows the mapping for the qdisc traffic element. The sequence number is generated and filled into the `seq` field of the Netlink header, as well as the Pid of the program.

Command	nlmsg_type	nlsmg_flags
add	RTM_NEWQDSC	NLM_F_EXCL | NLM_F_CREATE
change	RTM_NEWQDSC	0
replace	RTM_NEWQDISC	NLM_F_CREATE | NLM_F_REPLACE
delete	RTM_DELQDISC	0
dump	RTM_GETQDISC	NLM_F_ROOT | NLM_F_MATCH

Table 3.3: Netlink header fields

3. `dev eth0 parent 1:1 handle 2:0` are the standard options for every class, and used for generating the TC_MESSSAGE. `eth0` device is mapped into integer 3, a unique kernel interface index and filled into the field `tcm_ifindex` [6] . User-id of the parent `1:1` is packed into a 32-bit field, with higher 16 bit carrying the major, and lower 16 bit carrying the minor, being thus `0x10001`. The value is placed in the `tcm_parent` field. Qdisc handle `2:0` is mapped into the 32 bit field as described (`0x20000` and placed in the `tcm_handle` field. `tcm_family` gets the value AF_UNSPEC and `tcm_info` is left 0.

4. `bfifo` is a qdisc kind. Each qdisc kind is associated to one structure implementing the interface towards the tc. The structure is named `qdisc_util` and consists of a ASCII field `id` carrying the unique qdisc name from the kernel mentioned before in this chapter, and several methods. Those are: `parse_qopt` for parsing the qdisc-specific options, `print_qopt` for printing the qdisc-specific optins when dumping information from the kernel, `parse_qopt` for parsing class options, and `print_qopt` for print class-specific options. Latter two are only implemented for classful queueing disciplines and are being called when specifying `class` as second argument to TC.
   When specifying `filter` as a second argument, same mechanism is used, yet using the structure `filter_util`, which contains a unique filter name (`id`) and two methods `parse_fopt` for parsing filter-specific methods, and `print_fopts` for printing those.
   Now, after localizing the `tc_util` named bfifo, the fist Rt-attribute is generated. It carries TCA_KIND as a type and bfifo in ASCII form as payload (attr-data). The method `parse_qopt` of the fifo object is called with the remaining arguments as parameters.

5. `limit 800` is parsed by the bfifo object. The option limit is placed in the structure `tc_fifo_qopt`, which contains only one field, `__u32 limit`. Limit is filled with 800 and the structure is packed in a payload of the (second) Rt-attribute TCA_OPTIONS [7] .

6. The Rtnetlink message is now complete and is being sent to the kernel. If the qdisc has successfully been replaced, acknowledgement from netlink is received. Otherwise, an error message is sent back to the application indicating the reason. A full list of error codes can be found in `include/asm-i386/errno.h`.

When dumping information from kernel, the tc utility receives a large Rtnetlink message with all the information requested. This is the point where `print` methods from the traffic control element - interface are used for analyzing messsage components and printing the human readable information to standard output.

[6] This mapping uses Rtnetlink by sending a request containing the name of the interface and receiving an interface index from the kernel.

[7] Kernel uses `u32` and user-space applications `__u32` for an unsigned integer 32 bits long. Using this type and others defined in `asm/types.h` makes programs more portable, since no assumptions are made about the hardware architecture. With other words `__u32` stays an unsigned 32bit integer on Intel, Alpha or Sparc.

TC doesn´t access the Rtnelink sockets directly, but uses the `libnetlink` library as e set of predefined and reusable functions. For more information on tc, refer to code and [34].

3.4 ATM in the Linux Kernel

Linux ATM kernel extension consists of a kernel patch, including device drivers and support for sockets among other stuff, and several user space programs used for administering ATM connections, the Signaling Daemon and ATM-ARP Daemon to name a few. In the following section, we shall shortly describe the structure of the ATM extension, concentrating on the components needed for our architecture.

3.4.1 ATM-Sockets

Physical ATM devices are represented through a structure `atm_dev`. The implementation of this structure is an ATM device driver. Every *virtual connection* (VC) is represented through a **`struct atm_vcc`**. This structure consists of a variety of fields needed for the frame transmission. `vpi, vci` identify the virtual connection and path, `atm_dev` is a backpointer to an ATM-device the frame is leaving by, `atm_qos` defines QoS parameters in receive and transmit direction and type of AAL used, `atm_options` is a bit-field, where for example the *Cell Loss Priority* (CLS) bit can be set. The ATM-Kernel patch extends the `sk_buff` structure with a pointer to a `struct atm_vcc`. This pointer has to be set before the skbuff reaches the send operation of a device driver

The figure 3.7 shows the rough structure of the ATM socket architecture. Depending on the family, the socket layer is implemented by the SVC or PVC `proto` structures. The setup and termination operations (`bind, close`) need ATM signaling for setting up and closing of a VC. Due to the complexity of the ATM-signaling and multiple protocol layers, the Signaling Damon is implemented in user space. The SVC module in kernel communicates with the sigd through a relatively simple *internal signaling protocol* (see [7] for a description of an older version). After receiving a request, sigd uses the signaling socket (signaling VC – 0/5) for UNI connection establishment and returns an acknowledgement back to the kernel notifying success.
Sending data over the open socket-connection calls the device´s send directly passing it to atm device driver `struct atm_dev::send`.
After receiving data, the device drivers allocates an skbuff and sets the pointer to the `atm_vcc` structure depending on the incoming VPI/VCI. Now the function `atm_vcc::push` is called, which pushes the skbuff up one layer. This function calls actually `atm_push_raw` belonging to the `atm_raw` device, which in turn pushes the data into the socket layer, waking up any blocking calls and sleeping processes.
As noted, when transmitting data, no device is used, the skbuff is passed directly to the device driver. When receiving packets, the so called atm-raw device is used for transferring data to the SVC sockets.

3.4.2 CLIP

The raw atm device described presents a thin layer between a socket and a device driver, not able of handling IP traffic. There a three devices (`struct device`) that accept IP-traffic:

- Multiprotocl over ATM - *mpoa* device.
- LAN Emulation Client - *lec* device, used with LANE.
- Classical IP over ATM - *CLIP* device – described below.

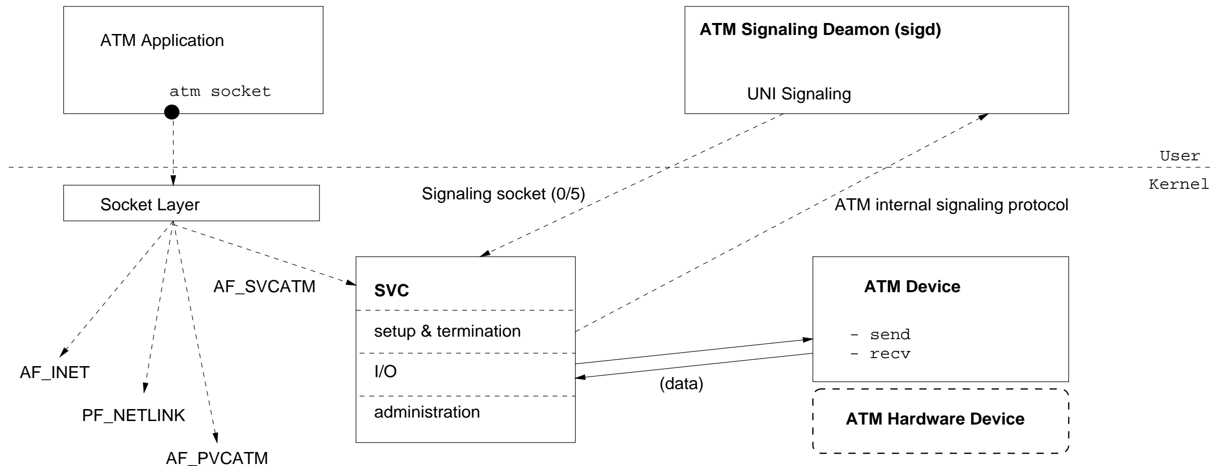

Figure 3.7: ATM Sockets internal structure

The CLIP implementation consists of two main modules: the *ATM-ARP Damon* executable in the user space and the clip-module in kernel. The daemon communicates with the atm-arp server in the Logical IP Subnet through an atm-arp protocol [9]. ATM - ARP protocol covers the obligation of registering within an ATM-ARP server, and the legitimacy of requesting an ATM-address of an IP address. The daemon also takes charge of the atm signaling - setting up a dedicated VC to the requested host in the ATM logical subnet.
The kernel CLIP module implements a network device (`struct device`) and manages open connections. If the destination address is not reachable by an open VC, the clip module notifies the ARP Daemon requesting for signaling. After some VC has been inactive for a specified amount of time, the connection is released.
Packet transmission is initiated when an skbuff reaches `dev_hard_xmit` of the CLIP device. This function adds the LLC (RFC1483) header (`skb_push`) depending on the 802.x protocol used and checks for an open connection. If the connection stands, the `atm_vcc` is added to the skbuff, which is then passed downwards to the device driver (`atm_dev::send`).

When receiving packets, the function `atm_vcc::push` calls `clip_push` thus forwarding the packet into the clip-module (and not to the atm socket layer). The LLC encapsulation is checked for the IP protocol and then removed (`skb_pull`). The skbuff is thereafter passed through `netif_rx` to an input queue, where it will be passed up to the IP layer with the next execution of the NET BH.

Manual CLIP Signaling

When establishing a socket connection to a remote host, the default service access point (SAP) is the raw-atm device passing all the data to the socket layer. If we try to establish a CLIP connection to a remote host, this raw atm device will not satisfy, because IP packets are handed out to the socket and not pushed up the IP protocol stack. In order to achieve this, two steps are necessary:

1. Before calling `connect`, the expected *SAP* type of the remote host has to specified. The `atm_sas` field of the atm specific socket address `atm_svcaddr` contains a field holding the E.164 address of the end host and a `blli` field holding the *Broadband Low Layer Information* (BLLI) attribute of the SAP. In case of IP, the abstract Logical Link Control (802.2) SAP

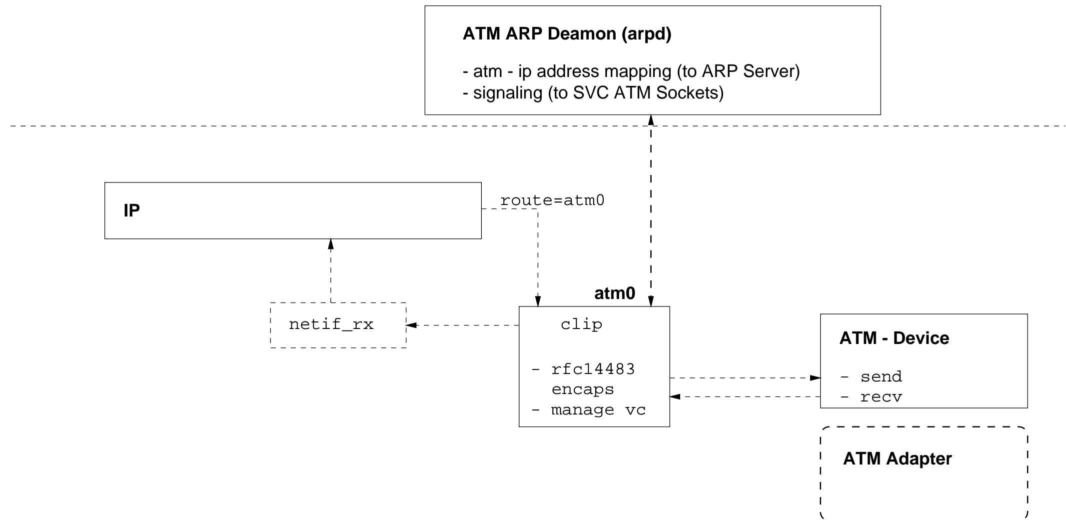

Figure 3.8: CLIP internal structure

is used as layer 2 protocol. LLC protocol as an entry is actually a multiplexer for diverse 802.x protocols which can be tunneled over an SVC. Thus, after establishing a connection, the input of the receiving socket is forwarded to the IP layer on the remote host. All the transmitted packets have to wrapped with the LLC encapsulation.

2. After the connection has been established, the local host data link layer has to be instructed that all the arrived packets are to be passed to the IP layer as well. Calling `ioctl` on the atm socket file descriptor with the MKIP parameter exchanges the `atm_vcc::push` function pointing to `push_raw` with `push_clip`. As before, all the IP packets have to be wrapped with the LLC encapsulation header, or otherwise they are discarded.

Chapter 4

DIANA Traffic Control Extension

This section presents the design and implementation of the DIANA Traffic Control extension for supporting RSVP over ATM.
As with all the work in kernel, most of the design decisions were dictated by already existing structures, leaving relatively little place for *own* ideas. It was rather a design compromising the requirements with structure of the traffic control, Rtnetlink and ATM modules.
As with all the work in a team, most of the design decisions were dictated by colleges working on the Atmtcd executable and asking for features in the kernel.

Several new traffic control elements were developed. DIANA-Prio (dprio) classfull queueing discipline is a simple priority scheduler, which served as the prototype and experimental platform for the ATM-dprio[1]. ATM-Dprio can be considered as a specialization (or subclass) of dprio and defines the one-to-one relation between class and virtual connection (VC). RPQ (*Rotating Priority Queues*) queue is a classless queueing discipline, that aims providing the so called bounded delay service in a simpler way then the earliest-deadline-first (EDF) queue. Simple DIANA RSVP (sdrsvpd) Filter is a reduced and simplified version of the RSVP-Filter, mostly implemented in the lack of understanding for the code architecture of the original RSVP-Filter.

We will explain single components of the extension first and come back to the overall architecture proposed in section 2.6.1 later.

4.1 Adding new elements to traffic control

Adding of new classfull qdisc is fairly easy, here a step by step introduction:

1. The file containing code should be placed in the `net/sched` directory and named `sch_`*qdiscname*. The methods exported by the classless qdisc should be implemented and pointers to those implementations saved in structure `Qdisc_ops` named *qdiscname*`_qdisc_ops`. The `id` field is initialized with the unique *qdiscname* name.

2. If the qdisc should be classful, i.e. containing classes and traffic filters, the methods of the classful interface are to be implemented too and placed in a structure `Qdisc_class_ops` named *qdiscname*`_class_ops`.

[1] since it is easier to experiment with ethernet, then with ATM

3. Set up the registration of the qdisc at boot time in sch_api.c by adding the macro `INIT_QDISC`*(pfifo)* in the function `__initfunc`.

4. Define the structures that are to be shared between the user-space and the kernel in include/linux/pkt_sched.h. Those components are rt-attributes specific to qdisc and structures which are carried as payload in Rtnetlink messages.

5. The new file should be added to the `Makefile` in the `net/sched` directory, kernel recompiled and computer rebooted.

6. TC utility in user-space can be extended by implementing the `qdisc_util` structure in a file named `q_`*qdiscname*. The makefile should be modified so that a new file is included in the executable. Finally, the `tc` is recompiled.

Modifying the `tc` program is not mandatory, but it enables easy manipulation of the newly implemented qdisc in the kernel.

4.2 Dprio

DIANA-Prio queueing discipline is a simple priority scheduler, with a limited number of different class priorities. The number of priorities has arbitrarily been chosen and set to 255.
The runtime structure of the *dprio* qdisc is the same as already shown in the figure 3.4. As already noted, filters determine the class of the skbuff and forward it to the qdisc belonging to a class. Each Dprio scheduler contains one default class, which has the lowest possible priority and manages traffic if there are no filters and no classes. Inner qdisc of this default class is a byte-limited FIFO qdisc. The default class is enumerated with `x:ffff`. This default class may not be deleted and can be considered a fixed part of the scheduler.

The figure 4.1 shows the object-oriented structure of the qdisc. Dprio manages a pointer to a linked list of filters, a linked list of classes and a pointer to a default class.

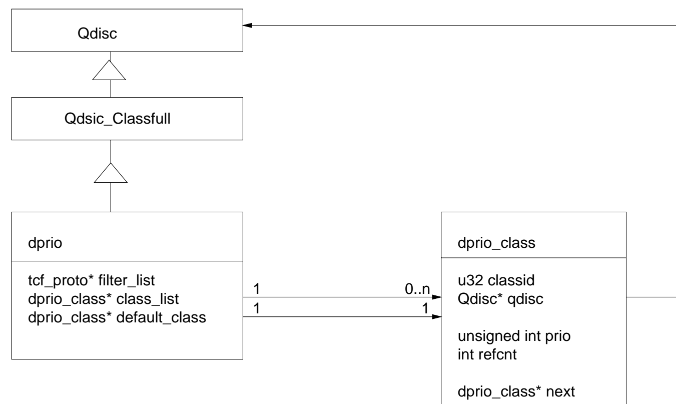

Figure 4.1: Dprio classful qdisc

Every dprio_class object has a unique classid, with the first 16 bit being the major of the parent-id, keeping the numeration scheme previously described – namely x:0 for qdiscs and x:y for classes .

The `qdisc` field of `dprio_class` points to a qdisc which manages skbuffs passed to a class. Being a pointer to a generic qdisc – an object oriented parent class of all the qdiscss, the same interface is used independent of the runtime type of a qdisc. The priority `prio` determines the position in the linked `class_list`. Higher priorities are placed before the lower priorities in the list, thus being scheduled earlier.
Reference count of a class is stored in the variable `refcount` and is being incremented each time a filter is added which uses the class as the forwarding point, and decremented when such a filter is deleted. Thus, we may prohibit deleting of a class if any filter still uses it.
Filters in the `filter_list` use the same priority mechanism, being added higher or lower depending on the `prio` field of `struct tcf_proto`. Thus if two filters potentially match an arrived skbuff, the filter with the higher priority determines the class.

Now we describe the functiolity of the methods. After calling `enqueue`, the method `classify` is called on every filter in the list. The returning structure `tcf_result` contains the class user-id and kernel-id. If any of the two fields is not null, the filter *matched* and no further classifications are executed. If kernel id (`unsigned long`) is set, it is simply casted into the structure `dprio_class`. On the other hand, user-id of a class requires one further traversal of the class list looking for a kernel-id. Now, the enqueue method of the qdisc in the class is called (`dprio_class::qdisc::ops::enqueue`). If the enqueue returns 0, the packet has been dropped, otherwise, the packet has been stored.
If none of the filters matched, the skbuff is enqueued in the default queue.

Calling `dequeue` traverses the class list top-down, i.e. from the higher to the lower priority. Dequeue is called for all the qdiscs belonging to the specific class. When the first dequeue returns an skbuff (an not NULL), this skbuff is returned. If none of the qdiscs had a packet to transmit, dequeue is called on the default classes qdisc.
Calling `drop` tries dropping one skbuff from the default queue. If this doesn´t succeed, the classes are traversed top-down looking for a packet to drop. Actually, it would be more natural to start looking for packets to drop at lower priority classes, but since we are not maintaining a doubly linked list, this bottom-up traversal would be highly inefficient.
Calling `requeue` classifies the packet (for the second time) and calls requeue of the selected qdisc.

Handles which are not explicitly specified from the user are generated in the following way: if having only one dprio qdisc on the interface, and this qdisc is identified by 1:0, the default class gets the first available handle from the top, namely `1:ffff` (in hex), the FIFO queue being grafted into the default class becomes the first unused qdisc-handle, namely `ffff:0`. Next added class is enumerated with `1:fffe` and the next qdisc `fffe:0`.

Rtnetlink Messages As explained in section 3.3.2 the rt-netlink message consists of three main parts: netlink-header, traffic control header and element-specific routing-attributes. Table 4.1 shows the values for the traffic control header fields.

Message-type	Field	Mandatory	U→ K	ECHO (K→U)
RTM_NEWQDISC	`tcm_handle`		qdisc-handle	generated qdisc-handle
RTM_NEWQDISC	`tcm_parent`	•	parent class-handle	parent class-handle
RTM_NEWTCLASS	`tcm_handle`		class-handle	generated class-handle
RTM_NEWTCLASS	`tcm_parent`	•	parent qdisc-handle	parent qdisc-handle
RTM_NEWTCLASS	`tcm_info`			generated qdisc-handle

Table 4.1: TC_MESSAGE Part of the Rtnetlink message with Dprio

As we can see, providing a parent class when adding a qdisc and parent qdisc when adding a class is mandatory. Qdisc - handle and class handle can be generated in the kernel and passed back if the ECHO flag of the Netlink header is set. When adding a new class, the auto-generated handle of the BFIFO qdisc that is automatically grafted is returned in the info field.

When adding, modifying or deleting a new qdisc (message types RTM_NEWQDISC and RTM_DELQDISC), only the predefined rt-attribute TCA_KIND is required, carrying the name of the qdisc (*dprio*) in plain ASCII form.
When adding a new class (message type RTM_NEWTCLASS) following optional rt-attributes may be included:

- DPRIO_PRIO carries the traffic class priority determining its position in the list and thereby the scheduler priority. Classes with higher priority are being server first – dequeued first. This attribute is not mandatory, the default class-priority is 255, being at the tail of the list, yet before the default class. When dumping the traffic control information from the kernel to the user space, this information element of type `unsigned int` carries the assigned priority.
- DPRIO_REFCNT attribute is only passed from kernel to user when dumping information. The `int`-typed field carries the reference count of the class, i.e. number of filters forwarding traffic to it.

Here a short summary: when adding a class, rt-attributes are as following: TCA_KIND carries text "dprio" in payload, TCA_OPTIONS carries the length of the following attribute, DPRIO_PRIO carries the priority as `unsigned int` in the 32 bit field; when dumping information from kernel to user-space, TCA_KIND carries the text, TCA_OPTIONS the length of the following two options, DPRIO_PRIO the assigned priority and DPRIO_REFCNT the reference count of type `u32`.

4.3 ATM-Dprio

ATM Dprio queueing discipline is a classful qdisc which ties one traffic class to exactly one open virtual connection. It a logical subtype of the Dprio qdisc, thus inheriting all the characteristics of dprio described in the previous section.
The main **requirement** for atmdprio originates from the proposed architecture for RSVP over ATM, which enables merging of multiple IS-flows into one VC. Translated in the ATM-Dprio architecture, those are multiple RSVP traffic filters forwarding traffic to one atmdprio class.
Viewed from the Atmtcd context in user space, Atmtcd decides that a dedicated connection (VC) to some next hop host in the logical ATM subnet is needed, which is then accordingly set up through signaling. Afterwards, a new class is added providing the kernel with the Atmtcd Process ID (PID) and the file descriptor of an open ATM socket. This pair (PID and File descriptor) can then be mapped into the `atm_vcc` structure at the kernel side and thus uniquely identify the open VC. As the figure 4.2 shows, the dprio_class is extended with some atm-specific data, with the pointer to the atm_vcc structure being its central part.
After the routing tables have decided that the skbuff is to leave on the *atm0* interface and the `dev` field of the skbuff has been set appropriately, `dev_queue_xmit` addresses the qdisc of the device and calls `enqueue`. `enqueue` traverses the filters and passes the skbuff to a class. Shortly after, `dequeue` is called asking for one skbuff. However, we are **not** supposed to return any packets, because these would be passed on to the CLIP device through `dev_hard_xmit`. This way, none of the packets would ever reach the newly signaled VC, because CLIP manages its own VCs to each host in the logical ATM subnet.

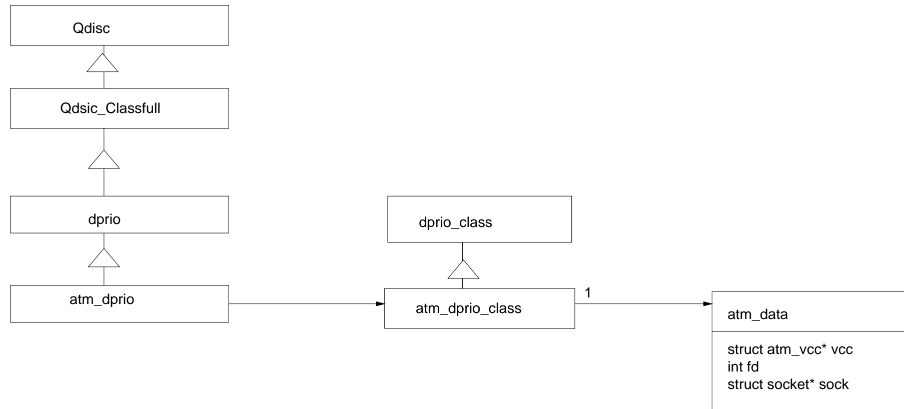

Figure 4.2: ATMDprio

This is why we use the following *hack* [2] : packets are still being stored by `enqueue`, but when `dequeue` is called, the transmission of the packet is initiated by calling send of the atm device driver (`atm_dev::send`). Dequeue finally returns NULL, as if no packets were available. Following steps are needed before the transmission can take place:

- Set the `atm_vcc` pointer of the skbuff to the `atm_vcc` of the VC we are about to use.
- Set skbuff atm-options to the options defined in the `atm_vcc`.
- Encapsulate the packet with the LLC encapsulation defined in RFC1483 by `skb_push`.

The default class of the Atmdprio qdisc is the class passing traffic to CLIP. There is no special functionality in the default class, `dequeue` returns the next skbuff queued, which is then passed down to the CLIP-send operation.
It is to note that default class logically covers more then one VC, actually one for each next hop reachable over ATM, at maximum. Other classes, on the other hand, do not have such multiplexing capability. Packets passed to other classes always end up at the host at the other side of the VC. For this purpose, traffic filters must guarantee that skbuffs passed to a class actually **are** addressed for the node behind the VC.

Rtnetlink Messages Table 4.2 shows the rt-attributes, their names, types and directions they are being sent to. ATM_DPRIO_PRIO and ATM_CLASS_REFCNT have the same semantics as their counterparts in the Dprio qdisc. The file descriptor and the PID are mandatory. VPI/VPI pair is only being passed back when dumping the information.

The `pid,fd` pair is needed for accessing the atm_vcc structure the class relies on. Obtaining this VC connection information is not straight forward, but requires some browsing of Linux kernel structures:

- The list of the tasks starting at global `init_task` is traversed looking for the *pid*.
- The file descriptor is used as an index in the file field of the task, obtaining the pointer to `struct file`.

[2] As seen in the atm qdisc.

Inform.Element Name	Type	U→K	Mand.	K→U	Assertion
ATM_DPRIO_PRIO	unsigned int	•		•	in Range 0..255
ATM_DPRIO_FD	int	•	•		file descript. valid and atm-socket
ATM_DPRIO_PID	pid_t	•	•		existing process
ATM_DPRIO_VPI	int			•	
ATM_DPRIO_VCI	int			•	
ATM_CLASS_REFCNT	int			•	

Table 4.2: Rt-attributes for message type RTM_NEWTCLASS

- Now the field `struct file::f_dentry::d_inode` is checked for existence and assured that it is a *socket*.
- The element `u.socket_i` points to a socket structure in the kernel.
- Now we check if this is an ATM socket and if the socket is connected at all.
- The field `sk::protinfo.af_atm` is the atm_vcc structure we are looking for.

4.4 RPQ

In section 2.2, we have already introduced the Static Priority and Earliest Deadline First schedulers. Rotating Priority Queues (RPQ) Scheduler [32] is a *novel* approach for providing bounded delay service scheduleability.
The main **application** of RPQ in our architecture is a scheduler for the merged guaranteed service flows. If multiple guaranteed service flows are being merged into one VC, we have to guarantee that each of them gets the quality of service desired. If the flows request 10, 20 and 30 ms delay latency, they can be serviced by the RPQ consisting of at least 3 queues.

4.4.1 Architecture

RPQ consists of $n+1$ FIFO-queues and n different priority levels. The queues are named $0, 1, 2, ..., n$ with the queue 0 having the highest, and queue n having the lowest priority. When the packets arrive, they carry their priority level between 1 and 0. This priority level determines in which queue the packet is being enqueued, packet carrying priority 2 in the second queue and so on. Since priority level 0 is not permitted, none of the packets are ever being enqueued in the 0th queue.
When the outgoing intrface becomes idle and is able to handle p packets, the 0th queue is asked for packet delivery, then the 1st queue, the 2nd and so on until p packets have been transmitted. This way, higher priorities are serviced first. Each RPQ has a so called *rotation interval*. Rotation interval is a period of time (in ms), after which the queues are being rotated. At the beginning the 0th queue is the current queue. After the first rotation, the queue with (old) priority 1 becomes the 0th queue, the priority 2 queue becomes queue number 1 and the 0th queue becomes the n queue.

Figure 4.3 shows a running RPQ. It consists of 4 FIFO queues, and thus 3 different priorities. After the arrival, packets are being enqueued according to their priority (1). At (2), we have the first rotation. At (3), the outgoing line has become idle and can accept 3 packets. One packet is taken from the 0th queue and 2 packets from the 1st queue. At the time (4), new packets arrive and are being queued appropriately. Step (5) rotates the queue one more time after the rotation

interval has elapsed. At (6) the outgoing line takes 2 packets from the 0th, and 2 more packets from the 1st queue.

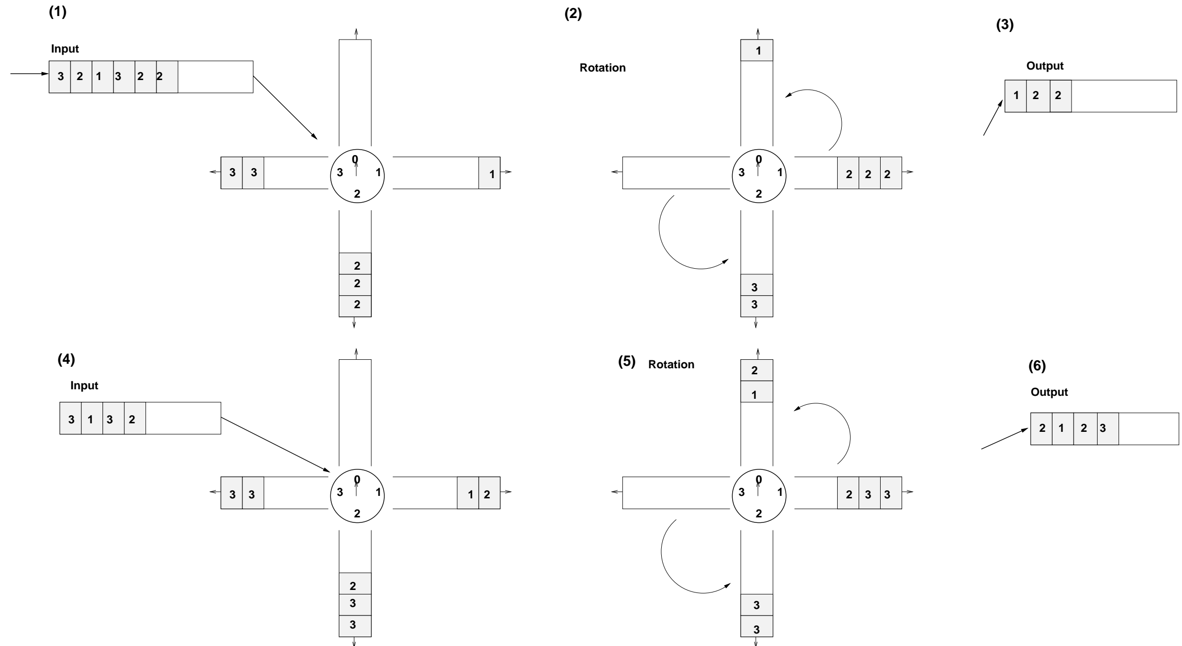

Figure 4.3: RPQ Example

As the figure shows, higher priority packets are being serviced first, decreasing packet delay. Lower priority packets, on the opposite, spend more time in the queue. Packets with the highest priority are queued for maximum time of the rotation interval, because they are served straight after the next rotation in the worst case. Packets in the second queue are thereafter served after *rotation-interval* $\times$ *2* in the worst case. Of course, the reserved bandwidth of the outgoing line must be able to handle the traffic volume of the incoming packets.

There are few more points to mention before we describe the implementation of the RPQ queueing discipline.

- The FIFO queues have limited capacity. As we are using the byte limited FIFO queueing discipline, these queues start dropping packets after the limit has been exceeded.
- RPQ is going to be used as a qdisc grafted into the highest class of the atmdprio qdisc. Since the outgoing side of the atmdprio is a simple priority scheduler, we shall have to limit the amount of data being transmitted in order to prevent the starvation of the traffic belonging to lower classes. The so called *service limit* defines the maximum amount of bytes being transmitted within one rotation interval. As an effect of service limit, we may have a situation where the scheduler requests packets for transmission, but the RPQ returns NULL, although there are packets to be sent. Due to the service limit, the RPQ acts as a simple shaper. However, the shaping is not fair, since large bursts of some low-priority class traffic may cause dropping of conformant traffic.

- The packets caught in the 0th queue at the rotation time are being dropped. Thus, the *n*th queue is always empty shortly after the rotation.
- Packet – skbuff priority cannot be determined by the qdisc. Traffic filters can identify flows and should thereafter be able to forward the skbuff to RPQ and handover the priority.

4.4.2 Implementation

Private data of the RPQ qdisc is shown in fig.4.4. The circular structure is modelled with a linear array using *modulo* for shifting back after one round.

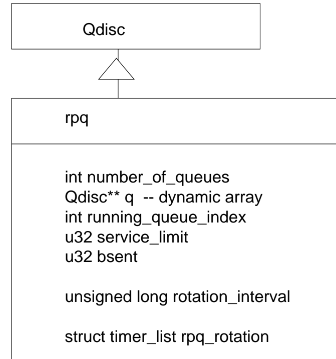

Figure 4.4: RPQ

FIFO queues are stored in a dynamic array `q` of the length `number_of_queues` decremented by one (NRQ-1). `Running_queue_index` (RQI) contains the index of the current 0th queue. At the beginning the RQI is 0, i.e. the current queue is the 0the queue, the queue with priority 1 is the queue with the index 1 etc. After the first rotation, RQI is incremented to 1. Now, the priority 1 queue is the queue at the index 2, priority queue 2 at index 3 and priority 3 at index 0. We should note that queues have a static index in the array, and a dynamic priority, dependent on the running queue. So computing the index i from the priority p is straight forward:

```
i = (p+RQI)%NRQ
```

When rotating we also use modulo in order to shift back to index 0.

```
RQI = (++RQI)%NRQ
```

In both cases, we do not actually use C-modulo (`%`), because it is very slow and inefficient. We should keep in mind that modulo is called at least twice for every skbuff given to the qdisc. Some simple measurements in user-space have shown that using the following macro executes almost twice as fast as a normal modulo:

```
/* faster-modulo (forall 0 <= a < 2b-1) */
#define MOD(a,b) ((a<b) ? a : (a-b))
```

When called, `enqueue` first determines the priority of the skbuff stored in the `priority` field of the skbuff. This field has to be filled with some instance in the protocol stack before the skbuff reaches the queue. In our case, the RSVP filter fills this field. Now the qdisc is accessed by the by the formula described above, and `enqueue` is invoked on the FIFO-qdisc. If the FIFO queue

overflows, the packet is dropped.

Calling `dequeue` starts looking for skbuffs at the current queue. Before any packet is returned, the `service_limit` variable is compared to the bytes sent `bsent`. If the service limit has been overrun, no further packets are returned. Otherwise, one skbuff is returned, and the `len` field of the skbuff is added to the variable `bsent` [3] . There is one important drawback with this mechanism. Let us assume that there is one skbuff in the 1st queue not being sent because the service limit has been overdrawn. Dequeue returns NULL on behalf of the other qdiscs lower in the scheduler waiting to be serviced. If for some time no further packets arrive, the `dequeue` function is not going to be called and the packet is going to be dropped when crossing the current-queue border, although it could have been sent. For this purpose, the backlog of the RPQ qdisc has to updated regularly and the function `qdisc_restart` has to be called to push the skbuffs out of the queue.
Strictly speaking, this problem is not always apparent. At first, after sending one packet, the network adapter raises a *transmission completed* interrupt, causing the net bottom half being run, and calling `dequeue` one extra time. The second point is that the so called watchdog-timer is called every 5 seconds, invoking `qdisc_restart` and taking care of the packets being stuck in one of the queues.

Shortly after the initialization of the RPQ qdisc, the timer is started with `add_timer` in rotation time. Each time the timer goes off, function `do_rpq_rotatation` is invoked. This function fulfills following housekeeping duties:

- Discards all the remaining skbuff in the current (0th) queue.
- Resets the `bsent` variable.
- Rotates the queues by incrementing the RQI.
- If the backlog is larger than 0, start `qdisc_restart`, in order to empty the queues.
- Reschedules itself in `rotation time`.

Rtnetlink Messages Options passed to the RPQ determine the number of queues, rotation time in and the service limit per rotation interval in bytes. Rotation time should not be larger then 100 ms, yet not smaller then 10 ms. The structure containing the fields is:

```
struct rpq_options
{
  __u8 number_of_queues;
  unsigned int rotation_time_in_ms;
  __u32 blimit;
};
```

The second rt-attribute is the length of the bfifo queueus inside the RPQ. All the options can be modified at runtime by a change command except the number of queues, which can be specified only once.

[3] At this point, `len` contains the length of the IP packet without data link layer header, e.g. without ethernet header.

Inform.Element Name	Type	U→K	Mandatory	K→U	Assertion
RPQ_OPTIONS	struct rpq_options	•	•	•	sane options
RPQ_SIZE_OF_BFIFO_QUEUES	int	•		•	

Table 4.3: Rt-attributes for message type RTM_NEWQDISC

4.4.3 Timers and Jiffies

The essence of time in the kernel is the global variable `jiffies` counting ticks since the computer has been turned on. On an Intel PC, one tick is generated every 10 ms, making 1 sec approximately 100 jiffies. However, a tick is a software abstraction caused by some hardware clock of much higher frequency.

As previously described, the rotation is implemented by using timers. Kernel timers have a clear and simple interface. The structure `timer_list` presents a timer abstraction. The fields of the structure are: the pointer function `function` is a function being invoked when the timer expires, the parameter passed to the function is stored in the field `data`, the variable `expires` contains the time value in jiffies specifying the earliest time the timer goes off. After the structure has been initialized `add_timer` is used to insert the timer into a global list of timers. `del_timer` can be used for deleting the timer before it goes off. The timer is invoked from the Timer bottom half (described later in this chapter), which is called 100 times a second (with HZ=100). Before executed, the timer is removed from the list. The RPQ timer inserts itself in the timer list at the end of the invoked function, achieving periodic execution.

The finest resolution one can get from timers is $10ms$. This is achieved by specifying the expiration time to actual jiffies incremented by one. The timer interrupt is raised every 10 ms invoking our RPQ rotation in the following Timer bottom half. If we need a finer resolution, the global variable HZ has to be set to more then 100, which is a default value. Incrementing HZ to 1000 gives us a resolution of 1 ms, yet having two negative consequences. First, the computer becomes slower, due to the overhead of interrupt handling and invocation of Timer BH and schedule (see section 4.7.1 and [42]). Second, the jiffies are stored in 32bit field and overflow after one and a third year with the default value of 100, however using 10000 causes jiffies to overflow in five days [42]. Jiffy-overflow can lead to unpredictable results, such as a kernel crash.

4.5 SDRSVP-Filter

After covering the atmdprio as classful qdisc and RPQ which is to be grafted into the class responsible for Guaranteed Load traffic, we now describe the *Simple DIANA RSVP Filter* (SDRSVP), which is used for differentiating and separating RSVP flows. The Linux traffic control implementation includes an RSVP filter, but since we had to adapt the filter for the RPQ, which itself distinguishes priorities, and the author did not understand the code at its full extent and since no documentation was available, a new RSVP filter has been implemented.
Following **requirements** are raised from the architecture proposed and the structure of the previously described traffic control elements:

- The filter should be able to differentiate packets according to the source address, source port, destination address, destination port and L4 protocol. The L3 protocol used is *always* IP. The only mandatory values given to the filter are the L4 protocol and the destination address.

- The filter should be able to handle *a large amount* of flows [4] , i.e. some more efficient data structure should be used then an ordinary linked list.
- The filter should fit into the atmdprio qdisc and should be able to fill the `priority` field of the skbuff, thus preparing it for the RPQ qdisc afterwards.

SDRSVP Filter is implemented as a two level filter, the main filter contains a hash-table where the references to subfilters are stored. The hash table has the length 256. The position in the hash table is computed from the destination IP-address. This storage type is more efficient then a plain linked list. The OO structure of those two elements is shown in figure 4.5

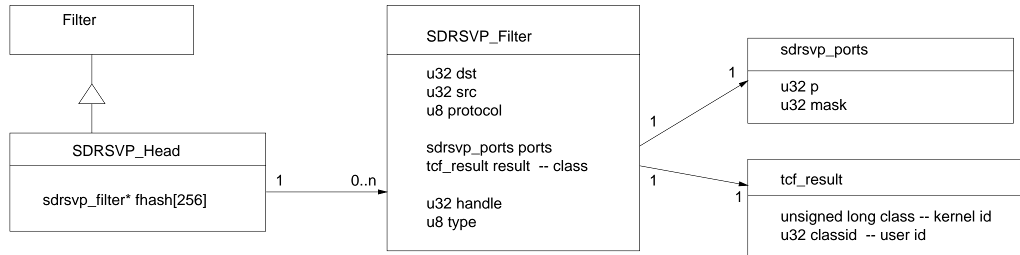

Figure 4.5: SDRSVP Filter

Filter Types Since not all the fields of a filters are mandatory, namely only the L4 protocol and the destination address, we face the following problem. Assuming we have two filters, one differentiating UDP traffic with the destination *a* and the second differentiating UDP traffic with the destination *a* and destination port 6006. When a UDP packet destinated to *a/6006* arrives at a filter, we have an unspecified case which filter should match.
In the first step, different types of filters are specified:

```
enum
{
  FT_DST_PORT_SRC_PORT,
  FT_DST_PORT_SRC,
  FT_DST_SRC_PORT,
  FT_DST_PORT,
  FT_DST,
};
```

The first type is a fully specified filter and the last one contains only the destination address. The second step is the definition of a semantic order relationship between the types. The first type is the most specific type and the last one the most general type. When classifying packets, the more specific filters should tested for matching first, followed by more general filters. The *larger then* relation is mapped into a higher position in the hash-list of filters being:

`FT_DST_PORT_SRC_PORT` $>$ `FT_DST_PORT_SRC` $\geq$ `FT_DST_SRC_PORT` $>$ `FT_DST_PORT` $>$ `FT_DST`

The $\geq$ relation stands for unspecified. An example of a runtime structure is shown in Figure 4.6.

[4] No accurate quantity specification here. An average number depends on the size of the LIS, structure of the IP subnets, application and user characteristics.

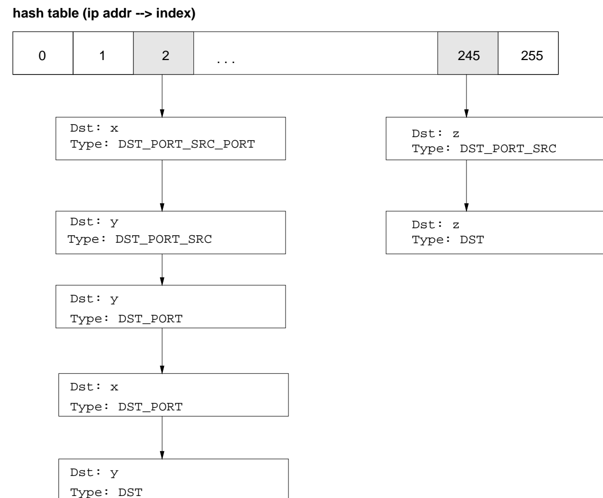

Figure 4.6: SDRSVP Subfilter types and positions in the hash table

Hash Table A hash table is used for efficient access to subfilters according to the destination address. With other words, when an skbuff arrives, the hash-position is computed from the destination address, and only the filters stored in the list at the computed index are checked if they match. Since we use a 256 (`0xff`) hash for at least theoretically ($2^{32} - 1$) IP addresses, there are always collisions. The simplest hash function transforming an address into an integer between 0 and 255 and producing the most uniform mapping is certainly modulo. As noted before, modulo has a large disadvantage of being slow. We should keep in mind that modulo is called at least once for every skbuff enqueued.

There is one simple trick, how to avoid modulo, use fast bit-shifting operations and achieve similar results [5] . Having a 32 bit field, we XOR the upper 16 bit with the lower, filling the upper part with 0s. Now having only a 16bit filled with data, we XOR the upper 8 bit with the lower 8 bit filling the upper part with 0s. The remaining data in the lower 8 bit is our hash value between 0 and 255. Here the hash function:

```
static __inline__ unsigned int hash_dst(u32* dst) {
  unsigned int h = *dst;
  h ^= h>>16;
  h ^= h>>8;
  return (h&0xFF);
}
```

Ports structure Resulting from the filter type, we distinguish between following four cases concerning ports: either both ports are specified, or only one of them – source or destination port, or none of ports are specified. Checking for all the cases when classifying an skbuff results in a relatively large *if then* statement. Although *if then* efficiency is far from being as poor as modulo, some care should be taken concerning code optimization. Here another simple trick of using bit-operations when comparing ports [6] .
The structure sdrsvp_ports contains the source port in the higher 16 bit of the field `p` and the

[5] Seen in code from A. Kuznetsov.
[6] Seen in code from A. Kuznetsov.

destination port in lower 16 bit of the same field. The field `mask` defines what ports are specified. If both ports should be checked for it contains 1s being `0xffffffff`, if none should be checked for it contains 0s. If only the source port is supposed to be checked for, it is `0xffff0000`, having only the higher 16 bit set. If only the destination port is of our interest, the mask contains `0x0000ffff`.

```
struct sdrsvp_ports
{
  __u32 p;
  __u32 mask;
};
```

So assuming that the ports are placed on the beginning of a transport layer header (`tlh`), following code compares the ports of the filter `f` concerning the mask:

```
!( ((*(u32*)tlh) & f->ports.mask) ^ f->ports.p)
```

Using AND masks only the ports we are interested in, and using XOR returns 0 if they matched. The Negation transforms the result in positive logics.

Classification Invoking `classify` can be broken down into several steps:

1. We need the IP header for destination address, protocol and eventually source address. We can save a pointer to the IP header of the skbuff directly:

   ```
   struct iphdr* iph = skb->nh.iph;
   ```

2. If the skbuff does´t originate from the local host, it is only being forwarded by the IP layer, so we cannot access skbuff fields `h.th` for TCP header or `h.uh` directly. The transport layer header is stored in the memory right after the IP header. Since we are only interested in source and destination ports, and this is always stored in the first 32 bit word of both TCP and UDP headers, we can safely cast the beginning of a transport layer header into a `u32`. The IP header length is stored in the `ihl` field, in 32-bit words. Before using pointer arithmetic, we should cast the pointer into `char` (`u8`) and thereafter move it up the memory segment. Multiplication with 4 ($<< 2$) is needed since the ihl field counts the length in 32 bit words and our pointer is only 8 bit long.

   ```
   u8* tlh = ((u8*)iph) + (iph->ihl<<2);
   ```

3. The hash position is computed from the destination address. The subfilters beneath the hash-position are compared to the skbuff-data stopping when the first one matches. First, the `priority` field is overwritten with the value `prio`, preparing the skbuff for RPQ. Second, `tcf_result` is returned carrying both user-id and kernel-id of a class.

Rtnetlink Messages The only rt-attribute supported by the SDRSVP is TCA_OPTIONS. It carries the following structure containing necessary data for the filter.

```
struct sdrsvp_options
{
  __u8 protocol; /* L4 */
  __u32 dst;
  __u32 src;
  struct sdrsvp_ports ports;
  __u8 prio; /* for RPQ */
  __u32 classid;
};
```

When adding a new subfilter, generating handles must be carried out by the kernel. The `tcm_handle` echoes the generated handle back to user-space. The `tcm_info` field consists of the priority of the main filter in the higher 16 bit, and the L3 protocol (IP) in the lower 16 bit.

Message-type	Field	Mandatory	U→ K	ECHO (K→U)
RTM_NEWTFILTER	`tcm_handle`	•	0	generated filter-id
RTM_NEWTFILTER	`tcm_parent`	•	parent qdisc-handle	parent qdisc-handle
RTM_NEWTFILTER	`tcm_info`	•	HI(prio) LO(L3 protocol)	

Table 4.4: TC_MESSAGE Part of the Rtnetlink message with sdrsvp

4.6 Parking Queue

RSVP takes the role of the signaling protocol in the Integrated Services network. Yet, RSVP control messages are passed through the best-effort service class and can occasionally be dropped in the congestion time. After the message has been lost, the RSVP Daemon retransmits a copy after approximately 30 seconds, which is a fairly long period. The proposed solution for avoiding the CLIP channel is a dedicated VC between each two hosts carrying *only* RSVP traffic.

Apparently, we do not want to use a large number of statically established PVCs. Rather, SVCs should be used. There is one problem with dynamic connection establishment; we do not know in advance when the SVC should be established, only when the RSVP message arrives the SVC should be set up by signaling and the packet forwarded through the new VC. There are two conceivable solutions:

- RSVP Daemon receives a message and notifies the Atmtcd of its arrival in a blocking manner. Atmtcd can now establish a connection, add a new class to atmdprio-qdisc, add a new filter describing the packet (RSVP protocol, destination address) and return. Now, the RSVP proceeds the path or reservation or any other message to the next hop, which is fetched in the kernel by the classifier and pushed through the VC. This solution is based on the application (user) side.
- Atmdprio-qdisc notices that a packet of RSVP protocol type is about to leave by CLIP. It can either signal a VC by itself and proceed the message, or notify Atmtcd that a VC should be established. Upon success, the RSVP control message is passed through the VC. The first solution described here is a pure kernel-based mechanism, the second one a hybrid mechanism involving both the kernel and Atmtcd.

After numerating the possibilities, we evaluate the mechanisms:

1. User-side mechanism would require some patches in the RSVP-Daemon, ATM-Adaptation module and Atmtcd. It is to note, that although the reception of a Reservation message concerning the *previous* router reachable by the ATM device is passed to Atmtcd, we need to accept the Reservations concerning the *next-hop* the Reservation message takes. Furtermore, Path messages are not forwarded to Atmtcd at all.
2. Kernel-mechanism is more complicated then it seems. First, a VC has to established from the kernel side, using the internal ATM signaling protocol to the Signaling Daemon. Second, open VCs should be managed in a CLIP similar way.

3. Hybrid-mechanism requires the implementation of a so called *Parking-Queue* where the RSVP messages are being parked, waiting for a VC being set up for them. Some patching of the Atmtcd executable is also necessary. On the other side, a lot of signaling code from the Atmtcd can be reused. This solution appeared easier to implement then other two and is being described in the following section.

4.6.1 Architecture

The queuing discipline managing RSVP messages is called a *parking qdisc*. All the RSVP traffic is passed to the Parking qdisc through some filter matching only RSVP traffic type, independent of the source or destination address. We call this filter RSVP traffic filter, which should not be mixed up with the standard RSVP-filter or SDRSVP filter.

This filter is placed low in the list of filters, yet before the default class is used. For example, when a Path message arrives from the IP output, SDRSVP filters are skipped, since they do not match, and the RSVP traffic filter forwards it to the Parking qdisc. Parking qdisc now notifies the Atmtcd with a Rtnetlink message that an RSVP message has arrived. Atmtcd can now establish a new SVC and an RSVP filter matching the destination of a parked skbuff. In the meantime, the skbuff is being requeued, and thus passed through the list of filters. A timer-task requeues the packets over and over again, making them cycle through the filters and fall through to the parking qdisc again. There are two ways a packet can leave. Firts, the Atmtcd has added a class to atmdprio, and an SDRSVP filter matching RSVP protocol with the destination pointing to this class. Since the filters are added logically higher then the RSVP filter, the requeued packet is being matched and leaves the queue over SVC. Second, if the Atmtcd fails for some reason, the parking qdisc notices the timeout-event and dequeues the skbuff, which leaves the host over the default CLIP VC.

Using timers for handling skbuff rotation until either a SVC and a filter is set, or a timeout arrives, is essential for two reasons. Since the skbuffs can be enqueued and dequeued in the control flow of the IP send procedure, which can be executed on behalf of some `send` system call originating from a user process, calling Rtnetlink would block the user space application for a long period of time. The second reason for timers is the recursion. When the packets cycle, the method `requeue` of the root-qdisc is called from the `requeue` of the parking qdisc. If the RSVP filter matches again, the control flow is passed recursively to `requeue` again, blocking the whole system. Asynchronous coupling through timers is essential for *breaking* the control flow.

Another feature of the qdisc is the function is the so called *notify and go* mode. In this operational mode, Atmtcd is notified of the arrival of the RSVP message, but the message is dequeued straight away, not being delayed by the Parking qdisc. With other words, we notify the arrival of the packet and pass it through the CLIP VC, yet leave Atmtcd a chance of establishing a VC for the *next* RSVP message.

Private data of the parking qdisc are following variables: skbuff list `qenq` – list of skbuffs being enqueued, `qreq` – list of skbuffs being requeued, `qdeq` – list of skbuffs leaving the queue with the next `dequeue`, `plimit` – packet limit for the requeue-queue and `delay` – maximum amount of time elapsed before the packets are transmitted.
The significant methods are described bellow:

- `enqueue` places the received skbuff into the `qenq` and schedules the timer-task unless it has been scheduled for execution already. Forwarded packets contain the arrival time stored in

the `stamp` field of the skbuff, filled by `netif_rx`. Local packets have no arrival timestamp set, so we set it to current time. The timestamp is used for determining the time spent is the parking queue. It is to note that the timestamp is not described in jiffies, but the `struct timeval` ([48]) is used presenting time in the seconds, microseconds format. Accessing current time in the user space is achieved by invoking the `gettimeofday` function. In the kernel, the global variable `xtime` of type `struct timeval` is used.

- Timer-task notifies the listener (Atmtcd) by sending a Rtnetlink message, dequeues one skbuff from the `enq` and calls `requeue` of the root - qdisc `struct Qdisc::dev::qdisc`. This is being repeated until there are no skbuffs left in the `qenq`.
  At the second step, the timer-task computes the difference between the skbuff´s timestamp (`stamp`) and the current time of all the skbuffs in the `qreq` list. If the time-difference exceeds the maximum delay, the skbuff is moved into the `qdeq`. If the time spent at the node is smaller then the maximum delay, the skbuff is requeued again.
  Finally `qdisc_restart` is called in case there are some packets waiting to be dequeued in `qdeq`.
- `requeue` places the skbuff in `qreq` and schedules the timer if it has not been set already.
- `dequeue` returns one skbuff from the `qdeq`. If there are no skbuffs in the outgoing queue, the maximum limit of the packets in the requeue-queue is checked against the actual requeue-queue length. If the available buffer has been overflown, one skbuff from `qreq` is returned.

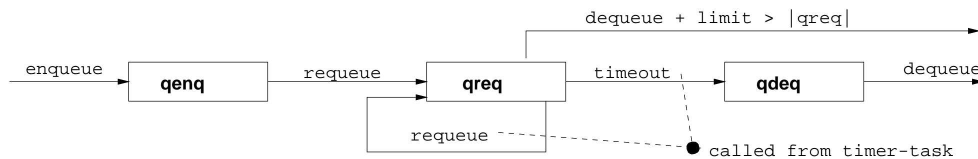

Figure 4.7: Inner queues in the Parking qdisc

Integration The described architecture requires following three new traffic control elements:

1. A classless Parking qdisc which is grafted into a parking-class, and manages timeouts, requeues packets and notifies the Atmtcd.
2. A special parking class, which differs from an ordinary atmdprio class, since it has no VC in the background, and uses CLIP after some skbuff has been timed out.
3. An RSVP traffic filter. Unfortunately, SDRSVP filter cannot be used for this purpose, since protocol *and* the destination addresses are mandatory. Again. SDRSVP uses a hash table, mapping the destination address into some list, so this would require restructuring of SDRSVP.

The easiest way of implementing the proposed architecture is making the three elements a fixed part of the **atmdprio qdisc** (Fig. 4.9).

Rtnetlink Messages Until now, the RTM_NEWQSDISC message did not support any Rt-attributes (except the mandatory TCA_KIND). The added information element is called ATM_DPRIO_PARKING and carries the following structure as payload.

```
struct atmdprio_parking_options
{
  pid_t notify_pid;
  u32 plimit;
  u32 max_delay; /* in seconds */
  u8 notify_and_go_mode;
}
```

PID specifies the user-space process interested in receiving notifications and being able to establish a VC. `plimit` is the buffer length of a queue. Maximum delay imposed on a RSVP message is in seconds. When appending an atmdprio qdisc, ATM_DPRIO_PARKING is optional. If missing, the default structure described before, without RSVP filtering is used.

Atmtcd needs only one significant information from the kernel: the IP address of the next hop. Atmtcd can resolve the address by contacting the ATM-ARP Daemon, signal a VC for RSVP traffic, add an atmdprio-class and an SDRSVP filter. The exact format of the Rtnetlink message passed up to the application layer carrying this information was not clear at the time of writing.

4.7 Avoiding Race Conditions

Race Condotins occur when two or more parallel or pseudo-parallel tasks share a variable, or more generally a memory segment and at least one of the tasks modifies the memory segment. When the modification of a variable is not atomic, the reading task can always be scheduled away while reading, and the writing task may change the variable in the meantime, feeding the reading task with an inconsistent value. Since the traffic control module is being called from different functions, belonging to a Rtnetlink control flow, or control flow originating from a user space, or even from the Network bottom half and since we timers are used within the RPQ and Parking queue, clear understanding of the matter is needed, in order to avoid mysterious crashes. Two further reasons served as a motivation for examining possible race conditions in our code: first, such bugs are hard to reproduce and second, kernel code is not very *debuggable*.

4.7.1 Bottom Half Scheduling

We will start this section with some clarification of the Linux scheduling. More details can be found in [43][45]. The runtime structure representing a running user-space process is a `task_struct`. The task-execution can either occur in user space or can be passed to the kernel when some system call has been invoked. The event of exchanging the running task in the user-space with one in the kernel is called a *context switch*. Only systems calls are of our interest, since normal processes do not modify any structures in the kernel. The invocation of any function being executed in the kernel may originate from one of the following control flows:

- *System calls* executed in behalf of user space processes.
- *Interrupt handlers* executed automatically after some hardware device raised an interrupt.
- *Bottom halves* executed after the fast interrupt handlers have finished their work.

The separation of interrupt handlers into the fast ones (upper halves) and slow ones (bottom halves) decreases the response time latency, since all interrupts are blocked while executing the upper half, and new hardware interrupts can arrive while executing the bottom half. Here is how this maps on an ethernet card receiving a packet: when the packet has been received, the ethernet adapter raises an interrupt starting the device driver´s interrupt handler. After reading some registers on the adapter, the handler learns that the interrupt cause was packet reception (and not completed transmission). The packet is copied from the hardware buffer in a newly allocated skbuff, `netif_rx` is called and the NET Bottom Half is marked for execution. This is the point where the upper half ends, and the assembler code in `ret_from_sys_call` is executed. Return-from-system-call routine is called after every system call *and* upon completition of the fast interrupt handler. The `ret_from_sys_call` call covers two main functional aspects. At first, all the *marked* bottom halves are run. This is the point where the NET BH gets executed pushing the skbuff from the packet input queue to the higher layers. Since bottom halves are *anonymous*, i.e. there is no knowledge of which interrupt caused its execution, the NET BH also executes the code in case the *transmission completed* interrupt had been raised. Transmission complete means idle line calling dequeue on the qdisc and passing the packet to `dev_hard_xmit`. At the time when the BHs are being run, some newly arrived hardware interrupt may stop the execution in behalf of the fast interrupt handler. In this case, the fast handler is executed, and after its completition, the interrupted bottom half finishes its job, followed by all the remaining or newly marked BHs. The second aspect of `ret_from_sys_call` is the user-process scheduling. After the *current* task has used its time-slice, some other process is scheduled (see [43] for details).

Except for the interrupts coming occasionally from the network adapter, there is one significant interrupt coming from the hardware: the clock tick. This hardware interrupt is masked through a software interrupt and raised (only) 100 times a second [7]. The timer interrupt handler increases the global variable `jiffies` and marks the TIMER_BH. All the tasks stored for execution by `add_timer` are run for the timer BH.
Before discussing the race conditions in our code, some relevant points are repeated:

- User space processes in the user-space context or kernel context (system call) can be interrupted by a hardware interrupt and therefore with any bottom halves
- Bottom halves run sequentially in respect to themselves and another bottom halves, i.e. bottom halves are not being scheduled away by other BHs or another BH task of the same kind. However, BHs can be interrupted by a hardware interrupt.

Figure 4.8 shows an example of the scheduling mechanism described above. The process executing `ping` is about to send one packet, and calls `send`. Send is executed as a system call, and causes a context switch. Send passes an skbuff down to the hardware device, which sends a packet and raises an interrupt. Interrupt is hadled in the interrupt routine which marks the Net BH for execution. Straight after the interrupt hander has finished its work, the `ret_from_sys_call` routine checks for marked interrupts and calls `schedule`; giving a scheduler a chance to switch to a new user-process. Marked NET BH is executed first, and the system call `send` is being ran to completition. When the `send` system call finishes, `ret_from_sys_call` is called again. This time, there are no bottom halves marked, but the scheduler decides that `ping` has used its time-slice and sets `emacs` as a running process. Shortly after being scheduled, a timer interrupt occurs and the timer interrupt routine is called, followed by a marked TIMER BH. TIMER BH does not run to completition, since it is interrupted by a network device receiving a packet. The network interrupt handler of a device is executed, marking the Net BH. When returning, the TIMER BH is run to an end, followed by

[7] 100 Hz on Intel PCs, 1024 Hz on Alphas, yet depending on the HZ value in the kernel, see [42].

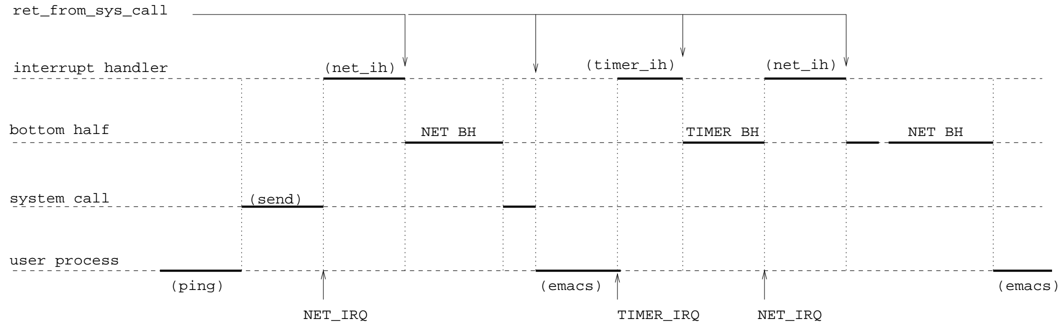

Figure 4.8: Scheduling Example

a newly marked Net BH. The scheduler called after the network interrupt handler decides that `emacs` should continue running.

4.7.2 Race Conditions in Traffic Control

In order to determine possible race conditions in the traffic control module, we should identify the callers and their context of execution. At first we take a look at the *net-interface* callers of a root qdisc:

1. IP output (Figure 3.3) calls `enqueue, dequeue` and eventually `requeue`. This block of code is protected with `start_bh_atomic` and `end_bh_atomic` guaranteeing the sequential execution in respect to any bottom half.

2. NET BH calls `dequeue` when the outgoing line has become idle, i.e. `device::tbusy` not 1, transmitting further skbuffs waiting in queues of traffic control module. This call is not being interwined with the calls in the point 1, since the `dequeue` originates from the BH - context.

3. Periodic watchdog timer `dev_do_watchdog` is called every 5 seconds and is used for recovering from hardware or software errors. The main error kind are packets queued in the traffic control, but not being dequeued directly from the code described in point 1 or by the netwok bottom half described in the point 2. Since all timers are executed from the TIMER BH, this execution does not interwine with points 1 and 2.

4. We use timers in the RPQ queueing discipline and in the Parking queue which in turn use the network-interface of the device´s qdisc. As noted in point 3, timer - tasks are executed from the TIMER BH and are accordingly not executed in parallel with code from points 1,2 and 3.

The *user interface* of the traffic control module is used exclusively from the Rtnetlink. The user-space processes use `send` or any similar command passing the Rtnetlink message, which is being transformed into a system call, and handled by netlink and Rtnetlink. As we have described before, these system calls can always be interrupted by the top and bottom half handlers. As a worst case, it can happen that class is being deleted in the system call, when an network interrupt arrives. After calling dequeue from the NET BH, the inconsistent state of some class list may lead to a memory fault.

In the first step, we identify the access art of each method of qdiscs, classful qdiscs and filters, specifically for RPQ, atmdprio and sdrsvp.

Access Kind	**TC Element**	**Method**
Write (safe-time)	qdisc	`init, graft`
	tfilter	`init`
Write	qdisc	`change, destroy, class_change, delete`
	tfilter	`change, delete, destroy`
Read	qdisc	`reset, dump, put, get, leaf, walk, dump_class`
	tfilter	`put, get, walk, dump`

Table 4.5: Access kinds of traffic control methods

Write methods executed at safe-time are constructors, which guarantee that no other methods uses an element, before it has been completely initialized. `graft` uses the `xchg` call [43] for exchanging two pointers in one processor instruction step (atomically).
Reading methods [8] do not modify local variables, so no extra locking mechanism is needed.
Writer methods may interwine with the network-interface described above. All the net-interface methods are either called from the (real) bottom half context, or wrapped through `start_bh_atomic` and `end_bh_atomic` simulating the BH context. By using the two `bh_atomic` calls in every writer method, we can avoid simultaneous access of user-interface and the net-interface methods, thus avoiding race conditions.

4.8 ATMTCD

We have considered the role and structure of the ATM Traffic Control Daemon in section 2.6.2 and explained the scope of the proposed extension. After describing the requirements, design and implementation of individual traffic control elements, we make a step back and take a look at the whole picture.

4.8.1 Kernel TC Architecture

The basis of the architecture is the atmdprio queueing discipline on the ATM interface. Best effort traffic is being scheduled by the default class, which uses the CLIP as an output device. As noted before, there are multiple VCs hidden behind the CLIP device.
Guaranteed service traffic to some next-hop router in the LIS is being scheduled by the RPQ queue including P priority levels [30]. Controlled Load service to some next-hop router is scheduled by the bfifo queue inside a class having lower priority then the GS-classes. The Parking queue takes care of the RSVP control traffic, and notifies Atmtcd when a RSVP message has been forwarded to it. Figure 4.9 shows the architecture.

4.8.2 ATMTCD-Interface to TC

The **tc_interface** module within the Atmtcd has been designed as a simple user interface to the relatively complex Rtnetlink-communication which uses the *libnetlink* library already mentioned in the description of the TC command line utility. The position of the module is described in Figure 2.7. The main user of the interface is the *Flow-to-VC* Module.

[8] `const` in C++.

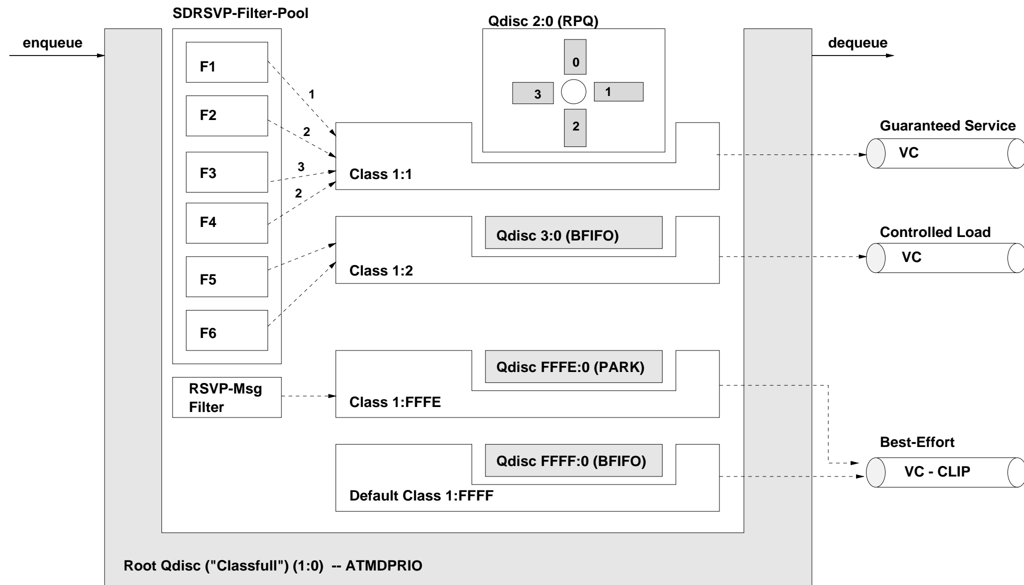

Figure 4.9: Runtime structure of the atmdprio qdisc

Preconditions The module makes some assumptions about the environment, which is normally set up by diverse shell-scripts before starting the RSVP - Daemon and Atmtcd. One of those assumptions is that an ATM interface (e.g. *atm0*) contains the atmdprio qdisc as a root queuing discipline. The handle of the root qdisc is 1:0.

Function Description All functions described below have the same architecture. Function parameters are transformed into a Rtnetlink-message, which is sent to the kernel. If no handle was specified, it is being generated by the kernel and echoed back to the sending application. A `select` call is used for receiving the acknowledgement from Netlink, the echoed Rtnetlink message and for handling timeout. All functions return 1 on success and -1 in case some error occurred. The error is being logged to a standard Atmtcd log file.

- `init_kernel_tc(), destroy_kernel_tc()` represent the constructor and destructor of the module respectively.
- `add_class` adds a new atmdprio class, the class handle may be specified, otherwise it is generated at the kernel level and returned back to the caller. The handle of the automatically grafted bfifo qdisc is returned as well, and it can not be specified in advance. PID and the file descriptor passed as options identify the VC, priority determines the scheduler priority.
- `del_class` deletes the class specified by the class handle, destroying the grafted qdisc as well. If the reference count of a class is positive, i.e. there are filters still pointing to a class, an error is returned.
- `add_filter` adds a new sdrsvp filter that allows to forward traffic to an existing class. The handle of a filter cannot be specified, yet the generated handle is returned back to the caller.
- `del_filter` deletes a sdrsvp filter specified by a handle, a parent qdisc and the priority of the main filter.

- `graft_rpq_into_class` grafts an RPQ qdisc into a specified class.
- `change_rpq` changes the attributes of the RPQ qdisc previously grafted into a GS class.
- `change_fifo` changes the attributes of the bfifo qdisc grafted automatically into a CL class.

4.8.3 Runtime Scenario

The following runtime scenario describes the relationship between the interfaces and protocols described up to this point. The example network topology consists of a *receiver host* connected through Ethernet with $router_1$ and the *sender host* connected through Ethernet with $router_2$. Both routers are interconnected through ATM, being a part of a Logical IP Subnet and using CLIP with RFC1483 encapsulation.

The receiver host has two relevant processes running, the RSVP-Daemon and the receiver application. The sender host has an RSVP-Daemon and the sender application running. Applications communicate with the RSVPD over RAPI and Unix domain sockets.
Routers have two relevant processes running, the RSVP Daemon patched for ATM (ATM Adaptation Module) and Atmtcd which communicate through Unix domain sockets, using the LLDAL interface-similar interface. These two routers use the atmdprio qdisc and our architecture on the ATM interface.
Note that this is an informative description, for more details on Atmtcd, one can refer to [30][31] and read RSVPD code.

Guaranteed Service Reservation The sender application uses RAPI and calls `rapi_sender`, which initiates a Path message sent to the receiver host. The Path message is forwarded over $Router_2$ and $Router_1$ and read by RSVPD at each host. At $Router_2$, it is eventually passed into a Parking qdisc, notifying Atmtcd that a dedicated VC towards $Router_1$ is needed. Arriving at its destination, a Path event (upcall) notifies the receiver application, which now calls Resv (RAPI) sending the Reservation message upstream. $Router_1$ reads the reservation and reserves a portion of ethernet towards receiver host using the CBQ qdisc. The Resv message is now passed on to $Router_2$, where it eventually lands in a Parking queue, notifying Atmtcd that an RSVP-VC is needed.
After RSVPD at $Router_2$ received the message, ATM adaptation module picks it up since the next reservation hop is reachable via ATM device and contacts Atmtcd with `AddFlowSpec`.
At the point annotated with (1), Atmtcd computes admission control parameters, checks for resource availability, initiates signaling of a GS VC towards $Router_1$ (CBR traffic class), and calls `add_class` of the tc_interface module. After sending the Rtnetlink message, the kernel adds a new atmdprio class associated to a VC newly established. In the second step, an RPQ qdisc is grafted into a class using `graft_rpq_into_class`. The second message received from the RSVPD is `AddFilter`. Atmtcd calls `add_filter` which sets up a new sdrsvp filter in the atmdprio qdisc forwarding traffic to a class added in the previous step. At this point, the Reservation message is forwarded to the sender, which can now start sending packets. Data packets sent from the sender-application use the CBQ reservation over ethernet on the link to the $Router_2$. Here, they are classified by the sdrsvp filter and passed over a GS VC to $Router_1$. On the link to their final destination, CBQ scheduler is used for guaranteeing QoS to the receiver host (and application). Figure 4.10 show the message sequence chart of the described mechanism. It is to note that the next hop reservation node of the $Router_2$ is actually $Router_1$ and the next hop path message hop of the $Router_1$ actually $Router_2$.

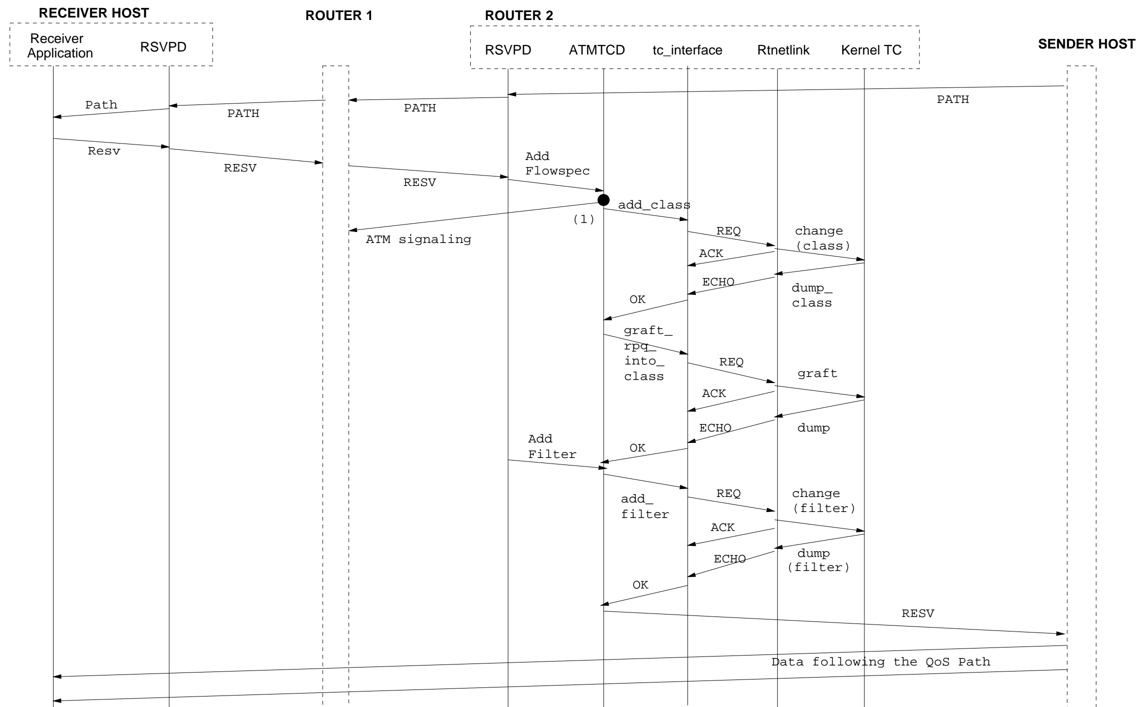

Figure 4.10: MSC for scenario 1

2nd Guaranteed Service Reservation If a second reservation for Guaranteed Service Class for the link $\text{Router}_2 \rightarrow \text{Router}_1$ arrives and Admission Control decides that another reservation can be handled with the existing QoS of the VC, only `add_filter` has to be used for redirecting flow traffic to the RPQ qdisc and defining the appropriate RPQ-priority.

3rd Guaranteed Service Reservation Assumeing a third reservation for Guaranteed Service for the link $\text{Router}_2 \rightarrow \text{Router}_1$ arrives and Admission Control decides that VC QoS does not suffice for yet another flow. The *VC-Setup and Modification module* renegotiates the QoS of the VC, broadening the bandwith and reducing maximum delay. `change_rpq` changes the parameters of the RPQ qdisc according to the renegotiation result. Finally, `add_filter` adds a new classifier for a new flow.

Controlled Load Reservation A Reservation message for Controlled Load follows the same message path as a GS reservation. The signalled VC uses the CBR traffic class too. Shortly after, `add_class` appends a new class for CL traffic and `change_fifo` adapts the size of an available buffer. `add_filter` classifies the CL traffic to the appended class.

Reservation Teardown Session closing is signaled to Atmtcd through a `DelFlowspec` and `DelFilter` messages from the RSVPD. If more then one flow is using a GS or CL atmdprio class, `del_filter` removes the filter. If the flow that is to be removed is the last one on the link towards Router_1, the VC can manually be torn down, followed by `del_class`. This way, the number of SVCs not being used anymore can be minimized.

Chapter 5

Component-Test and Traffic Generation

This chapter describes the testing of the software developed. Passing the control flow to specific pieces of code in the kernel is not as obvious as in user space. Moreover, some user space applications must be used, which then invokes our functions through a system call.
In the course of this work, we have permanently differentiated between the user-interface and the network interface of the qdisc. This logical split between the interfaces remains present and becomes evident in the test-phase.

Test of the user interface is possible, because the results of methods are predictable and definable. For example, if the interface *eth0* has no root qdisc, and the tc program appends a dprio qdisc to an interface, we expect that the dprio is the root qdisc of *eth0* if everything worked well.

Testing the network-interface requires sending packets, which are passed through `enqueue` and `dequeue`. Testing (or simply running) `requeue` and `drop` is not directly possible. Another problem with testing the network interface is that there are no results which determine whether the action taken actually passes the test. So, instead of testing, we perform *measurements* over the network interface, logging packet loss, average packet delay and available bandwidth over some period of time.

5.1 Testing the User-Space Interface

Two important assumptions had to be met, in order to effectively test the code produced:

- The interface has to clearly defined and accessable.
  The user-space interface to traffic control, namely the `tc` command utility was used to modify the elements in the kernel. Theoretically, using this interface leads to simultaneous testing of the tc-executable, Netlink, Rtnetlink and finally our code. However, providing some kind of direct interface to the parts of the kernel under test (e.q. ioctl) would cause a large overhead and completely discard all the advantages of the Rtnetlink-concept.
- The state of the module under test has to be observable. The black-box transparency can easily be used with the tc command utility examining the runtime state of the module. Tc namely dumps all the elements known to the kernel. In order to test the code in a white box

manner, we should log the actions from the kernel in the Kernel Log Daemon [1] and compare those to expected result.

- The results of the commands effecting the module under test have to *deterministic* and repeatable.

For a more formal introduction and proposal for *serious* testing, refer to [54].
After clarifying the problems concerning successful testing of our code, we can describe one simple way of testing the user-space interface.

1. Reset the traffic control of the interface (delete all qdiscs).
2. Add new qdiscs, classes or filters.
3. Dump the state of the traffic control on the interface using `tc`.
4. Compare the results with the expected results.
5. Delete the changes made and reset the traffic control of the interface.

Point 1 is the assurance that the test *precondition* applies. In the point 2, some action taken to modify the state of the module. Point 3 reads the state of the module, which is then compared to the *postcondition* required. Finally, the module is set to the initial state.

5.1.1 DejaGnu

DejaGnu [53] is a testing framework used to provide a single front end for all tests. The testing environment consists of a DejaGnu framework, written in Expect [2] and test-cases bundled into a test-suite. Test cases themselves are also written in Expect, which enables automatic execution of command line utilities and evaluation of the output normally written to standard output.

Here is a simple test-case, adding a dprio qdisc to an Ethernet interface:

```
catch {exec $tc qdisc del dev eth0 root}   # clean up
spawn bash   # start shell
send "$tc qdisc add dev eth0 root handle 10:0 dprio\n"   # add qdisc
expect $shellprompt   # wait
send "$tc qdisc list dev eth0\n"   # list all qdiscs
expect {
      -re "qdisc dprio 10:"      { pass "basic dprio (1)"}
      timeout                    { fail "(timeout) basic dprio (1)"}
  }
}
catch {exec $tc qdisc del dev eth0 root}   # clean up
```

In the first line, the state of the interface is initialized and the bash shell started. The test-action is a call to the `tc` utility adding the qdisc. Now, the state of the module is read by calling `tc` with `list` parameter. The information passed to standard output is compared to the expected result. Either `pass` or `fail` is called to define the test-result. In the last line, the interface qdisc is removed.

Each test case is written in a separate file. All the test files together make up the test-suite, which van be ran by calling `runtest`. DejaGnu logs all the information and prepares a report telling which test-cases have passed and which not.

[1] For more information on `klogd` see [42].
[2] Expect is an extension of Tcl scripting language [52].

5.2 Traffic Generation

Generating traffic is necessary in order to test the network interface of the qdisc. Traffic can either be generated on the local host, whose qdisc is being tested, or on any other host forwarding packets through the host acting as a router.

There is a variety of traffic generators and meauserment tools, both commericial and freely available [3] . However, none of those seemed to be capable of measuring delay and being extandable at the same time. Some freely available tools do not run under Linux, or do not come with the source code. The publicly available `ttcp` [38] can only generate TCP traffic and can only measure bandwidth. *Stampgram* [4] is a reletively simple prgram designed for BSD, which could have been used. But, it did not compile *out of the box* for Linux and there was no documenetation and no understanding how to integrate other features, like bandwidth measurement.

5.2.1 Requirements

The requirements for the traffic generation and meausrement tool were the following:

- It sholud be able to measure end-to-end delay of the packets.
- It should be able to measure the packet loss.
- The traffic characteristsics should be configurable, adding new traffic models should be enabled.
- It should *eventually* allow support for RSVP signaling.

IP Performance Metrics Working Group [5] has investigated and proposed a framework for meausing IP traffic [55]. Measuring end-to-end delay makes the results fully dependable on the synchronised host-clocks. Of course, these clocks do not have to be accurate, but synchronized to one another. Network Time Protocol (NTP, RFC1305) addresses the problem of synchroning clocks, but concentrates on a long term synchronisation. Another point to mention is that the packets may not be fragmented by the IP layer. Otherwise, the fragmentation and refregmentation is measured as well. Many IP-metrics are based on the *wire-time*, time between the first bit has reached the physical medium and time when the last bit has left the physical medium. Such measurements should be carried out as low in the protocol stack as possible. Berkely Packet Filter (BPF) [38][39] allows accesss to the link layer directly from the user-space. Furthermore, every Ethernet card can function in promiscous mode, allowing some third party in the subnet to detrmine when the IP packet has been transmitted.
However, the use of BPF in our architecture was not considered, because the queueing delay occuring in the L3 is the value we are interested in, and it should be measured.

5.2.2 Architecture

The arcitecture of the measurement and traffic generation tool developed is shown in Fig. 5.1. The client uses an UDP socket to transmit packets to the server. The timestamp and the sequence number are filled in every packet. Here the structure which is passed as a payload in every UDP packet:

[3] See http://www.caida.org/Tools/meastools.html for a survey.
[4] http://addler.nttjoker-unet.ocn.ne.jp/ott-tool.html
[5] http://www.ietf.org/html.charters/ippm-charter.html

```
struct packet {
        unsigned long seq;
        struct timeval ts;
        char buffer[MTU];
};
```

The `char` array `buffer` contains a random byte pattern. Whe the packet arrives, the server reads its local time and computes the difference.
Client implments differnt traffic types, full blast traffic, periodic traffic, and Poissson periodic traffic, to name a few.

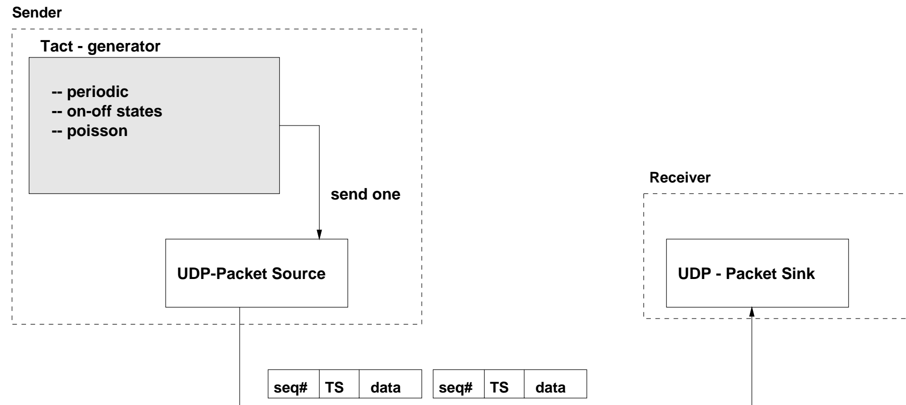

Figure 5.1: Client/Server architecture of traffic generation

Sender

The overall structure of the sender is simple. The `send_one` function prepares a packet by filling a sequence number and a timestamp fetched by `gettimeofday`, and sends it to the receiver. The caller of the send-one function is an actual traffic generator. All the traffic generators use the `ualarm` C library function. This function is similar to `alarm` ([48]) execpt that it works with finer time-resolution in microsecond range. Calling `ualarm` with microseconds schedules the arrival of `SIGALARM` signal in given time. When the signal arrives, the signnal handler for `sigalarm` is called, which subsequently calls `send_one`. After sending a packet, the new alarm can be scheduled. However, Linux as a non real time operating system does not guarantee that the signal actually arrives on hard real time basis, and that none of the signals are masked while handling the upcall routine of the previous one. Yet, for our purposes, an underloaded computer suffices.

Traffic generators export a generic interface to the main sender application consisting of the following methods:

- `read_options` reads the traffic generator specific options and fills them into local variables. For example, the periodic traffic generator stores the time-period for the next packet-transmission.
- `start` overrides the generic signal handler for SIGALARM by using the `signal` system call, giving it a pointer to a function which actually calls `send_one`.

- *Signal handler* calls `send_one` and rechedules the next alarm-time, in order to be called again.

The main program executes a forever loop waiting until the given number of packets has been transmitted. `pause` system call is used in order to avoid busy waiting. Another signal is relevant in our architecrure, namely SIGINT, which arrives when the user has issued a Ctrl-C on the working terminal. SIGINT ends the program gracefully and prints the accumulated information on standard output.

Following traffic generators have been implemented:

- Full Blast traffic sends the packets as fast as the signal arrives. The alarm is scheduled periodicallly every 1 microsecond. Since the transmission lasts longer then a one microsecond, some signals are masked. When the signal handler function returns, another packet is sent straight away.
- Periodic traffic sends a packet and rechedules another alarm after the transmission has been completed.
- Poisson Periodic traffic uses the Poisson distribution to determine the time of the next packet transmission. Using simple periodic sampling has a disadvantage [55] of possible network synchronisation when using multiple sources. Further, a pure periodic behavior is not an efficent model of real Internet traffic.
  We use the following algorithm for genarating the Poisson distribution [55]:
  1. For an average transmission period p, we have $\lambda = 1/p$
  2. Generate uniform distribution $U_1, U_2, ..., U_n$ between 0 and 1.
  3. Compute time interval E_i with $E_i = -log(U_i)/\lambda$
  4. Send a packet and wait E_i long, i.e. use `ualarm` to schedule the next signal. Repeat the last step until all the packets have been sent.
- On-Off traffic consists of two states: the On-state sending packets periodically (t_p), and the Off-state not sending packets at all. The length of the On-state is a Poisson distribution of a period t_{ON}.

Receiver

Receiver executable is a simple server using some specific port and waiting for UDP traffic. The receiver is connectionless, i.e. it does not regard or is aware of the sending application. When a packet arrives, the sequence number and the transmission timestamp are read and the arrival timestamp is fetched. Finally, the server dumps obtained information onto the standard output in the following order: sequence number, transmission time, arrival time and diffrence between the later two. The receiver has intentionally been kept simple, moving further complexity of computing average delay or packet loss to small scripts analying the output.

5.2.3 Usage

As noted before, measuring one way delay is fully dependent of the acurate synchronisation of the sender´s and receiver´s clock. NTP has been designed with long term accuracy in mind, adjusting clocks dependent on the configuration, approximatly once an hour. Measurable delay is described in terms of microseconds, while the clock drift in an hour can sum up to one second and more. Another objective by measurements is the fact that if we use the same line (ineterface,

shared medium or subnet) for synchronising clocks and generating traffic, the accuracy of the synchronisation is likely to be detracted. Therefore, we should make sure that the line connecting the sender and receiver with the time-server is underloaded.
Both the server and the client accept the `-h` or `--help` options and print a short usage message.

1. Terminate the NTP-Daemon (client) on the receiver and on the sender (`kill -9`). Use `ntpdate` to force the synchronisation of local clocks to the time server on both hosts. The follwing shell script will do:

```
TimeServer = ...
Ntpdate = ...
while true
  do
  $Ntpdate $Timeserver
  $Ntpdate $Timeserver
  sleep 1
done
```

2. Start the server on the receiver host, specifyng the port and redirecting the standard output to some file.

```
./server -p 7007 > server.7007.output
```

3. Start the sender. We can specify the local host name (if multihomed), local port, destination host name, destination port (7007 obviously), number of packets to be sent (100), size of the payload (400 bytes), not counting UDP, IP and ethernet headers and kind of traffic. The kind of traffic used in the following example is peroidic traffic generation every 100 ms.

```
./client -lp 6006 -lh localhost -s 400 -p 7007 -n 100 remotehost periodic -t 0.1

==> Packet size: 442 bytes (eth,ip,udp Header 42 + Payload 400)
==> 44200 bytes (100 packets) transmitted in 10.002057 seconds
==> 0.035 Mbps
```

4. Terminate the server. The file `server.7007.output` contains the logged data from the server. The fields are: sequence number, transmission time meausured at the sender, reception time measured at the receiver, difference between the two timestamps.

```
seq:0 (942772607,112545)(942772607,112185)(0,360)
seq:1 (942772607,117156)(942772607,117018)(0,138)
```

   Obtaining relevant information from the log-file is left to various Tcl scripts. The following script is used for computing the delay relevant information.

```
> ./srv_average_delay.tcl server.7007.output
Avarage Delay: 0.000116 s
Min-Delay: 0.000091 s
Max-Delay: 0.001438 s
```

   Computing packet loss uses sequence numbers and encounters holes between lines. However, since the server has no information how many packets were actually sent, it cannot encounter losses at the end of the transmission.

```
> ./srv_loss.tcl server.7007.output
0 packets lost (out of 100)
```

The following script computes the delay between transmissions at the sender host. This way, one can determine how accurate the `ualarm` function has been.

```
> ./srv_period.tcl server.7007.output
1 0.099011
2 0.099959
3 0.100032
4 0.099952
...
```

One can easily come up with new scripts fitting specific needs. For example, given the payload size, the server output can be used to determine the actual bandwidth arriving at the server. Note that UDP is unackwnoledged, so the client does not know how many packets have actually made it to ther server.

Another help by measuring the effectiveness of the qdiscs in the kernel is the `tc` utility called with the statistics options `-s`. If the qdisc collect statistics, they are printed onto standard output. For example, the implementation of the RPQ manages separete statistics for each logical priority queue. Here an example of the `tc` output for a RPQ containing 4 queues.

```
qdisc rpq 1: dev eth0 nr_of_queues 4 rotation-time 10ms
byte-limit 12500 bfifo_queue_size 125000
  logical-queue 0 -  Sent 0 bytes 0 pkts (dropped 0, overlimits 0)
  logical-queue 1 -  Sent 12400 bytes 10 pkts (dropped 1, overlimits 0)
  logical-queue 2 -  Sent 0 bytes 0 pkts (dropped 0, overlimits 0)
  logical-queue 3 -  Sent 15000 bytes 12 pkts (dropped 0, overlimits 0)
 Sent 27400 bytes 22 pkts (dropped 1, overlimits 0)
```

There is another script which uses the RPQ specific data to produce `gnuplot` complient output. The script `messung.tcl` calls the `tc` utility periodically and fills the collected statistics on the RPQ qdisc in differnt files. For each RPQ logical queue, files contaning the relative time and bytes transmitted, packets transmitted and the number of packets dropped are written.

We have presented the usage of the traffic generation and measurement framework consisting of the sender, the receiver and various scripts. This framework can be used for gathering information about the architecture. Presenting results of these measurements is out of the scope of this work.

Chapter 6

Graphical Frontend for Traffic Control

Runtime structure of traffic control in the Linux kernel is read by the tc application and dumped on standard output. One can specify what information should be passed back, for example root qdisc and its classes, traffic filters of a classfull qdisc, statistical information etc. This chapter describes how all this information is fetched and drawn by a simple Tcl/Tk based graphical user interface (GUI).
Presented simple (and nice) graphical frontend visualizes the hidden kernel internals showing dependicies and hierarchical structure of teh traffic control elements.

6.1 Tcl/Tk, incr Tcl and TclX

The scripting language *Tcl* accompanied by its set of widgets, *Tk*, is the fastest way of developing a graphical interface. First, compact widget syntax gives a powerful set of graphical components requiering very little code. Second, as a scripting (interpreted) language, it shortens the development cost by exchanging the code-compile-run with code-run cycle.
Object oriented paradigm has found its way to all the domains of software development, includung graphical interfaces, where it has been used successfully by different object oriented frameworks. *Incr Tcl* is an extension to Tcl, adding object-oriented concepts to the language, and following the C++ syntax.
TclX is another Tcl extension adding a variety of Unix low-level calls. For example, signal handling, ioctls, and select to name a few.
In this work, a Tcl/Tk version extended by incr Tcl and TclX has been used.

6.2 Architecture

The GUI consists of two coupled executables:

- The Tcl/Tk executable is the graphical frontend, using the `tc` command utility for dumping the information about the runtime structure of the traffic control in the kernel. The hierarchical structure is drawn onto a canvas. The filter window shows detailed information about traffic filters and uses the `nslookup` command utility for transforming the numerical IP addresses to hostname. The atmtcd window shows relevant information conecerning atmtcd, like flows, VPI/VCI and QoS parameters.

- Rt-listener is a small executable, which has to be run with root-suid. It receives Rtnetlink messages concerning traffic control and is thus aware of every modification some apllication . Every time a message is received, SIGUSR1 signal is sent to the Tcl executable. Semantically, the signal carryies an update message telling the GUI to reread its information.

Figure 6.1 shows the class diagramm of a Tcl executable. Interface_tc, Atmtcdlist and FilterList are graphical classes, serving as frames for the main window, atmtcd log file frontend and filter frontend respectively. All other classes in the lower part of the digramm map the runtime structure of traffic control in the kernel in the runtime structure in Tcl code with appropiate objects.

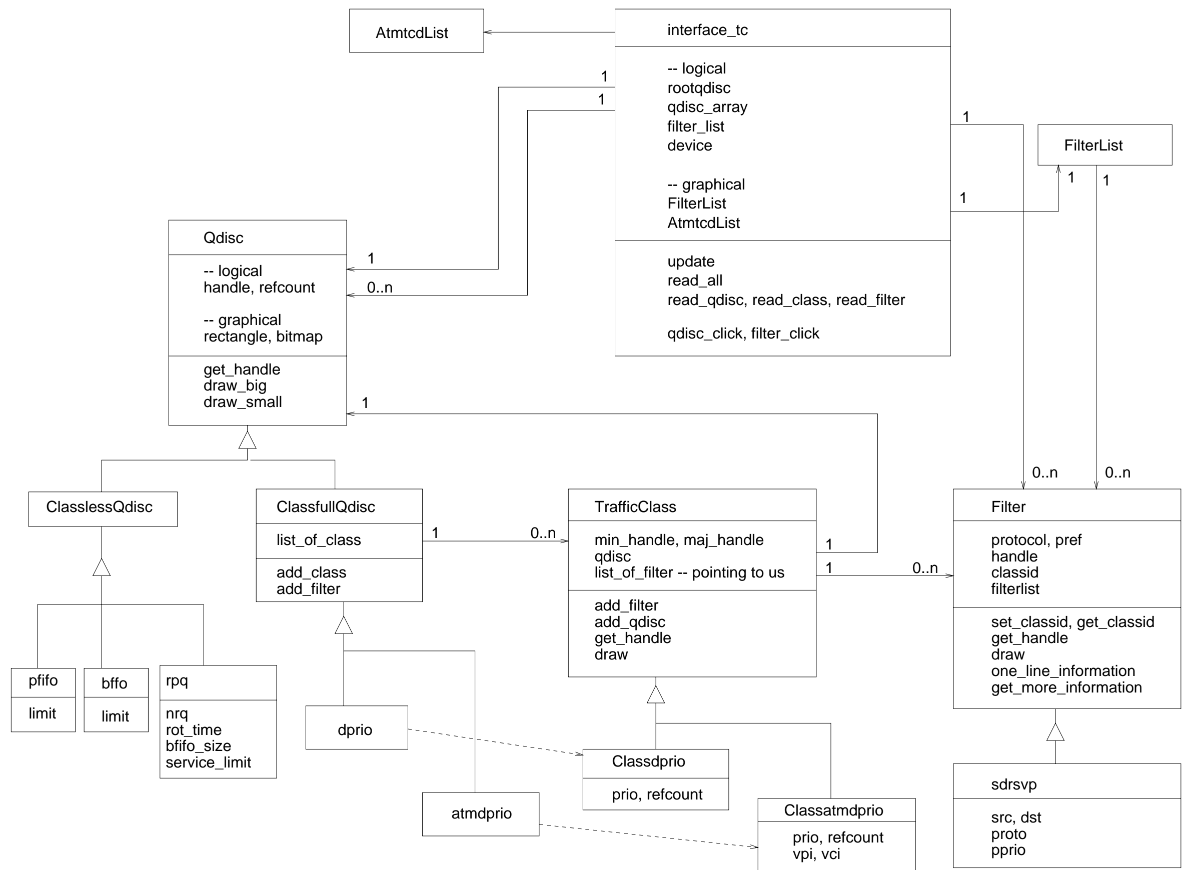

Figure 6.1: UML view of the incr Tcl code

The Interface_tc (Fig. 6.2, up) is the main (root) class consisting of logical and graphical class variables. Graphical class variables manage the menues and the canvas on which the runtime structure of the traffic control are drawn. Logical class variables are filled with the pointer to a root qdisc, list of other qdiscs and list of filters. These structures are filled by invoking `read_all` subsequently `read_qdisc, read_class` and `read_filter` which use the TC-Programm for reading the necessary information. Each time a Tcl process receives a SIGUSR1 signal, the information is reread (updated).

While reading the information dumped from TC by analysing the standard output, the runtime structure of the kernel is mapped into the runtime structure of Tcl classes. Depending on the kind

of the element, appropiate classes are used for the instantiation of the objects. Incr Tcl enables such dynamic (runtime) class selection[1]
When the information has been read, `draw_big` is called, which then calls the `draw` method of all the classes and filters, and the `draw_small` method of a qdisc.

The FilterList (Fig. 6.2, down) class is a simple Listbox widget displaying details about filters not shown in the canvas by using the local list of filters. Each time a filter entry is clicked on, `/etc/services` is read for obtaining possibly available textual information about ports used. In the second step, the `nslookup` program is used for obtaing the textual form of IP addresses in the filter. Asynchronous communiaction with the nslookup is used (`fileevent`), since nslookup blocks while contacting the next NameServer.

The AtmtcdList is another Listbox widget displaying the logged data from the atmtcd. AtmtcdList reads the Atmtcd log file asynchronously, displaying all the read data as entries in the list. The log-file consists of lines starting with *add* or *delete* indicating the construction or destruction of flows. Additional data carries the record-id, class-id, reserver and used bandwidth and QoS parameters.

6.3 Usage

The usage of the GUI is simple. Here a short overview. The Tcl executable is called with the interface name (*atm0*) as a parameter.

```
>./tc_interafce atm0
```

The Rtnetlink executable listening for changes in the traffic control is started from the main script. It receives a PID of the Tcl programm for sending the SIGUSR1 signal.
The file menu contains an entry *Update*, which forces the GUI to reread the traffic control state. The Windows menu is used for openening the Filterlist window or the Atmtcd-Log window.

Filter details When clicking on a rectangle presenting a traffic filter in the main window, the Filterlist windwo is opened in case it is not present on the screen already. The selected filter is inversed in the Filterlist. The entry in the list shows more detaiöed information about an sdrsvp filter. The upper part of the Filter list shows host-names and services using ports.

[1] Somewhat like the functionality of metaclasses in Smalltalk.

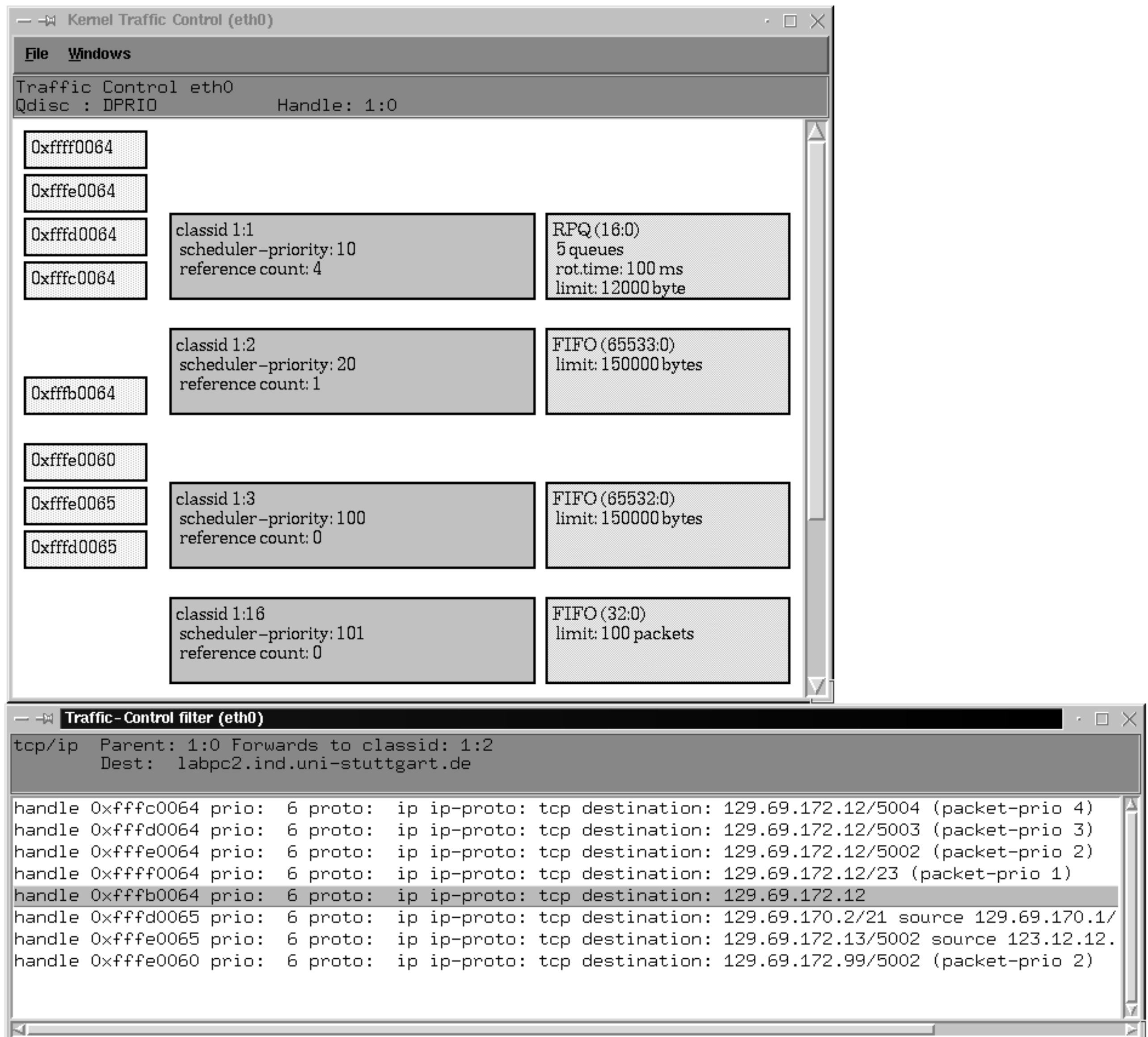

Figure 6.2: TC-GUI Snapshot

Chapter 7

Summary

In this last chapter, we shall present the results of this theses and take a look ahead at the furher possible developments.

Linux Developing software which is a part of the Linux kernel is very different from the ordinary user space programming. Linux is often called a *moving target*. The kernel structures change often and the changes are not always good documented. However, there are several good Linux kernel programming books, which accompanied with patience and code-reading exercise make development of new architectures possible.
As a part of this work, Rtnetlink sockets interface has been documented. It is a rather novel approach of connecting user applications with the kernel. Further, the basic structure of the ATM part of the kernel has been clarified.
While implementing the SDRSVP filter, many design decisions from the RSVP filter have influenced the work. Different optimization mechanisms relevant in the kernel and tricks used in the networking code have been described.
The work begann with the 2.2.1 kernel version, but ended up in the upgrade on to the version 2.2.9.

Software development The author has used RCS for configuration management of the code. Using a simple, but powerful tool like RCS had two important advantages. Firstly, having a checked-in working versions of the code makes experimenting and implementing new features easier, since one knows that the old running version still exists in the configuration management hierarchy and that it can be checked out any time. Second, at the time of the upgrade from 2.2.1 to 2.2.9, both versions of code were developed in parallel. RCS enabled clear separation of the code (branching) designed for the old and new kernel release respectively. The feature of combining revisions into a release (freezing) has made stepwise development of our architecture much easier.
The DejaGnu testing framework was used for definition of test cases and a test suite for the code. Although not many test cases have been written, they present an elementary, solid basis for gaining assurance that the software works. Furthermore, three different releases of the `tc` utility and two different kernel versions were used during this work and the regression test helped identify the changed features of the underlying modules.

RSVP over ATM We have described a step-by-step introduction of how a classful queueing discipline has been developed. The atmdprio queueing discipline associates a traffic class to a VC and uses RFC1483 encapsulation for the IP packets.

The implementation of the RPQ classless queueing discipline has been described, as well as the use of timers in the kernel. Further, we have clarified the scheduling issues concerning bottom halves, as well as where race conditions occur and how they are avoided.
The SDRSVP filter has been used a classifier, which reads the IP header and the ports stored in the UDP/TCP header of a packet. It is a simple filter containing subfilters managed in a hash table and different subfilter types depending on the amount of information given to it.
Various ad hoc tests using TCP have shown that the code developed is efficient, and that a packet being read by some SDRSVP filter and forwarded to a RPQ queueing discipline experiences approximately 5% extra delay due to the computation overhead.
Although the ATM Traffic Control Daemon and the atmdprio were developed in the same context, loose coupling was required for the two modules. Atmtcd is fully dependent of the atmdprio queueing discipline in the kernel, but not vice versa. With other words, the correct function of the kernel is guaranteed independent of an eventual crash of the Atmtcd.

Traffic Generation The simple traffic generation and measurement framework has been developed and used for some ad hoc tests. The framework consists of the two client-server based programs exchanging packets over UDP sockets and a set of Tcl and Shell scripts used for gathering information about user-side statistics and kernel-side qdisc specific statistics. The definition of the framework has been a first and important step towards the test and measurement phase of the proposed RSVP over ATM architecture.

Look Ahead This work has concentrated on the interoperation of Integrated Services and ATM networks. It is a small contribution towards the Next Generation Internet, a worldwide packet-based network supporting real-time services on top of different link-layer architectures and network technologies. The implementation issues in the Linux kernel described in this thesis could be used for similar projects mapping Integrated Services onto existing LAN and WAN technologies, as well as for support of wireless protocols. This work is a contribution to an idea of an Integrated Services Internet in a heterogenous networking environment.

Bibliography

ATM

[1] O. Kyas. *ATM-Netzwerke*. International Thompson Publishing. 1998

[2] R.O. Onvural, R. Cherukuri. *Signaling in ATM Networks*. Artech Hause. 1997

[3] A. Alles. *ATM Internetworking*. http://cell-relay.indiana.edu/cell-relay/docs/cisco.html 1995

[4] D.E, McDysan, D.E. Spohn. *ATM - Theory and Application*. McGraw-Hill. 1994

[5] The ATM Forum. *Traffic Management Specification - Version 4.0.* http://www.atmforum.com 1996

[6] W. Almesberger. *Linux ATM API, Draft, Version 0.4.* http://icawww1.epfl.ch/linux-atm/doc.html 1996.

[7] W. Almesberger.*Linux ATM internal signaling protocol, Version 0.2.* http://icawww1.epfl.ch/linux-atm/doc.html 1996

[8] W. Almesberger. *High-speed ATM networking on low-end computer systems.* http://icawww1.epfl.ch/linux-atm/doc.html 1996

[9] M. Laubach. *Classical IP and ARP over ATM*. RFC1577. 1994

[10] The ATM Forum. *Native ATM Services: Semantic Description. Version 1.0.* http://www.atmforum.com 1996

[11] M. Lin et al. *Distributed Network Computing over Local ATM Networks.* http://www.eantc.de/Documents/OtherPapers/IPoverATM/atmperf.ps 1995

Networks

[12] A.S. Tanenbaum. *Computer Networks*. Prentice Hall. 1996

[13] K. Rothermel. *Rechnernetze I*. Vorlesungsmanuskript. Universität Stuttgart. 1998

[14] P.J. Kühn. *Nachrichtenverkehstheorie*. Vorlesungsmanuskript. Universität Stuttgart. 1999.

[15] C. Partridge. *Gigabit Networking*. Addison Wesley. 1994

Integrated Services & RSVP

[16] D. Clark; S. Shenker; L. Zhang. *Supporting Real-Time Applicationsin an Intergrated Services Packet Network: Architecture and Mechanism*. ACM Computer Communication Review, vol. 22, Oct. 1992. SIGCOMM '92 Symposium.

[17] R. Braden et al. *Integrated Services in the Internet Architecture: an Overview*. RFC1633. 1994

[18] S. Floyd, V. Jacobson. *Link-sharing and Ressource Management Models for Packet Networks*. IEEE/ACM Transactions on Networking. Vol.4.No.4 August 1995

[19] P.P. White, J. Crowfort. *RSVP and Integrated Services in the Internet: A Tutorial*. IEEE Communications Magazine. May 1997

[20] L. Zhang et al. *RSVP: A New Resource ReSerVation Protocol*. IEEE Network. September 1993

[21] S. Shenker, L. Breslau. *Two Issues in Reservation Establishment*. ftp://parcftp.xerox.com/pub/net-research/opwacr.ps 1995

[22] R. Braden et al. *Resource ReSerVation Protocol (RSVP) – Vesion 1 Functional Specification*. RFC2205. 1997

[23] R. Braden, L. Zhang. *Resource ReSerVation Protocol (RSVP) – Vesion 1 Message Processing Rules*. RFC2209. 1997

[24] J. Wroclawski. *The Use of RSVP with Integrated Services*. RFC2210. 1997

[25] S. Shenker, J. Wronclawski. *General Characterisation Parameters for Integrated Service Network Elements*. RFC2215. 1997

[26] J. Wroclawski. *Specification of the Controlled-Load Network Element Service*. RFC2211. 1997

[27] S. Shenker et al. *Specification of Guaranteed Quality of Service*. RFC2212. 1997

[28] R. Braden, D. Hoffman. *RAPI - An RSVP Application Programming Interface Version 5*. Internet-Draft. 1998

DIANA

[29] M. Lorang. *A QoS Guaranteeing Framework for the Integration of IP and ATM in DIANA*. In EUNICE'98 Open European Summer School on Network Management and Operation, pp. 178-187, Munich, Germany, August 31 - September 3, 1998.

[30] E. Aarstad, L. Burgstahler, M. Lorang *Description of the Flow-to-VC Mapping Control Module in DIANA's RSVP over ATM Architecture*. In QoS Summit'99, Paris, France, November 16-19, 1999.

[31] L. Burgstahler et al. *Prototype Implementation of a RSVP/IP and ATM Network Integration Unit*. Deliverable 3 of the E.C project ACTS/DIANA, AC319/DIANA/WP2/K/P/000/A1, October 1999

[32] J. Liebeherr et al. *Exact Admission Control for Networks with a Bounded Delay Service.* IEEE/ACM Transactions on Networking. Vol.4,No.6. Dec.1996

[33] W. Almesberger. *Linux Traffic Control - Implementation Overview.* ftp://lrcftp.epfl.ch/pub/people/almesber/pub/tcio-current.ps.gz 1999

[34] V. Sironja. *Kommunikation zwischen den Traffic Control Elementen in User- und Kernel-Space.* Internal IND Publication. 1999

ATM - RSVP

[35] M. Garett et al. *Interoperation of Controlled-Load Service and Guaranteed Service with ATM.* RFC2381. 1998

[36] E. Crawley et al. *A Framework for Integrated Services and RSVP over ATM.* RFC2382. 1998

UNIX Programming

[37] W.R. Stevens. *Advanced Programming in the UNIX Environment.* Addison-Wesley. 1992

[38] W.R. Stevens. *TCP/IP Illustrated, Volume 1.* Addison-Wesley. 1994

[39] G.R. Wright, W.R. Stevens. *TCP/IP Illustrated, Volume 2.* Addison-Wesley. 1995

[40] W.R. Stevens. *UNIX Network Programming, Volume 1.* Prentice-Hall. 1998

[41] D.A. Curry *UNIX Systems Programming for SVR4.* O´Reilly & Associates. 1996

[42] A. Rubini. *Linux Device Drivers.* O´Reilly & Associates. 1998

[43] M. Beck et al. *Linux-Kernel Programmierung.* Addison Wesley. 1997

[44] A. Cox. *Network Buffers And Memory Management.* http://lx5cmd.inp.nsk.su/cmd2/manuals/LDP/khg/HyperNews/get/net/net-intro.html 1996

[45] D.A. Rusling. *The Linux Kernel.* http://linux.greynet.net/LDP/LDP/tlk/tlk-title.html 1998

Software

[46] B.W. Kerninghan, D.M. Ritchie. *The C Programming Language.* Prentice-Hall. 1988

[47] D.R. Hanson. *C Interfaces and Implementations.* Addison-Wesley. 1997

[48] S. Loosemore et.al.. *The GNU C Library Manual Version 2.00 Beta.* http://www.gnu.org/manuals 1996.

[49] B. Meyer. *Object-oriented software construction.* Prentice Hall. 1997

[50] S. Alhir. *UML in a Nutshell.* O´Reilly & Associates. 1998

[51] D. Bolinger, T. Bronson. *Applying RCS and SCCS.* O´Reilly & Associates. 1995

[52] D. Libes. *Exploring Expect.* O´Reilly & Associates. 1995

[53] R. Savoye. *The DejaGnu Testing Framework.* http://www.gnu.org/manuals 1996

[54] ANSI/IEEE Std 1008-1987. *IEEE Standard for Software Unit Testing.* 1987.

Measurement

[55] V. Paxon et al. *Framework for IP Performance Metrics.* RFC2330. 1998

[56] G. Almes et al. *A One-way Delay Metric for IPPM.* RFC2679. 1999

Tcl/Tk

[57] M.J. McLennan. *Object-Oriented Programming with [incr Tcl].* In: M. Harrsion. *Tcl/Tk Tools.* O´Reilly & Associates. 1997

[58] M. Harrison, M.J. McLennan. *Effective Tcl/Tk Programming.* Addison-Wesley. 1998

[59] De Clarke. *TclX.* In: M. Harrsion. *Tcl/Tk Tools.* O´Reilly & Associates. 1997

Appendix A

Mapping OO to C

We have presented most of the design diagrams using UML Notation [50]; the obvious question is, how do those map into C programming language. In this section, we present mostly used techniques in the Linux-kernel. A short, but well explained text on the topic can be found in [49], more on C in [46][47]. The presented ideas here show only one possible way of mapping OO to C.

We show the ideas using the simple structure in Fig.A.1. The class A has 2 class-variables and 2 methods. It associates to class B. Class C inherits from class A, it adds one new variable, redefines one method and adds one method.

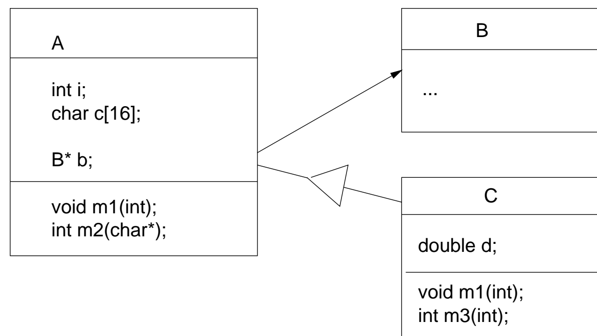

Figure A.1: Example class associations

Class-type can easily be represented through a structure. The *class-variables* are mapped into structure fields. Of course, there is no way of accomplishing the protection levels like private or protected – all variables are public, other protection levels can only be achieved through programming discipline. So we have:

```
typedef struct {
    int i;
    char c[16];
    B* b;
} A;
```

Each variable of type `A` is an instance (object) of class A. After defining the structure B the same

way, we can easily achieve *association* through a pointer to struct B.

Methods are represented through pointers to functions. We can define two pointers to functions being elements of structure A, but it would lead to highly inefficient memory management, since *every* instance of class A would have 2 pointers. A more extreme example is a class having 10 methods and 100 instances making 1000 references necessary. The workaround is a definition of structure containing only methods, and one pointer to this structure as a field. Following the example mentioned above, only $100 + 10 = 110$ references are needed. Except for mapping the methods into pointers to function, we also add one *constructor* and one *destructor*. Methods either modify or read class-variables of an object. Since there is no *this* construct, we have to pass the reference of the structure A as a parmeter to every function having:

```
struct A_ops {
    void (*init)(A*);
    void (*destroy)(A*);
    void (*m1)(A*, int);
    int  (*m2)(A*, char*);
};
```

The final structure containing the class A in a structure containing class variables and a pointer to the methods-structure:

```
typedef struct A {
    int i;
    char c[16];
    B* b;
    struct A_ops* ops;
} A;
```

Now we can implement the interfaces `m1` and `m2` through real functions `m1_impl` and `m2_impl`, implement the constructor `init_impl` and destructor `destroy_impl` and finally initialize the structure:

```
static struct A_ops aops = {
    &init_impl,
    &destroy_impl,
    &m1_impl,
    &m2_impl
};
```

Following code instantiates a new object of type A, cleans the memory segment belonging to it, associates implmented methods to the object, fills the class variables and class the constructor.

```
A* a = (A*)malloc(sizeof(A)); /* alloc */
memset(a,0.sizeof(A)); /* clear */
a->ops = &aops; /* init methods */
a->i = ...; a->c = ...; /* fill data */
a->ops->init(a);
```

After the instantiation, we can use the object by invoking methods through `a->ops->m1(a,2)`. Making copies is also relatively simple using `memcpy`. After calling a destructor, the memory segment is freed:

```
a->ops->destroy(a);
free(a);
```

Implementing *inheritance* is slightly tricky. As a first step, the class C is implemented in the same manner as class A, having:

```
typedef struct C {
    double d;
};

static struct C_ops bops_impl = { &m3_impl };
```

Class A has to be prepared for being inherited from class B by adding a pointer to the methods added by class C and for the class-variables being added by class C:

```
typedef struct A {
    ... /* as before */
    struct C_ops* cops;
    char subtype[0];
};
```

The pointer `subtype` is supposed to hold the private data of the subclass (C) and the pointer `C_ops` references the methods of the subclass. Objects of class C still have the type of class A, making interchangeable usage of an object as a class A and a class C possible (*subtyping*), but not the other way around. In the following code, we use the variable `a1`, belonging logically to class C, yet being of type A.
The first line allocates memory for both structures (A and B). The lower part of the memory segment is thus attached to the `subtype` pointer, which we intentionally placed at the end. In order to manipulate the class variables of C, we first cast the `subtype` pointer into *C*. This casting is needed in the method `m3` as well, since it uses the private data of the class C.

```
A* a1 = (A*)malloc(sizeof(A)+sizeof(B)); /* alloc */
C* c = (C*)a1->subtype;
c->d = ...;
a1->cops = &cops_impl;
a1->cops->init(a);
```

After setting the methods pointer and invoking the C-specific constructor, the variable `a1` can be used as type A and type C. Functions which need to know if the instance is actually of type A or C can check for existence of the cops pointer.

```
if (a1->cops)
    a1->cops->m3(a,i);
```

Redefining the method `m1` is straight forward. After reimplementing the method, the appropriate pointer to the new implementation has to be set before the object is used.
Implementing a second level of inheritance or multiple inheritance makes things a little awkward and complicated. Yet, we have shown a simple and efficient way of mapping basic object oriented ideas into C, which satisfy when for *some* reason C++ can, or may not be used.

Ich versichere, daß ich diese Arbeit selbständig verfaßt und nur die angegebenen Hilfsmittel verwendet habe.

Stuttgart, den 1. Dezemeber 1999 ________________

Mir ist bekannt und ich erkenne an, daß ich an den Ergebnissen meiner Diplomarbeit keine eigenständigen Verwertungsrechte habe. Eine Verwertung der Arbeit einschließlich der Ausarbeitung darf nur nach Zustimmung durch das Institut für Nachrichtenvermittlung und Datenverarbeitung erfolgen. Ich erkläre weiterhin, daß ich diese Arbeit selbständig verfaßt und keine anderen als die in meiner Ausarbeitung angegebenen Hilfsmittel für die Durchführung der Arbeit verwendet habe.

Stuttgart, den 1. Dezember 1999 ________________