SCALABLE RESOURCE RESERVATION FOR THE INTERNET

THÈSE N^{O} 2051 (1999)

PRÉSENTÉE À LA SECTION DE SYSTÈMES DE COMMUNICATION

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES TECHNIQUES

PAR

Werner ALMESBERGER

Dipl. Inf.-Ing. ETH

de nationalité autrichienne

acceptée sur proposition du jury:

Prof. J.-Y. Le Boudec, directeur de thèse
Prof. E. Biersack, rapporteur
Dr J. Heinanen, rapporteur
Dr M. Potts, rapporteur
Prof. B. Stiller, rapporteur

Lausanne, EPFL
1999

Scalable Resource Reservation for the Internet

Acknowledgements

I am very grateful to Prof. Jean-Yves Le Boudec who was all I could hope for in a PhD director and who has spent a lot of time guiding and contributing to this work.

Among my colleagues at the LRC and then ICA institute at EPFL, I owe special thanks to Silvia Giordano for all the good public relation work for Arequipa and SRP, and for quietly enduring my search for the perfect background music, to Leena Chandran-Wadia and Philippe Oechslin for working with me on Arequipa, to Giuseppe Di Fatta for designing the scalable policing framework for SRP, Paul Hurley for porting my SRP simulator to ns-2, and to Patrick Thiran and Arnis Ziedins for making valuable contributions to the estimators of SRP.

I would like to thank them, and all my other colleagues, past and present, at LRC/ICA for many stimulating discussions and for making our lab a friendly place to work at: Ljubica Blazevic, Heiko Boch, Catherine Boutremans, Olivier Crochat, Hans Einsiedler, Xavier Garcia, Eric Gauthier, Constant Gbaguidi, Maher Hamdi, Thorsten Kurz, Leila Lamti, Sam Manthorpe, Raffaele Noro, Kestutis Patiejunas, Stephan Robert, and Olivier Verscheure.

I would also like to thank our system managers Bruno Dufresne and Jean-Pierre Dupertuis, and our secretaries Danielle Alvarez, Angela Devenauge, and Yvette Dubuis for all their work behind the scenes.

Many wonderful improvements of SRP have resulted from the work Tiziana Ferrari did as a student at the doctoral school, and then later, when writing her own thesis. Also Arrate Muñoz made many valuable contributions to SRP and the simulation environment while at the doctoral school. Many thanks to both of you !

There are many students who worked hard on improving ATM on Linux and whose questions often made me realize that the answers I thought I had were wrong: Edouard Lamboray, André Meier, Fabien Modoux, Pedro Paiva, Jean-Michel Pittet, Roman Pletka, Cameron Pope, and Olivier Voumard.

The ATM on Linux project has benefitted greatly from the help of many people. In particular, I would like to thank Juha Heinanen for spending countless hours debugging my code, and for initiating the projects of Marko Kiiskilä and Heikki

Vatiainen, who have made amazing contributions on LAN Emulation, ANS, and MPOA over a very long time. Thanks go also to Rui Prior for maintaining a very popular driver, to Scott Shumate for writing the ILMI demon, and to Steven Wright for introducing me to the ATM Forum, and for his help with Arequipa.

Likewise, the Diffserv on Linux project has had many contributors. My special thanks go to Jamal Hadi Salim and Alexey Kuznetsov with whom I had the privilege to work on the design and the implementation.

Almost all of the implementation work related to my thesis was done on Linux. It would be moot to even try to list all the great people who have contributed to this unique system. I am specially indebted to Alan Cox, David S. Miller, and of course our "benevolent dictator", Linus Torvalds.

I spent a particularly enjoyable two years working in the DIANA project. I would like to acknowledge the help of many members of this project, in particular Piergiorgio Cremonese, Martin Lorang, John Loughney, and Martin Potts.

The Swiss Office Fédéral de l'Education et de la Science and ACTS programme have financed my research during my five years at EPFL and I acknowledge their support.

Abstract

For some time now, the design of architectures for providing dependable Quality of Service (QoS) on multimedia networks has been a busy topic in academia and industry. In this work we look at practical aspects of Quality of Service, and in particular at resource reservation for the Internet.

Initially, our focus was on ATM, which looked like the most promising QoS technology at the time. We proposed, implemented, and demonstrated Arequipa, a mechanism that uses ATM to provide QoS for TCP/IP, and which only requires end-to-end connectivity, but no changes in the ATM network.

While ATM still has its place, the Differentiated Services architecture, which returns to the paradigm of packet-oriented processing in the network, is nowadays a more likely candidate for providing a basis for supporting dependable Quality of Service in the Internet. We therefore concentrated on Differentiated Services when looking for solutions to overcome some of the problems we found in the reservation models in Arequipa and the underlying ATM.

Our work culminates in the development of SRP, the "Scalable resource Reservation Protocol", a highly scalable yet conceptually simple reservation protocol for TCP/IP. SRP achieves scalability by aggregating flows in the network, and by controlling flow admission by making simple per-packet decisions, which are based on estimates of the bandwidth available for reservation.

We give a detailed description of the SRP architecture, including examples of estimation algorithms and an introduction to a novel approach for scalable policing. We describe how we integrated SRP in the `ns` simulator, and show simulation results. Finally, we have implemented SRP on top of Differentiated Services on Linux, and again we describe the implementation and give measurement results.

We also discuss some of the infrastructure-building activities that have resulted from our work, namely the ATM on Linux project, documentation of Linux traffic control, and the Diffserv on Linux project.

Zusammenfassung

Die Entwicklung von Architekturen, die eine verlässliche Dienstqualität (Quality of Service, QoS) in Multimedianetzen ermöglichen, ist bereits seit einiger Zeit ein in Forschung und Industrie mit grossem Interesse verfolgtes Thema. In der vorliegenden Arbeit achten wir auf praktische Aspekte von QoS, und speziell auf Ressourcenreservation für das Internet.

Anfangs konzentrierten wir uns auf ATM, das damals die vielversprechendste Architectur für QoS zu sein schien. Wir haben Arequipa, einen Mechanismus der ATM verwendet um QoS für TCP/IP bereitzustellen, entwickelt, implementiert, und demonstriert. Arequipa benötigt ATM am ganzen Weg zwischen kommunizierenden Knoten, braucht sonst aber keine Änderungen im Netzwerk.

Heutzutage ist die "Differentiated Services" Architektur (Diffserv), die zum Prinzip der paketorientierten Datenverarbeitung im Netzwerk zurückkehrt, ein wahrscheinlicherer Kandidat für QoS im Internet als ATM. Wir haben uns deshalb auf der Suche nach Lösungsansätzen für Probleme, die sich im Rahmen unserer Arbeit mit Arequipa gestellt haben, fortan auf Diffserv konzentriert.

Unsere Arbeit gipfelt in der Entwicklung von SRP, dem "Skalierbaren RessourcenreservationsProtokoll", einem hochgradig skalierbaren, dennoch aber konzeptionell einfachen Reservationsprotokoll für TCP/IP. SRP erreicht Skalierbarkeit durch Aggregieren von Datenflüssen im Netzwerk, und durch das Kontrollieren der Verbindungsannahme durch einfache, auf Schätzungen der verfügbaren Bandbreite gestützte Entscheidungen, die für einzelne Pakete getroffen werden.

Wir geben eine ausführliche Beschreibung der SRP-Architektur, mit Beispielen für Algorithmen zur Bandbeitenabschätzung, und skizzieren auch einen neuartigen Ansatz für skalierbares Policing. Wir beschreiben, wie wir SRP in den `ns` Simulator integriert haben, und zeigen Simulationsergebnisse. Schliesslich haben wir SRP auf Diffserv für Linux implementiert, und auch hier beschreiben wir dir Implementierung und zeigen Messergebnisse.

Weiter beschreiben wir einige der Aktivitäten zur Bereitstellung von Infrastruktur, die von unserer Arbeit ausgegangen sind, nämlich das "ATM on Linux" Projekt, Dokumentation für Linux Traffic Control, und das Projekt Diffserv für Linux.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this chapter, we briefly introduce the concept of Quality of Service. Then we describe major QoS architectures, to which our work is closely related. After this preparation, we can discuss our activities in the area of QoS research. This chapter finishes with a look at related work, and an outline of the rest of the book.

1.1 Quality of Service

Transferring multi-media data and other time-critical data over networks may need a dependable Quality of Service (QoS). This is the case for all situations where a minimum guaranteed bandwidth is required. There are fundamentally two approaches to provide Quality of Service:

- The network can be dimensioned in such a way that traffic requiring quality of service is very unlikely to ever exceed the available resources.
- An alternative is to explicitly reserve resources for specific data flows [1]. This line of thinking finds its origin in the past experience in telephony networks.

Enlarging the network until all delay and throughput requirements are met on all links is a reliable way of ensuring that applications obtain the service they need. Unfortunately, given sufficiently diverse needs, this can quickly lead to economically infeasible requirements on the network.

Traditional reservation architectures tend to require explicit specification of resource requirements and careful tracking of reservation state. This makes them complex to design, expensive to implement, difficult to use, and may also cause scalability problems.

A compromise between telephone-style reservations and provisioning is the use of different priority levels, with higher priority given to flows that are expected to receive high quality of service. This is the approach taken in proprietary architectures such as [2]. The work currently underway at the Internet Engineering Task Force (IETF) on the topic of Differentiated Services [3] also provides solutions in that direction.

Later on, we will show how resource reservations can be built upon Differentiated Services, such that the dependability of reservations can be obtained without sacrificing the elegance and scalability of a priority scheme.

1.2 Quality of Service architectures

This section discusses the evolution of the nowadays most visible Quality of Service architectures for integrated services networks. It also briefly characterizes the approaches introduced in the course of this work on reservation mechanisms for the Internet. Some of the mechanisms are then described in much more detail later in this chapter.

1.2.1 IP precedence and TOS

The concept of a non-uniform service has been part of the IP architecture since the very beginning. Already the original specification of the Internet Protocol [4] provides a type of service (TOS) byte containing a set of eight precedence values and a set of TOS bits that can be used to indicate the need for short delay, high throughput, or reliability.

The use of the TOS bits has been later re-defined in [5], which only allows a single bit to be set at a time, and which also introduces a new service type called "minimize monetary cost". Finally, [6] added the somewhat dubious concept of

"security" (mainly in the sense of confidentiality) as a fifth non-default service.

The original use of the TOS byte was a mixed success: the use of precedences to give traffic necessary for the operation of the network infrastructure priority over "user traffic" was generally accepted. This was mainly applied to routing traffic.

Setting the TOS bits is supported by most operating systems and also by many applications, in particular by those using protocols listed in section "IP TOS PARAMETERS" of [7]. However, only few networks or routers were actually configured to differentiate packet treatment based on TOS bits. Due to this lack of consistent deployment, no meaningful end-to-end service could be constructed, except in trivial or highly homogeneous cases.

Recently, the interest in using the TOS byte has increased again. For example Cisco's "Committed Access Rate" [8] allows the assignment of precedence values in IP packets based on packet classification and on rate metering. The Differentiated Services architecture generalizes this concept even more and is discussed in a separate section below.

1.2.2 ATM

One network technology for providing reservations is the Asynchronous Transfer Mode (ATM) [9, 10], which promises to provide a scalable network architecture. The design of ATM considered reservation mechanisms from the very beginning. ATM networks therefore now offer reliable reservation mechanisms and well-understood traffic management concepts. Corporate and public ATM networks are already a reality in many places. Applications that are specifically written to use ATM, so-called *native* ATM applications, already guarantee end-to-end QoS today. However, one major problem is that the vast majority of networked applications is written to TCP/IP service interfaces, not to ATM. If you want to use TCP/IP applications in such environments, the standard solution is to run IP over ATM.

A more detailed description of native ATM can be found in chapter A. Technologies for carrying IP over ATM are discussed in chapters A and B, and in particular in section B.2.

1.2.3 Integrated Services (intserv)

The IETF identified the need for supporting integrated services years ago [11, 12], and has been working on the design of reservation mechanisms for TCP/IP. One major result of this activity are the Resource reSerVation Protocol (RSVP [13]), the definition of a guaranteed and a controlled-load service [14, 15], and the corresponding mappings to specific link layers, which are largely still in draft status. This approach is based on the concept of *integration*: network nodes (here: routers) need to be upgraded in order to support an additional set of functions required by the reserved services. Once and if RSVP is deployed across the Internet, it is possible to use TCP/IP applications with some end-to-end quality of service.

1.2.4 Differentiated Services

Traditional resource reservation architectures that have been proposed for integrated service networks (RSVP [13], ST-2 [16], Tenet [17], ATM [10, 18], etc.) all have in common that intermediate systems (routers or switches) need to store per-flow state information. The more recently designed Differentiated Services architecture [19] offers improved scalability by aggregating flows and by maintaining state information only for such aggregates. The basic architecture does not include resource reservation but depends on additional mechanisms for signaling (e.g RSVP [20]) and for resource allocation (e.g. provisioning or so-called "bandwidth brokers" [21]).

1.2.5 RSVP over Diffserv

Recently, hybrid approaches combining RSVP and Differentiated Services have been proposed (e.g. [22]) to overcome the scalability problems of RSVP. Unlike SRP, which runs end-to-end, they require a mapping of the INTSERV services onto the underlying Differentiated Services network, and a means to tunnel RSVP signaling information through network regions where QoS is provided using Differentiated Services.

1.3 In quest of QoS

We summarize in this section the work presented in more detail in the rest of this book. We describe the research activities in chronological order, and we also mention the reason why we decided to examine those specific approaches.

1.3.1 Why Linux ?

Since Linux is a re-occurring theme throughout this book, a little historical note on why we selected this operating system seems appropriate before we discuss what we used it for.

In the middle of 1999, this question may sound a little strange. Linux has gained so much popularity and its benefits for advanced research and development work are so obvious, that it seems difficult to imagine using any other platform for work which is likely to involve modifications at various places in the operating system, including the kernel.

This wasn't always so. Back at the end of 1994, when our work started, Linux was still a bit of an oddball system, cherished by its adepts, but largely ignored and dismissed as a hacker's toy by most people looking for a "serious" platform. The usual route taken by researchers who wanted to gain actual implementation experience was either to confine themselves to user space, or to use a commercial Unix kernel and to do their work under non-disclosure agreements, typically allowing them to publish their findings and to release their software in compiled (binary) form, but limiting access to source code, which made it difficult if not impossible for other researchers or developers to build upon their work, and for students to learn from it.

This is a situation we clearly wanted to avoid. In order to gain a deeper understanding of real-life performance of QoS architectures, and in order to be able to design and implement our own improvements, an open, flexible, and easily extensible platform with state of the art support for QoS techniques was needed. We found openness, flexibility, and extensibility in Linux, but QoS support was lacking. Our research has therefore spawned a number of strongly implementation-oriented

activities which provided the infrastructure for experimenting with more advanced designs.

An alternative to Linux could have been BSD, which shares many attractive properties with Linux, and which, particularly in 1994, was reputed to have a TCP/IP stack vastly superior to the one found in Linux. The decision for Linux was motivated mainly by prior experience and the perception of a much more dynamic and also more coherent[1] development process. Also, we were confident that any major shortcomings in parts of Linux would be rectified soon, which turned out to be correct. Linux has served us well, and meanwhile, it has gained so much momentum, also compared to BSD, that we are very happy with our choice also under this aspect.

The infrastructure-building projects that have resulted from our work are the ATM on Linux implementation, which provides a complete ATM stack from the device drivers, via several mechanisms for running IP over ATM, up to the signaling protocols; the documentation of Linux traffic control; and the implementation of support for Differentiated Services on Linux. All of these activities were well received in the Linux community and have become de facto standards on Linux.

1.3.2 Arequipa

Initially, we perceived ATM as the future network architecture for quality of service. However, the all the wonderful capabilities of ATM were useless for most applications, which were written for TCP/IP, because no mapping of QoS characteristics between TCP/IP and ATM was available at the time. We designed Arequipa to address this issue.

Arequipa is a method for providing the quality of service of ATM to TCP/IP applications without requiring any cooperation in the network between IP and ATM. It does not need any modifications in the ATM or IP networks; however, it requires end-to-end ATM connectivity.

[1] Due to various reasons, BSD development had splintered into several competing and mutually hostile efforts. In fact, fragmentation of the BSD community was so rapid that people jokingly referred to the ensemble of 386BSD, FreeBSD, NetBSD, OpenBSD, etc. as "BSD du jour". Fortunately, the situation has improved.

The approach taken by Arequipa is that of service integration in hosts only. It relies on the fact that TCP/IP implementations do not follow a strict layer separation: the IP destination tables in hosts are typically set per socket pair, rather than per IP destination address. This makes it possible to select a given ATM connection for one specific application flow, instead of for one IP destination address. Service integration in hosts rather than in the network makes it possible to use QoS immediately, since ATM commercial networks are already in operation.

1.3.3 From Arequipa to SRP

There is a number of lessons we learned from the implementation and deployment of Arequipa, but here we would like to focus on one major lesson. It has to do with the observation that, contrary to our expectation when we started the project in 1994, the penetration of end-to-end ATM remains minuscule. One obvious reason is the fact that ATM requires specific communication adapters, and cannot run today on the existing hosts, which, for the vast majority of them, use Ethernet. However, our work on Arequipa may give us some additional clues about the reasons for this state of affairs. Certainly, the lack of ATM penetration is *not* due to the difficulty of making QoS available and visible to the user. Indeed, with Arequipa, we have a solution readily available for the Unix environment, and we conjecture that porting that solution to the market dominating operating system would not be a major effort. We also do not believe that the minor changes required to Web clients or servers are a major drawback, since the time between releases of this type of software is usually less than a year. If there would be a massive push to obtain QoS in hosts, then the ATM penetration would be higher. Therefore, we are lead to think that making QoS visible to the user is an idea that simply did not meet its market. Many users would like to have it, but hardly any organization is willing to invest in the network technology required to support it. This also leads us to conjecture that approaches based on RSVP that would attempt at making QoS visible to the end user will equally suffer from the same lack of penetration, because introducing RSVP into the Internet is also a major investment.

If we follow this line of thought, we conclude that it may not be a good idea to

let the QoS be visible to the application, except maybe for niche application settings where the investment is justified.

1.3.4 SRP

The realization that traditional reservation mechanisms were too complex, both for rapid deployment, and for addressing user needs, led us to look for a much more light-weight approach. SRP is the result of this work.

SRP extends upon simple aggregation by providing a means for reserving network resources in routers along the path taken by flows, using a single end-to-end protocol. It does so without explicit signaling of flow parameters, and without requiring routers to maintain per-flow state. Instead, routers monitor the aggregate flows of reserved traffic and maintain a running estimate of what amount of resources is required to serve them with the appropriate quality of service.

1.4 Related work

1.4.1 IETF and ATM Forum

IETF and ATM Forum have specified a plethora of protocols to address various aspects of providing Quality of Service. We have already mentioned some of them in section 1.2. Section B.2 compares several approaches for transmitting IP packets over ATM in more detail.

1.4.2 Guaranteed Internet Bandwidth

This is an architecture that, similar to Arequipa, uses service integration to obtain dependable QoS. We discuss it in the appendix on Arequipa, in section B.2.6.

1.4.3 Ticket Signalling Protocol

The "Ticket Signalling Protocol" [23] looks at first sight quite similar to SRP, because it also aggregates flows everywhere in the network. The main difference is

that TSP still uses an explicit form of signaling, the so-called tickets. Where SRP can rely on easily repeatable measurements, TSP has to keep track of the state of its tickets, thereby sacrificing much of the simplicity gained by aggregation.

1.4.4 Phantom Circuit Protocol

The "Phantom Circuit Protocol" (PCP) has recently been proposed in [24]. PCP is very similar to SRP and also uses a concept of "probes" and "reserved" packets, where the probe traffic is a model of the future reserved traffic. Since the packet types are sufficiently similar, we will use the SRP terms *request* and *reserved* when discussing PCP in the remainder of this section.

PCP differs from SRP in that routers are not required to explicitly remove *request* packets (i.e. either by dropping or degrading them), but just have to treat them worse than *reserved* packets. The destination can then determine the bandwidth available for *reserved* traffic by measuring the properties of the *request* flow.

Details on many architectural considerations, such as how successful reservations are communicated back to the source, or the effect probes can have on best-effort traffic, are still missing. Furthermore, although not generally required, the preferred mechanism for treating *request* packets worse than *reserved* packets seems to be dropping. There also seems to be the concept of waiting until the full reservation has been obtained before admitting actual user traffic. SRP offers more flexibility by allowing failed *requests* to be degraded, so that immediate end-to-end communication is still possible, although without reservation.

[24] claims not to require bandwidth estimates in the routers. However, in order to prevent *reserved* traffic from allocating arbitrarily large amounts of bandwidth (only limited by the link capacity, and perhaps the jitter experienced by individual micro-flows), some bandwidth limiting of *request* and *reserved* traffic is needed, which is, in fact, also reflected by the suggested use of Weighted Fair Queuing in [25].

Also, it is not clear if PCP is susceptible to reservation drifts caused by *requests* receiving sufficiently good forwarding behaviour in times of congestion, because jitter in *reserved* may momentarily lower the bandwidth perceived at the router.

1.5 Structure of this book

Chapter 2 introduces the fundamental design of SRP, specifies the protocols, gives examples for the various measurement algorithms, and discusses more advanced issues, such as scalable policing and multicast.

In chapter 3, we describe the implementation of an SRP simulator, and give a set of simulation results illustrating the performance of SRP.

Chapters 4 and 5 describe the general design of Linux Traffic Control and of the extensions we made in order to support Differentiated Services. This serves as the basis for the implementation of SRP on Linux.

In chapter 6, we continue with the topic of actual implementations, and describe how we extended the Diffserv infrastructure on Linux to support SRP. In chapter 7, we show the performance of this implementation in a small test network.

The historical work that preceded SRP is described in the following appendices: Appendix A gives a short introduction to ATM and details selected aspects of the ATM on Linux implementation. In appendix B, we describe how we implemented Arequipa and made it publicly available in Linux, how we applied it to the Web, and finally, how we tested it on a European ATM wide area network.

Chapter 2

The Scalable Resource Reservation Protocol

In this chapter, we describe the basic SRP concepts and algorithms, we show how SRP can be integrated with the current Internet architecture, and we introduce an approach for policing aggregated flows in a scalable way.

This chapter is organized as follows. Section 2.2 provides a more detailed protocol overview. Section 2.3 describes the role of the traffic estimators and discusses algorithms for their implementation. Section 2.4 addresses policing issues and outlines an approach for designing a scalable policing mechanism.

2.1 Introduction

The Scalable Reservation Protocol (SRP) provides a light-weight reservation mechanism for multimedia traffic in the Internet. Our main focus is on good scalability to very large numbers of individual flows. End systems (i.e. senders and destinations) actively participate in maintaining reservations, but routers can still control their conformance. Routers aggregate flows and monitor the aggregate to estimate the local resources needed to support present and new reservations. There is neither explicit signaling of flow parameters, nor maintenance of per-flow state by routers.

Reservation mechanism In short, our reservation model works as follows. A source that wishes to make a reservation starts by sending data packets marked as *request* packets to the destination. When receiving a *request* packet, a router determines whether hypothetically adding this packet to the flow of *reserved* packets would still allow it to meet the quality of service goals.[1] If so, the *request* packet is accepted and forwarded towards the destination, while still keeping the status of a *request* packet. In the opposite case, the *request* packet is degraded to a lower traffic class, such as best-effort, and forwarded towards the destination. A packet sent as *request* will reach the destination as *request* only if all routers along the path have accepted the packet as *request*.

The destination periodically sends feedback to the source indicating the amount of *request* and *reserved* packets that have been received. This feedback does not receive any special treatment in the network (except possibly for policing, see below). Upon reception of the feedback, the source can send packets marked as *reserved* according to a profile derived from the feedback. If necessary, the source may continue to send more *request* packets in an attempt to further increase the reservation.

Thus, in essence, a router accepting to forward a *request* packet as *request* allows the source to send more *reserved* packets in the future; it is thus a form of implicit reservation.

Aggregation Routers aggregate flows on output ports, and possibly on any contention point as required by their internal architecture. They use estimator algorithms for each aggregated flow to determine their current reservation levels and to predict the impact of accepting *request* packets. The exact definition of what constitutes an aggregated flow is local to a router.

Likewise, senders and sources treat all flows between each pair of them as a single aggregate and use estimator algorithms for characterizing them. The estimator algorithms in routers and hosts do not need to be the same. In fact, we expect hosts to implement a fairly simple algorithm, while estimator algorithms in routers may evolve independently over time.

[1] Quality of service is loss ratio and delay, and is defined statically.

Fairness and security Denial-of-service conditions may arise if flows can reserve disproportional amounts of resources or if flows can exceed their reservations. We presently consider fairness in accepting reservations a local policy issue (much like billing) which may be addressed at a future time.

Sources violating the agreed upon reservations are a real threat and need to be policed. A scalable policing mechanism to allow routers to identify non-conformant flows based on certain heuristics is the subject of ongoing research; an architecture and preliminary results are presented in section 2.4. Such a mechanism can be combined with more traditional approaches, e.g. policing of individual flows at network edges.

2.2 Architecture overview

The proposed architecture uses two protocols to manage reservations: a reservation protocol (sections 2.2.1 and 2.2.2) to establish and maintain them, and a feedback protocol (sections 2.2.3 and 2.2.4) to inform the sender about the reservation status.

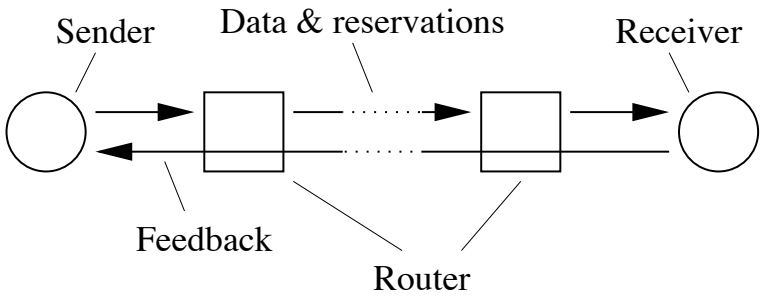

Figure 2.1: Overview of the components in SRP.

Figure 2.1 illustrates the operation of the two protocols:

- Data packets with reservation information are sent from the sender to the receiver. The reservation information consists in a packet type which can take three values, one of them being ordinary best-effort. Routers process

this information to control reservation increases, and to estimate the effective resource usage.

- The receiver sends feedback information back to the sender. Routers only forward this information; they do not need to process it.

2.2.1 Reservation protocol

The reservation protocol is used in the direction from the sender to the receiver. It is implemented by the sender, the receiver, and the routers between them. As mentioned earlier, the reservation information is a packet type which may take three values:

Request This packet is part of a flow which is trying to gain *reserved* status. Routers may accept, degrade or reject such packets. When routers accept some *request* packets, then they commit to accept in the future a flow of reserved packets at the same rate. The exact definition of the rate is part of the estimator module.

Reserved This type identifies packets which are inside the source's profile and are allowed to make use of the reservation previously established by *request* packets. Given a correct estimation, routers should never discard *reserved* packets because of resource shortage.[2]

Best-effort No reservation is attempted by this packet.

Packet types are initially assigned by the sender, as shown in figure 2.2. A traffic source (e.g. the application) specifies for each packet if that packet needs a reservation. If no reservation is necessary, the packet is simply sent as *best-effort*. If a reservation is needed, the protocol entity checks if an already established reservation at the source covers the current packet. If so, the packet is sent as *reserved*, otherwise an increase of the reservation is requested by sending the packet as *request*.

[2]A router will typically perform acceptance checks also on *reserved* packets, e.g. as protection from failures and for policing.

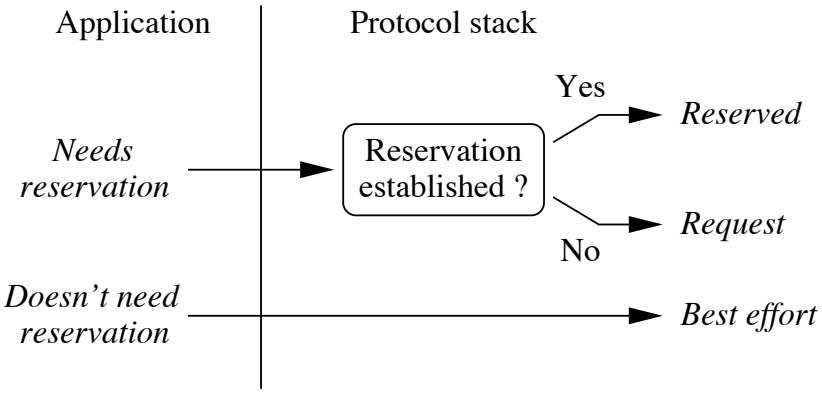

Figure 2.2: Initial packet type assignment by sender.

Each router performs two processing steps (see also figure 2.3). First, for each *reserved* packet the estimator updates its current estimate of the resources used by the aggregate flows. For each *request* packet, the router decides whether to accept the reservation increase, considering the resource estimate (packet admission control). Then, packets are processed by various schedulers and queue managers inside the router.

- When a *reserved* packet is received, the estimator updates the resource estimation. The packet is forwarded unchanged to the scheduler where it will have priority over best-effort traffic and normally is not discarded.

- When a *request* packet is received, then the estimator checks whether accepting the packet will not exceed the available resources. If the packet can be accepted, its *request* label is not modified and the resource estimate is changed accordingly. If the packet cannot be accepted, then it is degraded, i.e. its type it changed to *best-effort* or it is discarded.

- If a scheduler or queue manager cannot accept a reserved or request packet, then the packet is either discarded or downgraded to *best-effort*.

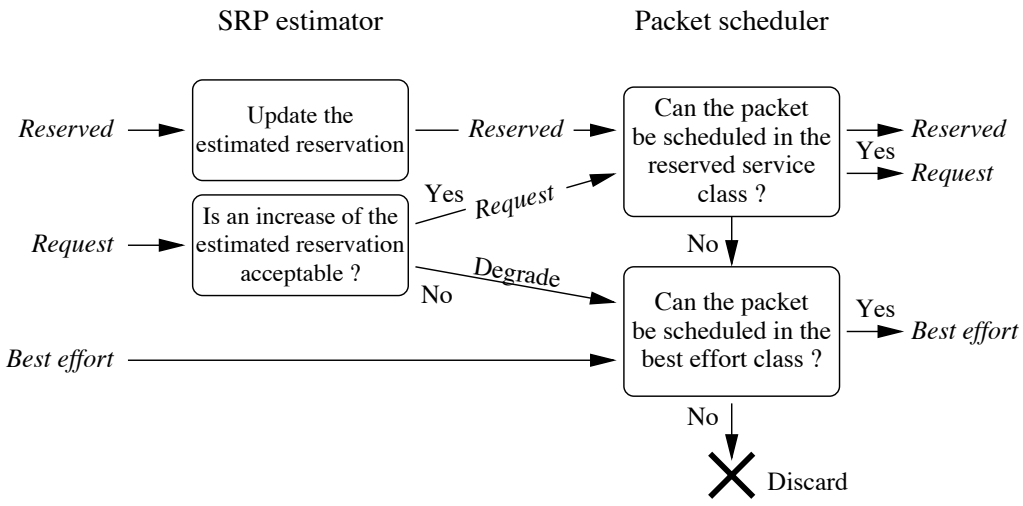

Figure 2.3: Packet processing by routers.

2.2.2 Packet type encoding

RFC2474 [26] defines the use of an octet in the IPv4 and IPv6 header for Differentiated Services (DS). This field contains the DS Code Point (DSCP), which determines how the respective packet is to be treated by routers (Per-Hop Behaviour, PHB). The DS field is ideally suited for expressing the SRP packet types. For SRP use, it is of particular importance that the DS architecture allows routers to change the content of a packet's DS field (e.g. to select a different PHB).

As illustrated in figure 2.4, SRP packet types can be expressed by introducing two new PHBs (for *request* and for *reserved*), and by using the predefined DSCP value 0 for best-effort. DSCP values for *request* and *reserved* can be allocated locally in each DS domain.

The suggested default DSCP values are **0x1b** for *request* and **0x1f** for *reserved*. Other values may be used as long as the following constraints are met:

- the values must be in one of the experimental/local use pools defined in [26]: **xxxx11** binary and **xxxx01** binary.
- they should occupy adjacent code points in the respective pool
- they must be smaller than 0x50 (otherwise, privileges are required to set them

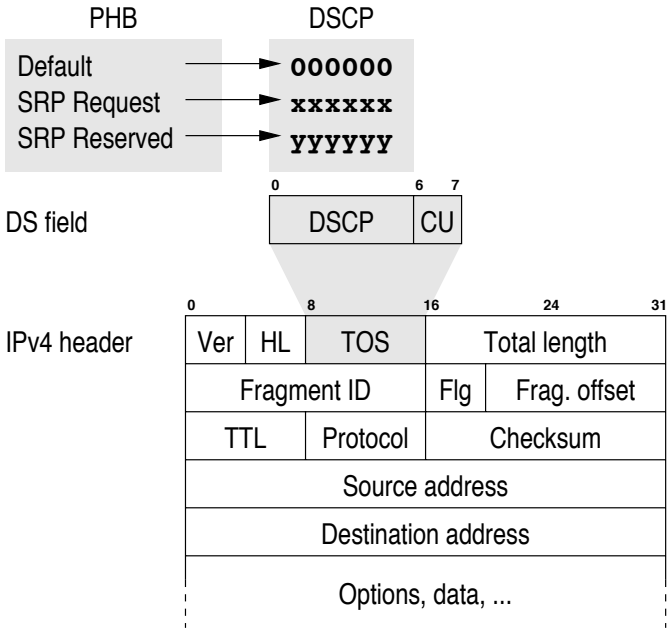

Figure 2.4: Packet type encoding using Differentiated Services (IPv4 example).

at the source using the `IP_TOS` socket option)

Note that, within the limitations described in RFC2475 [19], the reservation protocol may "tunnel" through routers that do not implement reservations. This allows the use of unmodified equipment in parts of the network which are dimensioned such that congestion is not a problem.

2.2.3 Feedback protocol

The feedback protocol is used to convey information on the success of reservations and on the network status from the receiver to the sender. Feedback information is collected by the receiver and it is sent directly to the sender. Unlike the reservation protocol, the feedback protocol does not need to be interpreted by routers.

The feedback contains the cumulative number of bytes in *request* and *reserved* packets that have reached the receiver, and the local time at the receiver at which the feedback message was generated.

Receivers collect feedback information independently for each sender and senders

maintain the reservation state independently for each receiver. Note that, if more than one flow to the same destination exists, attribution of reservations is a local decision at the source.

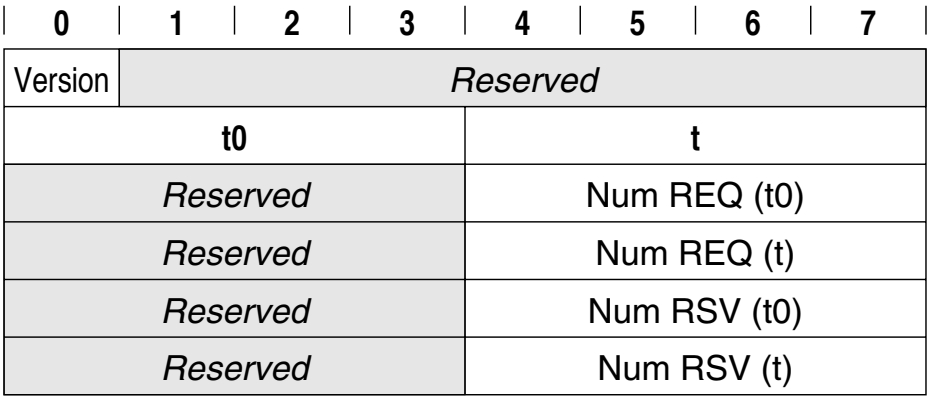

Figure 2.5: Feedback message format.

Figure 2.5 illustrates the content of a feedback message: the time when the message was generated (t, measured in microseconds), and the number of bytes in *request* and *reserved* packets received at the destination (REQ and RSV). All counters can start with arbitrary values and they wrap back to zero when they overflow.

Feedback messages are transported over UDP [27]. The default port number to which to send feedback is 4360 decimal.[3] Feedback receivers should ignore all messages originating from a port different from 4360 as a basic measure against tampering. The octets in figure 2.5 are shown in the order of transmission. In each field, the most significant octet is transmitted first. The version number field must be set to one by the sender of a feedback message. Feedback receivers implementing version one of the protocol must ignore all messages with version 0, and process all messages with version one or above.

Feedback messages received out of sequence are ignored. In order to improve tolerance to packet loss, also the information sent in the previous feedback message (at time t0) is repeated. Portions of the message are reserved to allow for future

[3]A request for a registered user port assignment by IANA is pending.

extensions.

2.2.4 Feedback scheduling

Feedback messages are not always sent at the same rate. Instead, the destination generates them only when there is useful information to report, e.g. when *request* packets reach the destination or when the reserved rate changes. We suggest the following algorithm for deciding when to send a feedback message:

We define t as the current time, t_0 as the time when the last feedback was sent, $\Delta t := t - t_0$, ΔN_{REQ} as the number of bytes in *request* packets received since t_0, $R(s)$ as the estimated reserved rate at time s, T_M as the minimum interval between two feedback messages, T_I as the largest interval in the absence of changes, T_R as the largest interval during reservation ramp-up, ϑ as the maximum change in the reserved rate for which no notification of the source is necessary, and N_{thres} as the maximum number of *request* bytes that can be received before a feedback needs to be generated.

Generally, we only send feedback if $\Delta t \geq T_M$. If $\Delta N_{REQ} = 0$, we send feedback if $\Delta t > T_I$ or $\frac{|R(t)-R(t_0)|}{R(t)} > \vartheta$. If $\Delta N_{REQ} > 0$, we send feedback if $\Delta t > T_R$ or $\Delta N_{REQ} > N_{thres}$.

2.2.5 Congestion control

In order to avoid interfering with congestion-controlled traffic (e.g. TCP) in an unfair way [28], and to prevent starvation among competing reservations, SRP senders must respond to congestion by limiting the rate at which they send *request* packets.

A mechanism similar to TCP congestion control can also be used with SRP: the sender maintains a congestion window, which grows while *request* packets pass the network unharmed, and which shrinks when congestion is detected. Congestion is assumed when *request* packets are lost. Their loss is detected by comparing the number of *request* and *reserved* bytes indicated in feedback messages with the values expected by the sender.

Congestion of *reserved* traffic (e.g. due to a partial network failure) is detected when the rate obtained from feedback messages is significantly below the rate expected by the sender (we assume that small decreases or apparent increases of the reserved are caused by jitter and ignore them). In this case, the sender lowers its own estimate to the indicated rate. A complete communication failure is assumed if no feedback is received while several request timeouts occur. In this case, the SRP component in the sender resets the reservation to zero and notifies its user.

Recently, the concept of a centralized congestion manager has been suggested in IETF. Such a component would be highly desirable for SRP, because SRP has no easily accessible two-way communication path from which congestion information could be obtained. If applications layered on top of SRP report their own congestion experience to a centralized component, SRP could directly benefit from this.

2.2.6 Example

Figure 2.6 provides the overall picture of the reservation and feedback protocols for two end-systems connected through routers $R1$ and $R2$. The initial resource acquisition phase is followed by the generation of *request* packets after the first feedback message arrives. Dotted arrows correspond to degraded *request* packets, which passed the admission control test at router $R1$ but could not be accepted at router $R2$ because of resource shortage. Degradation of *request*s is taken into account by the feedback protocol. After receiving the feedback information the source sends *reserved* packets at an appropriate rate, which, in this case, is smaller than the one at which *request* packets were generated.

2.2.7 Multicast

The key strength of SRP is to provide scalable reservations for unicast flows. Nevertheless, we consider it important for SRP to also support multicast traffic. To this end, we propose a design that slightly extends the reservation mechanism described above. Refinement of this design is still the subject of ongoing work. Additional details on the proposed mechanism can also be found in [29].

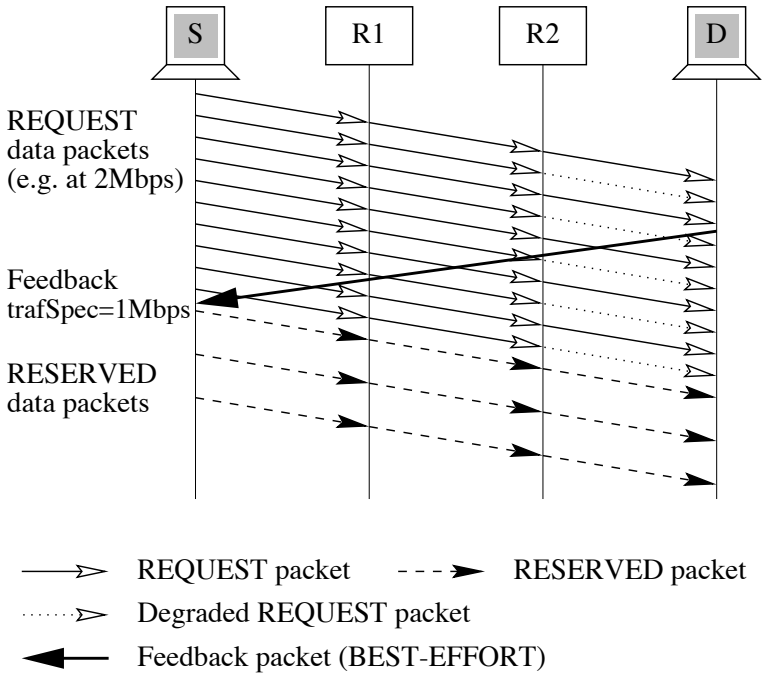

Figure 2.6: Reservation and feedback protocol diagram.

The extensions concern the feedback and the reservation protocol at the source. They are needed to cope with several problems which are typical in a multicast environment:

- the joining mechanism: how to establish reservations to a new group member without affecting the reservation already in place;
- transparency: events like route instability, topology changes, joining and leaving of some group members and situations like heterogeneous connectivity should only affect their limited scope, i.e. they should be transparent to the remaining session members.
- feedback implosion: the feedback protocol which works well in a unicast scenario does not scale well in a multicast environment.

Establishing reservations in a multicast tree Members of a multicast session are divided into two sets:

1. joining members, forming the *request* multicast group;

2. "old" members, forming the *reserved* multicast group.

Receivers wishing to receive a multicast flow first join the request group. The join request is issued hop-by-hop toward a multicast router already on the reserved tree (or to the source). Routers already receiving reserved traffic start sending the multicast traffic to the new member after receiving the join request. In addition to that, they also switch the *reserved* flag to *request*. Members of the request group can compare their reservation estimate to the target amount indicated by the source. If the reservation offered is acceptable, then the member can leave the request group and join the reserved group. Figure 2.7 illustrates the group structure.

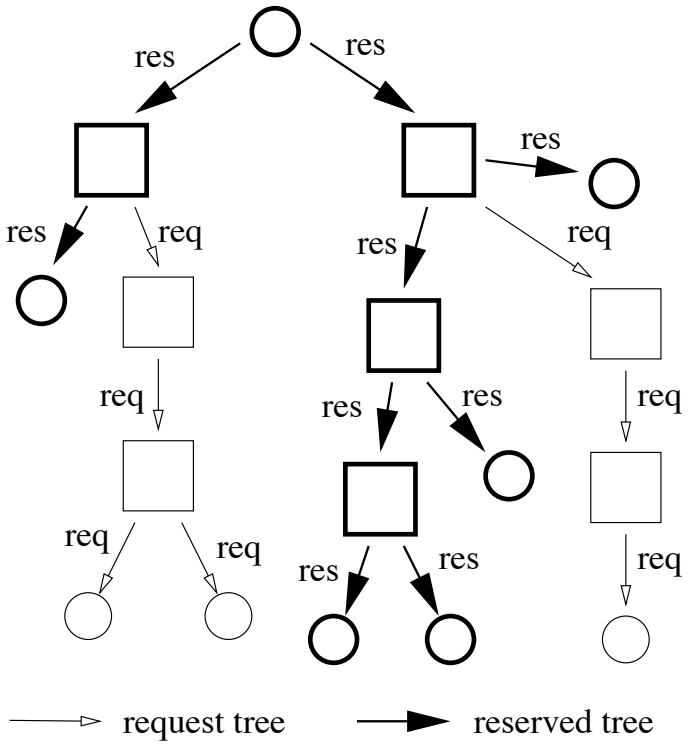

Figure 2.7: Request and reserved multicast group.

The algorithm executed by the multicast router when a multicast packet is received, is the following:

```
if ((packet_addr is multicast) and
        (packet_type == RES)) {
    forward packet to reserved group;
    if (router is in the request group) {
        newpacket = copy(packet);
        newpacket_type = REQ;
        forward newpacket to request group;
    }
}
```

Transparency In a network with bottlenecks the algorithm should avoid that the link with worst connectivity (e.g. with the lowest bandwidth availability) limits the reservation offered to each member of the group. To cope with this heterogeneity multicast members could be grouped into separate sets and layered coding [30] could be used.

Different coding layers representing different levels of quality are sent to different multicast groups. Typically, all the receivers are included in a common multicast tree for the distribution of the fundamental coding layer, and each member can join additional groups depending on the quality of its connectivity.

Feedback The problem of feedback implosion is solved by simply not sending any explicit feedback but by using group membership as an implicit indicator instead. The multicast source can fix an a priori value for the minimum amount of reservations required to forward the traffic of a given coding layer. After joining the request group the receiver does flow acceptance control. If the estimated reservation is acceptable compared to the target set by the source, then it can leave the request group and join the reserved, otherwise it leaves the request group and gives up.

2.3 Estimation modules

We call *estimator* the algorithm which attempts to calculate the amount of resources that need to be reserved. The estimation measures the number of *request*s sent by sources and the number of *reserved* packets which actually make use of the reservation.

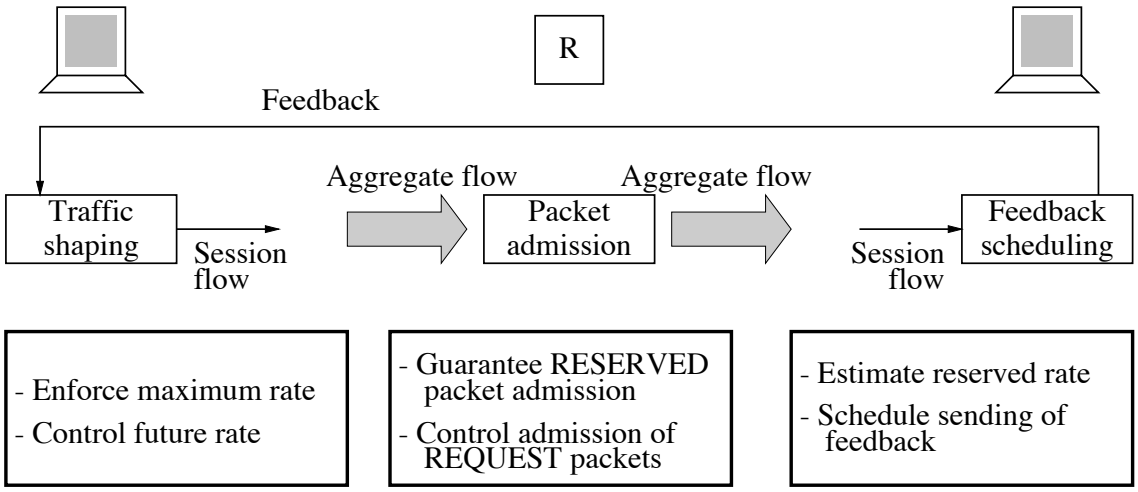

Figure 2.8: Use of estimators at senders, routers, and receivers

Estimators are used for several functions.

- Senders use the estimator for an optimistic prediction of the reservation the network will perform for the traffic they emit. This, in conjunction with feedback received from the receiver, is used to decide whether to send *request* or *reserved* packets.

- Routers use the estimator for packet-wise admission control and perhaps also to detect anomalies.

- In receivers, the (simple) estimator is fed with the received traffic and it generates a rough estimate of changes in the reservation. This is used to schedule the sending of feedback messages to the source.

Figure 2.8 shows how the estimator algorithm is used in all network elements.

Our architecture is independent of the specific algorithm used to implement the estimator. Sections 2.3.1 and 2.3.2 describe two different solutions. The definition and evaluation of algorithms for reservation calculation in hosts and routers is still ongoing work. A detailed analysis of the estimation algorithms and additional improvements can be found in [29].

2.3.1 Basic estimation algorithm

The basic algorithm we present here is suitable for sources and destinations, and could be used as a rough estimator by routers. This estimator counts the number of requests it receives (and accepts) during a certain *observation interval* and uses this as an estimate for the bandwidth that will be used in future intervals of the same duration.

In addition to requests for new reservations, the use of existing reservations needs to be measured too. This way, reservations of sources that stop sending or that decrease their sending rate can automatically be removed. For this purpose the use of reservations can be simply measured by counting the number of *reserved* packets that are received in a certain interval.

To compensate for deviations caused by delay variations, spurious packet loss (e.g. in a best-effort part of the network), etc., reservations can be "held" for more than one observation interval. This can be accomplished by remembering the observed traffic over several intervals and using the maximum of these values (step 3 of the following algorithm). Given a hold time of h observation intervals, the maximum amount of resources which can be allocated Max, res and req (the total number of *reserved* and *request bytes* received in a given observation interval), the reservation R (in bytes) is computed by a router as follows. Given a packet of n bytes:

```
if (packet_type == REQ)
    if (R + req + n < Max) {
        accept;
        req = req + n;   // step 1
    }
    else degrade;

if (packet_type == RES)
    if (res + n < R) {
        accept;
        res = res + n;   // step 2
    }
    else degrade;
```

where initially $R, res, req = 0$. At the end of each observation cycle the following steps are computed:

```
for (i = h; i > 1; i--) R[i] = R[i-1];
R[1] = res + req;
R = max(R[h],R[h-1],...,R[1]); // step 3
res = req = 0;
```

The same algorithm can be run by the destination with the only difference that no admission checks are needed.

Examples of the operation of the basic algorithm can be found in [31].

This easy algorithm presents several problems. First of all, the choice of the right value of the observation interval is critical and difficult. Small values make the estimation dependent on bursts of *reserved* or *request* packets and cause an overestimation of the resources needed. On the other hand, large intervals make the estimator react slowly to changes in the traffic profile. Then, the strictness of traffic acceptance control is fixed, while adaptivity would be highly desirable in order to make the allocation of new resources stricter as the amount of resources reserved gets closer to the maximum. These problems can be solved by devising an adaptive enhanced algorithm like the one described in the following section.

2.3.2 Enhanced estimation algorithm

Instead of using the same estimator in every network component, we can enhance the previous approach so that senders and receivers still run the simple algorithm described above, while routers implement an improved estimator.

We describe an example algorithm in detail below. It consists of the principal components illustrated in figure 2.9: the effective bandwidth used by *reserved* and accepted *request* packets is measured and then smoothed by calculating an exponentially weighted average (γ). This calculation is performed for every single packet.

The estimate γ is multiplied with a correction factor β in order to correct for systematic errors in the estimation. Packets are added to a virtual queue (i.e. a counter), which is emptied at the estimated rate. If the estimate is too high, the virtual queue shrinks. If the estimate is too low, the virtual queue grows. Based on the size of the virtual queue, β can be adjusted.

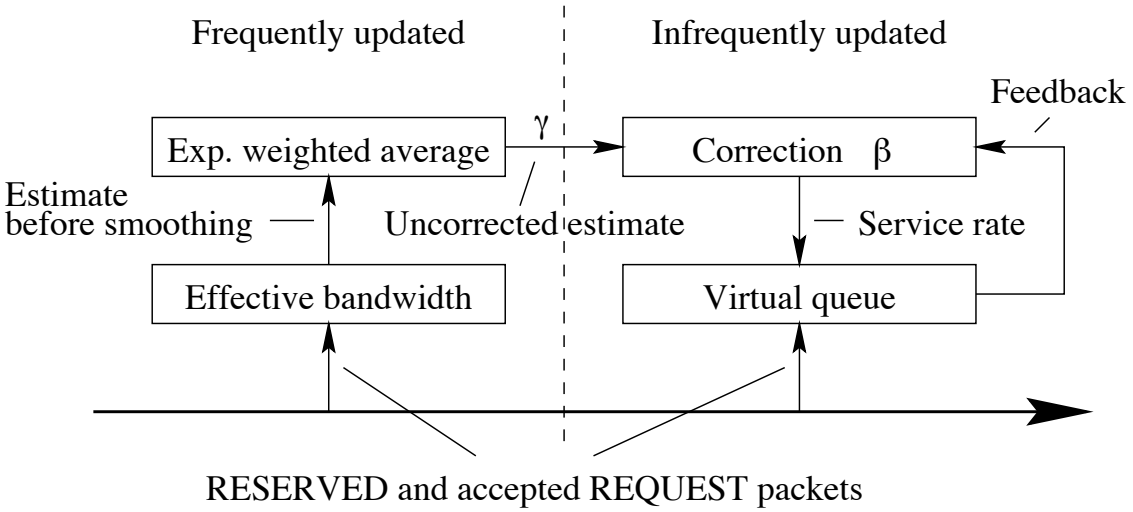

Figure 2.9: Schematic design of an adaptive estimator.

Packet admission control and low pass filter In order to filter out small scale traffic profile variations in a way close to the real node behavior, we borrow the concept of *deterministic effective bandwidth* from network calculus [32]. Given an arrival curve α and a delay bound D, the corresponding deterministic effective bandwidth e_D is defined as:

$$e_D = \sup_{s \geq 0} \frac{\alpha(s)}{s + D}$$

By applying this definition to our model and by assuming that observation starts at time 0, we obtain that:

$$e = \sup_{1 \leq i \leq j} \frac{n_i + \ldots + n_j}{t_j - t_i + D}$$

where $t_1, \ldots, t_k$ are the time instants at which packets arrive, n_i is the number of bytes in packet number i (only *reserved* or *request* packets are taken into account) and D is a fixed parameter: the delay objective.

e represents the bandwidth required for the flow with smoothed peaks, as packets are queued in a buffer system requesting a maximum queueing time of D. Since the traffic profile of a flow may change, the capacity estimated for a given flow should vary accordingly. To achieve this we estimate the effective bandwidth e at any

arrival time t_k of a *reserved* or *request* packet over a sliding window w:

$$e_k = \sup_{1 \leq i \leq j \text{ and } t_i, t_j \in [t_k - w, t_k]} \frac{n_i + \ldots + n_j}{t_j - t_i + D} \tag{2.1}$$

Then, in order to smooth out changes in e_k as a function of the packet rate of a flow we eventually calculate γ by taking the exponentially weighted average of e_k and we assume that the amount of bandwidth allocated by a router at time t_k per input and output port is equal to γ:

$$\gamma_k = \alpha^d \gamma_{k-1} + \left(1 - \alpha^d\right) e_k \tag{2.2}$$

where α is a parameter such that $0 < \alpha < 1$, and $d = t_k - t_{k-1}$. α^d is the weight, which depends on the time between packets. Parameters α and w define the behavior of the low pass filter, in particular the resource release process of the estimator when a given flow stops, and the reservation keeping during temporary silences.

The packet admission procedure is devised in such a way that *reserved packets* are always considered in the estimator, while *request* packets have to pass an admission control test. If the *k-th* packet is *reserved*, then equations 2.1 and 2.2 are computed and the packet is accepted, even if it could be discarded later by the scheduler. On the other hand, if the packet type is *request*, the following test is applied:

$$\text{if } \gamma_k \beta \leq C_{max} \text{ then accept else refuse}$$

where C_{max} is a fixed parameter representing the maximum amount of bandwidth which can be reserved on a given output interface, and β is a correction factor computed according to the algorithm presented in the following paragraph. If the packet is accepted then the estimated bandwidth is updated, otherwise the packet is downgraded to *best-effort* and we let $\gamma_k = \gamma_{k-1}$ (i.e. if a packet is rejected by the admission test, its arrival is ignored by the estimator).

Adaptivity in packet admission control Adaptivity in packet admission control is obtained by making parameter β vary as a function of the number of reserved bytes lost. There are two independent variables: L_r, the number of reserved bytes *really* lost by the router, and L_v, the number of reserved bytes *virtually* lost as defined in formula 2.3.

L_r is the measure of real losses of *reserved* and (accepted) *request* packet, which occur when the amount of reserved traffic reaches the capacity C_{max}. We assume that L_r is counted over intervals $(t-\theta, t]$ (see below).[4]

In order to tune β before *reserved* traffic reaches the capacity C_{max}, we calculate at each packet arrival the maximum buffer occupancy L_v, counted in bytes, of a virtual queue served at rate $\gamma\beta$ (the current estimate of the bandwidth required by the flow), and the maximum virtual queue size L_v^{max}:

$$L_v := \max(0, L_v + n_k - \gamma_{k-1}\beta(t_k - t_{k-1})) \tag{2.3}$$

$$L_v^{max} := \max(L_v^{max}, L_v)$$

where n_k is the size of the current packet, $t_k - t_{k-1}$ is the time since the previous packet was received, and γ_{k-1} is the value of γ computed after the last packet reception. The initial values of L_v and L_v^{max} are 0.

If our estimation procedure is correct, we should have $L_v^{max} \leq \gamma\beta D$, otherwise we need to increase the value of β. Conversely, if L_v^{max} is very small, then we have to decrease β.

To determine how to change β, we use L_v^{max} to calculate the rate $\gamma\beta'$ at which we have to serve the virtual queue to reach the length corresponding to the delay goal D at the present rate $\gamma\beta$:

$$\gamma\beta' = \gamma\beta + \frac{L_v^{max} - \gamma\beta D}{B^{-1}}$$

or

$$\beta' = \beta + B\left(\frac{L_v^{max}}{\gamma} - \beta D\right)$$

where B^{-1} is the time after which the length goal should be reached. β is updated with period θ as follows:

$$\beta := \beta \underbrace{+ A\frac{L_r}{N_r}}_{\text{if } L_r>0} + B\left(\frac{L_v^{max}}{\gamma} - \beta D\right) \tag{2.4}$$

where N_r is the amount of data received in *reserved* and (accepted) *request* packets since the last update of β, L_r is the amount of such data lost in the same

[4]The actual measurement and filtering method for L_r is argument of further study.

interval, γ is the current bandwidth estimate, and A and B are fixed parameters to be tuned by simulation. The initial value of β is 1. L_v^{max} is reset to L_v after computing (2.4).

The possibility to make β a function of the rejection rate of *request* packets and the tuning of the parameters used in the algorithms described above, are arguments for future work.

2.4 Policing

In the previous sections, we assumed that network elements operate in conformance with the described behaviour. We will now describe some general aspects of the non-conformance we expect to experience in real networks and we propose a scalable approach for protecting the network and its users from the effects of non-conformant traffic.

We distinguish two main types of non-conformance: (1) use of resources without prior permission (theft of service), and (2) denial of service. Both types of non-conformance have in common that traffic for which no adequate reservation exists is added to the network. Policing must therefore be based on the comparison of the current traffic with the expected traffic. If the former exceeds the latter, packets not belonging to conformant flows must be identified and counter-measures can be performed, e.g. offending packets can be discarded.

2.4.1 SRP constraints

The architecture of SRP imposes a few constraints on a policing solution: (1) reservations can decrease along the path, (2) no per-flow information is available, (3) only the addition of new reserved traffic is announced, but not its removal.

The first constraint stems from the fact that each router sees only the reservation requests that have been accepted by upstream nodes, and it can either accept the whole reservation or only a part of it. Upstream routers may therefore initially accept more of a reservation than downstream routers. A source sending request packets at a rate R and following them up with reserved packets at the same rate

may therefore go undetected at upstream routers, even if the bandwidth along the entire path (i.e. the reservation obtained at the bottleneck router) is below R.

Note that reserved traffic and feedback packets may use different paths. Reservations can therefore not simply be verified by examining feedback traffic (which indicates the minimum reservation obtained along the path). However, in some cases, it may be possible to restrict the choice of possible routes in order to enable also this type of policing.

The second constraint implies that traditional policing approaches, which are based on measuring each individual flow, cannot generally be applied for SRP without sacrificing its scalability.

The third constraint limits our ability of detecting non-conformance of an aggregate without additional information on its internal structure: when sources stop sending, their resources are released implicitly by measuring the overall reduction of the reserved traffic volume. If non-conformant sources inject traffic that is equivalent in volume to the traffic previously generated by now silent sources, the aggregate does not seem to change, and the abuse may go undetected.

2.4.2 Three complementary approaches

We consider three approaches for policing. First, at the boundary to access networks, monitoring individual flows may be the most economic approach, because the number of flows will typically be small at such points, and it is comparably easy to route forward and backward traffic through the policing point.

The second approach applies the same principles, but works on aggregates instead. Both, reserved traffic and feedback messages are examined and policed. This way, the policing entity can ensure that only the accepted amount of traffic passes. Heuristics on the internal structure of the aggregate, as discussed below, can be used to ensure that not only the volume but also the paths reserved traffic takes correspond to the reservation.

Finally, we propose a heuristic approach that works by capturing certain characteristics of an aggregate and comparing new traffic against these characteristics. If the traffic volume exceeds certain bounds, which we assume to be caused by

non-conformant traffic, packets are classified by their similarity to the stored characteristics. Packets are discarded if they are not similar to the expected traffic, or if they are similar to excess traffic. We describe the heuristic approach in more detail below.

2.4.3 Architecture of the heuristic approach

Our approach is based on the observation that packets belonging to conformant flows will exhibit certain characteristics that can be used to detect them with a certain probability. Likewise, probable characteristics of packets belonging to non-conformant flows can be determined, and incoming packets can be matched against those characteristics.

We identify each flow by a unique bit pattern in the IP headers of packets belonging to it, e.g. the source and the destination address. We call this the *packet signature*.

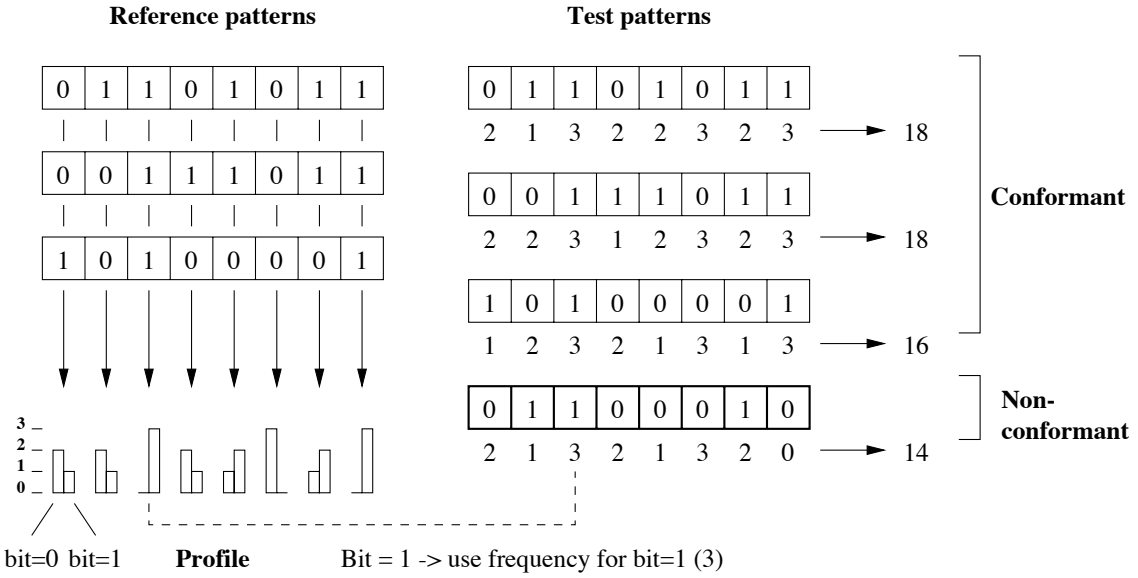

Figure 2.10: The frequency of bit patterns.

The characteristic we are using is the frequency of bit patterns in those signatures. Figure 2.10 illustrates a case where three conformant flows generate a profile of their bit pattern frequency, and then bit patterns of the three conformant flows

and one non-conformant flow are compared with this profile. The comparison is done by adding the frequencies with which the bits in the pattern were observed.

We notice that the patterns used for calculating the profile obtain a higher ranking, indicating their higher probability of being among the reference patterns.

2.4.4 General information flow

Figure 2.11 shows the general architecture. First, we measure if the current traffic exceeds the expected traffic by some tolerance margin (1). This will determine how aggressively we try to detect and to eliminate non-conformant traffic.

In parallel, we calculate the probability that the current packet belongs to a conformant ("good") or non-conformant ("bad") flow. We do this by comparing it with the profiles obtained for supposedly conformant packets (2), and for packets suspected to belong to non-conformant flows (3).

In order to make the calculation independent from the actual bit patterns occurring in signatures, we expand the packet signature to a larger vector using a hash function (4). The hash function is keyed with a random number only known to the router. It is therefore difficult if not impossible for an attacker to construct packets in a way that their signatures will have a predictable relation to the signatures of packets belonging to specific other flows.

The probabilities for conformance and non-conformance are then added (5) and a decision is made whether the packet in question should be accepted or not (6), and, depending on this decision, the packet is either enqueued for forwarding or it is discarded (7).

In parallel, a decision is made if the packet should be counted for the next estimate of the profile of conformant flows (8). After a certain time interval, the profile of accepted packets is copied to the profile of conformant flows, and the difference between the previous profile of conformant flows and the current profile of all received packets is used as the next prediction for non-conformant flows (9).

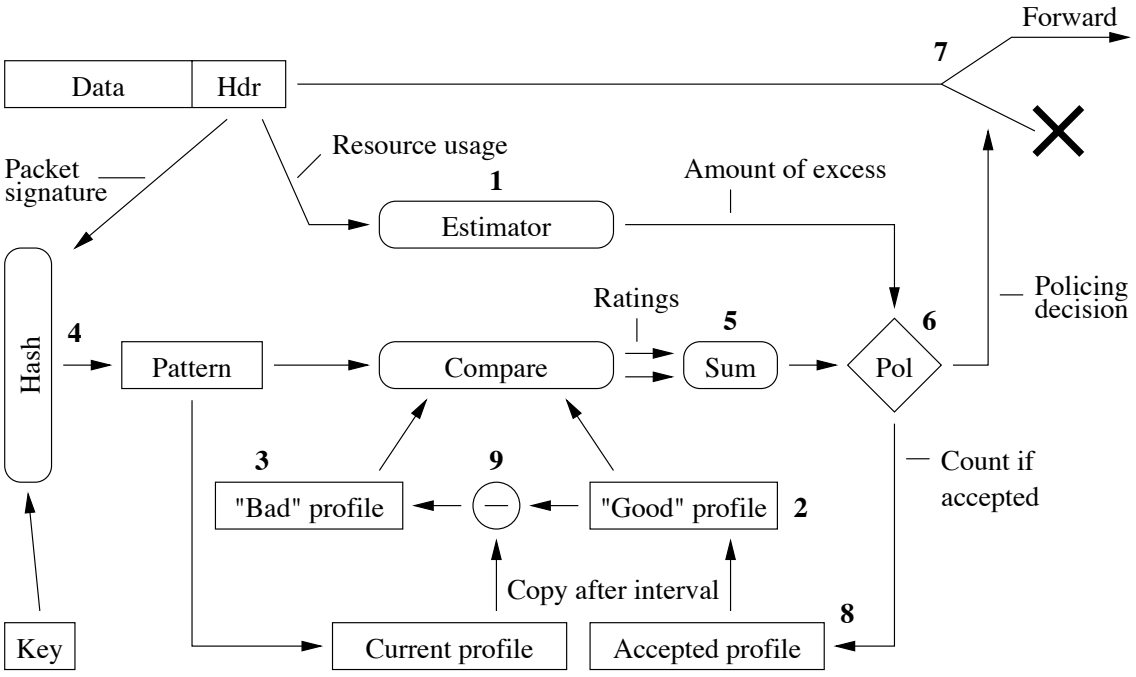

Figure 2.11: Policing architecture overview.

2.4.5 Formal description

The signature of each incoming packet is hashed to a bit vector w of length d. Then we count the relative frequency of ones at each bit position in a set W of such vectors:

$$p_i(W) = \frac{1}{|W|} \sum_{w \in W} w_i$$

where w_i is the value of the ith bit. We call $p(W)$ the *profile* of W.

We define a discriminant function $g(W, w)$ to reflect the probability that a vector w belongs to a set of vectors W as

$$g(W, w) = \frac{1}{d} \sum_{i=1}^{d} x_i \text{ with } x_i = w_i p_i(W) + (1 - w_i)(1 - p_i(W))$$

We assume that the d components of the vectors in W are independent Bernoulli variables. The probability that the event $\{w_i = 1\}$ occurs exactly k times in the $|W|$ vectors is binomial. The statistical properties of the discriminant function can

be determined. Using

$$\begin{aligned} P\left(x_i = \frac{k}{|W|} \middle| w \notin W\right) &= \binom{|W|}{k} \frac{1}{2^{|W|}} \\ P\left(x_i = \frac{k}{|W|} \middle| w \in W\right) &= \frac{2k}{|W|} P\left(x_i = \frac{k}{|W|} \middle| w \notin W\right) \end{aligned}$$

$(1 \leq k \leq |W|)$, we obtain the mean and the variance of the probability distribution of $g(W, w)$:

$$\begin{array}{llll} \mu_{w \in W} = \frac{1}{2} + \frac{1}{2|W|} & \sigma^2_{w \in W} = \frac{1}{4d|W|} + \frac{1}{2d|W|^2} - \frac{3}{4d^2|W|^2} \simeq \frac{1}{4d|W|} & \text{for } w \in W \\ \mu_{w \notin W} = \frac{1}{2} & \sigma^2_{w \notin W} = \frac{1}{4d|W|} & \text{for } w \notin W \end{array} \tag{2.5}$$

where the approximation is valid for $|W| \gg d$.

We measure the profile of actual reserved traffic over short time intervals. In interval T_j, given the profile $p(A)$ of conformant traffic (which is initially obtained from accepted requests), and the measured traffic M_{j-1} of the previous interval T_{j-1}, we can determine the profile of non conformant traffic B as $p(B) = p(M_{j-1}) - p(A)$. We can now determine the function $g(A, w)$, which represents the probability that a new packet with vector w is part of the conformant traffic A, as well as the function $g(B, w)$, the probability of it to be part of traffic that was previously found to be in excess of the agreed upon reservation.[5]

We accept packets if

$$z(w) > \text{threshold, with } z(w) = g(A, w) - g(B, w)$$

Using formula 2.5 we can approximate the condition for obtaining good separation of the probability distributions of $g(A, w)$ and $g(B, w)$:

$$d > 4\frac{|A| \cdot |B|}{|A| + |B|}$$

A full paper covering the mathematical background of the proposed approach is in preparation. The development of a more comprehensive model that also takes into account effects like rate decreases, jitter, etc., and the study of applicability of this approach to other aggregation-based architectures than SRP, are the subject of ongoing research.

[5]For simplicity, we omit reduction of the traffic volume in this discussion.

2.5 Conclusion

We have proposed a new scalable resource reservation architecture for the Internet. Our architecture achieves scalability for a large number of concurrent flows by aggregating flows at each link. This aggregation is made possible by delegating certain traffic control decisions to end systems – an idea borrowed from TCP. Reservations are controlled with estimation algorithms, which predict future resource usage based on previously observed traffic. Furthermore, protocol processing is simplified by attaching the reservation control information directly to data packets.

In this chapter, we described the general concepts, gave examples for implementations of core elements, including the design of estimator algorithms for sources, destinations and routers, and we introduced a policing framework. In the next chapter, we will focus on simulations. The remaining chapters of this book focus on implementation issues, which finally lead to a prototype of SRP.

Chapter 3

Simulation

We have implemented a model of SRP in the UCB/LBNL/VINT network simulator `ns` [33]. In this section, first describe the implementation, and then we show the network behaviour obtained by simulating a large number of SRP sources sending over a bottleneck link.[1]

3.1 Simulator implementation

This section gives a brief overview of the structure of the SRP-related extensions to `ns-2`, and their use in simulation scripts.

Figure 3.1 illustrates the interaction of the extensions with each other and with existing parts of `ns`. SRP-specific extensions are shown in dark grey.

3.1.1 SRP agents

In `ns`, applications generating or receiving traffic are represented by so-called agents. SRP uses two agents: the `SRPAgent` (`srpagent.cc`), which controls traffic generation and reception of feedback, and the `SRPSink` (`srp-sink.cc`), which acts as the destination of SRP traffic. Both agents provide only the framework for SRP and feedback traffic and leave packet type selection and scheduling to the estimators.

[1] The changes to `ns-2` and the configuration scripts for the simulation are available on `http://icawww1.epfl.ch/srp/`

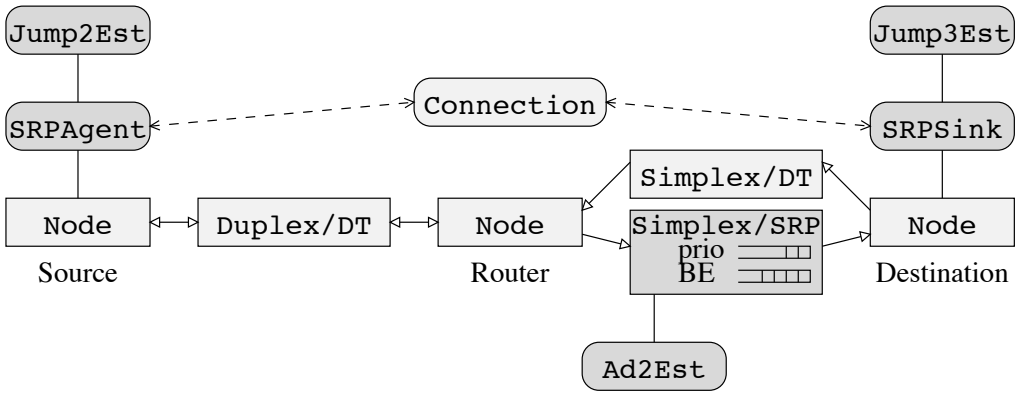

Figure 3.1: SRP-related components in the `ns` simulator.

`SRPAgent` has a built-in constant rate traffic generator. Optionally, a more advanced traffic generator can be registered. That generator can also access the estimator module and may therefore adapt traffic generation to the available reservation.

3.1.2 SRP estimators

All of the "intelligence" of the SRP simulator is in the estimators. They control packet admission, perform rate estimation, schedule feedback emission, and process incoming feedback messages.

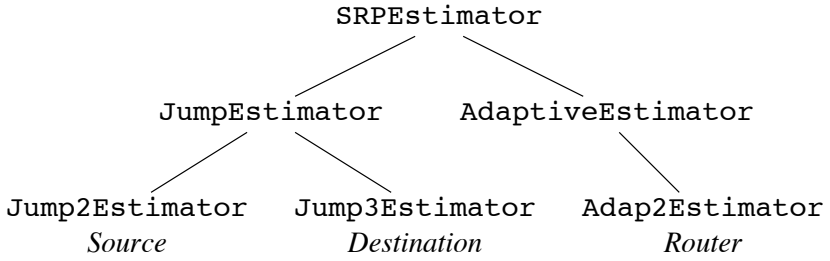

Figure 3.2: Class hierarchy of SRP estimators.

Figure 3.2 shows the class hierarchy of SRP estimators: `JumpEstimator` (`srp_jumpest.cc`) implements the basic algorithm described in section 2.3.1, and is divided into the more specialized sub-classes `Jump2Estimator` (`srp_jump2.cc`; only

for the SRP source), and `Jump3Estimator` (`srp_jump3.cc`; only for the SRP destination).

`AdaptiveEstimator` (`srp_adest.cc`) and `Adap2Estimator` (`srp_ad2.cc`) implement the adaptive estimation algorithm described in section 2.3.2.[2] `Adap2Estimator` reduces the overly expensive calculation of the effective bandwidth to a simple average over a fixed time interval, which yields equally good results in our simulations.

The parent class `SRPEstimator` (`srp_estimator.cc`) defines, among others, the following methods for SRP estimators:

`admit` Decide whether to admit a packet with its current type. If the packet is not admitted, the caller may change the packet type (e.g. from *reserved* to *request*) and try again. `admit` returns a non-zero value if the packet is admitted, zero otherwise.

`packet` Count the specified packet. This function is invoked exactly once for each packet as the packet is passed to the queue.

`loss` Loss of the specified packet, because the queue was full. Note that `loss` is invoked after `packet`.

`get_feedback` Obtain the feedback information gathered by the estimator. This method needs to implemented only by estimators used at the destination.

`set_feedback` Process incoming feedback. This method needs to be implemented only by estimators used at the sender.

`Jump3Estimator` can optionally be compiled such that it pretends successful reception also of *request* packets which have been degraded. This causes the source to generate non-conformant traffic, which still reacts to congestion and is therefore not easily detected. This extension was used to generate traces of non-conformant traffic for evaluation of the statistic classification described in section 2.4.

[2]Note that, for historical reasons, the division of code between `AdaptiveEstimator` and `Adap2Estimator` is somewhat arbitrary, and that `Adap2Estimator` overrides most methods of `AdaptiveEstimator`.

3.1.3 SRP queue

`SRPQueue` (`srpqueue.cc`) implements a queue which consists internally of two drop-tail queues, of which one is served with higher delay priority than the other. This corresponds to the simplest way of implementing the isolation of *reserved* SRP traffic from other traffic in a router.

`SRPQueue` invokes the estimator to decide whether *request* packets should be admitted, and it also notifies the estimator when packets are enqueued or lost.

In order to test the stability of estimations in the presence of jitter in the network, a jitter generator was added to the SRP queue. A detailed description of this can be found in [34]. This extension was used to ensure that the estimation of *reserved* traffic at the source does not drift due to jitter.

3.1.4 Example

The following `ns` script sets up the configuration shown in figure 3.1:

```
set ns [new Simulator]
```

Create a new instance of the simulator.

```
set S [$ns node]
set D [$ns node]
set R [$ns node]
```

Create the source, destination, and router node.

```
$ns duplex-link $S $R 10Mb 15ms DropTail
```

Link the source to the router with a bidirectional 10 Mbps link.

```
$ns simplex-link $D $R 5Mb 15ms DropTail
$ns simplex-link $R $D 5Mb 15ms SRP
set srpqueue [[$ns link $R $D] queue]
[$srpqueue get-priority-queue] set limit_ 150
[$srpqueue get-best-effort-queue] set limit_ 1000
set est [new SRPEstimator/Adaptive/Adap2]
$est set cmax 0.8
$srpqueue estimator $est
```

Create the link between the router and the destination. In the forward direction, this link uses SRP. In the backward direction, it has a normal drop-tail queue. The two SRP queues are limited to 150 and 1000 packets, respectively. Then an estimator is attached and the maximum bandwidth allocation for reserved traffic is set to 80% of the link bandwidth (`cmax`).

```
set src [new Agent/SRP]
$ns attach-agent $S $src
$src estimator [new SRPEstimator/Jump/hola]
```

An SRP agent is attached to the source and its estimator is set to `Jump2Estimator`.

```
set dst [new Agent/SRPSink]
$ns attach-agent $D $dst
$dst estimator [new SRPEstimator/Jump/dest]
```

Likewise for the destination.

```
$ns connect $src $dst
```

Source and destination agent are connected.

```
$src set packet-size 1000
$src set target-rate 1000
```

The target rate of the source is set to 1000 packets per second. Each packet has a size of 1000 bytes.

```
$ns at 0 "$src start"
$ns at 9 "$src stop"
$ns at 10 "exit 0"
$ns run
```

The source sends for nine seconds. One second later, the simulation ends.

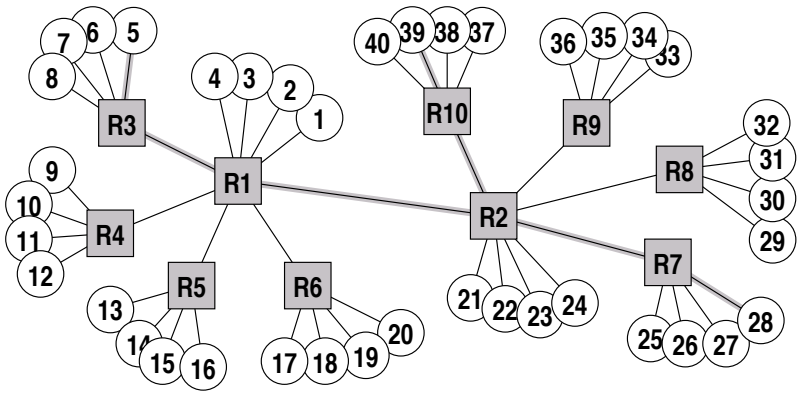

Figure 3.3: Configuration of the simulated network.

3.2 Simulation scenario

The network configuration used for the simulation is shown in figure 3.3. The grey paths mark flows we examine below.

There are ten routers (labeled **R1**...**R10**) and 40 hosts (labeled **1**...**40**). Each of the hosts **1**...**20** tries occasionally to send to any of the hosts **21**...**40**. Connection parameters are chosen such that the average number of concurrently active sources sending via the **R1**–**R2** link is approximately 150. Flows have an on-off behaviour, where the on and off times are randomly chosen from the intervals $[5, 20]$ and $[0, 30]$ seconds, respectively. The bandwidth of a flow remains constant while the flow is active and is chosen randomly from the interval $[1, 150]$ packets per second.

All links in the network have a bandwidth of 10'000 packets per second and a delay of 15 ms.[3] We allow up to 90% of the link capacity to be allocated to reserved traffic. The link between **R1** and **R2** is a bottleneck, which can only handle about 54% of the offered peak traffic. The delay objective D of the queue at **R1** is 10 ms. The queue size at the bottleneck link is limited to 150 packets (15 ms) for *reserved* and *request* packets.

[3]Small random variations were added to link bandwidth and delay to avoid the entire network from being perfectly synchronized. Also, some simplifications have been made in order to reduce simulation overhead: only **R1** performs acceptance control, feedback is sent at constant intervals, the estimator in **R1** only approximates the algorithm in [31], and congestion avoidance at sources uses a trimmed version of the mechanism described in section 2.2.5.

3.3 Simulation results

Figure 3.4 shows the **R1**–**R2** link as seen from **R1**. We show the total offered rate, and the smoothed actual rates of *request* and *reserved* packets. Figure 3.5 shows the queuing delay at **R1** for all three packet types. The system succeeds in limiting the queuing delay for *reserved* and *request* packets to less than the delay goal of 10 ms.

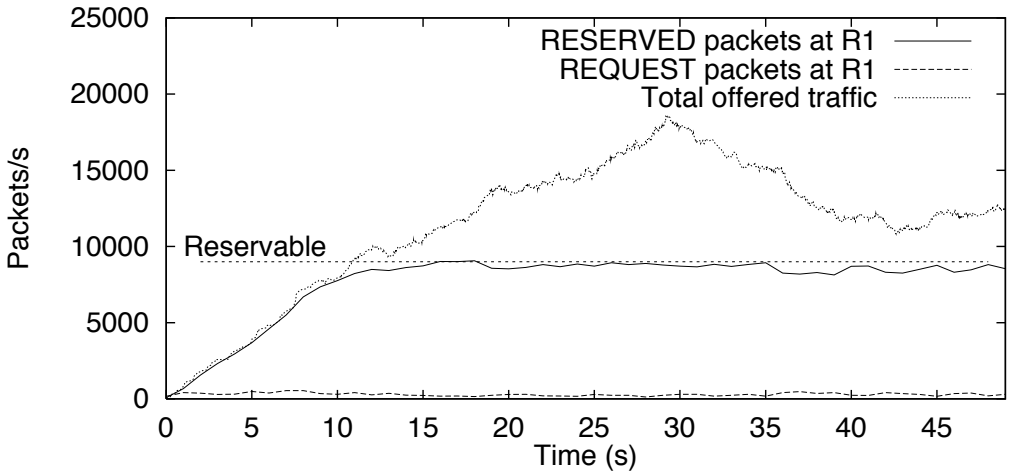

Figure 3.4: Traffic at **R1** towards **R2**.

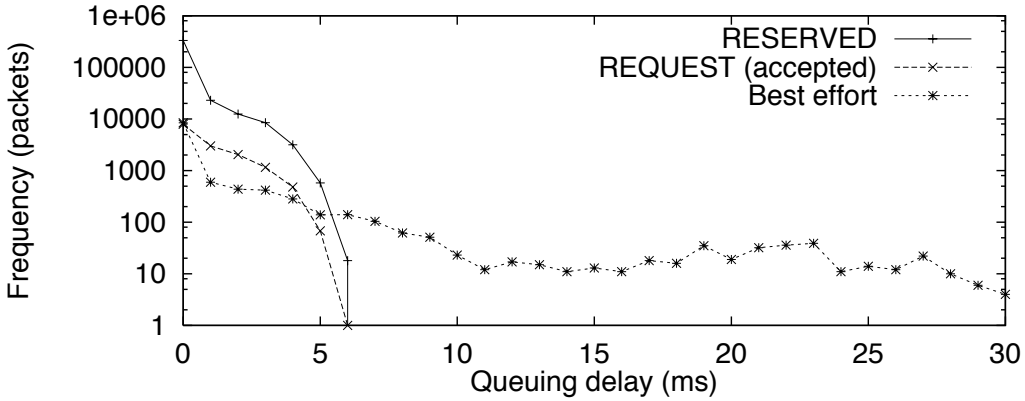

Figure 3.5: Queuing delay at **R1** on the link towards **R2**.

Finally, we examine some end-to-end flows. Figure 3.6 shows a successful reservation of 149 packets per second from host **5** to **39**. The requested rate, the rate of *request* packets sent, and the rate of *reserved* packets received are shown. Similarly, figure 3.7 shows the same data for a reservation host **5** attempts later to **28**, at a time when the offered traffic already exceeds the bandwidth available at the bottleneck. The reservation for 97 packets per second does not succeed initially, so the sender continues to send *request* packets at a reduced rate until the desired reservation is eventually obtained.

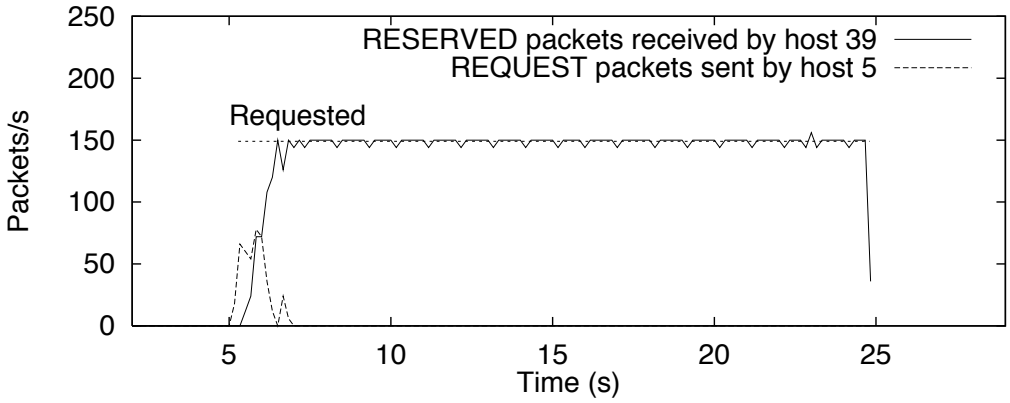

Figure 3.6: End-to-end reservation from host **5** to host **39**.

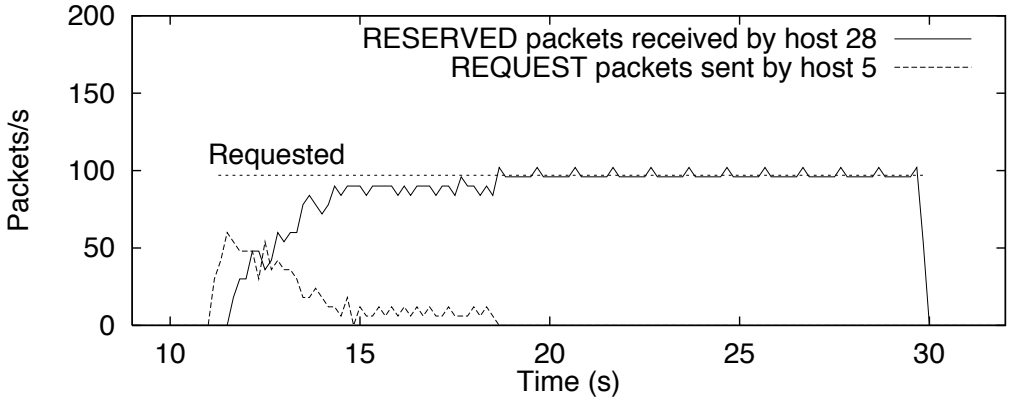

Figure 3.7: End-to-end reservation from host **5** to host **28**.

During the entire simulated interval of 50 seconds, 14'657 *request* packets and 379'648 *reserved* packets were sent from **R1** to **R2**. This is 98% of the total offered traffic, limited by the reservable bandwidth of that link.

3.4 Conclusion

We have extended the popular `ns` simulator such that it can be used to validate SRP. The simulations we have done suggest that SRP is does indeed work.

While working on the simulations, it became clear that the most sensitive components in SRP are the estimators at the source and, even more so, in routers. The SRP architecture allows estimators to be easily replaced, such that more advanced estimators can be deployed where necessary.

Chapter 4

Linux Traffic Control

Linux offers a rich set of traffic control functions. This chapter gives an overview of the design of the respective kernel code, describes its structure, and illustrates the addition of new elements by describing a new queuing discipline.

4.1 Introduction

Recent Linux kernels offer a wide variety of traffic control functions. The kernel parts for traffic control, and several user-space programs to control them have been implemented by Alexey Kuznetsov <kuznet@ms2.inr.ac.ru>. That work was inspired by the concepts described in [35], but it also covers the mechanisms required for supporting the architecture developed in the IETF "intserv" group [36], and will serve as the basis for supporting the more recent work of "diffserv" [3]. See also [20] for further details on how intserv and diffserv are related. This document illustrates the underlying architecture and describes how new traffic control functions can be added to the Linux kernel. The kernel version we used is 2.2.6.

Figure 4.1 shows roughly how the kernel processes data received from the network, and how it generates new data to be sent on the network: incoming packets are examined and then either directly forwarded to the network (e.g. on a different interface, if the machine is acting as a router or a bridge), or they are passed up to higher layers in the protocol stack (e.g. to a transport protocol like UDP or TCP) for further processing. Those higher layers may also generate data on their own

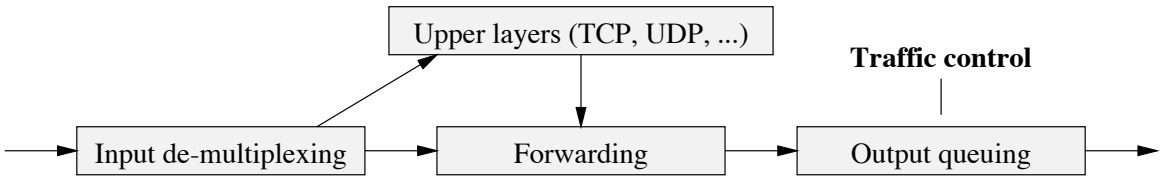

Figure 4.1: Processing of network data.

and hand it to the lower layers for tasks like encapsulation, routing, and eventually transmission.

"Forwarding" includes the selection of the output interface, the selection of the next hop, encapsulation, etc. Once all this is done, packets are queued on the respective output interface. This is the point where traffic control comes into play. Traffic control can, among other things, decide if packets are queued or if they are dropped (e.g. if the queue has reached some length limit, or if the traffic exceeds some rate limit), it can decide in which order packets are sent (e.g. to give priority to certain flows), it can delay the sending of packets (e.g. to limit the rate of outbound traffic), etc.

Once traffic control has released a packet for sending, the device driver picks it up and emits it on the network.

Sections 4.2 to 4.4 give an overview and explain some terminology. Sections 4.5 to 4.8 describe the elements of traffic control in the Linux kernel in more detail. Section 4.9 describes a queuing discipline that has been implemented by the author.

4.2 Overview

The traffic control code in the Linux kernel consists of the following major conceptual components:

- queuing disciplines
- classes (within a queuing discipline)
- filters

- policing

Each network device has a *queuing discipline* associated with it, which controls how packets enqueued on that device are treated. A very simple queuing discipline may just consist of a single queue, where all packets are stored in the order in which they have been enqueued, and which is emptied as fast as the respective device can send. See figure 4.2 for such a queuing discipline without externally visible internal structure.

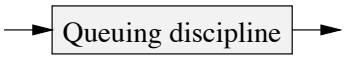

Figure 4.2: A simple queuing discipline without classes.

More elaborate queuing disciplines may use *filters* to distinguish among different *classes* of packets and process each class in a specific way, e.g. by giving one class priority over other classes.

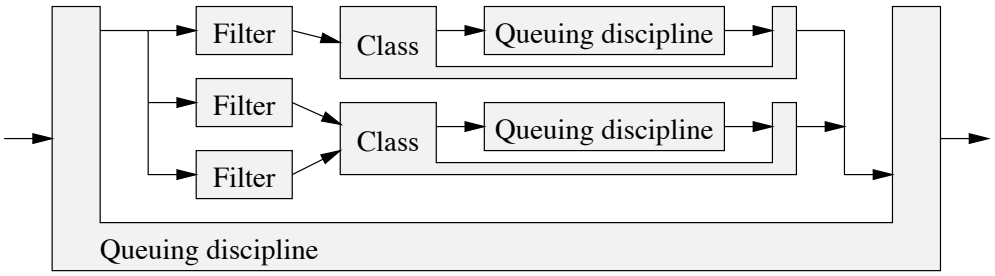

Figure 4.3: A simple queuing discipline with multiple classes.

Figure 4.3 shows an example of such a queuing discipline. Note that multiple filters may map to the same class.

Queuing disciplines and classes are intimately tied together: the presence of classes and their semantics are fundamental properties of the queuing discipline. In contrast to that, filters can be combined arbitrarily with queuing disciplines and classes as long as the queuing discipline has classes at all. But flexibility doesn't end yet – classes normally don't take care of storing their packets themselves, but they use another queuing discipline to take care of that. That queuing discipline

can be arbitrarily chosen from the set of available queuing disciplines, and it may well have classes, which in turn use queuing disciplines, etc.

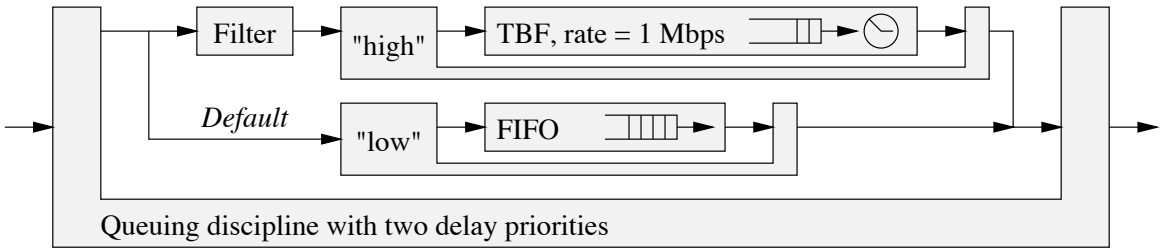

Figure 4.4: Combination of priority, TBF, and FIFO queuing disciplines.

Figure 4.4 shows an example of such a stack: first, there is a queuing discipline with two delay priorities. Packets which are selected by the filter go to the high-priority class, while all other packets go to the low-priority class. Whenever there are packets in the high-priority queue, they are sent before packets in the low-priority queue (e.g. the `sch_prio` queuing discipline works this way). In order to prevent high-priority traffic from starving low-priority traffic, we use a *token bucket filter* (TBF), which enforces a rate of at most 1 Mbps. Finally, the queuing of low-priority packets is done by a FIFO queuing discipline. Note that there are better ways to accomplish what we've done here, e.g. by using *class-based queuing* (CBQ) [37].

Packets are enqueued as follows: when the `enqueue` function of a queuing discipline is called, it runs one filter after the other until one of them indicates a match. It then queues the packet for the corresponding class, which usually means to invoke the `enqueue` function of the queuing discipline "owned" by that class. Packets which do not match any of the filters are typically attributed to some default class.

Typically, each class "owns" one queue, but it is in principle also possible that several classes share the same queue or even that a single queue is used by all classes of the respective queuing discipline. Note however that packets do not carry any explicit indication of which class they were attributed to. Queuing disciplines that change per-class information when dequeuing packets (e.g. CBQ) may therefore not work properly if the "inner" queues are shared, unless they are able either to repeat

the classification or to pass the classification result from `enqueue` to `dequeue` by some other means.

Usually when enqueuing packets, the corresponding flow(s) can be policed, e.g. by discarding packets which exceed a certain rate.

We will not try to introduce new terminology to distinguish among algorithms, their implementations, and instances of such elements, but rather use the terms queuing discipline, class, and filter throughout most of this document, to refer to all three levels of abstraction at the same time.

4.3 Resources

Linux traffic control is spread over a comparably large number of files. Note that all path names are relative to the base directory of the respective component, e.g. for the Linux kernel this is `/usr/src/linux/`, for the `tc` program `iproute2/tc/`.

`tc` is a user-space program used to manipulate individual traffic control elements. Its source is in the file `iproute2-`*version*`.tar.gz`, which can be obtained from `ftp://linux.wauug.org/pub/net/ip-routing/`.

The kernel code resides mainly in the directory `net/sched/`. The interfaces between kernel traffic control elements and user space programs using them are declared in `include/linux/pkt_cls.h` and `include/linux/pkt_sched.h`. Declarations used only inside the kernel and the definitions of some inline functions can be found in `include/net/pkt_cls.h` and `include/net/pkt_sched.h`.

The *rtnetlink* mechanism used for communication between traffic control elements in user-space and in the kernel is implemented in `net/core/rtnetlink.c` and `include/linux/rtnetlink.h`. rtnetlink is based on *netlink*, which can be found in `net/netlink/` and `include/linux/netlink.h`.

The kernel source can be obtained from the usual well-known places, e.g. from `ftp://ftp.kernel.org/pub/linux/kernel/v2.2/`.

The example in section 4.9 is included in the ATM on Linux distribution, which can be downloaded from `http://icawww1.epfl.ch/linux-atm/dist.html`.

The Differentiated Services on Linux project (`http://icawww1.epfl.ch/`

`linux-diffserv/`) has produced further examples for extensions of Linux traffic control and their use.

4.4 Terminology

Unfortunately, the terminology used to describe traffic control elements is far from consistent in literature, and there are some variations even within Linux traffic control. The purpose of this section is to help to put things into context.

Figure 4.5 shows the architectural models and the terminology used in the IETF groups "intserv" [13] and "diffserv" [26, 19], and how elements of Linux traffic control are related to them. Note that classes play an ambivalent role, because they determine the final outcome of a classification and they can also be part of the mechanism that implements a certain queuing or scheduling behaviour.

Table 4.1 summarizes the keywords used at the `tc` command line, the file names used in the kernel (in `net/sched/`), and the file names used in the source of `tc`.

Element	`tc` keyword	File name prefix	
		Kernel	`tc`
Queuing discipline	`qdisc`	`sch_`	`q_`
Class	`class`	(`sch_`)	(`q_`)
Filter	`filter`	`cls_`	`f_`

Table 4.1: Keywords and file names used for traffic control elements.

4.5 Queuing disciplines

Each queuing discipline provides the following set of functions to control its operation (see `struct Qdisc_ops` in `include/net/pkt_sched.h`):

`enqueue` enqueues a packet with the queuing discipline. If the queuing discipline has classes, the `enqueue` function first selects a class and then invokes the `enqueue` function of the corresponding queuing discipline for further enqueuing.

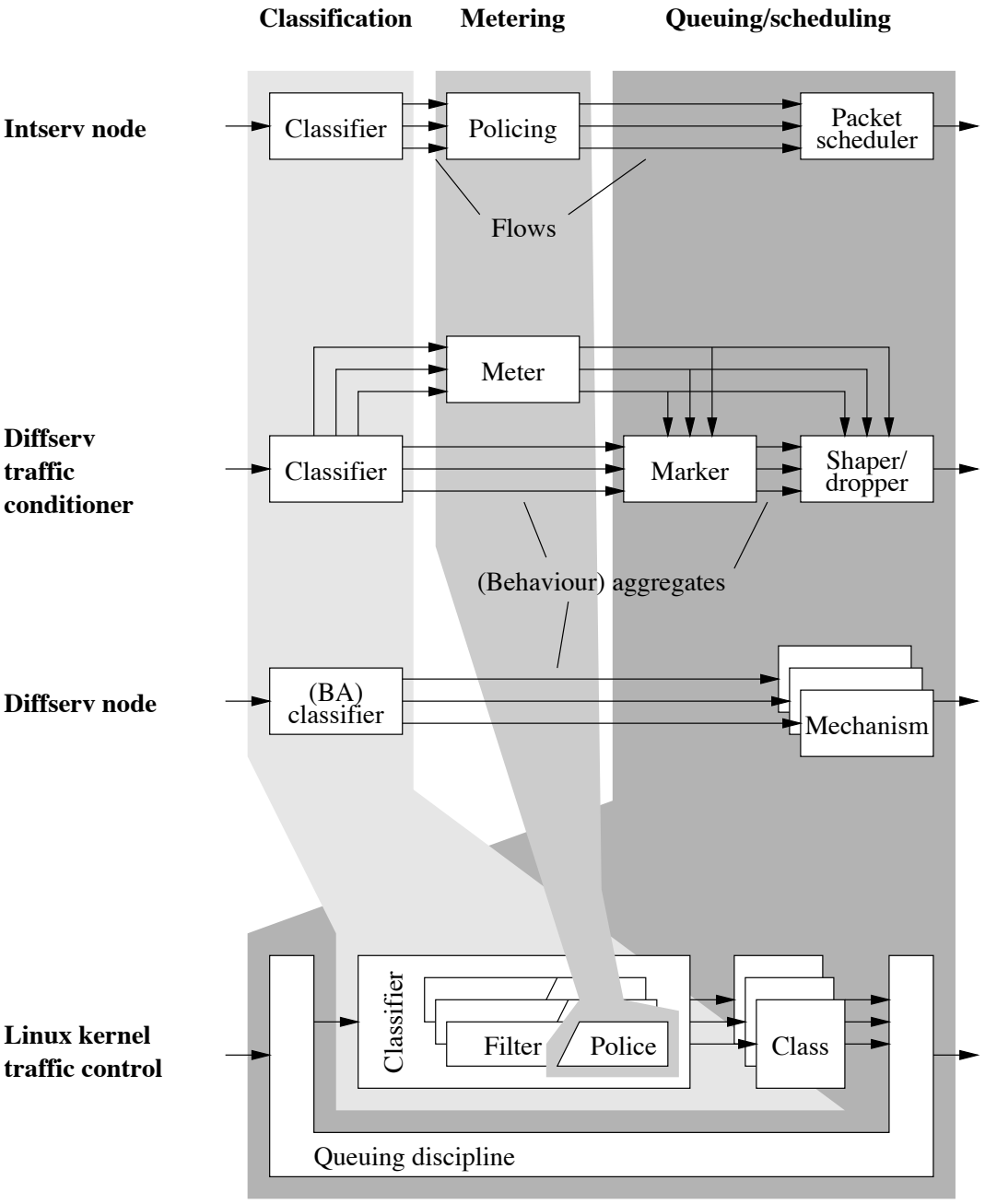

Figure 4.5: Relation of elements of the intserv and diffserv architecture to traffic control in the Linux kernel.

`dequeue` returns the next packet eligible for sending. If the queuing discipline has no packets to send (e.g. because the queue is empty or because they're not scheduled to be sent yet), `dequeue` returns NULL.

`requeue` puts a packet back into the queue after dequeuing it with `dequeue`. This differs from `enqueue` in that the packet should be queued at exactly the place from which it was removed by `dequeue`, and that it should not be included in the statistics of cumulative traffic that has passed the queue, because that was already done in the `enqueue` function.

`drop` drops one packet from the queue.

`init` initializes and configures the queuing discipline.

`change` changes the configuration of a queuing discipline.

`reset` returns the queuing discipline to its initial state. All queues are cleared, timers are stopped, etc. Also, the `reset` functions of all queuing disciplines associated with classes of this queuing discipline are invoked.

`destroy` removes a queuing discipline. It removes all classes and possibly also all filters, cancels all pending events and returns all resources held by the queuing discipline (except for the data structure describing the queuing discipline itself).

`dump` returns diagnostic data used for maintenance. Typically, the `dump` functions returns all sufficiently important configuration and state variables.

For all these functions, queuing disciplines are usually referenced by a pointer to the corresponding `struct Qdisc`.

When a packet is enqueued on an interface (`dev_queue_xmit` in `net/core/dev.c`), the `enqueue` function of the device's queuing discipline (field `qdisc` of `struct device` in `include/linux/netdevice.c`) is invoked. Afterwards, `dev_queue_xmit` calls `qdisc_wakeup` in `include/net/pkt_sched.h` on that device to try sending the packet that was just enqueued.

`qdisc_wakeup` immediately calls `qdisc_restart` in `net/sched/sch_generic.c`, which is the main function to poll queuing disciplines and to send packets. `qdisc_restart` first tries to obtain a packet from the queuing discipline of the device, and if it succeeds, it invokes the device's `hard_start_xmit` function to actually send the packet. If sending fails for some reason, the packet is returned to the queuing discipline via its `requeue` function.

`qdisc_wakeup` can also be invoked by a queuing discipline when that queuing discipline notices that a packet may be due for sending, e.g. on expiration of a timer. TBF is an example of such a queuing discipline. `qdisc_restart` is also called via `qdisc_run_queues` from `net_bh` in `net/core/dev.c`. `net_bh` is the "bottom-half" handler of the networking stack and is executed whenever packets have been queued up for further processing.

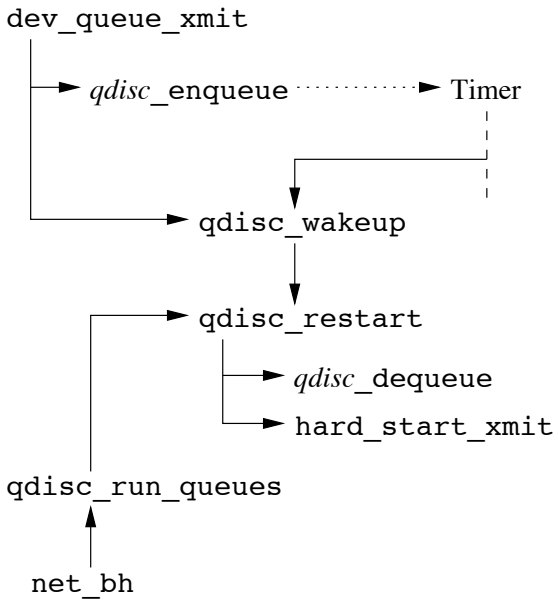

Figure 4.6: Functions called when enqueuing and sending packets.

Figure 4.6 illustrates the procedure. For simplicity, calls made by the queuing discipline (e.g. for classification) are not shown.

Note that queuing disciplines never make direct calls to delivery functions. Instead, they have to wait until they are polled.

If a queuing discipline is compiled into the the kernel, it should be registered by `pktsched_init` in `net/sched/sch_api.c`. Alternatively, is can also be registered from some other place using `register_qdisc`, e.g. from the `init_module` function if the queuing discipline is compiled as a module.

When creating or changing an instance of a queuing discipline, a vector of options (type `struct rtattr *`, declared in `include/linux/rtnetlink.h`) is passed to the `init` function. Each option is encoded with its type, the length of the value, and the value (i.e. zero or more data bytes). Option types and the data structures used for values are declared in `include/linux/pkt_sched.h`. The option vector is parsed by calling `rtattr_parse`, which returns an array of pointers to the individual elements, indexed by the option type. The length and content of an option can be accessed via the macros `RTA_PAYLOAD` and `RTA_DATA`, respectively.

Option vectors are passed between user-space programs and the kernel using the rtnetlink mechanism. Explaining rtnetlink and the underlying netlink is beyond the scope of this work. The location of the respective source files is described in section 4.3.

Instances of queuing disciplines are identified by 32-bit numbers, which are split into a major and a minor number. The usual notation is *major*:*minor*. For queuing disciplines, the minor number is always zero. Note that these major and minor numbers are not related to the numbers used for device special files.

4.6 Classes

Classes can be identified in two ways: (1) by the *class ID*, which is assigned by the user, and (2) by the *internal ID*, which is assigned by the queuing discipline. The latter has to be unique within a given queuing discipline and may be an index, a pointer, etc. Note that the value 0 is special and means "not found" when returned by `get`. The class ID is of type `u32`, while the internal ID is of type `unsigned long`. Inside the kernel, the usual way to refer to a class is by its internal ID. Only `get` and `change` use the class ID instead.

Note that multiple class IDs may map to the same internal class ID. In this

case, the class ID conveys additional information from the classifier to the queuing discipline or class.

Class IDs are structured like queuing discipline IDs, with the major number corresponding to their instance of the queuing discipline, and the minor number identifying the class within that instance.

Queuing disciplines with classes provide the following set of functions to manipulate classes (see `struct Qdisc_class_ops` in `include/net/pkt_sched.h`):

`graft` attaches a new queuing discipline to a class and returns the previously used queuing discipline.

`leaf` returns the queuing discipline of a class.

`get` looks up a class by its class ID and returns the internal ID. If the class maintains a usage count, `get` should increment it.

`put` is invoked whenever a class that was previously referenced with `get` is dereferenced. If the class maintains a usage count, `put` should decrement it. If the usage count reaches zero, `put` may remove the class.

`change` changes the properties of a class. `change` is also used to create new classes, where applicable – some queuing disciplines have a constant number of classes which are created when the queuing discipline is initialized.

`delete` handles requests to delete a class. It checks if the class is not in use, and de-activates and removes it in this case.

`walk` iterates over all classes of a queuing discipline and invokes a callback function for each of them. This is used to obtain diagnostic data for all classes of a queuing discipline.

`tcf_chain` returns a pointer to the anchor of the list of filters associated with a class. This is used to manipulate the filter list.

`bind_tcf` binds an instance of a filter to the class. `bind_tcf` is usually identical to `get`, except when the queuing discipline needs to be able to explicitly refuse

class deletion. (E.g. `sch_cbq` refuses to delete classes while they are referenced by filters.)

`unbind_tcf` removes an instance of a filter from the class. `unbind_tcf` is usually identical to `put`.

`dump_class` returns diagnostic data, like `dump` does for queuing disciplines.

Classes are selected in the `enqueue` function of the queuing discipline usually by invoking `tc_classify` in `include/net/pkt_cls.h`, which returns a `struct tcf_result` (in `include/net/pkt_cls.h`) containing the class ID (`classid`) and possibly also the internal ID (`class`), see section 4.7. The return value of `tc_classify` is either -1 (`TC_POLICE_UNSPEC`) or the policing decision returned by the filter (see section 4.8). The return values of `tc_classify` are declared in `include/linux/pkt_cls.h`.

There is also a shortcut for classification of locally generated traffic: if `skb->priority` contains the ID of a class of the current queuing discipline, that class is used and no further classification is attempted. `skb->priority` (`struct sk_buff` in `include/linux/skbuff.h`) is set to `sk->priority` (`struct sock` in `include/net/sock.h`) when locally generating a packet. `sk->priority` can be set with the `SO_PRIORITY` socket option (`sock_setsockopt` in `net/core/sock.c`). This type of classification can be useful for implementing functionality like the one provided by Arequipa [38].

Note that kernels up to at least 2.2.3 limit the value that can be set with `SO_PRIORITY` to the range $0 \ldots 7$, so that this shortcut classification does not work. However, all queuing disciplines support it. Also note that `skb->priority` can contain other priority values, e.g. the priority obtained from the TOS byte of the IPv4 header. All such values are below the smallest valid class number, 65536.

After selecting the class, the `enqueue` function of the respective inner queuing discipline is invoked. The way how this queuing discipline is stored in the data structure(s) associated with the class can vary among queuing discipline implementations.

The option vector passed to the `change` function is of the same structure as

the vectors passed to the `init` functions of queuing disciplines. The corresponding declarations are also in `include/linux/pkt_sched.h`.

4.7 Filters

Filters are used by a queuing discipline to assign incoming packets to one of its classes. This happens during the enqueue operation of the queuing discipline.

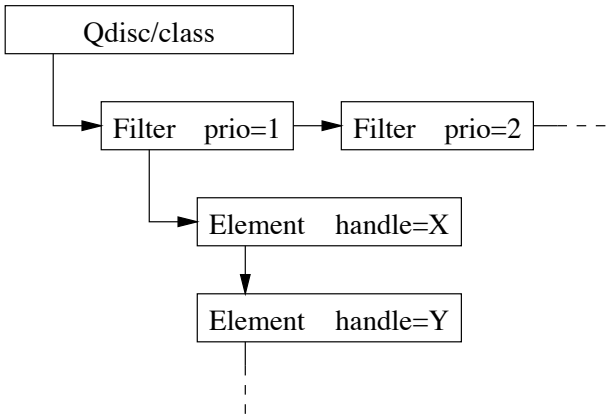

Figure 4.7: Structure of filters, with a list of elements belonging to the first filter, and no internal structure for the second filter.

Filters are kept in filter lists which can be maintained per queuing discipline or per class, depending on the design of the queuing discipline. Filter lists are ordered by priority, in ascending order. Furthermore, the entries are keyed by the protocol for which they apply. Those protocol numbers are also used in `skb->protocol` and they are defined in `include/linux/if_ether.h`. Filters for the same protocol on the same filter list must have different priorities.

A filter may also have an internal structure: it may control internal elements, which are then referenced by 32-bit handles. These handles are similar to class IDs, but they are not split into major and minor numbers. Handle 0 always refers to the filter itself. Like classes, also filters have internal IDs, which are obtained with the `get` function. The internal organization of a filter can be arbitrary. Figure 4.7 shows a filter with a list of internal elements.

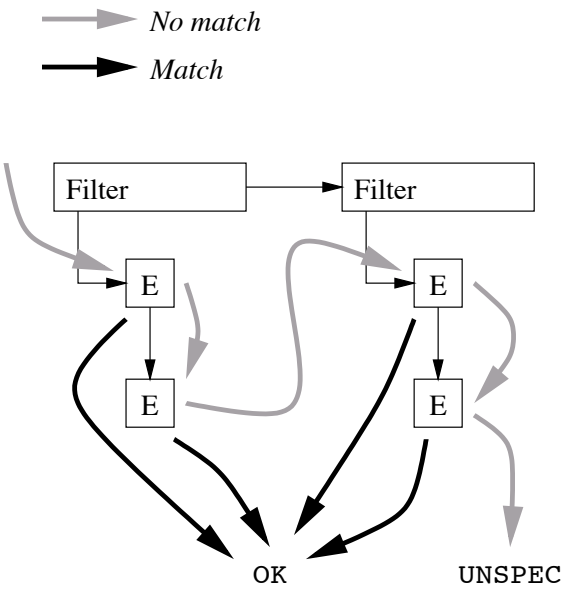

Figure 4.8: Looking for a match.

Figure 4.8 shows the order in which filters and their elements can be examined. A linked list that is processed sequentially is of course only one of many possible internal structures of a filter.

Filters are controlled via the following functions (see `struct tcf_proto_ops` in `include/net/pkt_cls.h`):

`classify` performs the classification and returns one of the `TC_POLICE_...` values described in section 4.8. If the result is not `TC_POLICE_UNSPEC`, it also returns the selected class ID and optionally also the internal class ID in the `struct tcf_result` pointed to by `res`. If the internal class ID is omitted, the value zero must be stored in `res->class`.

`init` initializes the filter.

`destroy` is invoked to remove a filter. Also the queuing disciplines `sch_cbq` and `sch_atm` use `destroy` to remove stale filters when deleting classes. If the filter or any of its elements were registered with classes, these registrations are canceled by calling `unbind_tcf`.

`get` looks up a filter element by its handle and returns the internal filter ID.

`put` is invoked when a filter element previously referenced with `get` is no longer used.

`change` configures a new filter or changes the properties of an existing filter. Configuration parameters are passed with the same mechanism as used for queuing disciplines and classes. `change` registers the addition of a new filter or filter element to a class by calling `bind_tcf`.

`delete` deletes an element of a filter. To delete the entire filter, `destroy` has to be used. This distinction is transparent to the user and is made in `net/sched/cls_api:tc_ctl_tfilter`. If the filter element was registered with a class, that registration is canceled by calling `unbind_tcf`.

`walk` iterates over all elements of a filter and invokes a callback function for each of them. This is used to obtain diagnostic data.

`dump` returns diagnostic data for a filter or one of its elements.

Note that the code for the RSVP filters is in `cls_rsvp.h`. `cls_rsvp.c` and `cls_rsvp6.c` only contain the right set of includes and set some parameters (mainly `RSVP_DST_LEN`), which control the type of filter generated from `cls_rsvp.h`.

Filters vary in the scope of packets their instances can classify: When using the `cls_fw` and `cls_route` filters, one instance per queuing discipline can classify packets for all classes. Those filters take the class ID from the packet descriptor, where it was stored before by some other entity in the protocol stack, e.g. `cls_fw` uses the marking functionality of the firewall code. We call such filters *generic*. They are illustrated in figure 4.9.

The other type of filters (`cls_rsvp` and `cls_u32`) needs one or more instances of the filter or its internal elements per class. We call such filters *specific*. Multiple instances of such a filter (or its elements) on the same filter list (e.g. for the same class) are distinguished by an *internal filter ID*, which is similar to the internal ID used for classes. However, unlike classes, filters have no "filter ID". Instead, they are identified by the queuing discipline or class for which they are registered, and their priority among the filters there.

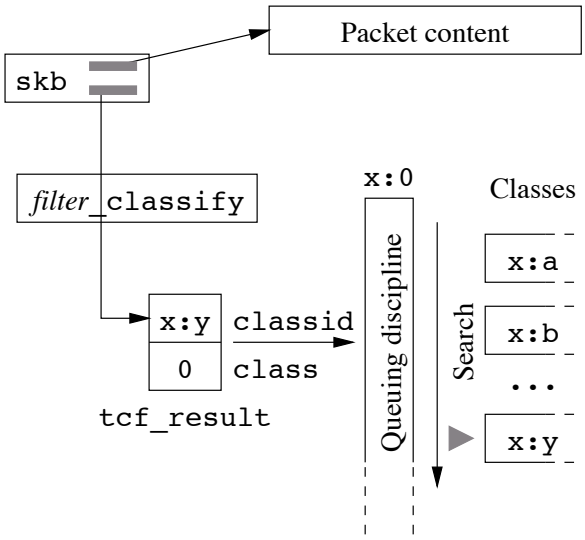

Figure 4.9: Generic filter.

Because specific filters have at least one instance or element per class, they can of course store the internal ID of that class and provide it as a result of classification. This then allows quick retrieval of class information by the queuing discipline. Figure 4.10 illustrates this scenario, where a pointer to the class structure is used as the internal ID. Unfortunately, generic filters have no means to provide this information. Therefore, they set the `class` field in `struct tcf_result` to zero and leave the lookup operation to the queuing discipline.

Starting with kernel version 2.2.5, also the generic filters `cls_fw cls_route` can become specific filters. This configuration change happens automatically when explicitly binding classes to them.

4.8 Policing

The purpose of policing is to ensure that traffic does not exceed certain bounds. For simplicity, we will assume a broad definition of policing and consider it to comprise all kinds of traffic control actions that depend in some way on the traffic volume.

We consider four types of policing mechanisms: (1) policing decisions by filters,

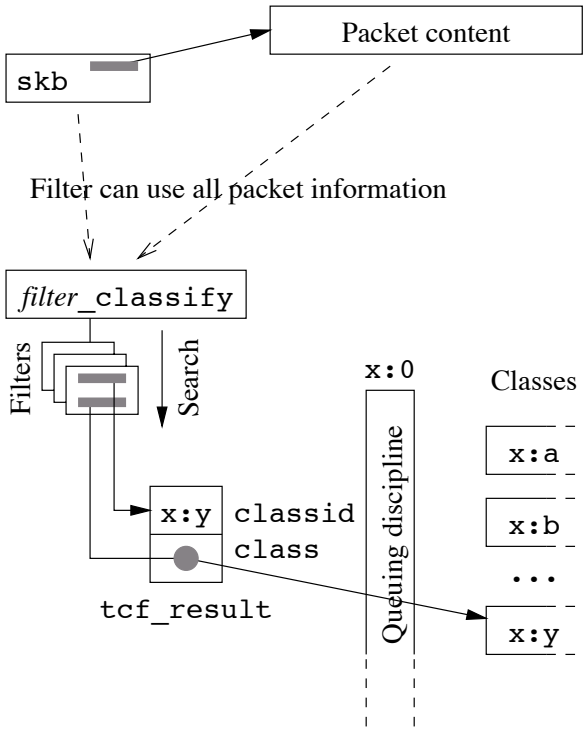

Figure 4.10: Specific filter, with a pointer to the class used as the internal class ID.

(2) refusal to enqueue a packet, (3) dropping of a packet from an "inner" queuing discipline, and (4) dropping of a packet when enqueuing a new one. Figures 4.11 to 4.15 illustrate the four mechanisms.

The first type of actions are decisions taken by filters (figure 4.11). The `classify` function of a filter can return three types of values to indicate a policy decision (the values are declared in `include/linux/pkt_cls.h`:

`TC_POLICE_OK` No special treatment requested.

`TC_POLICE_RECLASSIFY` Packet was selected by filter but it exceeds certain bounds and should be re-classified (see below).

`TC_POLICE_SHOT` Packet was selected by filter and found to violate the bounds such that it should be discarded.

Currently, the filters `cls_rsvp`, `cls_rsvp6`, and `cls_u32` support policing. The policing information is returned via `tc_classify` (in `include/net/pkt_cls.h`) to

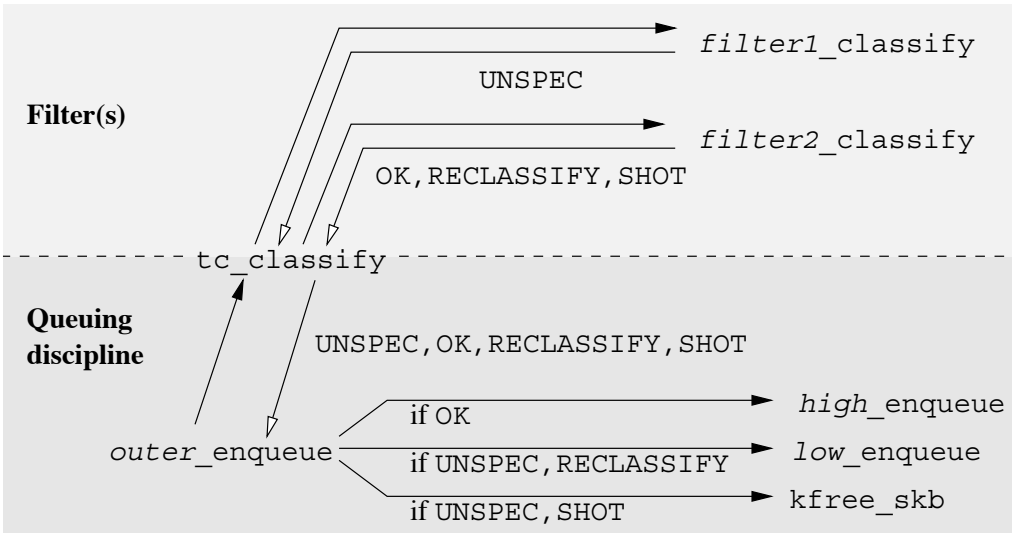

Figure 4.11: Policing when enqueuing; decision taken by filter.

the `enqueue` function of the queuing discipline. It is then up to the queuing discipline to take an appropriate action. The queuing disciplines `sch_cbq` and `sch_atm` handle `TC_POLICE_RECLASSIFY` and `TC_POLICE_SHOT`. The `sch_prio` queuing discipline ignores any policing information returned by `tc_classify`.

Filters can use the function `tcf_police` (in `net/sched/police.c`) to determine if the flow they select conforms to a token bucket. The bucket parameters (declared in `struct tc_police` in `include/linux/pkt_cls.h` and later on stored in `struct tcf_police` in `include/net/pkt_sched.h`) are roughly the same as for TBF: maximum packet size (`mtu`), average rate (`rate`), peak rate (`peakrate`), and bucket size (`burst`). The field `action` contains the policy decision code returned when accepting the packet would exceed the limits. If the packet can be accepted, `tcf_police` updates the meter and returns the decision code stored in `result`.

If no matching filter was found, `tc_classify` returns `TC_POLICE_UNSPEC`. In this case, a queuing discipline will typically either discard the packet or treat it with low priority.

Sometimes, it is desirable to police traffic with respect to more than a single token bucket, e.g. to partition traffic into "low", "high", and "excess" packets. In order to build such configurations, multiple policing functions need to be consulted.

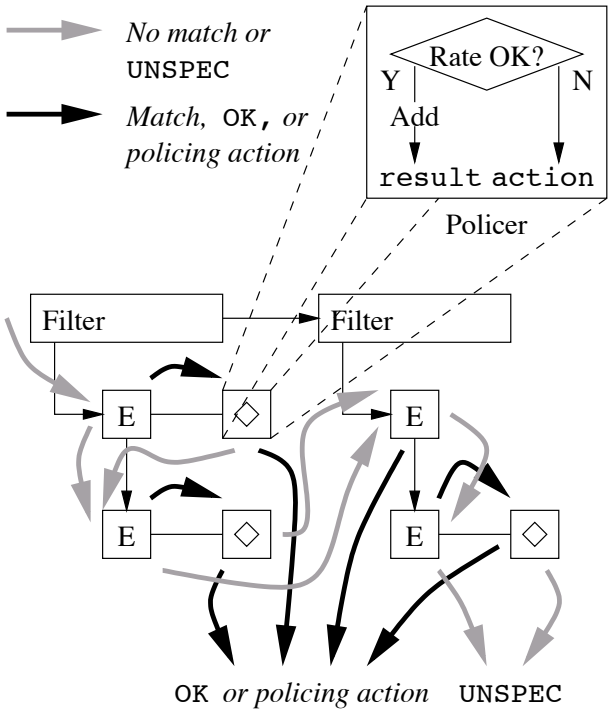

Figure 4.12: Looking for a match, with policing.

To accomplish this, `tcf_police` returns `TC_POLICE_UNSPEC`, upon which the filter proceeds with the next element, or, if the current filter has no more eligible elements, the next filter is invoked. An example of such a configuration is given in [39].

Figure 4.12 illustrates how the matching process changes when policing is involved.

The second type of policing occurs when a queuing discipline fails to enqueue a packet (figure 4.13). In this case, it normally simply discards the packet (i.e. by calling `kfree_skb`). Some queuing disciplines also provide more sophisticated feedback to the calling queuing discipline and give it a second chance for enqueuing the packet: if the `reshape_fail` callback function has been set (in `struct Qdisc`), the "inner" queuing discipline may invoke it instead to allow the "outer" queuing discipline to select a different class. If `reshape_fail` is not set or if it returns a non-zero value, the packet must be discarded. Currently, only `sch_cbq` provides a `reshape_fail` function. `sch_fifo` and `sch_tbf` make calls to `reshape_fail`, if available.

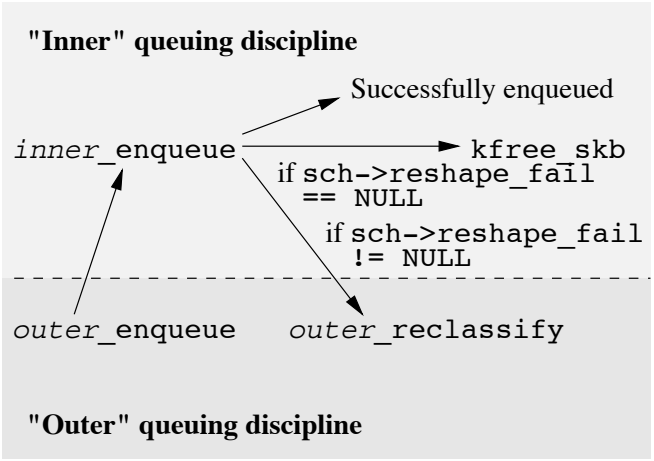

Figure 4.13: Policing when enqueuing; decision taken by "inner" queuing discipline.

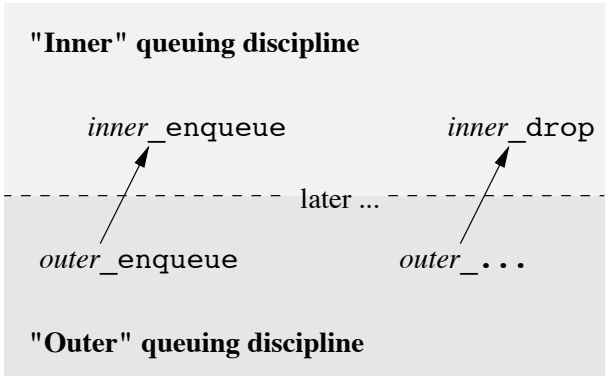

Figure 4.14: Policing after enqueuing; decision taken by "outer" queuing discipline.

The third policing mechanism is applied if a queuing discipline decides to drop a packet from an "inner" queuing discipline after that packet was enqueued, e.g. in order to create space for packets of a more important class (figure 4.14). This is done using the `drop` function. The `cbq_dequeue_prio` function of `sch_cbq` uses this via `cbq_under_limit` to remove packets from classes which are over limit.

Also the fourth mechanism (figure 4.15) discards packets that have already been successfully enqueued: if the `enqueue` function of a queuing discipline considers a new packet to be more important than some older one, it can discard the old packet and enqueue the new one instead. It indicates this to the caller by returning zero.

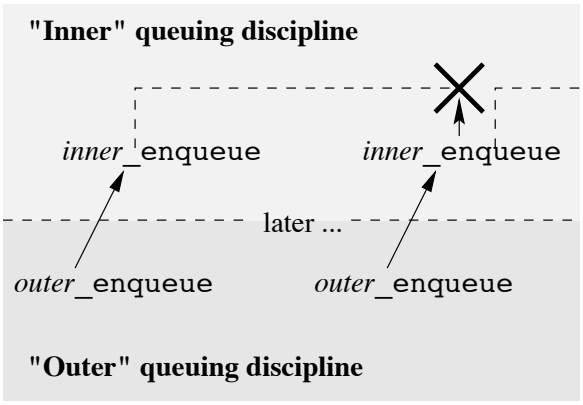

Figure 4.15: Older packet is discarded to make room for new packet.

4.9 The `sch_atm` queuing discipline

As an example of how new traffic control elements can be added, we examine the ATM queuing discipline in more detail. It is used to re-direct flows from the default path (e.g. through a given interface) to ATM VCs. Each flow can have its own ATM VC, but multiple flows can also share the same VC. Figure 4.16 illustrates the structure of this queuing discipline.

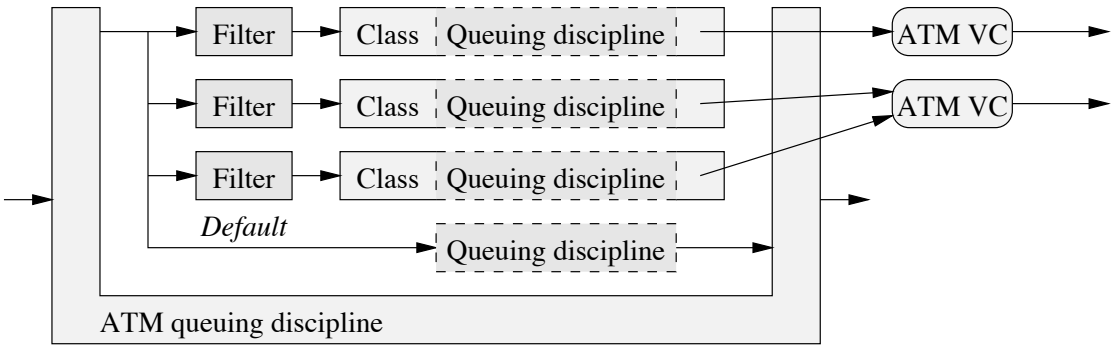

Figure 4.16: The ATM queuing discipline.

While its classification and queuing part is fairly generic, the ATM queuing discipline differs from other queuing disciplines in that packets enqueued on it may leave via other paths than through the `dequeue` function or being dropped: whenever `dequeue` is called, it first checks all inner queuing disciplines for packets to send,

and sends them over the respective ATM VCs. After that, it returns whatever it gets from the default queue, which receives the packets that don't get attributed to any of the classes.

In order to prevent VCs from being removed while the queuing discipline is still using them, the reference count of the corresponding socket is increased when attaching a VC to a class of the ATM queuing discipline. This happens in the function `sockfd_lookup` in `net/socket.c` which `atm_tc_change` calls to translate the socket descriptor number to a pointer to the socket structure. When the class is removed, it returns the socket using `sockfd_put`, which then decrements the reference count. This pair of functions performs roughly the equivalent of `fdopen` and `close`.

The ATM queuing discipline supports the policing responses `TC_POLICE_SHOT` and `TC_POLICE_RECLASSIFY`. The latter can be handled in two different ways: (1) by assigning the packet to a new class (as configured by the user), or (2) by setting the cell loss priority bit in outgoing ATM cells.

The code of the ATM queuing discipline is in `net/sched/sch_atm.c`. In addition to that file, `include/linux/pkt_sched.h` contains the option types (prefix `TCA_ATM_`), and `net/sched/sch_api.c` contains the initialization. Furthermore, the usual changes had to be made to `net/sched/Config.in` and `net/sched/Makefile` to include the new queuing discipline in the configuration and build process.

The use of the ATM queuing discipline is described in the file `atm/extra/tc/README` in the ATM on Linux distribution.

4.10 Conclusion

Linux traffic control consists of a large variety of elements, which interact with each other in many ways. The modular approach chosen results in a very versatile design that can be readily applied to most current traffic control tasks, and which can be easily extended to accommodate less typical applications, such as the link-layer selection implemented in the ATM queuing discipline. It also forms the basis for the Linux implementation of Differentiated Services, which unify and advance

many of the existing traffic control concepts.

We have described queuing disciplines, classes, filters, and elements within filters, we have illustrated the most important interactions between these components, and we have briefly introduced the design of a new queuing discipline. We hope this information to be useful for people aiming to understand the inner workings of Linux traffic control, and in particular also to implementors of new traffic control functions.

Chapter 5

Differentiated Services on Linux

As shown in the previous chapter, recent Linux kernels offer a wide variety of traffic control functions, which can be combined in a modular way. We have designed support for Differentiated Services based on the existing traffic control elements, and we have implemented new components where necessary. In this chapter we give a brief overview of the structure of Linux traffic control, we describe our prototype implementation in more detail, and we show measurement results to illustrate its performance.

This chapter is structured as follows. Section 5.2 introduces the concepts of the Diffserv architecture. Section 5.3 discusses where traffic control functions in the Linux kernel needed to be extended. Section 5.4 describes the new components in more detail.

In section 5.5, we explain examples of configuration scripts. We conclude with measurement results obtained using our implementation in section 5.6.

5.1 Introduction

The Differentiated Services architecture (Diffserv; [19]) provides an infrastructure for applications, users, or providers to select the network service that best suits their needs. Services may differ in many ways, such as delay or loss goals.

Diffserv defines local node services in terms of the forwarding behavior of individual routers (the so-called Per-Hop-Behavior; PHB). Diffserv defines only PHBs

which can be used to define end-to-end services, however the actual use of these building blocks to define end-to-end services is beyond the current scope of the IETF Diffserv Working Group [3].

When forwarding a packet, a node selects the PHB to apply based on the content of the Diffserv field (short "DS field") in the IP header [26]. This value is called the Diffserv Code Point (DSCP). Note that each network may decide on its own mapping between DSCP values and PHBs. Nevertheless, each PHB definition also proposes a default DSCP value.

The Diffserv design allows PHBs to be defined, implemented, and deployed in a largely independent way. It is therefore important to preserve this flexibility in any implementation.

We have developed a design to support basic classification and DS field manipulation required by Diffserv nodes. The design enables configuration of the first PHBs that are being defined in the Diffserv WG. We have implemented a prototype of this design using the traffic control framework available in recent Linux kernels. The source code, configuration examples, and related information can be obtained from `http://icawww1.epfl.ch/linux-diffserv/`

The main focus of our work is to allow maximum flexibility for node configuration and for experiments with PHBs, while still maintaining a design that does not unnecessarily sacrifice performance.

5.2 Differentiated Services

Figure 5.1 shows the general structure of the forwarding path in a Diffserv node.

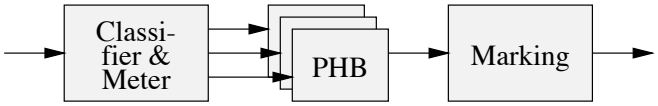

Figure 5.1: General Diffserv forwarding path.

Depending on the implementation, marking may also occur at different places, possibly even several times.

5.2.1 Classification and metering

Diffserv distinguishes two types of classification: a "behavior aggregate classifier" distinguishes packets based only on their DS fields. A "micro-flow classifier" may take into account the whole packet, e.g. the source and destination IP addresses, port numbers, etc.

Classification based on packet contents may also be supplemented by metering of traffic flows, e.g. in order to accept only limited traffic for a given PHB.

5.2.2 Marking

The process of setting or modifying the DS field is called marking. Marking is necessary in several cases, for example:

- Whenever a packet from a non-Diffserv network reaches the edge of a Diffserv network, its DS field has to be initialized to the appropriate DSCP.
- Diffserv-capable hosts need to be able to set the DS field of packets they originate.
- Since different parts of a network may use different DSCP to PHB mappings, edge routers may have to change the DS field in packets crossing such a boundary.
- A PHB group may use multiple PHBs and hence multiple DSCPs to convey additional information (e.g. some form of congestion indication). In this case, the DS field may change at any Diffserv-capable node along the path.

5.2.3 PHBs

Three groups of PHBs are currently being defined in the Diffserv WG:

- PHBs for compatibility with historical use of the IPv4 TOS byte (defined in [26])
- Expedited forwarding, a simple high-priority PHB [40]

- Assured Forwarding, a group of PHBs with different delay and drop priorities [41]

5.3 Diffserv extensions to Linux traffic control

The traffic control framework available in recent Linux kernels [42] already offers most of the functionality required for implementing Diffserv support. We therefore closely followed the existing design and added new components only where it was deemed strictly necessary.

5.3.1 Overview

The classification result may be used several times in the Diffserv processing path, and it may also depend on external factors (e.g. time), so reproducing the classification result may not only be expensive, but actually impossible.

We therefore added a new field `tc_index` to the packet buffer descriptor (`struct sk_buff`), where we store the result of the initial classification. In order to avoid confusing `tc_index` with the classifier `cls_tcindex`, we will call the former `skb->tc_index` throughout this document.

`skb->tc_index` is set using the `sch_dsmark` queuing discipline, which is also responsible for initially retrieving the DSCP, and for setting the DS field in packets before they are sent on the network. `sch_dsmark` provides the framework for all other operations.

The `cls_tcindex` classifier reads all or part of `skb->tc_index` and uses this to select classes.

Finally, we need a queuing discipline to support multiple drop priorities as required for Assured Forwarding. For this, we designed GRED, a generalized RED. `sch_gred` provides a configurable number of drop priorities which are selected by the lower bits of `skb->tc_index`.

5.3.2 Classification and marking

The classifiers `cls_rsvp` and `cls_u32` can handle all micro-flow classification tasks. Additionally, the `ipchains` firewall is also capable of tagging microflows into classes. Behavior aggregate classification could also be done using `cls_u32` and `ipchains`, but since we usually already have `sch_dsmark` at the top level, we use the simpler `cls_tcindex` and retrieve the DSCP using `sch_dsmark`, which then puts it into `skb->tc_index`.

When using `sch_dsmark`, the class number returned by the classifier is stored in `skb->tc_index`. This way, the result can be re-used during later processing steps.

Nodes in multiple DS domains must also be able to distinguish packets by the inbound interface in order to translate the DSCP to the correct PHB. This can be done using the `route` classifier, in combination with the `ip rule` command interface subset.

Marking is done when a packet is dequeued from `sch_dsmark`. `sch_dsmark` uses `skb->tc_index` as an index to a table in which the outbound DSCP is stored and puts this value into the packet's DS field.

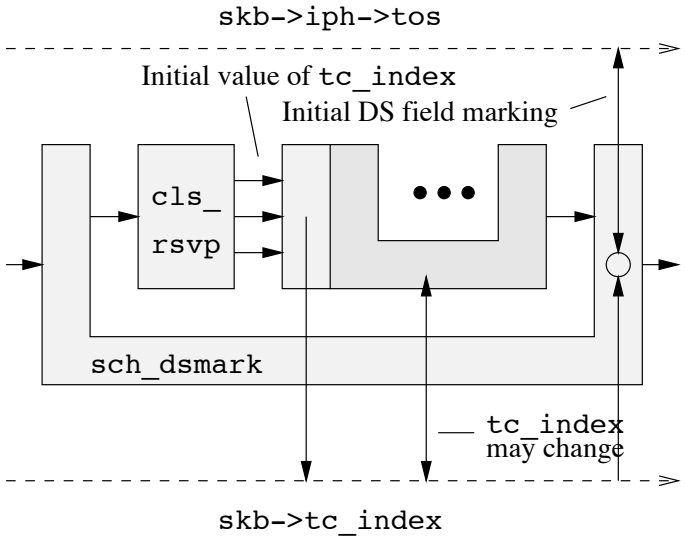

Figure 5.2: Micro-flow classifier.

Figure 5.2 shows the use of `sch_dsmark` and `skb->tc_index` in a micro-flow classifier based on `cls_rsvp`. Figure 5.3 shows a behavior aggregate classifier using

`cls_tcindex`.

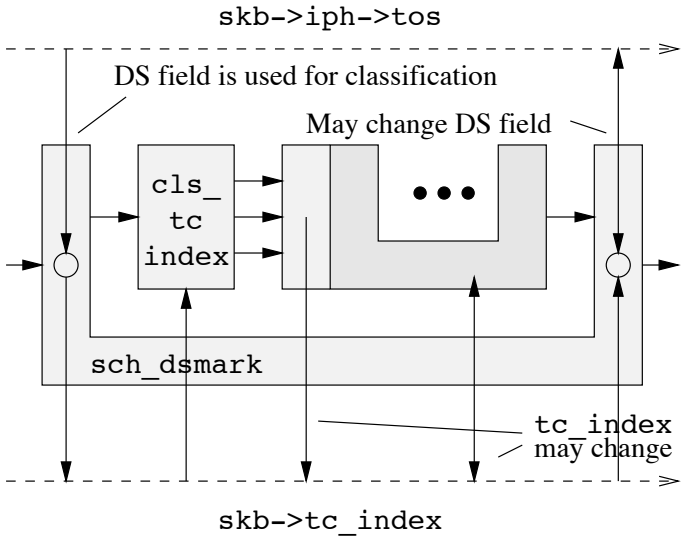

Figure 5.3: Behaviour aggregate classifier.

5.3.3 Cascaded meters

Multiple meters are needed if traffic should be assigned to more than two classes, based on the bandwidth it uses. As an example, such classes could be for "low", "high", and "excess" traffic.

Our current implementation supports a limited form of cascading at the level of classifiers. We are testing a cleaner and more efficient solution at the time of writing.

5.3.4 Implementing PHBs

PHBs based only on delay priorities, e.g. Expedited Forwarding [40], can be built using CBQ [37] or the more simple `sch_prio`. (See section 5.5.)

Besides four delay priorities, which can again be implemented with already existing components, Assured Forwarding [41] also needs three drop priorities, which is more than the current implementation of RED supports. We therefore added a new queuing discipline which we call "generalized RED" (GRED). GRED uses the

lower bits of `skb->tc_index` to select the drop class and hence the corresponding set of RED parameters.

5.3.5 Shaping

The so-called Token Bucket Filter (`sch_tbf`) can be used for shaping at edge nodes. Unfortunately, the highest rate at which `sch_tbf` can shape is limited by the system timer, which normally ticks at 100 Hz, but can be accelerated to 1 kHz or more if the processor is sufficiently powerful. Note that Linux traffic control supports more granular clocking for droppers (i.e. shapers without buffer).

CBQ can also be used to do shaping.

Higher rates can be shaped when using hardware-based solutions, such as ATM.

5.3.6 End systems

Diffserv-capable sources use the same functionality as edge routers, i.e. any classification and traffic conditioning can be administratively configured.

In addition to that, an application may also choose to mark packets when they are generated. For IPv4, this can be done using the `IP_TOS` socket option, which is commonly available on Unix, and of course also on Linux. Note that Linux follows the [5] convention of not allowing the lowest bit of the TOS byte to be different from zero. This restriction is compatible with use for Diffserv. Furthermore, the use of values corresponding to high precedences (i.e. DSCP 0x28 and above) is restricted. This can be avoided either by giving the application the appropriate capabilities (privileges), or by re-marking (see below).

Setting the DS field with IPv6 is currently very awkward. In the future, an improved interface is likely to be provided that unifies the IPv4 and IPv6 usage and that may contain additional improvements, e.g. selection of services instead of "raw" DS field values.

An application's choice of DS field values can always be refused or changed by traffic control (using re-marking) before a packet actually reaches the network.

5.4 New components

The prototype implementation of Diffserv support requires the addition of three new traffic control elements to the kernel: (1) the queuing discipline `sch_dsmark` to extract and to set the DSCP, (2) the classifier `cls_tcindex` which uses this information, and (3) the queuing discipline `sch_gred` which supports multiple drop priorities and buffer sharing.

Only the queueing discipline to extract and set the DSCP is truly specific to the differentiated services architecture. The other two elements can also be used in other contexts.

Figure 5.2 shows the use of `sch_dsmark` for the initial packet marking when entering a Diffserv domain. The classification and rate control metering is performed by a micro-flow classifier, e.g. `cls_rsvp`, in this case.

This classifier determines the initial TC index which is then stored in `skb->tc_index`. Afterwards, further processing is performed by an inner queuing discipline. Note that this queuing discipline may read and even change `skb->tc_index`.

When a packet leaves `sch_dsmark`, `skb->tc_index` is examined and the DS field of the packet is set accordingly.

Figure 5.3 shows the use of `sch_dsmark` and `cls_tcindex` in a node which works on a behavior aggregate, i.e. on packets with the DS field already set. The procedure is quite similar to the previous scenario, with the exception that `cls_tcindex` takes over the role of `cls_rsvp` and that the DS field of the incoming packet is copied to `tc_index` before invoking the classifier.

Note that the value of the outbound DS field can be affected at three locations: (1) in `sch_dsmark`, when classifying based on `skb->tc_index`, which contains the original value of the DS field; (2) by changing `skb->tc_index` in an inner queuing discipline; and (3) in `sch_dsmark`, when mapping the final value of `skb->tc_index` back to a new value of the DS field.

5.4.1 sch_dsmark

As illustrated in figure 5.4, the sch_dsmark queuing discipline performs three actions based on the scripting invocation:

- If set_tc_index is set, it retrieves the content of the DS field and stores it in skb->tc_index.
- It invokes a classifier and stores the class ID returned in skb->tc_index. If the classifier finds no match, the value of default_index is used instead. If default_index is not set, the value of skb->tc_index is not changed. Note that this can yield undefined behaviour if neither set_tc_index nor default_index is set.
- After sending the packet through its inner queuing discipline, it uses the resulting value of skb->tc_index as an index into a table of (mask,value) pairs. The original value of the DS field is then replaced using the following formula: ds_field = (ds_field & mask) | value

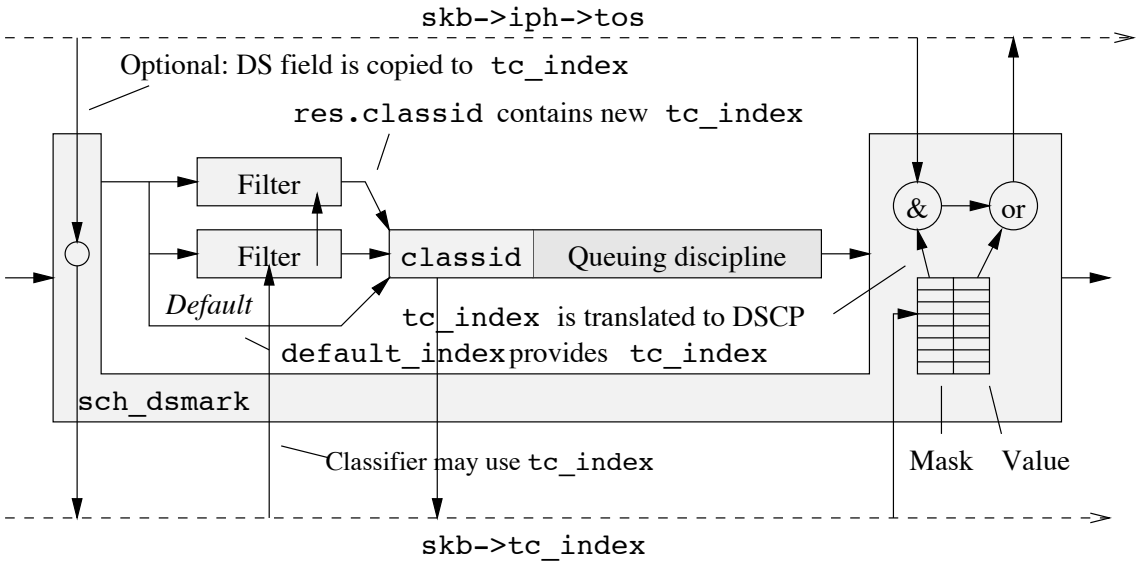

Figure 5.4: The dsmark queuing discipline.

Table 5.4.1 lists the parameters that can be configured in the dsmark queuing discipline. The upper part of the table shows parameters of the queuing discipline

itself. The lower part shows parameters of each class.

Variable name / `tc` keyword	Value	Default
`indices`	2^n	none
`default_index`	0...`indices`-1	absent
`set_tc_index`	none (flag)	absent
`mask`	0...0xff	0xff
`value`	0...0xff	0

Table 5.1: Configuration parameters of `sch_dsmark`.

`indices` is the size of the table of (mask,value) pairs.

5.4.2 `cls_tcindex`

As shown in figure 5.5, the `cls_tcindex` classifier uses `skb->tc_index` to select classes. It first calculates the lookup key using the algorithm

```
key = (skb->tc_index >> shift) & mask
```

Then it looks for an entry with this handle. If an entry is found, it may call a meter (if configured), and it will return the class IDs of the corresponding class.

If no entry is found, the result depends on whether `fall_through` is set. If set, a class ID is constructed from the lookup key. Otherwise, it returns a "not found" indication and the search continues with the next classifier. We call construction of the class ID an "algorithmic mapping". This can be used to avoid setting up a large number of classifier elements if there is a sufficiently simple relation between values of `skb->tc_index` and class IDs. An example of this trick is used in the AF scripts on the web site.

The size of the lookup table can be set using the `hash` option. `cls_tcindex` automatically uses perfect hashing if the range of possible choices does not exceed the size of the lookup table. If the `hash` option is omitted, an implementation-dependent default value is chosen.

Table 5.4.2 shows the parameters that can be configured in the `tcindex` classifier. The upper part of the table shows parameters of the classifier itself. The lower part shows parameters of each element.

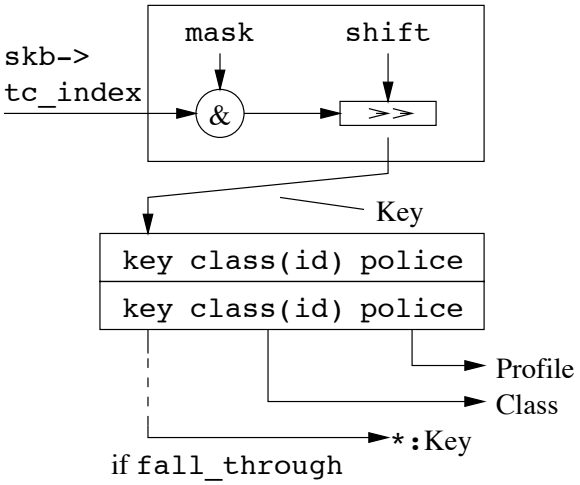

Figure 5.5: The `tcindex` classifier.

Variable	`tc` keyword	Value	Default
`hash`	`hash`	1...0x10000	implementation-dependent
`mask`	`mask`	0...0xffff	0xffff
`shift`	`shift`	0...15	0
`fall_through`	`fall_through`/ `pass_on`	flag	`fall_through`
`res`	`classid`	*major*:*minor*	none
`police`	`police`	Profile	none

Table 5.2: Configuration parameters of `cls_tcindex`.

Note that the keyword used by `tc` (the command-line tool used to manually configure traffic control elements) does not always correspond to the variable internally used by `cls_tcindex`.

5.4.3 `sch_gred`

Figure 5.6 shows how `sch_gred` uses `skb->tc_index` for the selection of the right virtual queue (VQ) within a physical queue. What makes `sch_gred` different from other Multi-RED implementations is the fact that it is decoupled from any one specific block along the packet's path such as a header classifier or meter. For

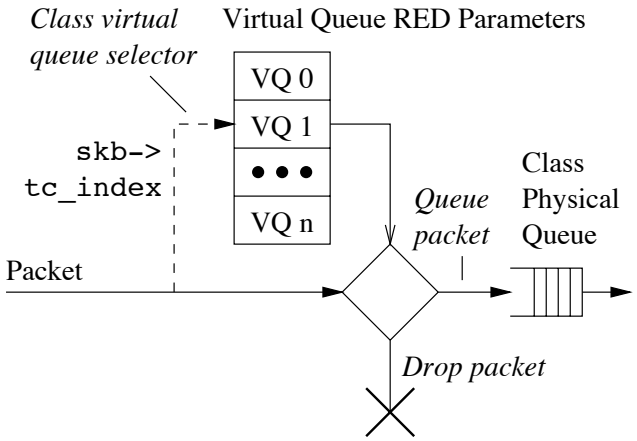

Figure 5.6: Generic RED and the use of `skb->tc_index`

example, CISCO's DWRED [43] is tied to mapping VQ selection based on the precedence bits classification. On the other hand, RIO [44] is tied to the IN/OUT metering levels for the selection of the VQ. In the case of GRED, any classifier, meter, etc. along the data path can affect the selection of the VQ by setting the appropriate value of `skb->tc_index`.

GRED also differs from the two mentioned multiple RED mechanisms in that it is not limited to a specific number of VQ. The number of VQs is configurable for each physical class queue. GRED does not assume certain drop precedences (or priorities). It depends on the configuration parameters passed on by the user. In essence, DWRED and RIO are special cases of GRED.

Currently, the number of virtual queues is limited to 16 (the least significant 4 bits of `skb->tc_index`). There is a one to one mapping between the values of `skb->tc_index` and the virtual queue number in a class. Buffer sharing is achieved using one of two ways (selectable via configuration):

- Simple setting of physical queue limits. It is up to the individual configuring the virtual queues parameters to decide which one gets preferential treatment. Sharing and preferential treatment amongst virtual queues is based on parameter settings such as the per-virtual queue physical limit, threshold values and drop probabilities. This is the default setting.

- A similar average queue trick as that is used in [44]. This is selected by the operator `grio` during the setup. Each VQ within a class is assigned a priority at configuration time. Priorities range from 1 to 16 at the moment, with 1 being the highest. The computation of the average queue value (for a VQ) involves first summing to the current stored average queue value all the the other average queue values of the VQs which are more important than it. This way a relatively higher priority (lower priority value) gets preferential treatment because its average queue is always the lowest; the relatively lower priority will still continue to send when the higher ones are not dominating the buffer space. A user can still configure the per-virtual-Queue physical queue limits, threshold values, and drop probabilities as in the (first) case when the `grio` option is not defined.

The second scheme is slightly slower than the first one (a few more per-packet computations).

GRED is configured in two steps. First (see also the upper part of table 5.4.3) the generic parameters are configured to select the number of virtual queues `DPs` and whether to turn on the RIO-like buffer sharing scheme (`grio`). Also at this point, a default virtual queue is selected so that packets with out of range values of `skb->tc_index` or misconfigured priorities in the case of `grio` buffer-sharing setup are directed to it. Normally, the default virtual queue is the one with the highest likelihood of having a packet discarded. The operator `setup` identifies that this is a generic setup for GRED.

The second step is to set parameters for individual virtual queues. (See also the lower part of table 5.4.3).

These parameters are equivalent to the traditional RED parameters. In addition, each RED configuration identifies which virtual queue the parameters belong to as well as the priority if the `grio` technique is selected. The mandatory parameters are:

- `limit` defines the virtual queue "physical" limit in bytes.

- `min` defines the minimum threshold value in bytes.

Variable	tc keyword	Value	Default
DPs	DPs	1...16	none
def	default	1...DPs	none
grio	grio	none (flag)	absent
limit	limit	bytes	none
qth_min	min	bytes	none
qth_max	max	bytes	none
n/a	avpkt	bytes	none
n/a	bandwidth	rate	10 Mbps
n/a	burst	packets	none
n/a	probability	$[0 \ldots 1)$	0.02
DP	DP	1...DPs	0
prio	prio	1...DPs	none

Table 5.3: Configuration parameters of sch_gred.

- max defines the maximum threshold value in bytes.
- avpkt is the average packet size in bytes.
- bandwidth is the wire-speed of the interface.
- burst is the number of average-sized packets allowed to burst. The Linux RED implementation attempts to compute an optimal W value for the user based on the avpkt, minimum threshold and allowed burst size. This is based on the equation:

$$\text{burst} + 1 - \frac{\text{qmin}}{\text{avpkt}} < \frac{1 - (1 - \text{W})^{\text{burst}}}{\text{W}}$$

  as described in [45].
- probability defines the drop probability in the range $[0 \ldots)$.
- DP identifies the virtual queue assigned to these parameters.
- prio identifies the virtual queue priority if grio was set in the general parameters.

5.5 Building sample configurations

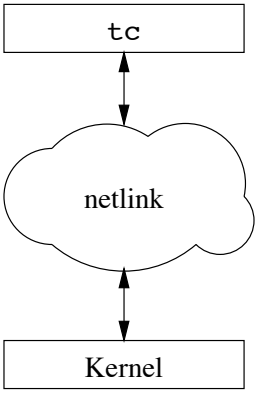

Figure 5.7: User space to kernel communication using `tc`

Communication and configuration of the kernel code or modules is achieved by a user level program tc written by Alexey. The interaction is shown in figure 5.5.

Given the flexibility of the code, there are many ways to reach the same end goal. Depending on the requirement, one could script the same PHB using a different combinations of qdiscs; e.g. one could build a core EF capable router using either CBQ to rate limit it and prioritise its traffic or instead use the PRIO qdisc with a Token Bucket attached to rate limit it. It is hoped that users of Linux Diffserv will be able to script their own flavored configurations. The examples below (as well as those on the Linux Diffserv web site) are simplistic, in the sense that they only assume one interface per node. One should easily be able to extend them for more than one interface

The normal recipe for creating a configuration script is:

- attach `sch_dsmark` to the output interface
- define the structure of the queuing discipline(s) inside `sch_dsmark`
- number the classes and decide on a numbering scheme to use for `skb->tc_index` (the latter may be trivial if `skb->tc_index` is only used within `sch_dsmark`.)

- identify which packets go to which classes and configure the classifier(s) of `sch_dsmark` accordingly

The script lines in the next subsections are numbered for clarity of the accompanying description below.

For clarity, we did not include handling of historical DS field values in our scripts.

5.5.1 Edge device: Packet re-marking

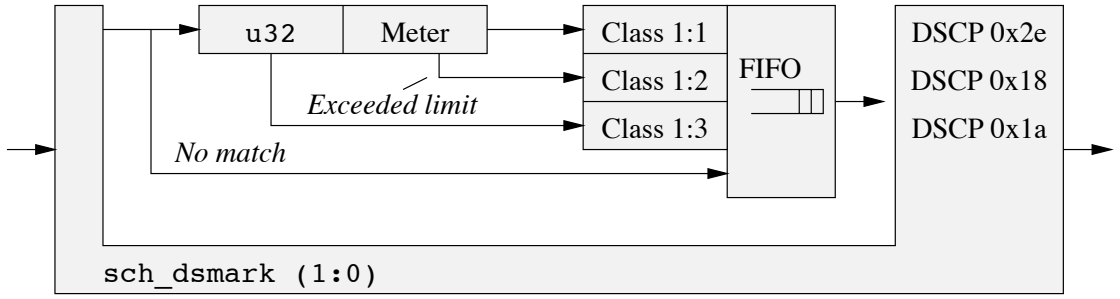

Figure 5.8: Packet re-marking at the edge.

```
1. tc qdisc add dev eth0 handle 1:0 root dsmark indices 64
2. tc class change dev eth0 classid 1:1 dsmark mask 0x3 value 0xb8
3. tc class change dev eth0 classid 1:2 dsmark mask 0x3 value 0x68
4. tc class change dev eth0 classid 1:3 dsmark mask 0x3 value 0x48
5. tc filter add dev eth0 parent 1:0 protocol ip prio 4 handle 1: u32
        divisor 1
6. tc filter add dev eth0 parent 1:0 protocol ip prio 5 handle 2: u32
        divisor 1
7. tc filter add dev eth0 parent 1:0 prio 4 u32
        match ip dst 10.0.0.0/24
        police rate 1Mbit burst 2K continue
        flowid 1:1
8. tc filter add dev eth0 parent 1:0 prio 5 u32
        match ip dst 10.0.0.0/24
        flowid 1:2
9. tc filter add dev eth0 parent 1:0 prio 4 u32
        match ip dst 10.1.0.0/16
        match ip src 192.1.0.0/16
        match ip protocol 6 0xff
```

```
match ip dport 0x17 0xffff
flowid 1:3
```

The first line attaches a dsmarker to the interface eth0 on the root node. The second line instructs the dsmarker to remark the DSCP of classid 1:1 by first masking out bits 6 and 7 then ORing that with a value of 0xb8. Note that: This is equivalent to ignoring the ECN bits, and setting the code point value to 0x2e (which happens to be the DSCP for EF). In a similar manner, the third line instructs the dsmarker to remark the CP of classid 1:2 to 0x1a (DSCP for AF31). The fourth line adds a remarking the class 1:3 DSCPs to 0x12 (DSCP for AF21). These three lines in effect are also registering the classes 1:2, 1:3 and 1:4.

Line 5 adds a u32 classifier with priority of 4. Line 6 adds another classifier of a lower priority. Line 7 maps all packets with a source IP address of 10.0.0.0/24 to class 1:1. Line 7 and 8 show how one can attach a meter to a classifier and the reaction to an exceeding of the rate. Basically, the trick is to define two filters matching the same headers with a higher priority one attached with a meter and policing action. The operator `continue` is used to allow a lookup of the next lower priority matching filter. In this case, should the metering be exceeded in class 1:1, the flow is reclassified to class 1:2. Line 9 selects all TCP packets from source subnet 10.1.1.0/16 destined towards subnet 192.1.1.0/16 and sends them to the queue for class 1:3.

The overall effect is: all packets coming in from source subnet address 10.0.0.0/24 will get their packets marked with a DSCP of 0x2e (EF class/PHB) up to a point where they start exceeding their allocated rate (of 1Mbps and burst of 2K). In this case, the packets are demoted to class 1:2 where they will be remarked to DSCP 0x18 (AF21). Any TCP packets of origin subnet 10.1.1.0/16 destination subnet 192.1.1.0/16 will be remarked to 0x1A (AF22). It is easy to see that one can build a multi-color marking scheme of large depths using using such cascading filter/metering schemes.

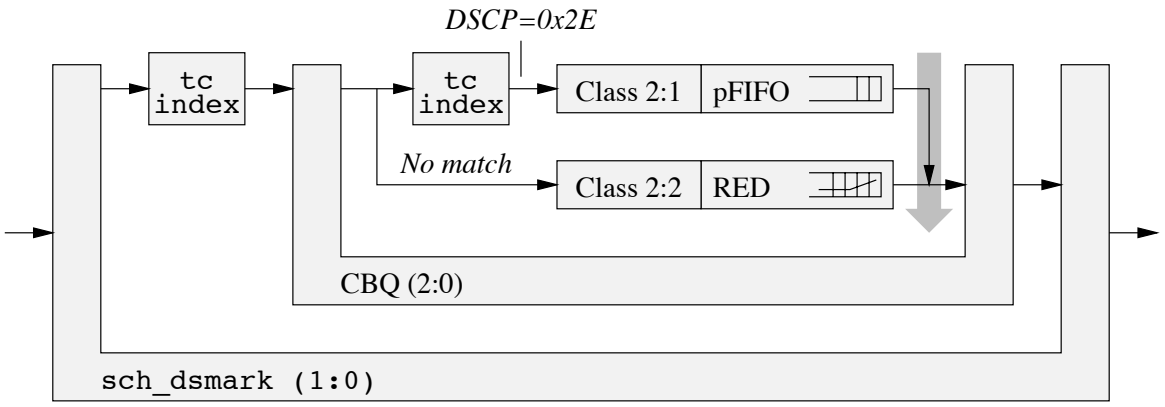

Figure 5.9: Configuring EF using CBQ.

5.5.2 Core device: EF using CBQ

The script below is the output of the EF Perl script on the Linux Diffserv Web site.

```
1. tc qdisc add dev eth0 handle 1:0 root dsmark indices 64
           set_tc_index
2. tc filter add dev eth0 parent 1:0 protocol ip prio 1
           tcindex mask 0xfc shift 2
3. tc qdisc add dev eth0 parent 1:0 handle 2:0 cbq
           bandwidth 10Mbit allot 1514 cell 8 avpkt 1000 mpu 64
4. tc class add dev eth0 parent 2:0 classid 2:1 cbq
           bandwidth 10Mbit
           rate 1500Kbit avpkt 1000 prio 1 bounded isolated
           allot 1514 weight 1 maxburst 10 defmap 1
5. tc qdisc add dev eth0 parent 2:1 pfifo limit 5
6. tc filter add dev eth0 parent 2:0 protocol ip prio 1
           handle 0x2e tcindex classid 2:1 pass_on
7. tc class add dev eth0 parent 2:0 classid 2:2 cbq
           bandwidth 10Mbit rate 5Mbit avpkt 1000 prio 7
           allot 1514 weight 1 maxburst 21 borrow
8. tc qdisc add dev eth0 parent 2:2 red limit 60KB min 15KB
           max 45KB burst 20 avpkt 1000 bandwidth 10Mbit
           probability 0.4
9. tc filter add dev eth0 parent 2:0 protocol ip prio 2
           handle 0 tcindex mask 0 classid 2:2 pass_on
```

Line 1 attaches to the root node on interface eth0 a dsmarker which copies the

TOS byte into `skb->tc_index`. Line 2 adds a filter to the root node which exists merely to mask out the ECN bits and extract the DSCP field by shifting to the right by two bits. A classful qdisc using CBQ is attached to node 2:0 (2:0 is the child of the root node 1:0) – this is in line 3. Two child classes are defined out of the 2:0 node. 2:1 is of type CBQ which is bound to a rate of 1.5 Mbps (line 4). A packet counting FIFO qdisc (`pfifo`) with a maximum queue size of 5 packets is attached to the CBQ class as the buffer management scheme (line 5). Line 6 adds a `tcindex` classifier which will redirect all packets with a `skb->tc_index` 0x2e (the DSCP for EF) to classid 2:1 – non 0x2e are allowed to fall through so they can be matched by another filter. Line 7 defines another CBQ class, 2:2, emanating out of node 2:0 – this is intended to be the Best Effort class. The rate is limited to 5 Mbps; however, the class is allowed to borrow extra bandwidth if it is not being used (via the operator `borrow`). Since the EF class does not lend its bandwidth (operator `isolated` line 4), the BE can only borrow up to a maximum of an extra 3.5Mbps. Note that in scenarios where there is no congestion on the wire, this might not be a very smart provisioning scheme since the BE traffic will probably get equivalent traffic performance as EF. The major differentiator in that case will be the priorities. The EF class' traffic will always be served first as long as there is something on the queue (prio 1 is higher than prio 8 in comparing line 4 and 7). Line 8 attaches RED as the buffer management scheme to be used by the BE class. Line 9 then maps the rest of the packets (without DSCP of 0x2e) to the classid 2:2. The description of the RED and CBQ parameters are beyond the scope of this document.

5.6 Measurements

We have measured the effect of Expedited Forwarding in a simple test network. In this section, we give some early measurement results to illustrate how our implementation can be used.

Figure 5.10 shows the configuration of our network. Additional details, traffic traces, and the configuration scripts can be found in `ftp://lrcftp.epfl.ch/pub/linux/diffserv/misc/test.19990401/` (select `jump.htm` for more convenient

browsing).

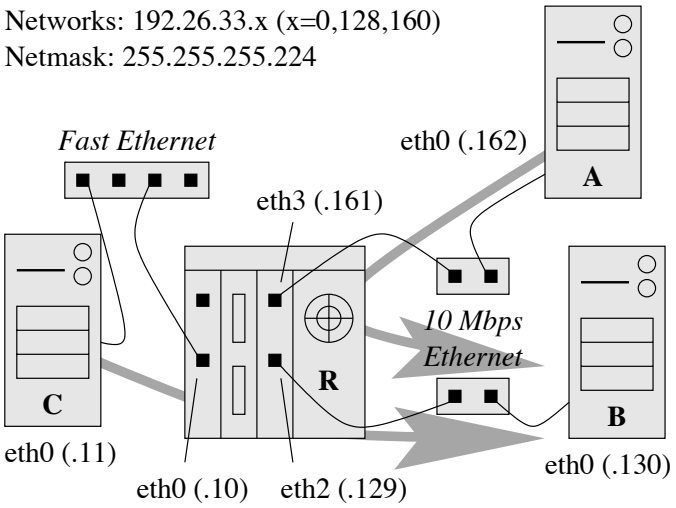

Figure 5.10: Configuration of the test network.

Hosts **A** and **C** send via router **R** to host **B**. **A** and **B** are connected via 10 Mbps Ethernet. **C** is connected via 100 Mbps Ethernet. **A** emits UDP traffic, marked with the EF default DSCP, which is shaped at the source to 100 kbps. **C** emits best-effort UDP background traffic as fast as it can. Table 5.4 summarizes the configuration.

Host	Link (Mbps)	Rate (Mbps)	Type	MAC packet size (bytes)
A	10	0.1	EF	100
B	10	n/a	n/a	n/a
C	100	≈ 55	BE	676

Table 5.4: Host configuration parameters.

The measurements show the time between consecutive packets arriving at **B**. Variation of this time is an indicator for how much the network load disturbs the flow from **A** to **B**. Note that we generated our test traffic using `ttcp`, which does not try to limit its sending rate. The shaper at **A** therefore already discards a large number of packets right at the source.

We first tested the network without background traffic (figure 5.11). As expected, traffic is very smooth. The average measured inter-packet time of 8.1 ms corresponds

to the expected 8 ms. The maximum time was 10.1 ms. (Shaper granularity was 10 ms, i.e. we did not raise the timer frequency (see section 5.3.5).

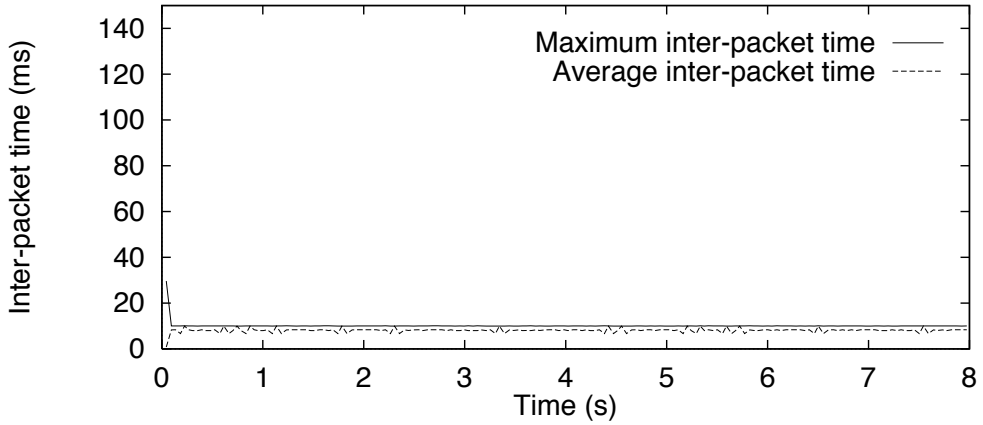

Figure 5.11: Inter-packet time for traffic from **A** in un-congested network (averaged over 40 ms)

In the following test (figure 5.12), we added a burst of background traffic from **C** but did not yet enable EF processing at router **R**. Figure 5.12 shows the effect of this on traffic from **A**. The maximum inter-packet time measured was 81 ms. 14% fewer packets from **A** reached the destination than in the previous test.

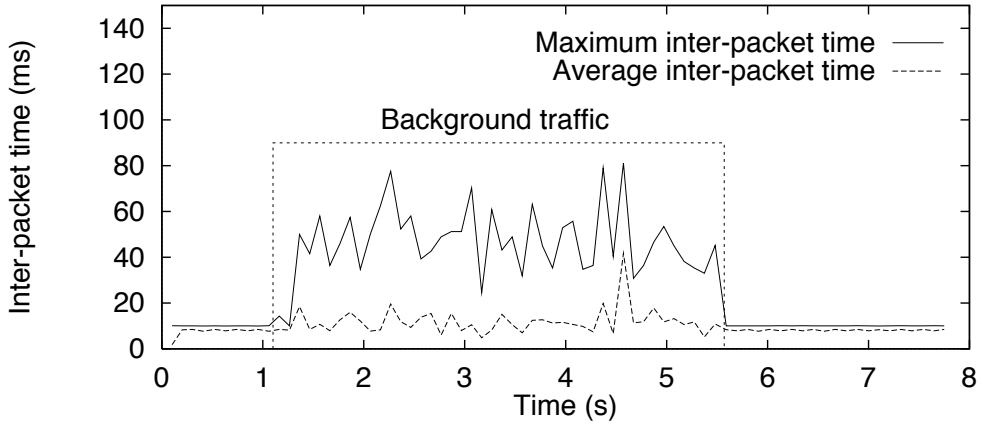

Figure 5.12: Inter-packet time for traffic from **A** with background traffic from **C**, without EF (averaged over 100 ms)

Finally, we used the script described in section 5.5.2 to enable EF in router **R** and

measured the resulting behaviour (figure 5.13). We see a significant improvement: the average inter-packet time decreases to the normal 8.1 ms, with a maximum of 18.4 ms, probably caused by losses. The number of packets arriving at **B** was only 3% lower than in the first test.

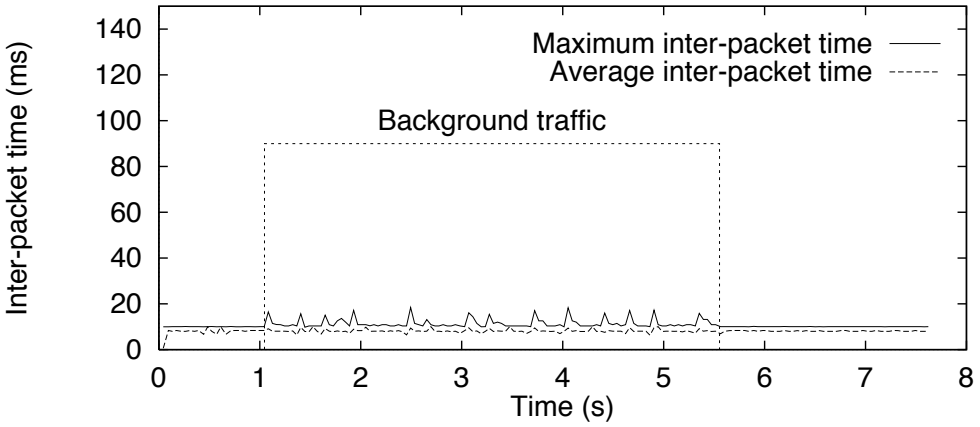

Figure 5.13: Inter-packet time for traffic from **A** with background traffic from **C**, using EF (averaged over 40 ms)

Our measurements show that we can indeed effectively shield EF flows from other traffic. Slight differences from the expected results (e.g. the 3% loss) will need further examination. Also, while these tests nicely illustrate the improvement obtained by using EF, the application-level semantics are hardly meaningful. Further experiments will therefore use less greedy sources and congestion-controlled sources. The effects of aggregation of multiple EF flows and the behaviour of the resulting traffic are also of interest beyond this specific implementation. Finally, we will also examine other PHBs than EF.

5.7 Conclusion

We have given a brief introduction to the Diffserv architecture, and we have explained how the existing infrastructure can be extended in order to support Diffserv. We have then shown how we implemented support for the Diffserv architecture in Linux, using the traffic control framework of recent kernels. We have also described

how nodes can be configured using our work, and we have given the results of measurements in a simple test configuration using Expedited Forwarding.

Our implementation provides a very flexible platform for experiments with PHBs already under standardization as well as experiments with new PHBs. It can also serve as a platform for work in other areas of Diffserv, such as edge configuration management and policy management.

Future work will focus on the elimination of a few restrictions that still exist in our architecture, the simplification of the configuration procedures, and of course in further trials to validate our implementation and to gain experience with the construction of services using Diffserv in general.

At the time of writing, efforts have begun to gradually introduce Diffserv support into what can be called "mainstream" Linux. Also, extensive tests with several applications and router equipment from various vendors are planned in a few research networks, e.g. in the Quantum Test Programme [46].

Chapter 6

SRP implementation

In this chapter, we describe the internals of a prototype implementation of SRP on Linux. The implementation is based on the support for Differentiated Services on Linux as described in chapter 5.

6.1 Overview

The implementation is structured such that only per-packet processing is done inside the kernel, while operations not directly involving individual packets are carried out by user-space processes. The user-space processes are called SRP demons. For simplicity, the implementation merges the SRP demons for sources, destinations, and routers in a single program called `srpd`.

At the sender and at the receiver, a new SRP-specific classifier called `cls_srp` monitors flows aggregated per remote IP address. `srpd` uses the metering results to estimate the reservation. At the source, `cls_srp` also selects the packet type based on rates configured by `srpd`.

In routers, a simple two-priority configuration is used, similar to Expedited Forwarding (see section 5.5). The amount of *request* and *reserved* traffic is monitored by `srpd`, which estimates the reservation and limits *request* traffic accordingly.

6.2 Traffic control elements

Only small changes had to be made to support SRP. The only entirely new component is the SRP-specific classifier `cls_srp`. `cls_srp` combines several selection and metering functions that would be difficult and inefficient to construct using existing traffic control elements.

The second change was to add a count of bytes accepted per entry to `cls_tcindex`. This is necessary, because `cls_tcindex` is the only place in a router where *reserved* and *request* traffic can be distinguished.

Finally, policers did not work properly if updated (replaced) frequently. We implemented a function that allows for a smoother transition from one policer to the next one.

6.2.1 `cls_srp`

The `cls_srp` classifier is used in the SRP source and in the SRP destination. It performs the following tasks:

- select local packets (optional)
- check if the packet is of type *reserved* or *request*
- retrieve rates and counters related to the remote address
- check rate and degrade if necessary (source only)
- add packet size to statistics

Figure 6.1 shows the structure of the `cls_srp` classifier. The local address check is only performed if a list of local addresses is present. Single-homed hosts can usually omit this check.

Note that, unlike `cls_tcindex`, `cls_srp` does not bit-shift the value of `skb->tc_index` before comparing it with the values for *reserved* and *request* packets. `cls_srp` returns `TC_POLICE_UNSPEC` for packets which are not SRP packets, i.e. which are neither *reserved* nor *request*.

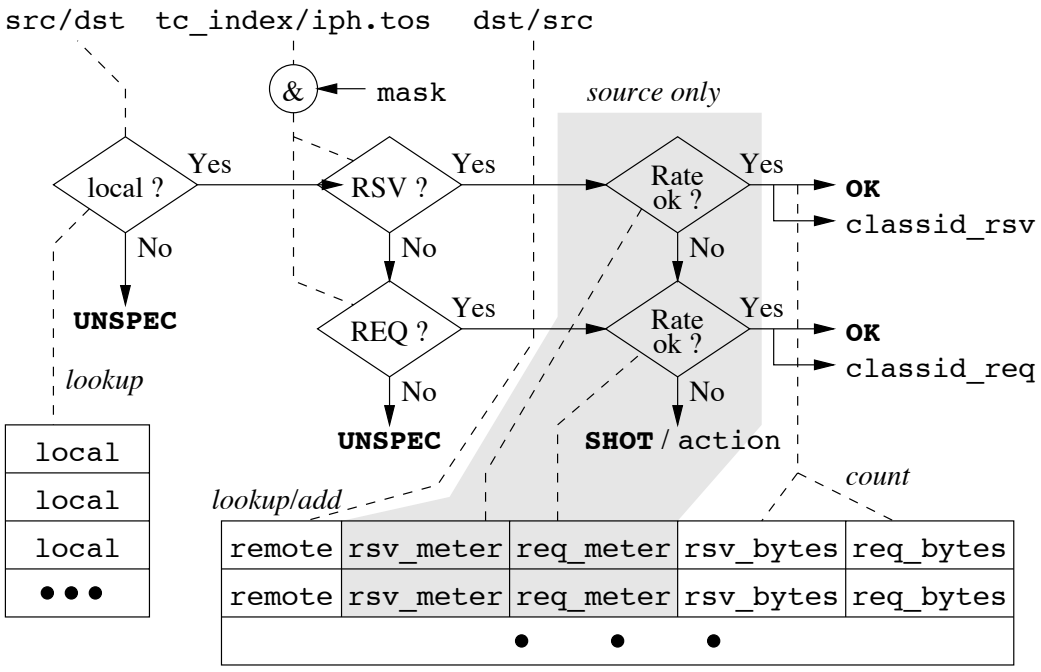

Figure 6.1: The SRP classifier.

When used at the destination, `cls_srp` can't use `skb->tc_index`, because it is not yet set at the time when the ingress classifier is invoked. Therefore, `cls_srp` uses directly the value of the IPv4 TOS byte. Note that the `mask` is still applied.

Next, the remote address of SRP packets is used to retrieve an entry containing the current policing information and statistics. If no entry exists, a new one is created. The new entry has no policing information, so any packets matching this entry are discarded until `srpd` adds this information.

If used at the source, `cls_srp` now checks the rate and degrades the packet, if necessary. This is accomplished with the mechanisms normally used for policing. If the packet fails the rate check for *request* traffic, `cls_srp` returns the `action` value returned by the last policing function checked. If no policing function was invoked, i.e. because none is configured, `cls_srp` returns `TC_POLICE_SHOT`, which normally causes the packet to be discarded.

Finally, if a packet has been accepted as either *request* or *reserved*, its size is added to the corresponding statistics counter, and the class ID for the packet type

is returned.

Table 6.2.1 shows the parameters that can be configured in the `srp` classifier. The upper part of the table shows parameters of the classifier itself. The lower part shows parameters of each remote address entry.

Variable name / `tc` keyword	Value	Default
`source`	none (flag)	absent
`local`	list of IPv4 addresses	empty
`mask`	0...0xffff	0xffff
`tcindex_rsv`	0...0xffff	0
`tcindex_req`	0...0xffff	0
`classid_rsv`	class ID	0:0 (no match)
`classid_req`	class ID	0:0 (no match)
`remote`	IPv4 address	none
`rsv_meter`	policing spec.	none
`req_meter`	policing spec.	none
`rsv_bytes`	`unsigned long` (read-only)	0
`req_bytes`	`unsigned long` (read-only)	0

Table 6.1: Configuration parameters of `cls_srp`.

Most of the parameters have already been discussed. `cls_srp` acts in source mode if `source` is specified, in destination mode otherwise. `tcindex_res` and `tcindex_res` contain the value `skb->tc_index` or `skb->nh.iph->tos` is compared with after applying the `mask`.

Note that `rsv_bytes` and `req_bytes` can only be read. Each counter wraps around to zero when passing `ULONG_MAX`.

6.2.2 `cls_tcindex` extension

The variable `bytes` was added to elements of `cls_tcindex`, as shown in table 6.2.2.

This variable can only be read. The counter wraps around to zero when passing `ULONG_MAX`.

Variable	`tc` keyword	Value	Default
`bytes`	n/a	`unsigned long`	0

Table 6.2: New parameter of `cls_tcindex`.

6.2.3 Smooth policer updates

Finally, when updating a policer, it is always reset to the maximum "credit". When making frequent small adjustments, the rate effectively accepted by the policer will therefore be far above the configured rate.

We added a new function `tcf_police_replace` that replaces an existing policer with a new one, computes the current "credit" of the old policer, and copies it over to the new policer. It also performs the actual replacement operation.

`cls_srp` and `cls_tcindex` were changed to use `tcf_police_replace` when updating policers.

6.3 Node configuration

This section describes how traffic control elements are combined at SRP nodes, and how they interact with the SRP demons.

6.3.1 Sender

As shown in figure 6.2, the traffic control part at an SRP sender is fairly straightforward: applications mark packets for which a reservation is desired using the `srp_reserve` API function (see section 6.4). `cls_srp` then determines whether packets fit in the rate budget for *reserved*, or at least *request* traffic to the respective destination, and classifies them accordingly. Then packets are queued, and finally marked and transmitted.

The queuing mechanism could use different drop or delay priorities, depending on the packet type. For simplicity, we assume no congestion at the source, so a simple FIFO is sufficient.

All of the intelligence of the SRP source resides in the SRP demon, which uses

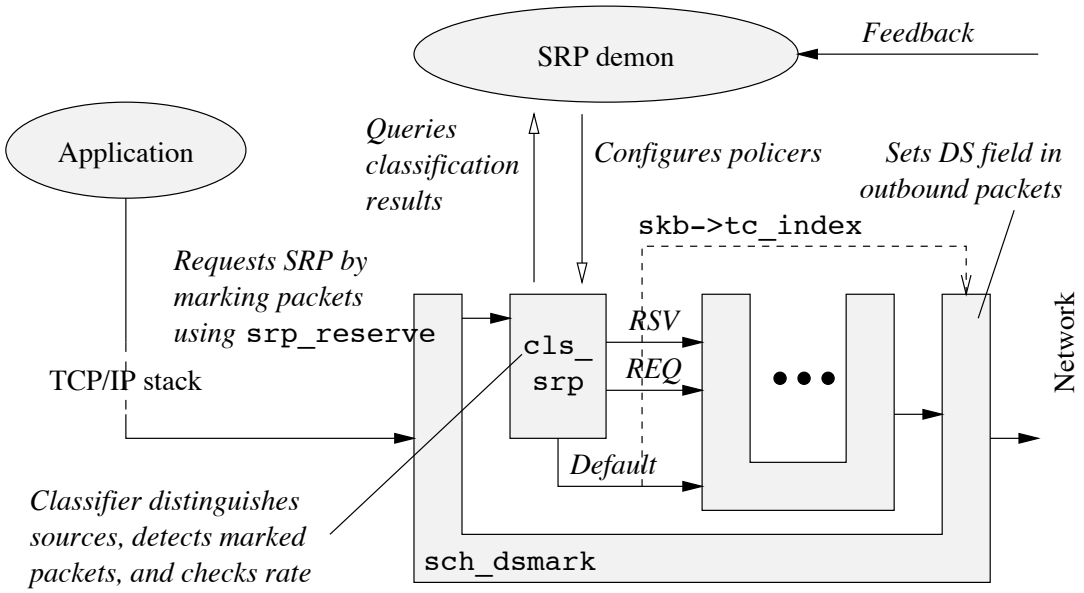

Figure 6.2: Traffic control elements at SRP sender.

statistics retrieved from `cls_srp`, and the feedback received from the destination(s) to calculate the reservation to each destination and the acceptable rate of *request* traffic. It then configures the policers in `cls_srp` accordingly.

6.3.2 Receiver

SRP receivers use the ingress policing infrastructure to pass the packet flow through `cls_srp`, as shown in figure 6.3.

`cls_srp` simply counts the bytes received from each remote SRP node. This information is periodically retrieved by `srpd` and then transmitted in feedback messages to the respective nodes. The SRP demon also removes stale entries from `cls_srp` after a while.

6.3.3 Router

Routers use a configuration very similar to what is also used for Expedited Forwarding (see section 5.5 for details). As shown in figure 6.4, the only difference is in the initial classification, which distinguishes three instead of only two classes.

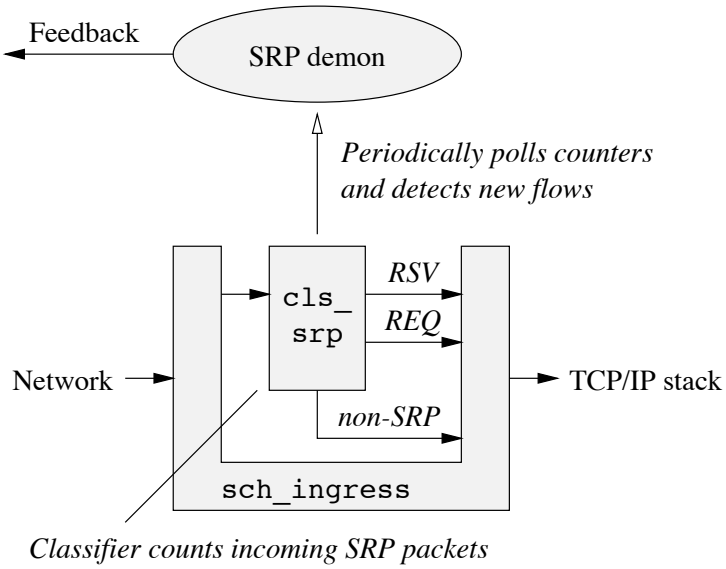

Figure 6.3: Traffic control elements at SRP receiver.

The first `tcindex` classifier identifies *reserved* and *request* packets, and limits the amount of accepted *request* packets. It may optionally also police *reserved* traffic.

The SRP demon polls the statistics gathered by the innermost queuing disciplines to estimate the current reservation. This estimate is then used to set the rate or credit available for *request* traffic.

When leaving the `dsmark` queuing discipline, packets are re-marked, e.g. the DSCP of *request* packets that have been downgraded to *best-effort* is changed accordingly.

6.4 SRP API

Only a very rudimental API is provided in the prototype implementation. It allows applications to attempt a reservation for the traffic they generate at a socket. No explicit feedback on the success of the attempt is provided. Instead, the application is expected to have its own means for detecting failure to communicate, and to abandon unsuccessful reservation attempts after a while.

Note that the SRP demon has final control over what is sent to the network, so

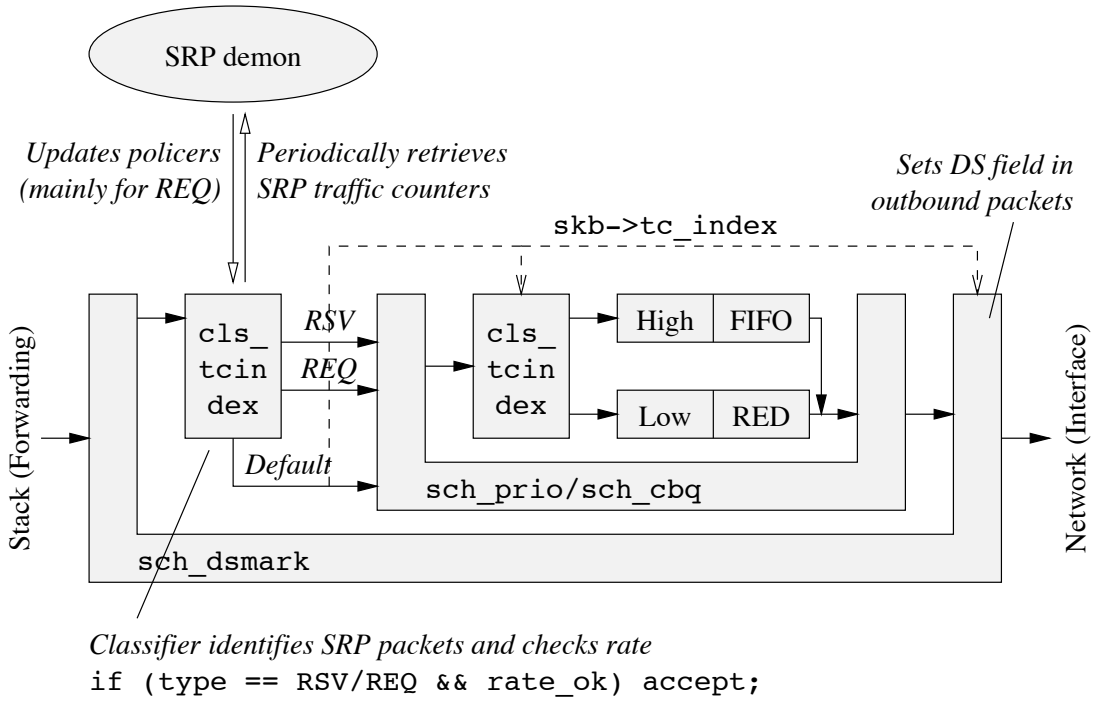

Figure 6.4: Traffic control elements at SRP router.

correct performance of the node can be ensured, even if an application persistently tries to obtain a reservation which is denied by the network.

The API consists of a single function, defined in `/usr/include/srp.h`:

```
int srp_reserve(int sd,int do_reserve);
```

If `do_reserve` is non-zero, a reservation is attempted. Otherwise, no reservation is made, and any existing reservation expires. `srp_reserve` returns zero if the attempt was successfully initiated. Otherwise, it returns a negative value and sets `errno`.

`srp_reserve` internally uses the `IP_TOS` socket option to set the DS field of outbound packets to *reserved*.

6.5 Known restrictions

At the time of writing, the prototype implementation of SRP has still the following restrictions:

- Only supports IPv4.
- Turn-around times are comparably long, because communication between `srpd` and the kernel is performed via `tc`.
- `cls_srp` should have a policers with an initial or global budget of bandwidth for *request* traffic, such that the first packets of a flow for which a new remote address entry is created are not lost.
- For simplicity, feedback is sent about once per second, independent of the rate at which changes occur.
- Likewise, the SRP senders and routers retrieve statistics and process feedback (senders) only once per second.

6.6 Conclusion

In this chapter, we have shown how a working prototype of SRP for sources, destinations, and routers, can be built on top of the existing support for Differentiated Services with the addition of only one new component, and minor modification to two other components. While the prototype is too constrained and also too inefficient for production use, it serves as a proof of concept, and allows to experiment with SRP in a real network, as shown in the next chapter.

Chapter 7

SRP measurements

We have run the SRP prototype described in the previous chapter in a simple test network. In this chapter, we discuss some measurements obtained during these test runs.

7.1 Test scenario

The purpose of the tests was to verify that the reservation properly ramps up, and that the router can control how much bandwidth a source claims. Isolation of reserved traffic from best-effort traffic has already been examined in section 5.6.

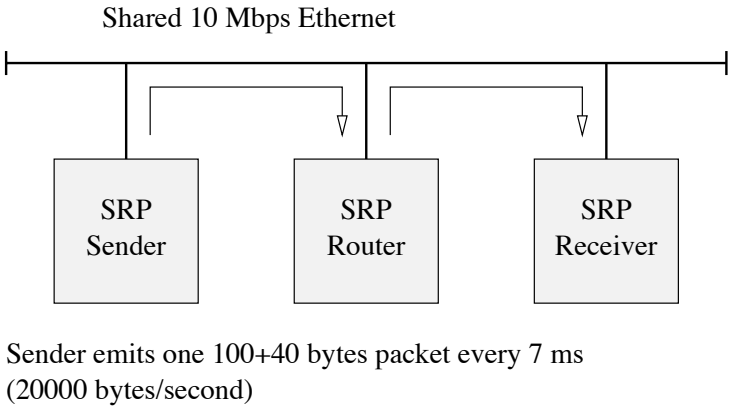

Figure 7.1: Test network configuration.

The test network consisted of three nodes on the same shared 10 Mbps Ethernet,

as shown in figure 7.1.

The source sends one UDP packet with a payload of 100 bytes every 7 ms. The resulting IP packet has a size of 140 bytes, so the required bandwidth is 20'000 bytes/s or 160 kbps. The source was active for 120 seconds in each test run.

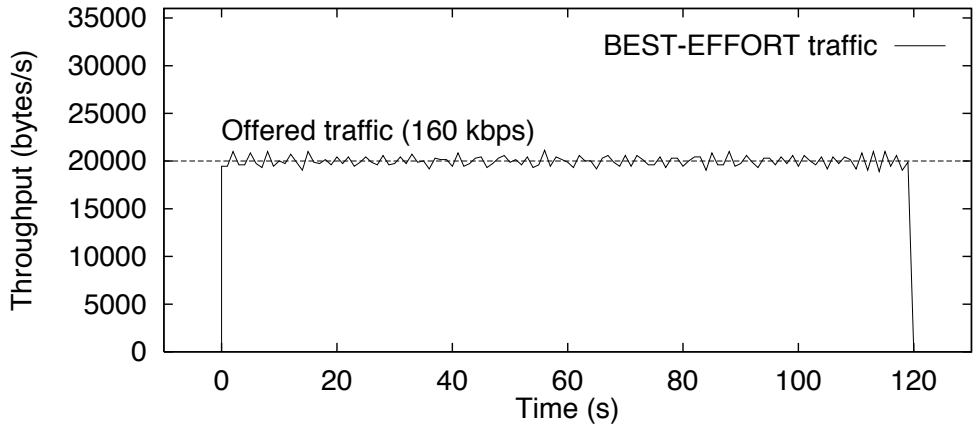

Figure 7.2: Over-provisioned best-effort network; seen from receiver.

7.2 Measurement results

We first tested the behaviour of the system when the available bandwidth is large enough that the router does not constrain the reservation.

In the first test (figure 7.2), SRP was not used at all. The unloaded network is more than sufficiently fast for the offered traffic.

The next test involved SRP, but there was still more bandwidth available than necessary. Figure 7.3 shows the traffic at the receiver. We can see that the reservation ramps up quickly and then remains stable.

A more interesting test is shown in figure 7.4. Now the bandwidth available for reservation is below the offered traffic. We can see that the sender tries to increase the reservation, but the router refuses most of the *request* packets.

The process of requesting more bandwidth is more clearly visible in the case shown in figures 7.5 and 7.6. Here the bandwidth is limited to 50 kbps. The source

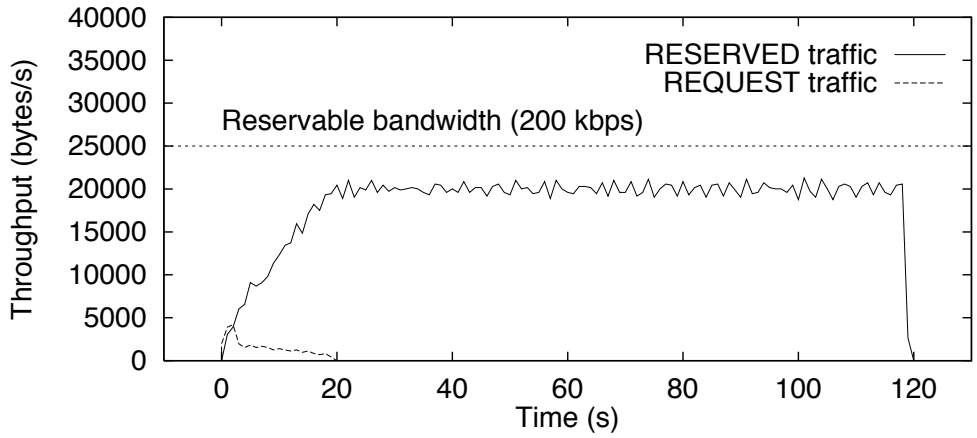

Figure 7.3: Sufficient reservable bandwidth; seen from receiver.

constantly emits *request* packets, but the router refuses to increase the reservation and discards most of them.

The spikes at the beginning of each transfer occur because policing functions in Linux traffic control are initialized to the maximum credit. The burst sizes at the source were 3 kB for *reserved* and 2 kB for *request* traffic.

7.3 Conclusion

We have shown that the prototype implementation of SRP behaves in the expected way in a simple configuration. Testing of SRP in more advanced scenarios is planned as future work. Also, our results suggest that more flexibility in the initialization of policing functions in Linux traffic control may be desirable.

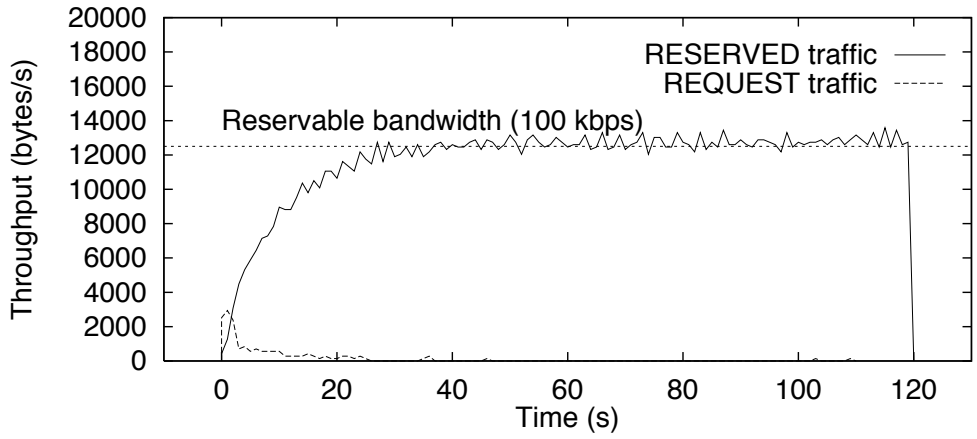

Figure 7.4: Limited to 100 kbps; seen from receiver.

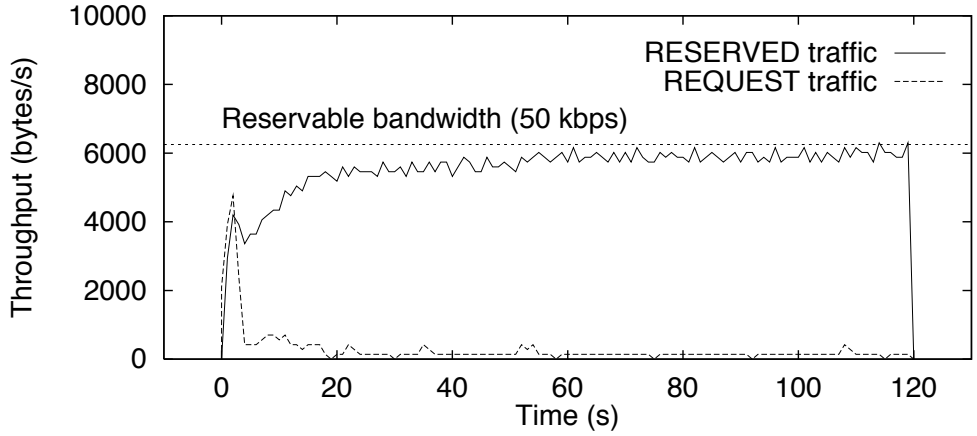

Figure 7.5: Limited to 50 kbps; seen from sender.

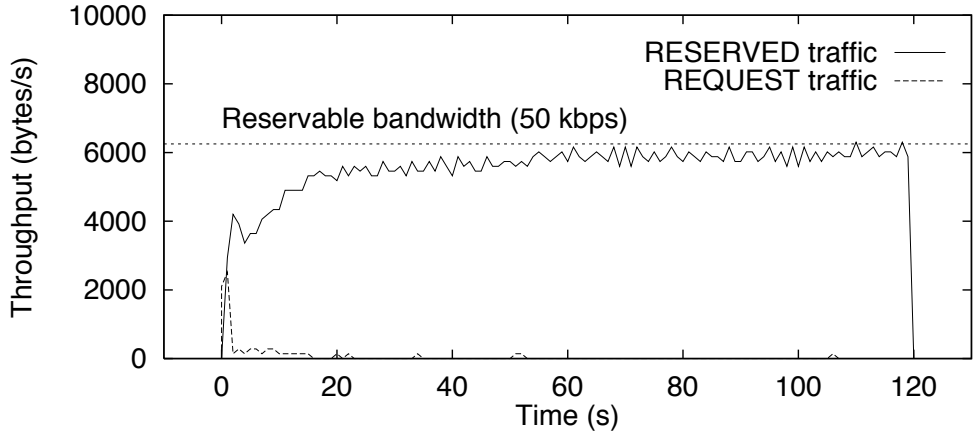

Figure 7.6: Limited to 50 kbps; seen from receiver.

Chapter 8

Conclusion

The chapter summarizes the work presented so far, gives a critical appraisal of the relevance of the work done in this project, and concludes with an outlook on the future evolution of the projects or their results.

8.1 Summary

The work presented in this thesis is strongly focused on direct practical applicability: ATM on Linux is intensely used in the whole world. Arequipa was completely implemented and tested with real-life equipment. The concepts of Linux traffic control have become more accessible due to the description produced in the course of this work, and there is now frequently discussion on new interesting uses and extensions on mailing lists related to Linux networking. The implementation of Differentiated Services on Linux is attracting a lot of interest, and is now also allowing the on-going standardization work in Diffserv to be verified against the properties of an existing implementation.

In the case of ATM and of Diffserv, concepts reaching beyond the immediate focus of industry have been designed and successfully implemented: Arequipa provided a rapid deployment option for users needing Quality of Service, and SRP is addressing the lack of coherent signaling in the Diffserv network architecture.

8.2 Relevance

The ATM on Linux project is widely recognized in industry and academia, and has yielded the de facto standard implementation under Linux. Although still quite young, the Diffserv on Linux project seems to be heading in the same direction.

While Arequipa's reliance on wide deployment of ATM confines it to a niche, general deployment of some form of Diffserv functionality in most core networks is likely to occur.

Current trends towards approaches that combine RSVP and Diffserv may seem to eliminate the need for a properly integrated, homogeneous reservation architecture for Diffserv. It should however be noted that, based on past experience with Diffserv (e.g. work on Assured Forwarding in the Diffserv group has shown that managing even a comparably small set of priorities can be surprising difficult and controversial), smooth integration of Intserv and Diffserv cannot be taken for granted. Also the often postulated concept of a bandwidth broker that oversees resource usage and allocation in large areas of a network will certainly need to be refined when concrete implementations are attempted.

All this is evidence that there is a place for rival architectures such as SRP. Also, elements originally developed in a different but sufficiently similar context (e.g. estimators for aggregate traffic or scalable policers) may be usefully applied in whatever will become the QoS architecture in the Internet.

8.3 Future work

The availability of a simulator and a running prototype of SRP allows independent verification of its properties and work on its shortcomings. In particular, at the time of writing, a research group is investigating the practical usefulness of the scalable policing design proposed for SRP, and another group inside ICA is working on improving the estimation in routers.

The documentation of Linux traffic control will continue to be maintained and extended, such that prospective implementors of new traffic control functionality will have a reference for the surrounding infrastructure, and that users of traffic

control elements will be able to develop a solid understanding of the underlying concepts.

The results of the ATM on Linux and Diffserv on Linux projects are currently being integrated into the mainstream code base of Linux, which will make them easily available for millions of users interested in learning, experiments, or production use. We hope that the availability of an openly available implementation will also help the on-going standardization process of Diffserv.

Appendix A

ATM on Linux

Since the beginning of 1995, ATM support is being developed for Linux. By now, Linux supports most functionality that is required for state of the art ATM networking. This chapter briefly introduces relevant ATM concepts and presents the current status of development on Linux.

A.1 Introduction

ATM (Asynchronous Transfer Mode, [9], see also [47] for a comprehensive overview of ATM technology) is a network technology for modern high-speed integrated services networks. It is not only popular in WANs, for high-speed backbones interconnecting LANs, and for access networks (e.g. ASDL), but ATM also offers a rich set of features to support guaranteed Quality of Service (QoS; bandwidth, end-to-end delay, etc.), which is necessary for many multimedia applications.

In order to create an ATM platform for research and education, the Laboratoire de Réseaux de Communication (LRC, later ICA) of EPFL is developing ATM support for Linux.

The Web main page of the ATM on Linux project with pointers to the latest ATM on Linux distribution and related news is `http://icawww1.epfl.ch/linux-atm/`

A.2 ATM basics

ATM is designed for demanding data and multimedia communication, such as audio and video transmission, and high-speed data transfer. The design of ATM has been strongly influenced by the telecommunication community, and therefore ATM differs in many ways from data network architectures like today's Internet. The probably most important differences are the following:

- ATM is connection-oriented
- ATM supports guaranteed QoS ("Quality of Service")
- ATM clearly distinguishes between end systems and "the network"

All these concepts have their counterparts in the telephony network: you have to establish a connection before you can communicate with the other party, the QoS (i.e. that you get reasonable bi-directional voice transmission) is guaranteed and doesn't depend on the network load, and your telephone is very different from, say, a PBX.

Another difference is that ATM sends data in tiny cells with a fixed size of only 53 bytes instead of in variable-size frames. While this difference is important at the lowest protocol layers, higher layers typically use larger units which are then transformed from/to cells by a so-called "ATM adaption layer" (AAL, [48]).

Figure A.1 shows the structure of an ATM network. The network itself consists of interconnected switches. Two types of networks are distinguished: "private" networks are typically company or campus networks, and "public" networks correspond to what is offered by telephone carriers. The standardized interface between end systems ("hosts") and the ATM network is called the "user-network interface" (UNI). The UNI defines several types of physical media (i.e. multi-mode fiber, UTP-5, etc.), many bit rates (ranging from only a few Mbps to 155 Mbps and more), line codings, configuration and signaling protocols, etc.

The most commonly used version of the UNI is 3.1 [49], but many providers of ATM stacks have already implemented the next version, 4.0 [10, 18], or are working on it. UNI 4.0 adds many new features to UNI 3.1, the most interesting ones

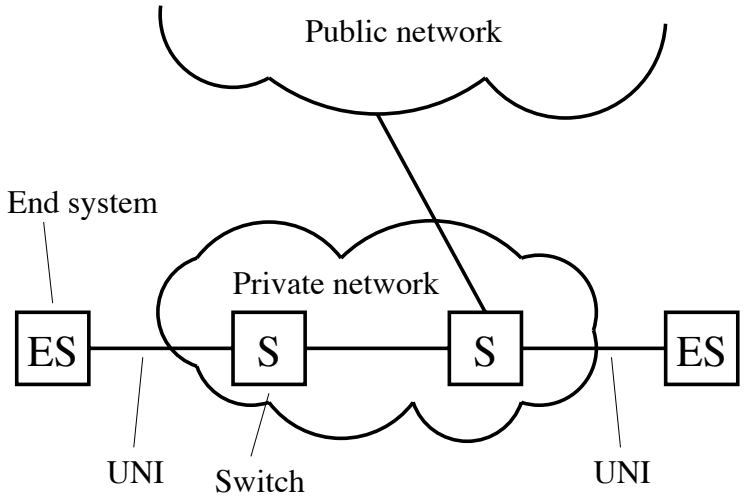

Figure A.1: General structure of an ATM network

are probably a scalable point-to-multipoint ("multicast") mechanism and support for ABR ("Available Bit Rate"), a traffic class with congestion control inside the network.

There are two mechanisms for setting up ATM connections: the simple way is to configure each switch individually (a bit like it was done in the early days of telephony, where operators had to physically connect calls on switchboards). Such connections are called "permanent virtual circuits" (PVCs). A more convenient way of setting up connections it to "dial" them, which is called "signaling" in ATM terminology. "Switched virtual circuits" (SVCs) are set up using signaling. ATM signaling is based on the protocols DSS2 (see Q.2931 [50] for unicast and Q.2971 [51] for multicast), which in turn use the so-called SAAL [52, 53, 54] to transport signaling messages.

Figure A.2 illustrates ATM signaling: first, the caller sends a SETUP message towards the destination (1). This message is processed at every single switch. If the destination accepts the call, it returns a CONNECT message (2). Again, this message is seen by all switches. When the CONNECT message reaches the destination, the data connection is established and data can be exchanged between both

end systems (3).[1] Note that the switches don't have to interpret what is sent on the data connection.

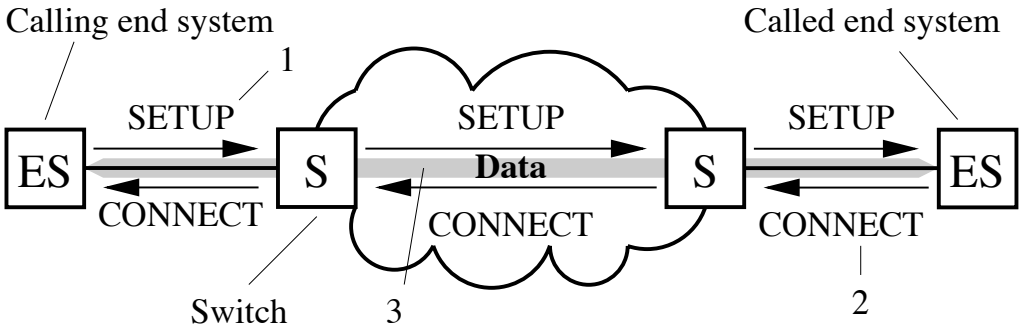

Figure A.2: Signaling message flows

Two mechanisms that are closely related to signaling are address configuration and a directory service. Addresses are configured either manually or automatically, using the "interim local management interface" (ILMI, [49]), which is based on SNMP [55].

An ATM NSAP address (see section 5.1.3.1 of [49]) has a length of 20 bytes. Human beings therefore usually prefer to use names instead of numeric addresses. This can be accomplished either by using a hosts file or by using a distributed directory service. ATM Forum has specified such a directory service called ANS ("ATM Name Service", [56]), which is based on BIND ("named").

At the time of writing, ATM on Linux supports PVCs and SVCs with UNI 3.1 signaling. Support for UNI 4.0 signaling is being worked on. The ILMI demon was contributed by Scott Shumate. ANS support was contributed by Marko Kiiskilä.

A.3 ATM and the real world

ATM purists may dream of a world where all computers, TV sets, telephones, etc., are connected to a big ATM cloud consisting of many interconnected ATM networks, but the real world is different: connectionless IP networks, typically without

[1] This is slightly simplified. ATM signaling also allows acknowledgements for the SETUP message and it requires an acknowledgement for CONNECT.

user-accessible QoS concepts, play the dominant role, and "native" ATM applications are a minority.

The first step in running IP over ATM is to have a means to carry IP packets on ATM. This is mainly an encapsulation issue, defined in RFC1483 [57]. With this alone, IP can be run over ATM using PVCs.

For SVCs, also a way to resolve IP addresses to ATM addresses is needed. The IETF currently uses an approach called "classical IP over ATM" that is based on an extension of ARP, called ATMARP [58, 59]. ATMARP works like this (see also figure A.3): each IP subnet has one ARP server (C). When a client (**A**, **B**) starts, it registers its own IP and ATM addresses at the ARP server (1). Now, if client **A** wants to send data to client **B**, but it only knows **B**'s IP address, it sends an ATMARP request (2) to the server. If the server knows **B**'s addresses, it responds with an ATMARP reply (3), containing **B**'s ATM address. **A** can now establish an SVC to **B** and send data (4).

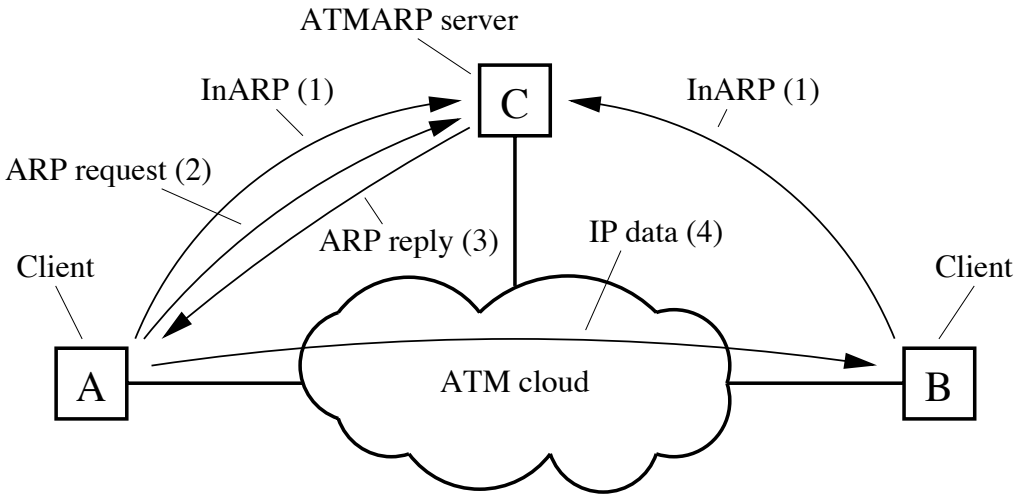

Figure A.3: ATMARP message flows

ATM Forum has defined a similar service, called "LAN Emulation" (LANE) [60, 61]. LANE tries to provide exactly the functionality one would obtain from, say, an Ethernet. Therefore, is can also carry other protocols than just IP and it supports multicast (and even broadcast). One disadvantage of LANE with respect to classical IP over ATM is that the maximum IP packet size is limited to 1500

bytes, like on Ethernet, whereas the default maximum IP packet size for classical IP over ATM is 9180 bytes ([62]).

Work from IETF and ATM Forum has been merged in MPOA ("MultiProtocol Over ATM", [63]), which aims to overcome many of the limitations of classical IP and LANE. In particular, MPOA allows the use of "shortcut" connections which bypass intermediate routers.

Furthermore, work has been done on integrating IP mechanisms for negotiating QoS parameters (e.g. RSVP [13]) with ATM [64].

ATM on Linux supports IP over ATM for PVCs and SVCs as defined by RFC1577 [58] and others. Comprehensive support for LANE, including complete LANE server functionality, has been contributed by Marko Kiiskilä [65]. Later on, Heikki Vatiainen has taken care of LANE maintenance and also also contributed an implementation of MPOA.

Because standard IP currently supports neither direct ATM end-to-end connectivity beyond subnet boundaries nor negotiation of QoS aspects, LRC has designed an extension of ATMARP called Arequipa ("Application Requested IP over ATM". Arequipa allows applications to request a direct ATM connection for their exclusive use with TCP/IP protocols. The applications can also determine exactly what QoS will be available to them. We describe Arequipa in appendix B.

A.4 ATM on Linux details

This section describes the development process of ATM on Linux and the current implementation.

A.4.1 Drivers

The first step in bringing ATM to Linux was to find ATM adapters that offered sufficient performance, that were available on the market, and for which programming information was openly available. The search for such adapters turned out to be quite difficult, mainly because at the end of '94, many companies only had products for Sun's SBus, and very few adapters for the PCI bus were available on

the market.

Eventually, we chose to use the products from ZeitNet and from Efficient Networks. In spring 1995, a driver for the Efficient Networks EN155p adapter was written, and a driver for the ZeitNet ZN1221 adapter followed soon thereafter. Both adapters are PCI bus cards and run ATM at 155 Mbps over multi-mode fiber.

A.4.2 ATM socket API

In order to send data over even only PVCs, a device driver alone isn't enough, but also an API is needed. Although ATM Forum is defining a semantic API [66], this description is far too general for any concrete implementation. Therefore, based on the BSD socket API, a native ATM API was defined for PVCs and later for SVCs too [67]. This was done in parallel with device driver development.

After some code was written to implement the ATM-specific socket and protocol functions, which interface between the common socket layer and the device drivers [68], early tests were possible. (See figures A.4 and A.5 for the protocol stack.) Since IP over ATM encapsulation is comparably easy to implement, support for classical IP over ATM over PVC was added shortly thereafter.

Figures A.4 and A.5 illustrate the user-space and kernel-space, respectively, parts of the Linux networking protocol stack. Figure A.5 shows the "traditional" IP over Ethernet (or SLIP, PPP, etc.) stack on the right side, the elements added by the ATM stack are on the left side.

A.4.3 Single-copy

At that time, performance tests revealed that throughput left much to be desired: instead of the theoretical maximum of 135.6 Mbps for user data with raw ATM, only a throughput of approximately 100 Mbps was obtained under ideal conditions. The results for IP over ATM were much worse. The culprit was easily found: because PCs tend to have a slow memory interface, the comparably large number of copy operations in the kernel created a bottleneck.

[1] This is only an interface to the signaling demon, which performs the actual exchange of Q.2931 signaling messages.

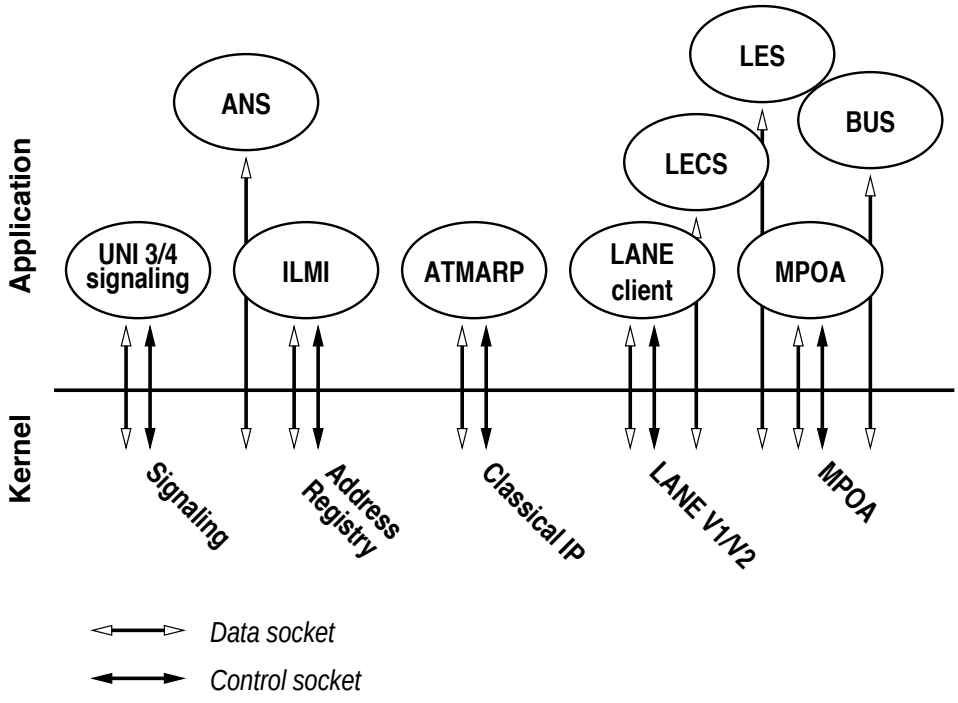

Figure A.4: User-space parts of the Linux networking stack.

The problem was resolved using a concept called "single-copy", where data is copied directly between user space and the device driver, without additional copying to kernel buffers [69]. With single-copy, transfer rates of up to 130 Mbps are possible on Linux PCs with native ATM when using sufficiently large datagrams.

Because this single-copy implementation was limited to native ATM connections, and because advances in commodity PC hardware have improved some of the performance bottlenecks, single-copy was removed in later versions of ATM on Linux. There is an on-going discussion among Linux kernel developers on the implementation of more general single-copy mechanisms, i.e. ones that are also applicable to other transports than native ATM.

A.4.4 Signaling

Since PVCs are too inflexible for most purposes, the logical next step was to start to implement signaling. ATM signaling mainly consists of the actual signaling protocol DSS2 and the transport protocol SSCOP [53]. Because those protocols are

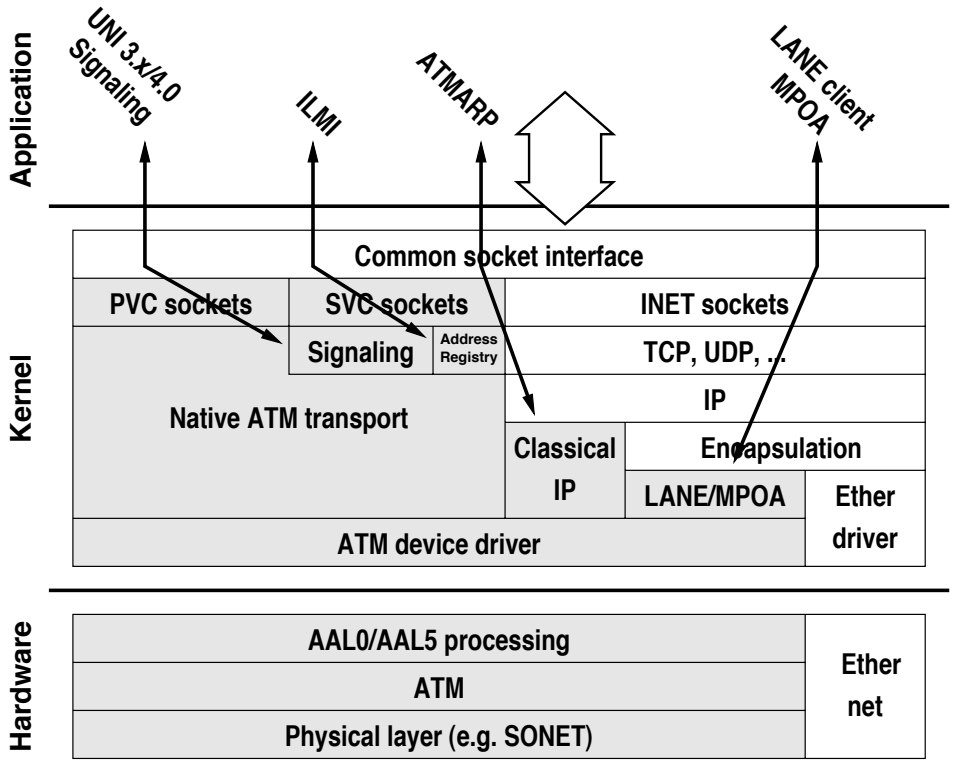

Figure A.5: Kernel-space parts of the Linux networking stack.

rather complex but do most of their work only when connections are established or torn down, we decided to implement them in a demon in user mode.

Figure A.6 illustrates a typical connection setup: When started, the signaling demon creates a PVC to communicate with the signaling entity in the network (1), and a special SVC socket (2), which is used to exchange signaling messages with the kernel. A very simple protocol is used for the communication between the kernel and the signaling demon.

When an application requests a connection to a remote party (3), the kernel sends a message to the signaling demon (4), which then performs the signaling dialog with the ATM network (5). When the connection is established, the signaling demon indicates this to the kernel (6), which then sets up the local part of the data connection (7) and notifies the application (8). Incoming calls are handled in a similar way.

Later on, a demon was also added for the "interim local management interface"

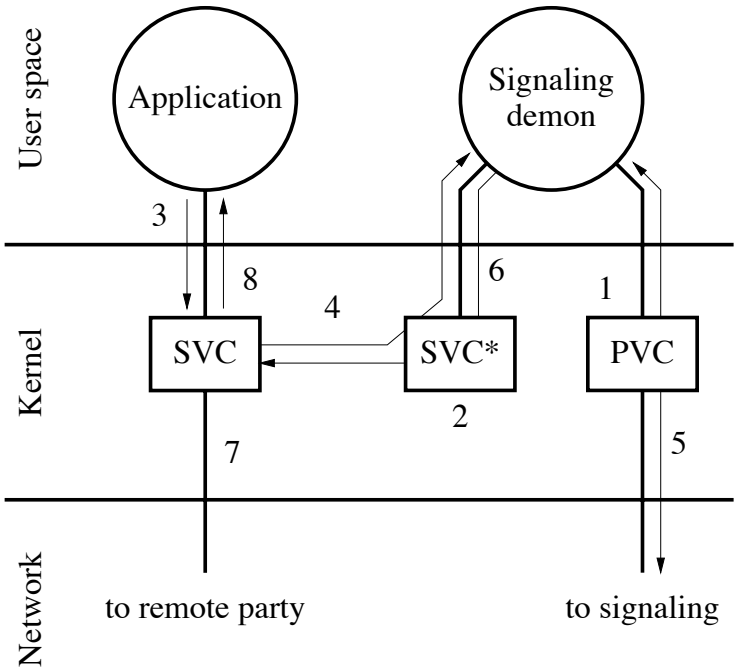

Figure A.6: Signaling procedure in the kernel

(ILMI, [49]) protocol, which is used mainly for configuration purposes, such as address auto-configuration. This demon was contributed by Scott Shumate of the University of Kansas.

A.4.5 ATMARP

After basic SVC functionality was available, ATMARP had to be implemented to make use of SVCs for IP over ATM too. The approach chosen is similar to signaling: a demon process implements the ATMARP protocol and only a simple table for ARP lookups is kept in the kernel, see figure A.7.

When started, the ATMARP demon creates a special socket (1) to communicate with the kernel. When an application wants to send data to an IP destination (2) on the same IP subnet, TCP/IP performs an ARP table lookup (3). If no ATM connection exists for that destination yet, the kernel sends a message to the ATMARP demon requesting resolution of the IP address (4). If the local machine acts as the ATMARP server for the IP subnet, only a lookup in the address resolution

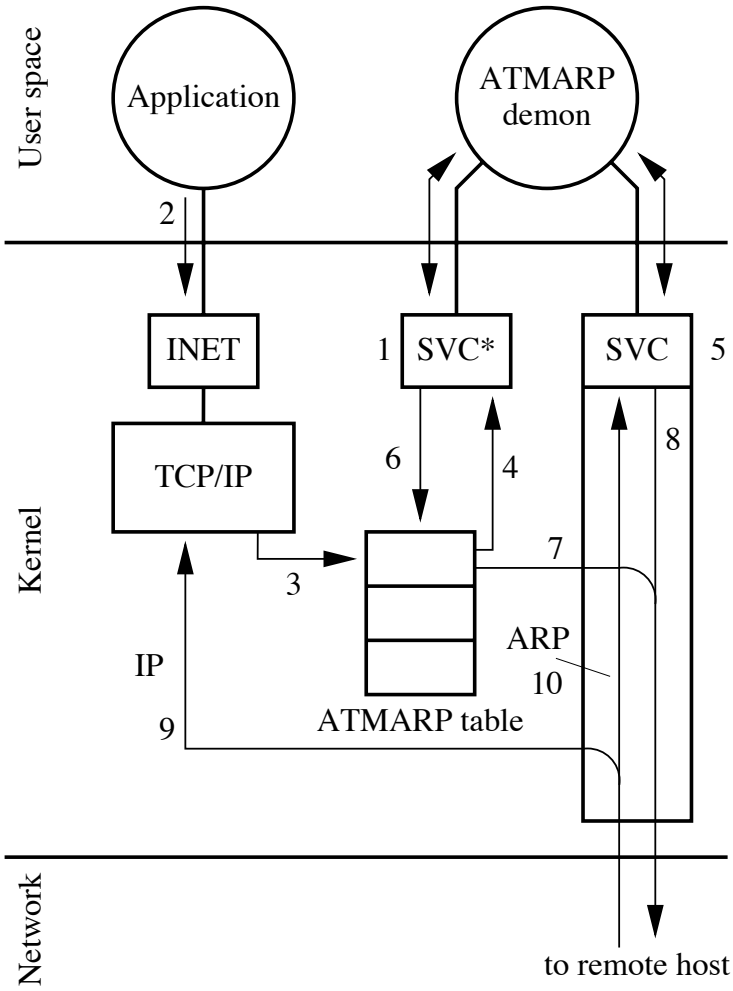

Figure A.7: ATMARP procedure in the kernel

table of the ATMARP demon is performed.[2] Otherwise, the ATMARP demon first searches its own table, and if no entry is found, it sends a resolution request to the ATMARP server. If the resolution succeeds, a new SVC is opened for the destination host (5) and it is entered in the kernel ATMARP table (6). Now, IP packets can be sent to the remote host (7).

Note that the ATMARP demon still owns the SVC and that it can send ARP packets to the remote host (8). Also, all incoming packets are examined and they're either sent to the TCP/IP stack (IP, 9) or to the ATMARP demon (ARP, 10).

[2]In addition to the entries that are also in the kernel ATMARP table, the ATMARP demon also caches mappings for hosts to which no connection exists.

A.5 Conclusion

A brief introduction to the most important concepts of ATM in today's networking world was given and it was illustrated that ATM on Linux supports all the respective mechanisms, and how this is accomplished. For some particularly interesting cases, details about the actual implementation were given.

At the time of writing, ATM on Linux has probably been installed at at least one thousand sites. Several other researchers have based their work on it, and also groups in industry are using it for various purposes. Integration into the "mainstream" Linux kernel is in progress, which will further simplify the installation and configuration process.

Appendix B

Arequipa

Arequipa is a method for providing the quality of service of ATM to TCP/IP applications without requiring any cooperation in the network between IP and ATM. It does not need any modifications in the ATM or IP networks; however, it requires end-to-end ATM connectivity.

In this chapter, we present the design, implementation and our first experience with Arequipa.

B.1 Introduction

A few years ago, it was commonly assumed that any architecture for providing Quality of Service on the Internet will be linked to RSVP in some way.

In this article, we report on the feasibility of an alternative approach, called Application REQuested IP over ATM (Arequipa). Our purpose with Arequipa is to show that providing the quality of service of ATM to TCP/IP applications is straightforward with minimum changes to the TCP/IP implementation in hosts. For two end-systems to communicate using Arequipa, it is necessary that they are connected (1) to a common ATM network and (2) to the Internet or to the same Intranet. However, there is no cooperation required between the two types of networks. We say that our approach is based on a concept of *segregation*. Arequipa allows applications to establish direct point-to-point end-to-end ATM connections with a given QoS at the link level. These connections are used exclusively by the ap-

plications that requested them. After setup of the Arequipa connection (namely, the ATM connection that is used for Arequipa), the applications can use the standard TCP/IP service to exchange data.

We made the conscious choice to let the user, or the application, explicitly control the ATM connection. In our implementation, we support unspecified bit rate (UBR) and constant bit rate (CBR) connections. In the latter case, the user or application has to specify the requested peak cell rate. The additional traffic or QoS parameters required by the ATM signalling procedure are set transparently by our implementation. We believe that it is reasonable to limit the information requested from the user or application to just the choice mentioned above, namely: UBR with no rate information, or CBR with a specified peak cell rate. Our choice to make the traffic specification visible to the application is an essential part of Arequipa; we believe that it will become more common in the future for a large variety of applications.It is based on the concept that QoS comes with a price, and therefore we expect a dialogue between application and user to take place before a guaranteed QoS is requested. However, this does come with a drawback: existing application code has to be modified. We did the modification to a web client and a web browser, as reported in section B.5. Similar work has been done for video and audio conferencing applications [70]. An alternative to our approach is to let the operating system choose the traffic parameters in lieu of the application. We do not follow this approach with Arequipa because we explicitly want to make quality of service visible to the end-user.

Considerable work has been devoted in the Broadband ISDN context on defining an application level signaling framework that would enable applications to negotiate services and determine service access points, depending on application profiles, terminal capabilities and service requirements by end-users [71]. We claim that such efforts are to a large extent redundant with the existing base of Internet applications (such as the Web). With Arequipa, it is possible for applications to use the Internet for exchanging short messages for purposes of service negotiation, address mapping, authentication, and then set up ATM connections as needed [72, 73].

The chapter is structured as follows: section B.2 describes mechanisms for run-

ning IP over ATM which are either currently in use or which are being defined by the IETF or the ATM Forum. In sections B.3 and B.4, we explain the concept of Arequipa and how we implemented it in a UNIX-like operating system. Section B.5 introduces a way of using Arequipa with the World-Wide Web (WWW, [74]). In section B.6, we describe how we tested Arequipa to transfer video data across Europe.

Although of high importance, the related issue of pricing for ATM services is not considered here.

B.2 Transmitting IP Packets over ATM

The IETF and the ATM Forum have defined various mechanisms that can be used to send IP traffic over ATM, and they continue developing new mechanisms and refining the existing ones. For what could be called the second generation of such mechanisms, both groups have joined forces and are closely synchronizing their efforts. This section briefly describes the current state of affairs.

B.2.1 Classical IP over ATM

The first standard developed by the IETF for running IP over ATM is the so-called *classical IP over ATM*, defined mainly in RFC1577 [58], but see also RFC1483 [57], RFC1755 [59], and RFC1932 [75]. In that scheme, IP hosts are grouped in Logical IP Subnets (LIS) which are typically interconnected with IP routers. The ATM network is treated much like a LAN and hosts within a LIS can obtain each other's ATM addresses through an address resolution protocol that maps IP addresses to ATM addresses. After obtaining the address of a destination (within the LIS), an ATM connection is established to it. If a packet has to go to another host outside the LIS, it is sent to a router which forwards it.

The advantage of this solution is that it works in the same manner as existing IP networks, hence the name. The disadvantage is that packets may be sent through a set of routers and ATM connections even if a direct ATM connection between the communicating hosts would be possible, as illustrated in figure B.1. Also, all

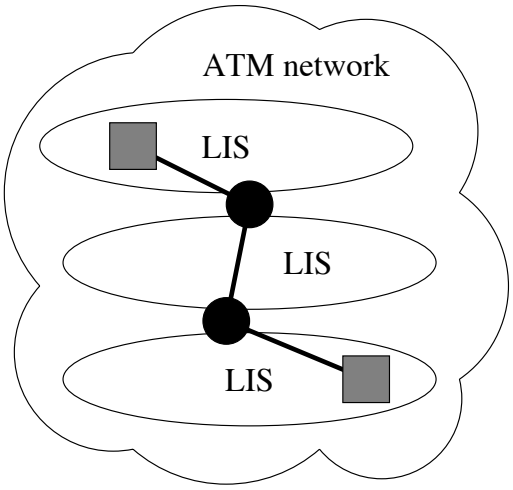

Figure B.1: Classical IP over ATM has to use routers even if a direct ATM connection could be established between communicating hosts

the data flowing between two machines typically uses the same ATM connection, making it impossible to request a QoS for one specific data stream.

B.2.2 LAN Emulation

LAN Emulation (LANE, [61]) is ATM Forum's equivalent to classical IP over ATM. Like the latter, it limits direct ATM connections to a comparably small cloud of systems, the so-called Emulated LAN (ELAN). The main differences to classical IP over ATM are that LANE uses IEEE 802 [76] MAC addresses instead of IP addresses, and that it also includes support for multicast and broadcast mechanisms.

LANE version 1 has no concept of honoring QoS requirements of upper layers. Support for this is planned for LANE version 2.

B.2.3 Next Hop Resolution Protocol

An improvement for classical IP over ATM is the Next Hop Resolution Protocol (NHRP, [77]). This protocol tries to resolve ATM addresses for hosts which are not in the same LIS. With the ATM address of a remote host, a direct ATM connection can be established, bypassing intermediate routers (see figure B.2). In cases were the

ATM address of a remote host can be resolved, NHRP can thus provide end-to-end ATM connections.

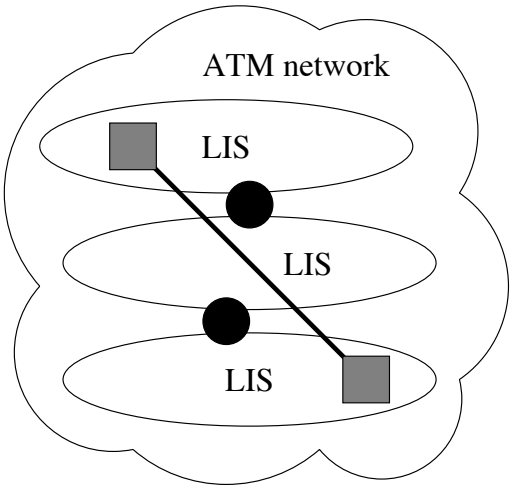

Figure B.2: NHRP can establish direct end-to-end ATM connections between hosts

However, NHRP has no mechanism to manage the QoS of such connections. Data from different applications may transit through the same end-to-end ATM connection and the QoS an application experiences depends on the traffic load generated by the others.

B.2.4 Multiprotocol over ATM

Multiprotocol over ATM (MPOA, [63]) merges protocols developed by the IETF and the ATM Forum, and extends them for using end-to-end ATM connections also with non-IP protocols, such as IPX. This includes mainly: NHRP and the multicast mechanism described in RFC2022 [78]. In addition, MPOA has mechanisms for flow classification, in order to decide automatically at layer 3 when an ATM shortcut should be established. It also supports the decoupling from the data and control paths in intermediate systems.

Like LANE, phase 1 of MPOA does not consider QoS, but phase 2 will.

B.2.5 RSVP

Current IP networks are designed to provide a best effort service. This explains why the aforementioned solutions for running IP over ATM do not pass the notion of QoS guarantees that ATM provides to the IP layer.

The standard approach for providing QoS guarantees in IP networks is the use of the Resource Reservation Protocol (RSVP). RSVP is part of the Integrated Services framework for the Internet. It typically hinges on mechanisms like packet schedulers, which make sure that data flows for which reservations have been made get their share of bandwidth on links. In this framework, RSVP is the signaling protocol, propagating information about available services and requests for reservation along the data path between sources and destinations.

B.2.6 Guaranteed Internet Bandwidth

A mechanism called "Guaranteed Internet Bandwidth" (GIB [79]) approaches the QoS issues by directing flows with QoS requirements over dedicated wide area network (WAN) connections (for example ISDN or ATM). End systems use a special signaling protocol to ask a GIB agent to change the routing tables of the gateway routers. Limiting flow selection to IP routes (namely, to the destination IP address) allows the use of standard routers, but makes the isolation of concurrent flows unreliable.

B.2.7 Discussion

The methods of running IP over ATM presented above (except for GIB) have one thing in common: They hide the fact that ATM is being used from the applications. Since ATM is used below the IP layer and IP has no notion of connections or QoS, the interoperation mechanisms hide those properties of ATM.

NHRP/MPOA and RSVP alleviate the problem by setting up end-to-end connections below the IP layer and by setting up reservations above it. This approach has the advantage of being very general but it adds some complexity and has the following restrictions.

- In order for NHRP to be effective, it must be deployed on all the nodes between communicating hosts. If this is not the case, NHRP is not able to create an end-to-end connection between the hosts and RSVP will not be able to guarantee reservations on the entire path. RSVP can operate even if some routers do not implement it, however, there is no reservation on such paths. On an ATM WAN, for example, end stations must rely on their service providers to deploy NHRP and RSVP over all parts of the WAN between them, in order to benefit directly from ATM guaranteed QoS.

  It is true that QoS guaranteeing services need only be deployed on the congested parts of the network. However, the congested parts are usually the backbones and long-distance links, which is where it is most complicated to install new services. Arequipa avoids this problem since it only needs to be installed on end systems.

- With public ATM connections being offered by telecom companies, it is now possible to have long distance end-to-end ATM connections, even across country borders. Thus two hosts in two distant ATM WANs may be able to open end-to-end ATM connections through their public network operators. However, the IP backbone on the long distance path between the WANs may well not be running on ATM. Thus NHRP may not be able to resolve the ATM addresses and to set up end-to-end connections.

In contrast to all methods presented above, Arequipa aims at providing the QoS of ATM directly to the application, and at letting the application control its use. It is not clear at this time which approach has superior benefits, but it should be emphasized that they pursue different objectives. Arequipa is based on network segregation, with integration in the hosts only; it is therefore less general but also considerably simpler.

B.3 Arequipa

Arequipa allows applications to establish end-to-end ATM connections under their own control, and to use these connections at the lower protocol layer to carry

the IP traffic of specific sockets.

Unlike the connections set up by classical IP over ATM or by LANE, Arequipa connections are used exclusively by the applications that requested them. The applications can therefore exactly determine what QoS will be available to them.

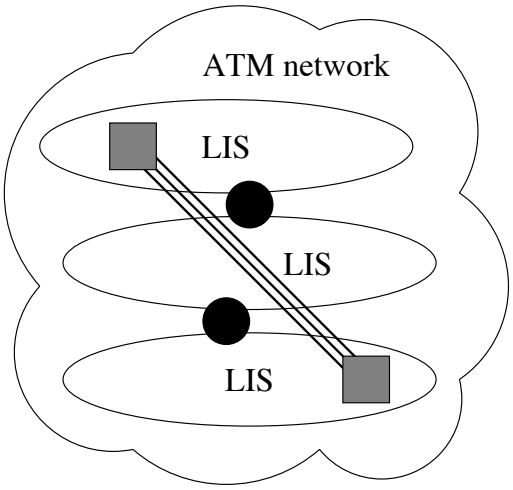

Figure B.3: End-to-end Arequipa connections for three applications with QoS requirements

Figure B.3 illustrates that Arequipa connections go end-to-end and that each flow has its own connection.

In its broadest sense, Arequipa offers a means to use properties of a network technology that is used to transport another network technology (e.g., IP on ATM) without requiring the explicit design and deployment of sophisticated interworking mechanisms and protocols.

Traditional protocol layering typically only allows access to functionality of lower layers if upper layers provide their own means to express that functionality. This approach can introduce significant complexity if the semantics of the respective mechanism are dissimilar. Also, if the upper layer fails to provide that interface, no direct access is possible and the lower layer functionality may be wasted or used in an inefficient way (e.g., if using heuristics to decide on the use of extra features). By allowing applications to control the lower layer, Arequipa enables them to exploit those properties.

Note that Arequipa coexists with "normal" use of the networking stacks, i.e., applications not requiring Arequipa do not need to be modified and they will continue to use whatever other mechanisms are provided.

B.3.1 Example

Figure B.4 illustrates the case of TCP/IP over ATM: TCP connections between applications are built by multiplexing their traffic over an upper layer (IP), which is in turn carried by a lower layer (e.g., Ethernet or ATM). Routers terminate lower layer segments in order to overcome scalability limitations of either layer or of the interface between the layers.

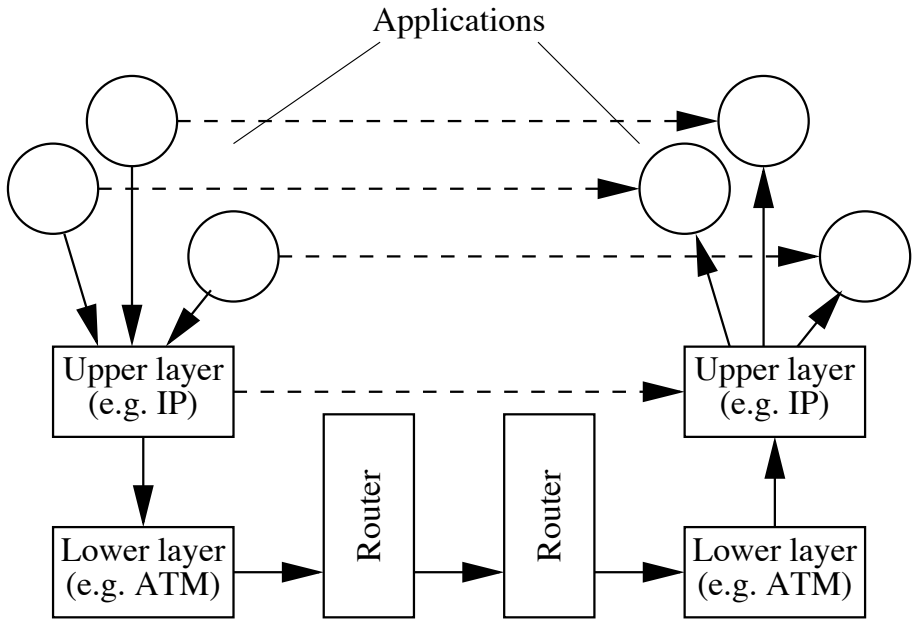

Figure B.4: Communication without Arequipa

Figure B.5 shows the same scenario, but this time using only Arequipa. The applications still have their TCP connections, but there is one dedicated end-to-end (Arequipa) connection at the lower layer for each of them.

Note that, although traffic between applications using Arequipa does not pass the normal routed IP path anymore, general IP connectivity may still be necessary, e.g. for ICMP messages or for traffic of other applications.

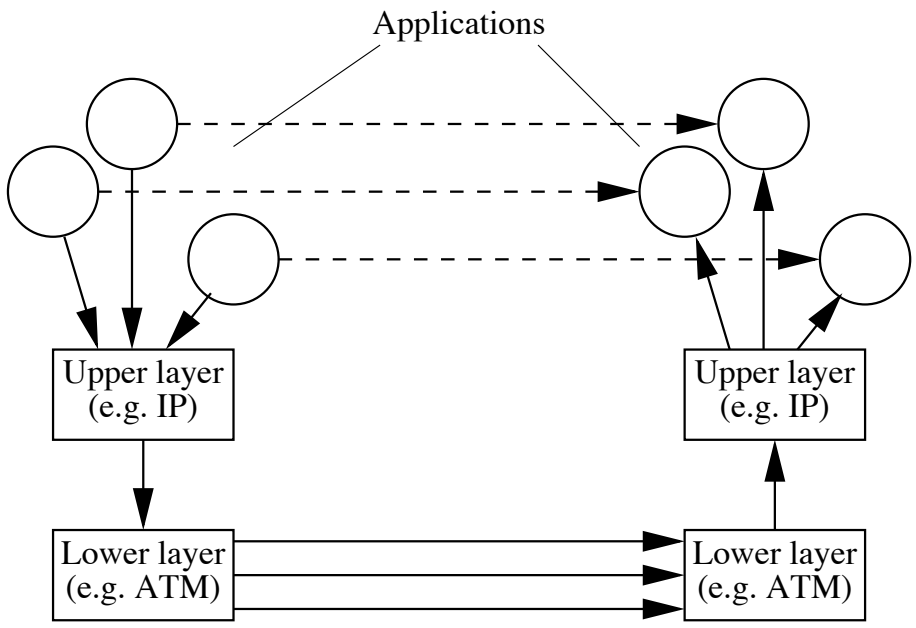

Figure B.5: Communication with Arequipa

B.3.2 Applicability

Arequipa is applicable if the following two conditions are met:

- applications can control "native" connections over the lower layer communication media
- the upper and the lower layer (e.g., IP and ATM) both allow communication between the same end-points (or they share at least a useful common subset of reachable end-points)

The next two conditions do not have to be met, but without them the use of Arequipa may be questionable:

- the upper layer is multiplexed over the lower layer (e.g., when using classical IP over ATM, all IP traffic between a pair of hosts typically shares the same ATM SVC)
- multiple lower layer connections are possible between a pair of end-points

In order to simplify interaction with the protocol stack, Arequipa assumes that data sent to destinations for which no Arequipa lower layer connection has been established will be delivered by some default mechanism.

Note that, despite its name (Application REQuested IP over ATM), Arequipa is not limited to IP and ATM only. The upper layer is typically IP or some similar protocol (e.g., IPX). The lower layer can be ATM, Frame Relay, N-ISDN, etc. Some of the advantages of using Arequipa in addition to the usual IP mechanisms are avoidance of routing overhead and the possibility of using dedicated connections with "hard" quality of service guarantees. This is of interest for flows with a lifetime which is long compared to the setup delay incurred by the lower layer.

B.3.3 API

The following primitives are available to applications using Arequipa:

```
int arequipa_preset(int sd,const struct sockaddr_atmsvc *addr,const
struct atm_qos *qos);
```

Presets the specified INET domain socket to use a direct ATM connection to `addr` with the QOS parameters specified in `qos`. If the socket is already connected, the ATM connection is set up immediately and data is redirected to flow over that connection.

```
int arequipa_expect(int sd,int on);
```

Enables (if `on` is non-zero) or disables (if `on` is zero) the use of Arequipa for return traffic on the specified INET domain socket. When enabling the use of Arequipa for return traffic, the Arequipa connection on which the next data packet or incoming connection for the socket is received is attached to that socket.

```
int arequipa_close(int sd);
```

Dissociates an Arequipa VC from the specified socket. After that, traffic uses normal IP routing. Note that the Arequipa connection is automatically closed when the INET socket is closed.

B.3.4 Use of ATM user-to-user signaling

ATM connections for Arequipa use are used almost exactly like connections for IP over ATM. However, in order to avoid conflicts with the IP over ATM entity, Arequipa connections are signaled in a slightly different way, so the rule is as follows:

An Arequipa connection is signaled by using the procedures and codings described in RFC1755 [59], with the addition that the Broadband High Layer Information (BHLI) information element be included in the SETUP message, with the coding shown in table B.1.

`bb_high_layer_information`		
`high_layer_information_type`		
	3	(vendor-specific application id.)
`high_layer_information`		
	00-60-D7	(EPFL OUI)
	01-00-00-01	(Arequipa)

Table B.1: Use of the Broadband High Layer Information information element with Arequipa

B.4 Implementing Arequipa in a Unix environment

This section describes general aspects of implementing Arequipa for IP over ATM in a socket-based operating system kernel. The organization of kernel internal data structures is assumed to be similar to the one found in the networking part of the Linux kernel [80].

B.4.1 Kernel data structures without Arequipa

Figure B.6 shows some of the kernel data structures that are typically associated with a TCP socket when not using Arequipa. Incoming data is demultiplexed by the protocol stack (in figure B.6, the circle with TCP/IP) and queued on the socket.

Outgoing data is multiplexed by the protocol stack and sent to the corresponding network interface.

Each data packet consists of a packet descriptor and the actual data. The packet descriptor contains information like the socket the packet belongs to, the interface on which it was received, etc.

Most modern TCP/IP implementations also cache routing information (including the network interface) for each socket, so that route lookups only need to be done when a new connection is established or if the routing table is modified.

B.4.2 Data structures for incoming Arequipa

When using Arequipa, incoming packets are handled as if they were using Classical IP over ATM: after little or no ATM-specific processing, they are passed to the protocol stack, which then performs the usual demultiplexing, etc. The only significant difference is that they are marked in order to identify them as originating from Arequipa (and from which VC) when they arrive at the socket.

Figure B.7 shows the data structures used when receiving from Arequipa. Note that all Arequipa VCs on a system can use the same Arequipa pseudo-interface.[1]

If the socket is not yet using Arequipa for sending (namely, if it has no associated Arequipa VC) and if it expects incoming Arequipa traffic (namely, if `arequipa_expect` has been invoked with the `on` argument set to a non-zero value), the Arequipa VC on which the packet has been received is attached to the socket, so that outbound traffic uses the VC.

Note that if a packet is received over an Arequipa VC, then it is possible that the VC no longer exists at the time the data is delivered to the socket. It is therefore necessary to verify the validity of the incoming Arequipa VC before attaching it to the upper layer socket (the normal closing procedures only ensure that both layers are synchronized *after* establishing the association).

This can be implemented as follows:

[1]The term "pseudo-interface" is used to make it clear that the Arequipa interface does not correspond to a physical network interface (namely, hardware) although the protocol stack interacts with it as if it did.

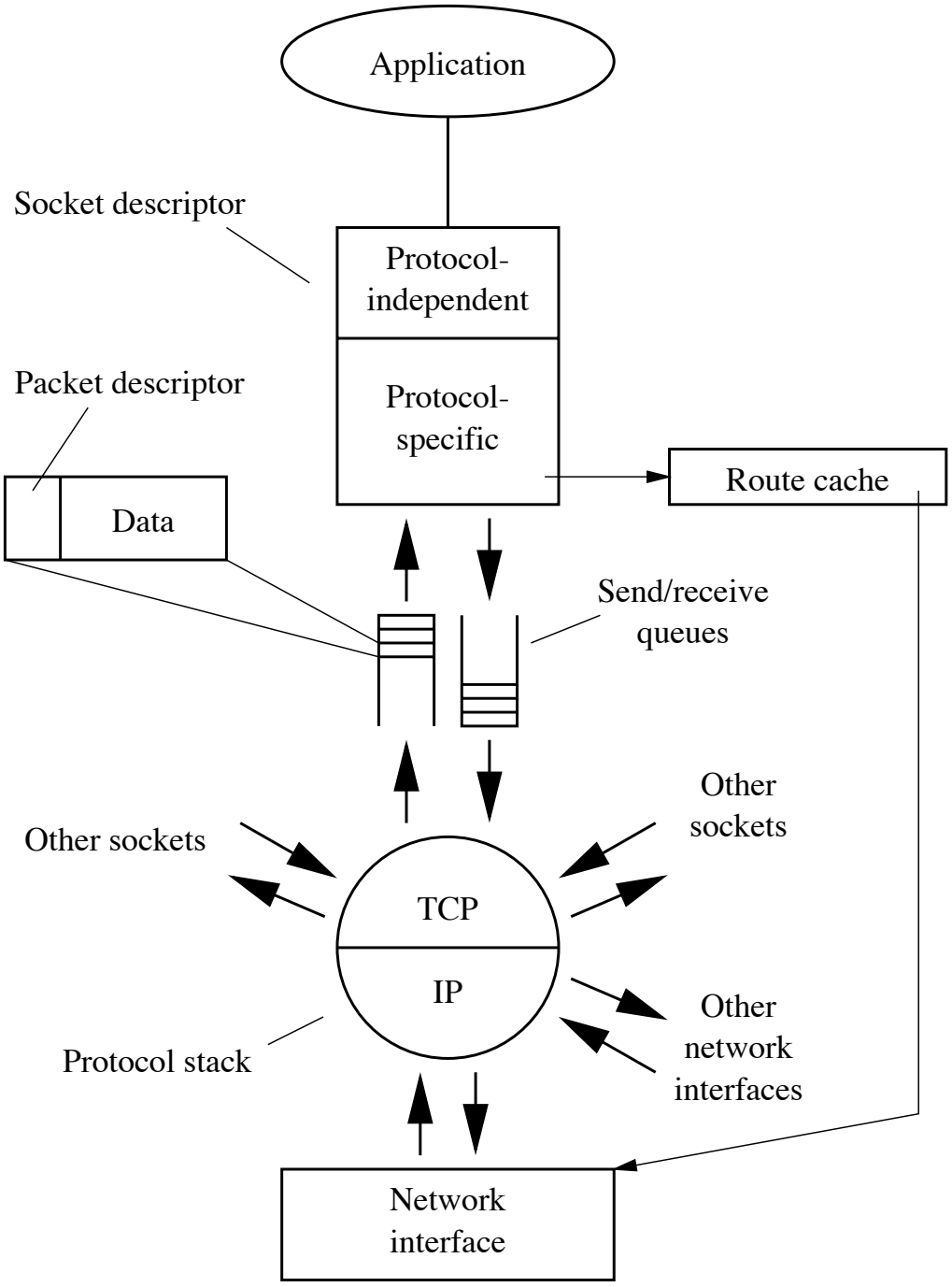

Figure B.6: Kernel data structures of a TCP socket (simplified)

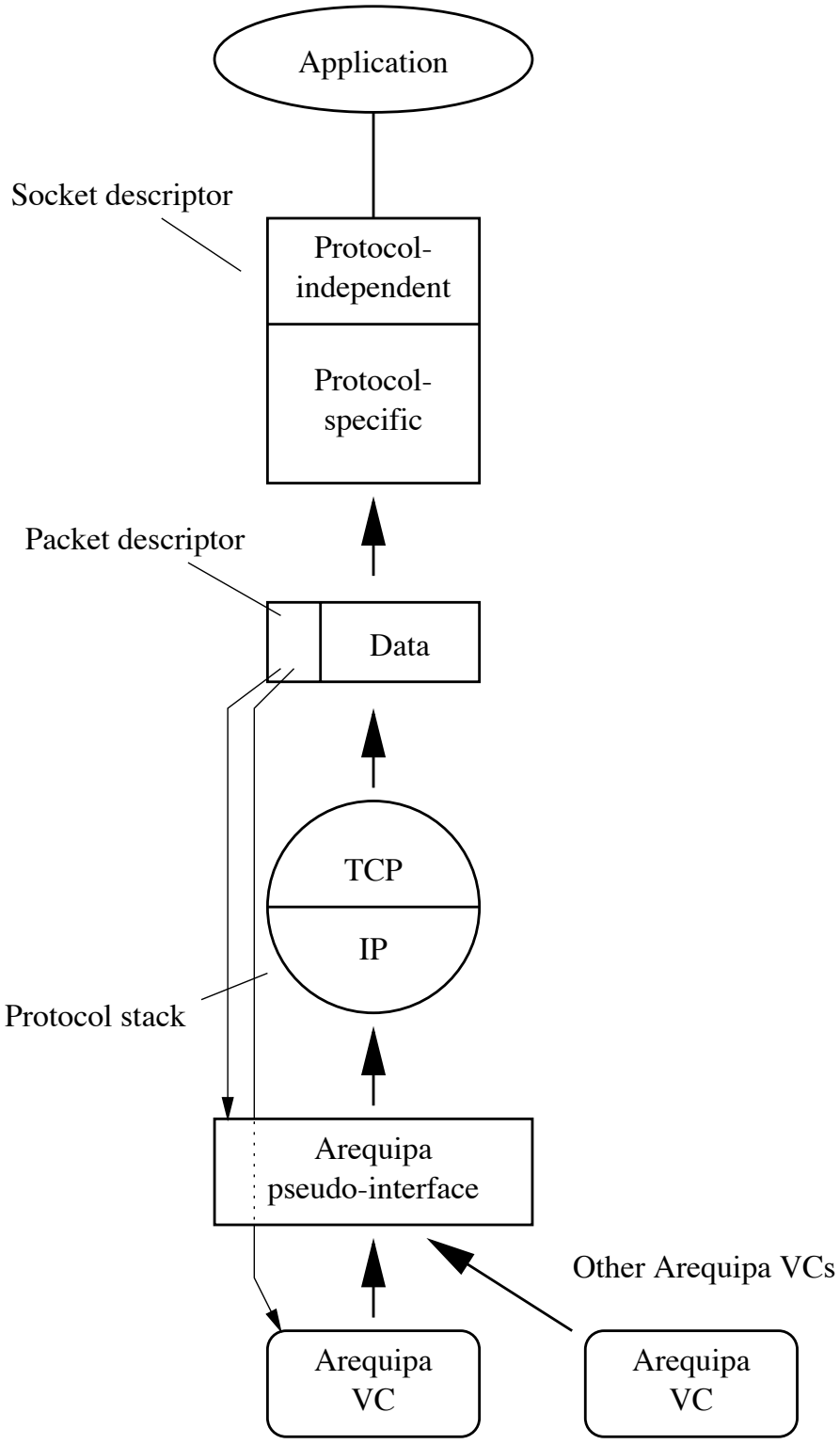

Figure B.7: Arequipa for incoming data

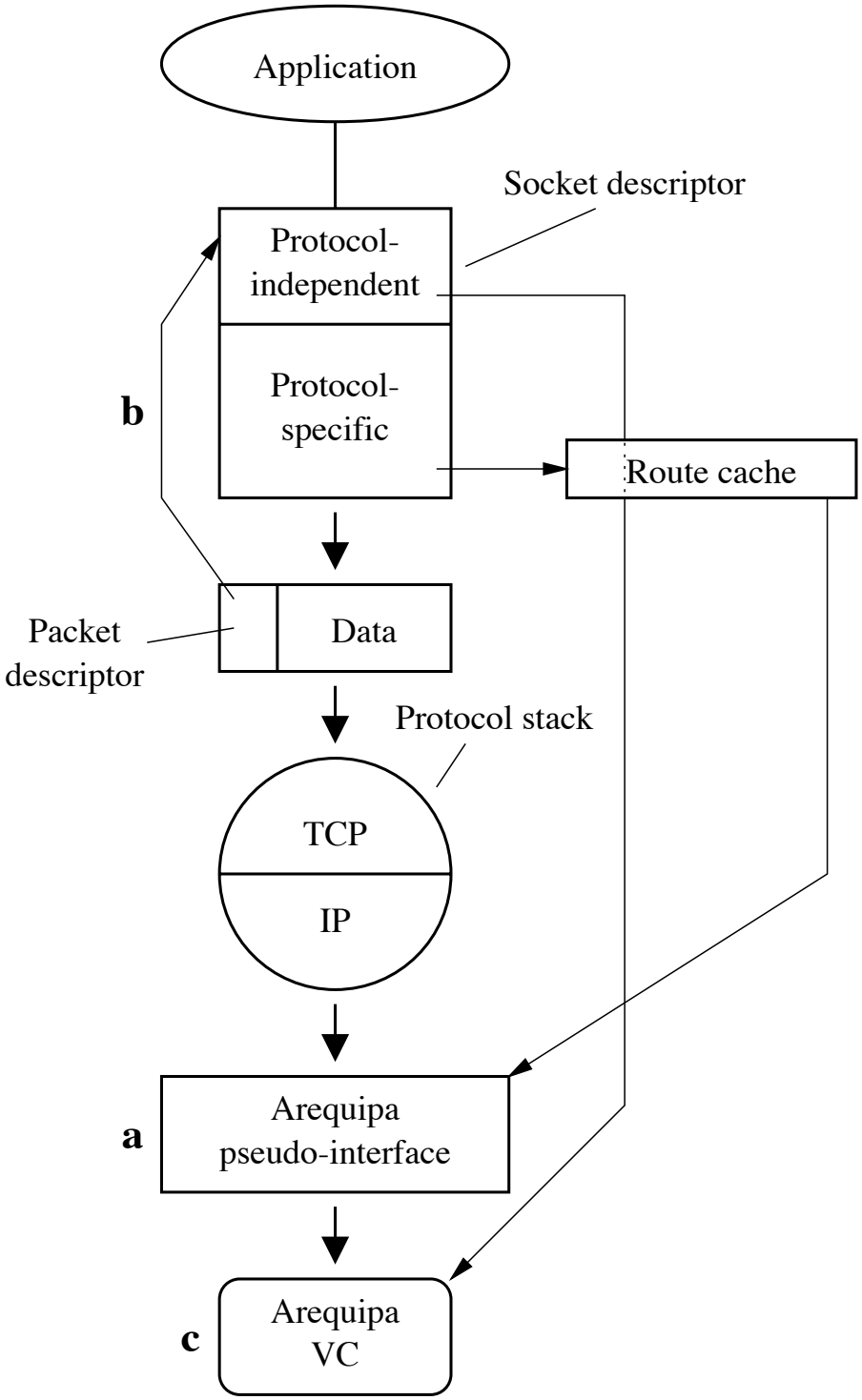

Figure B.8: Arequipa for outgoing data

- all incoming Arequipa VCs are registered in a list (while they are dangling because they are not attached)
- there is a global generation number, which is incremented whenever a new Arequipa VC is created. The generation number at the time of VC creation is stored in the VC descriptor.
- the generation number of the Arequipa VC is recorded in the descriptor of each data packet arriving on that VC

A VC is still valid when the packet is delivered to the socket only if a reference to that VC is on the list and if its generation number matches the one stored in the packet descriptor.

B.4.3 Data structures for outgoing Arequipa

When sending from an Arequipa socket, outbound packets must be associated with the corresponding Arequipa VC. As illustrated in figure B.8, this is done by sending them all through an Arequipa pseudo-interface (a) which then looks up a back pointer (b) to the originating socket in the packet descriptor. The originating socket contains a pointer to the descriptor (c) of the VC over which the data has to be sent.

Note that an Arequipa connection may be removed (e.g. because the remote party has closed it, because of a network failure, etc.) without notification at the socket. In this case, the Arequipa route is removed and all outbound traffic is sent with the "normal" IP mechanisms again.

B.4.4 Networking code changes

If using the approach outlined in the previous sections, the networking code has to be modified at least at the following places:

- when creating a socket, the Arequipa information (namely, if Arequipa is in use on that socket, if the socket expects incoming Arequipa traffic, etc.) needs to be initialized

- when connecting a UDP or TCP socket, a cached Arequipa route exists if `arequipa_preset` was invoked before the `connect` system call. This cached route must be preserved.

- when delivering data from Arequipa to a socket, the Arequipa VC is attached to the socket if

  - the socket expects incoming Arequipa traffic, and
  - the socket does not currently use Arequipa, and
  - the Arequipa VC is not already attached to a different socket

- if an incoming TCP connection is received on a listening socket which expects incoming Arequipa traffic, the new socket (the one returned by `accept`) is also set to expect incoming Arequipa traffic and, if the packet has arrived via Arequipa and if the constraints listed above are met, the Arequipa VC is attached to the new socket

- when an upper layer socket is closed, the underlying Arequipa connection has to be closed too

- when forwarding IP packets, packets received over an Arequipa connection must be discarded (see [38], section 6)

Additional modifications may be necessary depending on how per-socket route caches are invalidated. Also, socket destruction may be interrupt-driven and may therefore need special care.

B.4.5 TCP issues

The use of TCP over Arequipa raises two specific problems: (1) if the Arequipa connection is attached after establishing the TCP connection, the maximum segment size (MSS) of TCP may be very small, typically increasing processing overhead. (2) there are no generally useful semantics for listening on a socket for which an Arequipa connection has already been set up.

TCP implementations frequently limit the MSS to a value which is based on the MTU of the IP interface on which the connection is started. If connections are set up over a media with an MTU size that is small compared to the default IP over ATM MTU size [62], that MSS will have to be kept even if Arequipa is later used for that socket (see RFC1122 [81], section 4.2.2.6). It is therefore recommended to invoke `arequipa_preset` before `connect` and to invoke `arequipa_expect` before `listen`.

Note that this is the only way to ensure that the use of Arequipa is known at both ends when exchanging the initial SYN segments. Applications that require the TCP listener to set up the Arequipa connection are therefore not able to ensure the use of a larger MSS.

Although the API could allow associating an Arequipa connection with a socket that is used to listen for incoming connections, the usefulness of such an operation is questionable.

Therefore, attempts to execute `arequipa_preset` on a listening socket or to `listen` on a socket for which an Arequipa connection already exists yield an error.

Further implementation details, including a step-by-step description of the changes that were necessary when adding Arequipa support to Linux, can be found in [82].

B.5 Transmitting Web documents with guaranteed QoS

One of the most popular applications on IP networks is the World Wide Web (WWW, [74]). Its popularity stems from the fact that it allows to access many different types of multimedia documents with a single intuitive user interface.

The Web is also a good example for an application that can benefit from dependable QoS: if the network can guarantee the required bandwidth, data with real-time constraints (e.g., video clips) can be displayed during reception and does not have to be downloaded and replayed from a file, as it is currently done. Also, users frequently have loose time constraints (e.g., the time to download stock exchange

information). QoS guarantees ensure that users can obtain an adequate service and won't be subjected to the vagaries of best-effort.

B.5.1 Arequipa and the Web

In order to use Arequipa for Web applications, three types of information are needed:

- The side that establishes the Arequipa connection must know the ATM address of the opposite side.
- Likewise, the side that establishes the Arequipa connection must be able to specify the QoS information.
- Finally, the side that establishes the TCP/IP connection (normally using either TCP or UDP) also needs to know the destination port.

We have chosen to let the Web server open Arequipa connections to the client, thus the server needs to know the ATM address of the client. This information can be sent conveniently as a pragma in the client's request [83]. This pragma can serve another purpose, namely to indicate to the server that the client is capable of using Arequipa.

The server also establishes the TCP/IP connection on which the document is transferred, so the client needs to indicate the port on which it expects the data. For convenience, it also includes the protocol type along with the port number.

For each document, we want to be able to specify whether a connection with guaranteed QoS should be used. If yes, we also want to specify what kind of service and what QoS should be requested. To specify this information, we use the notion of meta-information for Web documents. Web servers are able to store meta-information for each document, either in the header of the document or in a separate file. We use this feature to store the ATM service and the QoS parameters to be requested.

Although not strictly required, the QoS information is also useful to the client, so it is included in the meta-information the server sends in response to a request. Figure B.9 illustrates the general information flow.

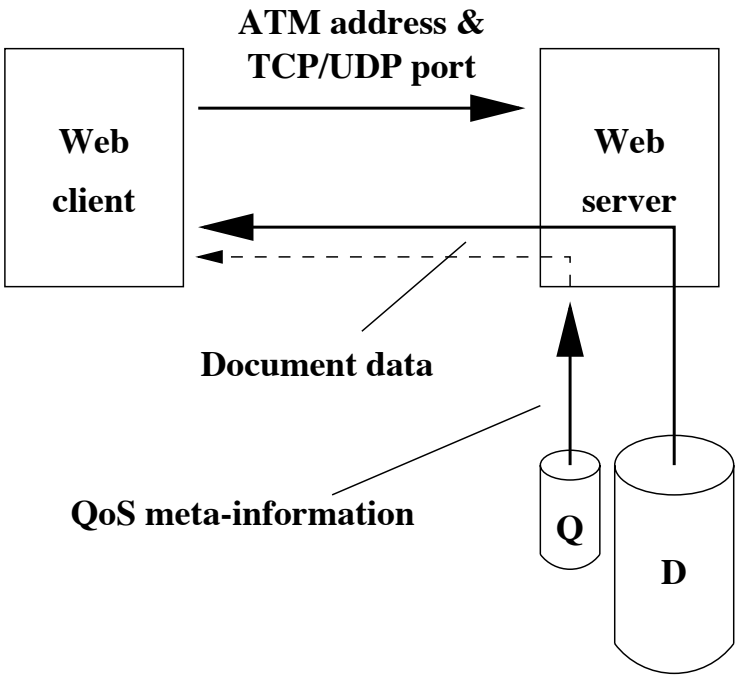

Figure B.9: General information flow when using Arequipa with the Web

Because a client may want to know the QoS attributes of a document before downloading it with Arequipa (e.g., because the client wants to ensure that sufficient local resources are available to handle the document, or because the user is charged for ATM connections and therefore only wants to use Arequipa for selected documents), we also need a mechanism to obtain only the headers, which include the QoS meta-information.

While a client could always send a `HEAD` request before issuing a `GET`, this would add one extra round-trip time for every request, whether or not the document in question is eligible for being transferred with Arequipa. This is clearly undesirable. We therefore extend the semantics of `GET` to only return the header of the document under the following conditions:

- the document has associated QoS information, and
- the client indicated that it supports Arequipa (by sending its ATM address), and

- the client did **not** include the destination port number.

If the client decides to retrieve the document using Arequipa, it issues a second request, this time with the destination port number. Note that a client can avoid the extra round-trip if it has a priori knowledge about the document (e.g., if the headers are cached) or if it wants to use Arequipa anyway, whatever the requested QoS is.

The extended behavior of the Web server is shown in the pseudo-code below:

```
if (request_has_ATM_address && document_has_QoS_metainfo)
    if (request_has_port_number)
        send_document_using_arequipa();
    else send_headers_only();
else /* non-QoS document or non-Arequipa client */
    send_document_standard_way();
```

Note that this extension is compatible to the standard HTTP protocol and that Arequipa capable servers and client will interact seamlessly with their standard counterparts.

B.5.2 HTTP extensions

When the client sends additional information required for Arequipa, it uses the following extra header fields:

`Pragma:` `ATM-address=`*pub_address*`.`*prv_address*

pub_address is the public E.164 address [84] of the client. If the client has no such address, that part of the field is empty. *prv_address* is the private ATM NSAP address of the client. If the client has no such address, that part of the field is left empty. Presence of the `ATM-address` pragma indicates that the client supports Arequipa and that it wishes to make this fact known to the server.

`Pragma:` `socket=`*protocol*`.`*port_number*

protocol is typically `TCP` or `UDP`. *port_number* is the corresponding port number or whatever information the protocol may use to identify end-points. Presence of the `socket` pragma indicates that the client wishes to retrieve the requested document over Arequipa, if the document is suitable for this, and if the server supports Arequipa.

QoS meta-information is sent by the server by adding the following new fields to the document header:

`ATM-Service:` *service*

service is either `UBR` or `CBR`.

`ATM-QoS-PCR:` *peak_cell_rate*

`peak_cell_rate` is the required peak cell rate in cells per second. This field can be omitted when using UBR.

B.5.3 Example

Figure B.10 shows a sample HTTP dialog when using Arequipa.

The client first sends its request without the port information, so that the server only returns the header. After asking the user for permission to request retrieval with Arequipa, the client sets up its socket and repeats the request, this time with the port information. The server can now establish the Arequipa connection and sends the document with the specified quality of service.

B.5.4 Arequipa with proxies

A proxy Web server (short "a proxy") is a Web server that requests documents from other Web server on behalf of clients. Typical uses for proxies include Web caches and application-level gateways through firewalls.

When used in conjunction with a proxy, Arequipa can even be useful to client that are not directly connected to ATM: If the network between the client and the proxy is dimensioned to offer enough bandwidth so that congestion is very unlikely (the typical situation in a LAN), it is sufficient if Arequipa is used only over the – possibly congested – WAN.

Figure B.11 illustrates the use of Arequipa with a proxy. The proxy uses Arequipa when transferring documents over the WAN from remote servers. The client uses the best-effort service of its LAN and doesn't even have to know about Arequipa.

The pricing question for cached documents is interesting, but, as mentioned earlier, is outside the scope of this work. Note that in our example, the cache is

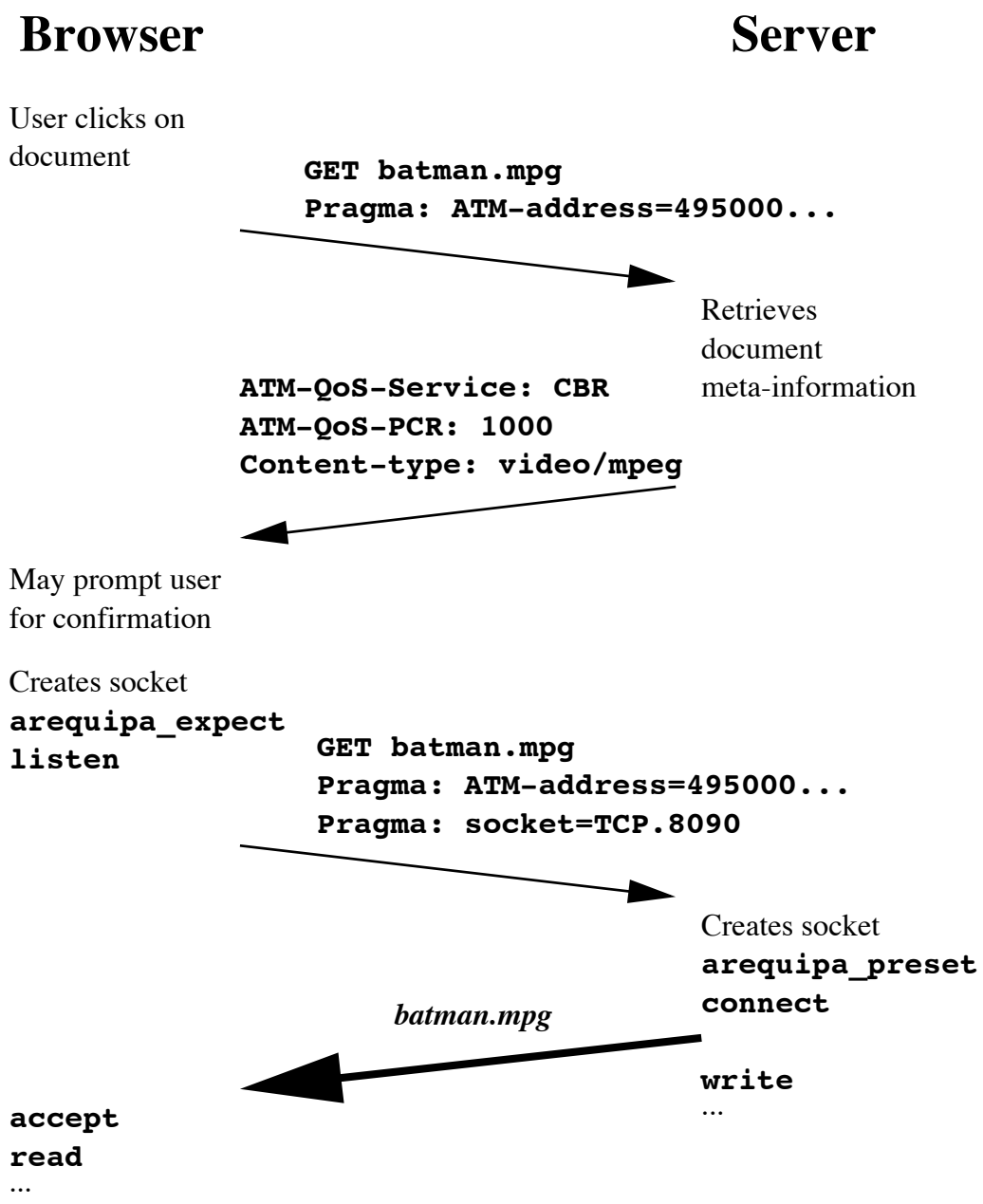

Figure B.10: Example request/response flow when using Arequipa on the Web

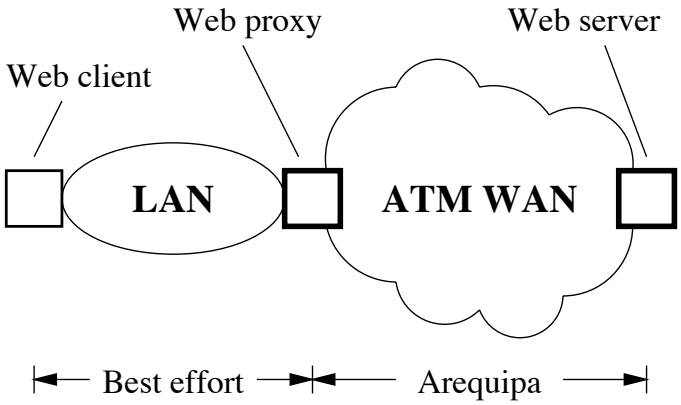

Figure B.11: Arequipa with a proxy Web server

located on the firewall or on the LAN of a company and all costs occurred by the users of the cache will be billed to the company.

B.6 An Arequipa test in the WAN

In October 1996, a demonstration of Arequipa was performed in an ATM WAN environment. This was done as part of an interim presentation of the "Web over ATM" project [73], which also comprises the work on Arequipa. A number of other tests and demonstrations of Arequipa were performed in 1997 as part of the European ACTS project.

The demonstration consisted of the transmission of raw uncompressed live video over TCP with Arequipa across Europe (see figure B.12). The purpose of this demonstration was to show how bandwidth-intensive applications can benefit from Arequipa.

Arequipa is part of the ATM-Linux distribution. The test reported here used computers with an Intel processor, a PCI bus and ATM adapters from two different vendors. The network used a number of different ATM switches from various vendors.

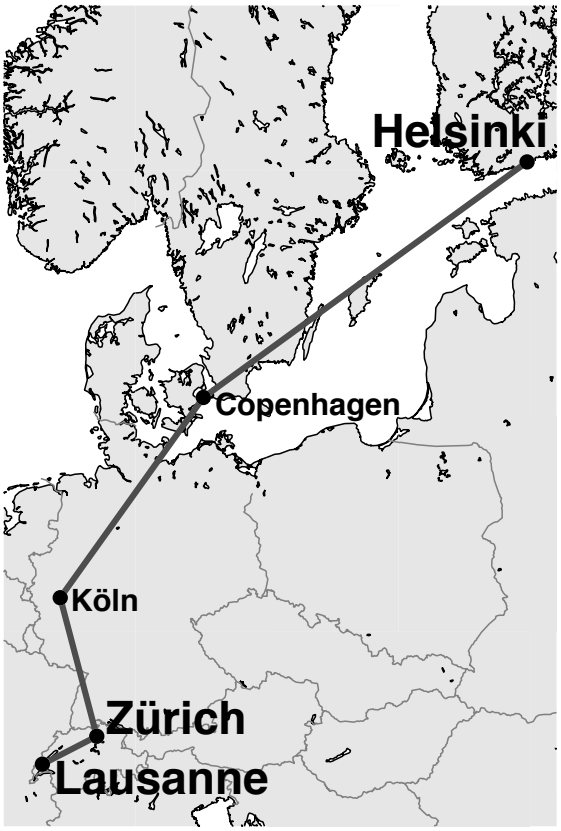

Figure B.12: Arequipa test in European WAN

B.6.1 The Network

The transmission was done from sites in Helsinki (Finland) to EPFL in Lausanne (Switzerland), using the JAMES (Joint ATM Experiment on European Services)[2] network. The partner sites in Finland were Nokia and Telecom Finland.

In order to experiment with the setup without wasting bandwidth in the international network, preliminary testing was done with ETH in Zürich (Switzerland). An overview of the sites involved is shown in figure B.13.

The WAN connections with Finland were virtual paths with a constant bit rate of 77'200 cells per second (corresponding to a user data rate of almost 30 Mb/s). The connection with ETH was a virtual path with a bandwidth of approximately

[2]See `http://btlabs1.labs.bt.com/profsoc/james/`

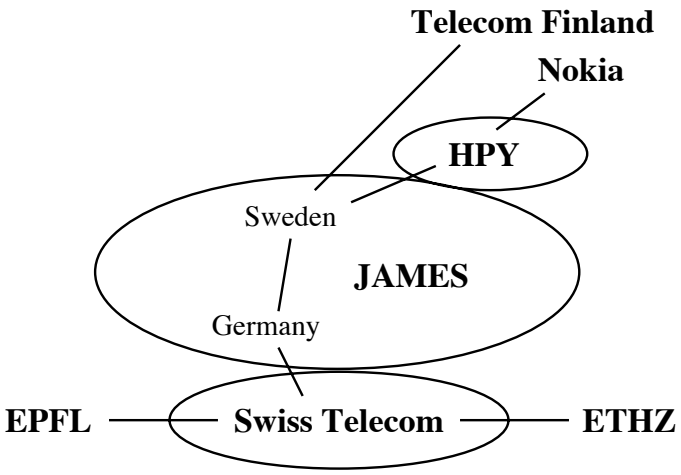

Figure B.13: Schematic overview of the network structure

34 Mb/s.

As described in section B.5, the Web was used to start and to control the video transmissions.

B.6.2 Results

The demonstration setup worked as expected and, using the video application with traffic shaping set to allow a user data rate of 27.3 Mb/s, a throughput of approximately 25 Mb/s was obtained for video traffic from Finland.

Also, the throughput for TCP over Arequipa without application overhead was tested on the 34 Mb/s virtual path with ETH. This benchmark was done with `ttcp`, a program that sends/receives to/from memory without doing any further data processing. With traffic shaping set to 33.3 Mb/s, we obtained a throughput of up to 33.0 Mb/s.

While those results tend to indicate that our implementation of Arequipa is able to sustain high rates, it should be noted however that the goal of Arequipa is *not* to obtain a high throughput. In contrast, Arequipa allows an application to receive a specified throughput, with hard guarantees, and in most cases for a price. This was illustrated in the demonstration using video images of a remote clock, with the display sent over a TCP connection using Arequipa. Depending on the selected peak

cell rate, the watch would either run too fast, too slow, or just at about the right speed, obviously something, you couldn't obtain with the normal Internet.

B.7 Conclusions and lessons learned

We have presented Arequipa, a method for providing the quality of service of end-to-end ATM connections to TCP/IP applications. It makes it possible to use ATM natively, in those cases where end-to-end ATM connectivity exists, while preserving the TCP/IP environment, and with only minimal changes to the application code. We have implemented Arequipa in Linux, tested it extensively, made it a part of the ATM-Linux distribution, and published an Internet RFC documenting it. We have described a way of using Arequipa with the Web without sacrificing compatibility with standard Web browsers and servers. We have indicated how Arequipa can be of use even for hosts not directly connected to ATM, using application level proxies. Finally, we have presented the results of a test of Arequipa in a European WAN setup.

B.8 Available software

An implementation of the Arequipa mechanisms is part of versions 0.13 to 0.32 of the ATM on Linux distribution. Support for Arequipa was discontinued with version 0.33, but the older distributions are still available from `http://icawww1.epfl.ch/linux-atm/`.

An application package for Arequipa is available on `http://icawww1.epfl.ch/arequipa/`. It includes the following components (with complete source code):

- Web server and proxy server: CERN `httpd` with Arequipa extensions
- Web browser: Arena with Arequipa extensions
- Video application: a modular video capture and playback package

Appendix C

Abbreviations

AAL ATM Adaption Layer

AF Assured Forwarding [41]

API Application Program Interface

ARP Address Resolution Protocol

ATM Asynchronous Transfer Mode, page 116

Arequipa Application REQested IP over Atm, page 133

B-ISDN Broadband ISDN (ATM)

BE Best-Effort

CAR Committed Access Rate [8]

CBR Constant Bit Rate

DSCP Differentiated Services Code Point

EF Expedited Forwarding [40]

IETF Internet Engineering Task Force, `http://www.ietf.org/`

ILMI Interim Local Management Interface, page 118

INTSERV IETF Integrated Services Working Group [36]

IP Internet Protocol [4]

ISDN Integrates Services Data Network

ISP Internet Service Provider

LANE LAN Emulation [61]

MPOA Multi-Protocol over ATM [63]

N-ISDN Narrow-band ISDN

PCP Phantom Circuit Protocol [24], page 9

PHB Per-Hop Behaviour [19]

PVC Permanent Virtual Channel

QOS Quality of Service

RFC Request for Comments

SRP Scalable resource Reservation Protocol

SVC Switched Virtual Channel

TCP Transmission Control Protocol [85]

TOS Type of Service

UBR Unspecified Bit Rate

UNI User-Network Interface

Bibliography

[1] Zhang, Lixia; Shenker, Scott; Clark, Dave; Huitema, Christian; Deering, Steve; Ferrari, Domenico. *Reservations or No Reservations*, `ftp://ftp.parc.xerox.com/pub/net-research/infocom95.html`, Proceedings of the Conference on Computer Communications (IEEE Infocom), April 1995, panel-discussion slides.

[2] Cisco. *Advanced QoS Services for the Intelligent Internet*, `http://www.cisco.com/warp/public/732/net_enabled/qos_wp.htm`, May 1997.

[3] IETF, Differentiated Services (diffserv) working group. `http://www.ietf.org/html.charters/diffserv-charter.html`

[4] RFC791; Postel, Jon. *Internet Protocol*, IETF, September 1981.

[5] RFC1349; Almquist, Philip. *Type of Service in the Internet Protocol Suite*, IETF, July 1992.

[6] RFC1455; Eastlake III, Donald. *Physical Link Security Type of Service*, IETF, May 1993.

[7] RFC1700; Reynolds, Joyce; Postel, Jon. *Assigned Numbers*, IETF, October 1994.

[8] Cisco Systems. *Committed Access Rate*, `http://www.cisco.com/warp/public/cc/cisco/mkt/ios/tech1/carat_wp.htm`, June 1999.

[9] Le Boudec, Jean-Yves. *The Asynchronous Transfer Mode: a tutorial*, Computer Networks and ISDN Systems, Volume 24, Number 4, 1992.

[10] The ATM Forum, Technical Committee. *ATM User-Network Interface (UNI) Signalling Specification, Version 4.0*, `ftp://ftp.atmforum.com/pub/approved-specs/af-sig-0061.000.ps`, The ATM Forum, July 1996.

[11] Clark, D.; Shenker, S.; Zhang, L. *Supporting Real-Time Applications in an Integrated Services Packet Network: Architecture and Mechanisms*, Proceedings of ACM SIGCOMM '92, pp 14-26, August 1992.

[12] RFC1633; Braden, Bob; Clark, David; Shenker, Scott. *Integrated Services in the Internet Architecture: an Overview.*, IETF, June 1994.

[13] RFC2205; Braden, Bob (Ed.); Zhang, Lixia; Berson, Steve; Herzog, Shai; Jamin, Sugih. *Resource ReSerVation Protocol (RSVP) – Version 1 Functional Specification*, IETF, September 1997.

[14] RFC2212; Shenker, Scott; Partridge, Craig; Guerin, Roch. *Specification of Guaranteed Quality of Service*, IETF, September 1997.

[15] RFC2211; Wroclawski, John. *Specification of the Controlled-Load Network Element Service*, IETF, September 1997.

[16] RFC1819; Delgrossi, Luca; Berger, Louis. *ST2+ Protocol Specification*, IETF, August 1995.

[17] Ferrari, Domenico; Banerjea, Anindo; Zhang, Hui. *Network Support for Multimedia - A Discussion of the Tenet Approach*, Computer Networks and ISDN Systems, vol. 26, pp. 1267-1280, 1994.

[18] The ATM Forum, Technical Committee. *ATM Forum Traffic Management Specification, Version 4.0*, `ftp://ftp.atmforum.com/pub/approved-specs/af-tm-0056.000.ps`, April 1996.

[19] RFC2475; Blake, Steven; Black, David; Carlson, Mark; Davies, Elwyn; Wang, Zheng; Weiss, Walter. *An Architecture for Differentiated Services*, IETF, December 1998.

[20] Bernet, Yoram; Yavatkar, Raj; Ford, Peter; Baker, Fred; Zhang, Lixia; Nichols, Kathleen; Speer, Michael; Braden, Bob. *Interoperation of RSVP/Int-Serv and Diff-Serv Networks* (work in progress), Internet Draft `draft-ietf-diffserv-rsvp-02.txt`, February 1999.

[21] Bernet, Yoram; Binder, James; Blake, Steven; Carlson, Mark; Carpenter, Brian; Keshav, Srinivasan; Davies, Elwyn; Ohlman, Borje; Verma, Dinesh; Wang, Zheng; Weiss, Walter. *A Framework for Differentiated Services* (work in progress), Internet Draft `draft-ietf-diffserv-framework-02.txt`, February 1999.

[22] Bernet, Yoram; Yavatkar, Raj; Ford, Peter; Baker, Fred; Zhang, Lixia; Speer, Michael; Braden, Bob; Davie, Bruce. *Integrated Services Operation Over Diffserv Networks* (work in progress), Internet Draft `draft-ietf-issll-diffserv-rsvp-02.txt`, June, 1999.

[23] Eriksson, Anders; Gehrmann, Christian. *Robust and Secure Light-weight Resource Reservation for Unicast IP Traffic*, Proceedings of IWQoS'98, pp. 168-170, IEEE, May 1998.

[24] Borgonovo, Flaminio; Capone, Antonio; Fratta, Luigi; Marchese, Mario; Petrioli, Chiara. *PCP: A Bandwidth and Delay Guaranteed Transport Service for IP networks*, IEEE ICC '99, Vancouver, Canada, June 1999.

[25] Borgonovo, Flaminio; Capone, Antonio; Fratta, Luigi; Petrioli, Chiara. *A PHB Description for PCP protocol*, `http://cerbero.elet.polimi.it/ntw/ip/phb.ps.gz`, Internal Report, May 1999.

[26] RFC2474; Nichols, Kathleen; Blake, Steven; Baker, Fred; Black, David. *Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers*, IETF, December 1998.

[27] RFC768; Postel, Jon. *User Datagram Protocol*, IETF, August 1980.

[28] Floyd, Sally; Mahdavi, Jamshid. *TCP-Friendly Unicast Rate-Based Flow Control*, http://www.psc.edu/networking/papers/tcp_friendly.html, Technical note, January 1997.

[29] Ferrari, Tiziana. *QoS Support for Integrated Networks*, http://www.cnaf.infn.it/~ferrari/tesidot.html, Ph.D. thesis, November 1998.

[30] McCanne, Steven; Jacobson, Van; Vetterli, Martin. *Receiver-driven Layered Multicast*, ACM SIGCOMM '96, August 1996.

[31] Almesberger, Werner; Ferrari, Tiziana; Le Boudec, Jean-Yves. *SRP: a Scalable Resource Reservation Protocol for the Internet*, Proceedings of IWQoS'98, pp. 107-116, IEEE, May 1998.

[32] Le Boudec, Jean-Yves. *Network calculus made easy*, http://icawww1.epfl.ch/PS_files/d4paper.ps, Technical Report 96/218, EPFL-DI, submitted to IEEE TIT, December 1996.

[33] UCB/LBNL/VINT. *UCB/LBNL/VINT Network Simulator – ns (version 2)*, http://www-mash.cs.berkeley.edu/ns/

[34] Arrate Muñoz. *Performance Evaluation of SRP*, SSC Doctoral School – Project Report, July 1998.

[35] Clark, David D.; Shenker, Scott; Zhang, Lixia. *Supporting Real-Time Applications in an Integrated Services Packet Network: Architecture and Mechanism*, Proceedings of SigComm'92, Baltimore, MD, August 1992. http://ana-www.lcs.mit.edu/anaweb/ps-papers/csz.ps

[36] IETF, Integrated Services (intserv) working group. http://www.ietf.org/html.charters/intserv-charter.html

[37] Floyd, Sally; Jacobson, Van. *Link-sharing and Resource Management Models for Packet Networks*, IEEE/ACM Transactions on Networking, Vol. 3 No. 4, pp. 365-386, August 1995.

[38] RFC2170; Almesberger, Werner; Le Boudec, Jean-Yves; Oechslin, Philippe. *Application REQuested IP over ATM (AREQUIPA)*, IETF, July 1997.

[39] Almesberger, Werner; Hadi Salim, Jamal; Kuznetsov, Alexey. *Differentiated Services on Linux* (work in progress), Internet Draft `draft-almesberger-wajhak-diffserv-linux-01.txt`, June 1999.

[40] RFC2598; Jacobson, Van; Nichols, Kathleen; Poduri, Kedarnath. *An Expedited Forwarding PHB*, IETF, June 1999.

[41] RFC2597; Heinanen, Juha; Baker, Fred; Weiss, Walter; Wroclawski, John. *Assured Forwarding PHB Group*, IETF, June 1999.

[42] Almesberger, Werner. *Linux Traffic Control — Implementation Overview*, `ftp://lrcftp.epfl.ch/pub/people/almesber/pub/tcio-current.ps.gz`, Technical Report SSC/1998/037, EPFL, November 1998.

[43] CISCO DWRED. *Distributed Weighted Random Early Detection*, `http://www.cisco.com/univercd/cc/td/doc/product/software/ios111/cc111/wred.htm`

[44] Clark, David; Wroclawski, John. *An Approach to Service Allocation in the Internet* (work in progress), Internet Draft `draft-clark-diff-svc-alloc-00.txt`, July 1997.

[45] Floyd, Sally; Jacobson, Van. *Random Early Detection Gateways for Congestion Avoidance*, IEEE/ACM Transactions on Networking, August 1993.

[46] DANTE. *TF-TANT: A joint activity between TERENA and DANTE*, `http://www.dante.net/tf-tant/`

[47] Alles, Anthony. *Internetworking with ATM*, `http://cell-relay.indiana.edu/cell-relay/docs/cisco.html`, Cisco Systems, May 1995.

[48] ITU-T Recommendation I.363. *B-ISDN ATM adaptation layer (AAL) specification*, ITU, March 1993.

[49] The ATM Forum. *ATM User-Network Interface (UNI) Specification, Version 3.1*, `ftp://ftp.atmforum.com/pub/UNI/ver3.1`, Prentice Hall, 1994.

[50] ITU-T Recommendation Q.2931. *Broadband Integrated Services Digital Network (B-ISDN) – Digital subscriber signalling system no. 2 (DSS 2) – User-network interface (UNI) – Layer 3 specification for basic call/connection control*, ITU, February 1995.

[51] ITU-T Recommendation Q.2971. *B-ISDN – DSS 2 – User-network interface layer 3 specification for point-to-multipoint call/connection control*, ITU, October 1995.

[52] ITU-T Recommendation Q.2100. *B-ISDN signalling ATM adaptation layer (SAAL) overview description*, ITU, July 1994.

[53] ITU-T Recommendation Q.2110. *B-ISDN ATM adaptation layer – service specific connection oriented protocol (SSCOP)*, ITU, July 1994.

[54] ITU-T Recommendation Q.2130. *B-ISDN signalling ATM adaptation layer – service specific coordination function for support of signalling at the user network interface (SSFC at UNI)*, ITU, July 1994.

[55] RFC1157: Schoffstall, Martin Lee; Fedor, Mark; Davin, James R.; Case, Jeffrey D. *A Simple Network Management Protocol (SNMP)*, IETF, May 1990.

[56] The ATM Forum, SAA/Directory Work group. *ATM Name Service (ANS) Specification Version 1.0*, `ftp://ftp.atmforum.com/pub/approved-specs/af-saa-0069.000.ps`, The ATM Forum, November 1996.

[57] RFC1483; Heinanen, Juha. *Multiprotocol Encapsulation over ATM Adaptation Layer 5*, IETF, 1993.

[58] RFC1577; Laubach, Mark. *Classical IP and ARP over ATM*, IETF, 1994.

[59] RFC1755; Perez, Maryann; Liaw, Fong-Ching; Mankin, Allison; Hoffman, Eric; Grossman, Dan; Malis, Andrew. *ATM Signaling Support for IP over ATM*, IETF, 1995.

[60] Truong, H. L.; Ellington, W. W. Jr.; Le Boudec, J.-Y.; Meier, A. X.; Pace, J. W. *LAN Emulation on an ATM Network*, IEEE Communications Magazine, May 1995, pp. 70–85.

[61] The ATM Forum, Technical Committee. *LAN Emulation Over ATM, Version 1.0*, `ftp://ftp.atmforum.com/pub/approved-specs/af-lane-0021.000.ps`, The ATM Forum, January 1995.

[62] RFC1626; Atkinson, Randall J. *Default IP MTU for use over ATM AAL5*, IETF, 1994.

[63] The ATM Forum, Technical Committee. *Multi-Protocol Over ATM, Version 1.0*, `ftp://ftp.atmforum.com/pub/approved-specs/af-mpoa-0087.000.ps`, July 1997.

[64] RFC1821; Borden, Marty; Crawley, Eric S.; Davie, Bruce S.; Batsell, Stephen G.. *Integration of Real-time Services in an IP-ATM Network Architecture*, IETF, August 1995.

[65] Kiiskilä, Marko. *Implementation of LAN Emulation Over ATM in Linux*, `ftp://viulu.atm.tut.fi/pub/misc/linux-lane.ps.gz`, Tampere University of Technology, October 1996.

[66] The ATM Forum, Technical Committee. *Native ATM Services: Semantic Description, Version 1.0*, `ftp://ftp.atmforum.com/pub/approved-specs/af-saa-0048.000.ps`, The ATM Forum, February 1996.

[67] Almesberger, Werner. *Linux ATM API*, `ftp://lrcftp.epfl.ch/pub/linux/atm/api/`, EPFL, July 1996.

[68] Almesberger, Werner. *Linux ATM device driver interface*, `ftp://lrcftp.epfl.ch/pub/linux/atm/docs/`, January 1996.

[69] Almesberger, Werner. *High-speed ATM Networking on Low-end Computer Systems*, Proceedings of IEEE IPCCC '96, pp. 336–343, March 1996.

[70] Almesberger, Werner; Chandran, Leena; Giordano, Silvia; Le Boudec, Jean-Yves; Schmid, Rolf. *Using quality of service can be simple: Arequipa with renegotiable ATM connections*, Computer Networks, Vol. 30, Issue 24, pp. 2327-2336, Elsevier, December 1998.

[71] RACE Project MAGIC. *Commission of the European Communities*, Final report, 1992.

[72] Le Boudec, Jean-Yves. *WWW over ATM: Issues and Prospects*, Proceedings of Interop 95, Paris, France, January 1995.

[73] Oechslin, Philippe. *Web over ATM – Intermediate Report*, `http://icawww1/WebOverATM/rapport/rapport.html`, Technical Report 96/209, EPFL, October 1996.

[74] Berners-Lee, Tim. *World-Wide Web - Summary*, `http://www.w3.org/pub/WWW/Summary.html`, 1991.

[75] RFC1932; Cole, Robert G.; Shur, David H.; Villamizar, Curtis. *IP over ATM: A Framework Document*, IETF, April 1996.

[76] IEEE Std. 802. *IEEE Standards for Local and Metropolitan Area Networks: Overview and Architecture.*

[77] RFC2332; Luciani, James V.; Katz, Dave; Piscitello, David; Cole, Bruce; Doraswamy, Naganand. *NBMA Next Hop Resolution Protocol (NHRP)*, IETF, April 1998.

[78] RFC2022; Armitage, Grenville. *Support for Multicast over UNI 3.0/3.1 based ATM Networks*, IETF, November 1996.

[79] Arango, Mauricio; Cortés, Mauricio. *Guaranteed Internet Bandwidth*, Proceedings of Globecom '96, vol. 2, pp. 862–866, November 1996.

[80] Cox, Alan. *Network Buffers and Memory Management*, Linux Journal, issue 30, October 1996.

[81] RFC1122; Braden, R. *Requirements for Internet Hosts – Communication Layers*, IETF, October 1989.

[82] Almesberger, Werner. *Arequipa: Design and Implementation*, `ftp://icawww1.epfl.ch/pub/arequipa/aq_di-1.tar.gz`, Technical Report 96/213, DI-EPFL, November 1996.

[83] RFC1945; Berners-Lee, Tim; Fielding, Roy T.; Frystyk Nielsen, Henrik. *Hypertext Transfer Protocol – HTTP/1.0*, IETF, May 1996.

[84] ITU-T Recommendation E.164/I.331. *Numbering plan for the ISDN era*, ITU, August 1991.

[85] RFC793; Postel, Jon. *Transmission Control Protocol*, IETF, September 1981.

Appendix D

Curriculum Vitae

Werner Almesberger was born in Zürich, Switzerland, in 1967. He obtained his diploma in Computer Science from ETH Zürich in 1992. While at ETH, he participated in design and realization of the site information system called "ezInfo". From 1993 to 1994, he developed ATM switch control software at the IBM Zurich Research Lab.

In 1994, he moved to the Laboratoire de Réseaux de Communication (LRC), which later became part of the Institute for computer Communication and Applications (ICA), of the Swiss Federal Institute of Technology in Lausanne (EPFL), where he worked on implementing ATM support on Linux, and the design and implementation of Arequipa. Later, he changed his focus to pure Internet technologies, and designed SRP. In the process of implementing SRP, he also participated in the design and implementation of support for Differentiated Services on Linux.

In much of his spare time, he enjoys improving the Linux system. The LILO boot loader and participation in the Linux-7k port are among his hobby projects.

Part of the work on Arequipa was done in cooperation with the European ACTS project EXPERT. Part of the work on SRP, and also participation in an implementation of RSVP over ATM (on Linux) were done in the ACTS project DIANA.

Appendix E

Publications

E.1 Journal papers

Almesberger, Werner; Le Boudec, Jean-Yves; Oechslin, Philippe. *Arequipa: Using TCP/IP over ATM with quality of service is simple... if you have ATM end-to-end*, Networking and Information Systems Journal, Volume 1, Number 4–5/1998, pp. 469–493. HERMES Science Publications, Paris, 1998.

Almesberger, Werner; Ferrari, Tiziana; Le Boudec, Jean-Yves. *SRP: a Scalable Resource Reservation Protocol for the Internet*, Computer Communications, Vol 21, Number 14, Special issue on "Multimedia networking", pp. 1200-1211, September 1998.

Almesberger, Werner; Chandran, Leena; Giordano, Silvia; Le Boudec, Jean-Yves; Schmid, Rolf. *Using quality of service can be simple: Arequipa with renegotiable ATM connections*, Computer Networks, Vol. 30, Issue 24, pp. 2327-2336, Elsevier, December 1998.

Gbaguidi, Constant; Einsiedler, Hans; Hurley, Paul; Almesberger, Werner; Hubaux, Jean-Pierre. *A Survey of Differentiated Services Proposals for the Internet*, Invited paper for IEEE Communication Surveys.

E.2 Book chapter

Almesberger, Werner; Ferrari, Tiziana; Le Boudec, Jean-Yves. *SRP: a Scalable Resource Reservation Protocol for the Internet*, to appear in *Workshop on Wide Area Networks and High Performance Computing*, Editors: Cooperman, G.; Jessen, E.; Michler, G. Lecture Notes in Control and Information Sciences (LNCIS), Vol. 249, pp. 21-36, ISBN 1 85233 642 0, Springer, 1999.

E.3 RFC

RFC2170; Almesberger, Werner; Le Boudec, Jean-Yves; Oechslin, Philippe. *Application REQuested IP over ATM (AREQUIPA)*, IETF, July 1997.

E.4 Conference papers

Almesberger, Werner. *High-speed ATM Networking on Low-end Computer Systems*, Proceedings of IEEE IPCCC '96, pp. 336–343, March 1996.

Almesberger, Werner. *ATM on Linux*, Proceedings of 3rd International Linux Kongress 1996, March 1996.

Almesberger, Werner; Le Boudec, Jean-Yves; Oechslin, Philippe. *Application Requested IP over ATM (AREQUIPA) and its Use in the Web*, Global Information Infrastructure (GII) Evolution, pp. 252–260, IOS Press, 1996.

Almesberger, Werner. *Cryptographic Authentication of Responses in ATM Traffic Control Protocols*, Proceedings of Globecom '96, vol. 3, pp. 2128–2132, November 1996.

Almesberger, Werner; Sasyan, Serge; Wright, Steven. *Quality of Service (QoS) in Communication APIs*, Proceedings of IWQoS'97, March 1997.

Almesberger, Werner. *ATM on Linux – The 3rd year*, Proceedings of 4th International Linux Kongress 1997, March 1997.

Almesberger, Werner; Ferrari, Tiziana; Le Boudec, Jean-Yves. *Scalable Resource Reservation for the Internet*, Proceedings of PROMSMmNet'97, pp. 18-25, IEEE, November 1997.

Almesberger, Werner; Ferrari, Tiziana; Le Boudec, Jean-Yves. *SRP: a Scalable Resource Reservation Protocol for the Internet*, Proceedings of IWQoS'98, pp. 107-116, IEEE, May 1998.

Almesberger, Werner; Giordano, Silvia; Le Boudec, Jean-Yves. *Reservation Models: From Arequipa to SRP*, Proceedings of Globecom98, Sydney.

Almesberger, Werner; Chandran, Leena; Giordano, Silvia; Le Boudec, Jean-Yves; Schmid, Rolf. *Quality of Service Renegotiations*, Proceedings of SPIE Int. Symp. on Voice, Video and Data Communications, Boston, November 1998.

Almesberger, Werner. *Linux Network Traffic Control – Implementation Overview*, Proceedings of 5th Annual Linux Expo, Raleigh, NC, May 1999, pp. 153-164.

Almesberger, Werner; Hadi Salim, Jamal; Kuznetsov, Alexey. *Using Differentiated Services on Linux*, To appear in Proceedings of IASTED Internet and Multimedia Systems and Applications Conference (IMSA'99), October 1999.

Almesberger, Werner; Hadi Salim, Jamal; Kuznetsov, Alexey. *Differentiated Services on Linux*, To appear in Proceedings of Globecom '99, December 1999.

E.5 Software

Related documents which are already mentioned above are not repeated in this section.

E.5.1 ATM on Linux and Arequipa

ATM on Linux, `http://icawww1.epfl.ch/linux-atm/`

Almesberger, Werner. *Linux ATM device driver interface*, `ftp://lrcftp.epfl.ch/pub/linux/atm/docs/`, January 1996.

Almesberger, Werner. *Linux ATM API*, `ftp://lrcftp.epfl.ch/pub/linux/atm/api/`, EPFL, July 1996.

Almesberger, Werner. *Linux ATM internal signaling protocol*, `ftp://lrcftp.epfl.ch/pub/linux/atm/docs/isp-current.tar.gz`

Almesberger, Werner. *Arequipa: Design and Implementation*, `ftp://icawww1.epfl.ch/pub/arequipa/aq_di-1.tar.gz`, Technical Report 96/213, DI-EPFL, November 1996.

E.5.2 Diffserv on Linux

Differentiated Services on Linux, `http://icawww1.epfl.ch/linux-diffserv/`

Almesberger, Werner; Hadi Salim, Jamal; Kuznetsov, Alexey. *Differentiated Services on Linux* (work in progress), Internet Draft `draft-almesberger-wajhak-diffserv-linux-01.txt`, June 1999.

E.5.3 SRP

Scalable Resource Reservation Protocol, `http://icawww1.epfl.ch/srp/`