Optimization of TCP/IP Traffic Across Shared ADSL

M.Sc. Thesis

Jesper Dangaard Brouer
<hawk@diku.dk>

26th January 2005

Department of Computer Science
University of Copenhagen
Denmark

Abstract

This thesis presents practical studies of the TCP performance problems caused by the asymmetric nature of ADSL connections. Previously, it has been shown on other types of asymmetric links that TCP throughput may be reduced due to a variable and imperfect ACK feedback. The upstream capacity of ADSL products in general does not disturb the ACK feedback mechanism, but we analyze and document that TCP traffic across ADSL in fact is affected by the asymmetric nature of ADSL, when utilizing the upstream capacity. For a single user ADSL installation it is manageable to avoid upstream congestion, but for larger networks, connected to the Internet by ADSL, the rise of peer-to-peer file sharing applications may result in a permanently congested upstream link.

The thesis provides evidence that the achievable downstream throughput is reduced significantly in case of a saturated upstream link. Saturation of the upstream link introduces a high queueing delay that effectively renders the connection useless for interactive and other delay-sensitive applications (like VoIP).

Previous work within the field of asymmetry has primarily been concerned with optimizing downstream throughput. Our solution incorporates latency as an important design parameter in order to support different types of network applications at the same time. The thesis also provides a practical solution to mitigate these problems. We demonstrate how it is possible to achieve full downstream and upstream utilization, while at the same time supporting different delay-sensitive applications, using the packet scheduler of a Linux based middlebox between the network and the ADSL connection.

On ADSL the available bandwidth for IP traffic varies significantly, which is caused by protocol overhead and packet/cell aligning at the ATM/AAL5 link layer. Depending on packet sizes the available bandwidth can be reduced up to 62 percentage. Therefore, a packet scheduler needs to account for the varying available bandwidth by accounting for the link layer overhead. As part of implementing a practical solution, based on Linux, the Linux Traffic Control system have been modified to perform accurate packet scheduling through modeling the ATM link layer overhead of ADSL.

Preface

This is a master's thesis in Computer Science written under the Department of Computer Science, University of Copenhagen. The work was conducted by Jesper Dangaard Brouer between December 2003 and January 2005.

Why English

I have chosen to write in English although English is a second language to me. It is a challenge as I am fairly untrained in written English. I have accepted this challenge for a reason.

Searching within the domain of this thesis I encountered various software packages which had basic install and readme information in English, but with the technical and theoretical foundation documented in a foreign language, e.g. in form of a university report like this.

This annoyed me because it rendered the technical and theoretical documentation unusable. The software packages where still functional but learning from and building upon the achieved knowledge and theoretical insight, was not possible. My reason for completing the thesis in English is to avoid having my work rendered useless to people outside Denmark.

Acknowledgments

In the process of writing this thesis, many generous people have provided me with help, advice, and guidance and deserves my thanks and gratitude.

First of all, my thanks goes to Per Marker Mortensen, my office mate in the final stage of this project, for technical discussions, advice, and guidance, plus a very detailed technical proof reading and for his ability to organize my thoughts when I myself could not.

My advisor Jørgen Sværke Hansen, for his guidance and for committing to write an article together based on this thesis, although the article was not accepted by ACM, we will prevail and try to submit the article again with the latest results from this thesis.

Eric Jul, for guidance amd employment as a research assistant under the Danish Center for Grid Computing (DCGC) during the writing of this thesis.

Niels Elgaard Larsen and Troels Blum, for trying out the "ADSL-optimizer" in production on their ADSL connection, which is shared by an apartment block.

The dormitory Kollegiegården, for allowing me to conduct experiments and optimizations on a production ADSL (8 Mbit/768 kbit) line shared by potentially 307 individuals.

Jon Bendtsen, for allowing me to conduct experiments on his personal ADSL (Tele2, 2 Mbit/512 kbit) line, on which most of the controlled experiments have been conducted.

Steffen Schumacher from TDC backbone and Kristen Nielsen for getting me in contact with TDC backbone.

Martin Lorensen from Tele2, for verifying the encapsulation protocol used on the Tele2 ADSL connection.

Peter “firefly” Lund, for nitpicking the report for incorrect grammar.

Martin Leopold, for modifying the DIKU logo and general competitive layout ideas.

Bjarke Buur Mortensen, for competing thesis writing through CVS commit-mails.

Allan Beaufour Larsen, for his excellent feedback after proof reading the thesis, and for sharing his knowledge of Makefiles.

Steffen Nissen, for proof reading and generally for being a likeable person.

The lunch club, for keeping me in contact with the real-world and accepting my strange combination of food products.

Thanks also goes to, authors of all the GNU programs and tools I have used throughout this thesis. Alexey Kuznetsov, for coding the Linux Traffic Control system. Martin Devera, for implementing the HTB packet scheduler under Linux, which I have modified and used as the primary packet scheduler in the solution. Christophe Kalt, for the RRDtool frontend `drraw` and for accepting my patches. Walter Karshat, for writing the original overhead patch to HTB and helping me to understand the rate table lookup system, which I have modified and partly moved into kernel space.

Finally, this thesis would not exist without the support of my wonderful girlfriend, Berit Løfstedt. She has put a tremendous effort into proof reading and correcting my English throughout the entire writing process. She kept me motivated when I had lost faith in the project, and above all love and for carrying my unborn child.

Contents

Chapter 1

Introduction

Broadband Internet connections, that are based on an asymmetric technology, are becoming more and more popular, especially Asymmetric Digital Subscriber Line (ADSL)[1]. The asymmetric nature of ADSL conflicts with the design of TCP/IP since TCP[63] builds upon an assumption of symmetric lines. The problems arise mainly due to the limited upstream capacity, which can be saturated with ease. The problems observed, when the upstream is saturated, are high latency and the inability to utilize full downstream capacity.

The ADSL technology was designed for a client-server usage pattern with a higher downstream demand than upstream, like home users using the World Wide Web. The increased use of peer-to-peer (P2P) file sharing applications has changed the usage pattern dramatically. Now, an ADSL upstream line can be permanently saturated. A single user can deal with this on the application level but if an ADSL connection is shared by a group of users, other methods are necessary.

This thesis provides a practical solution to mitigate these observed problems by introducing a packet scheduling "middlebox" between the local network and the ADSL modem. We provide a fully functional solution in the form of a software package for a Linux based router. Our solution is currently operational on 3 sites and has proven its worth on a real network of about 307 autonomous users connected to the Internet by a single shared ADSL connection[2]. The design of the software package is not the focus of this thesis, we only use it to demonstrate that our solution works in a real-world scenario.

The thesis is divided into three parts. The first part is focused on analyzing and documenting how, and to what extent, TCP traffic on a unmodified ADSL is affected by its asymmetric nature. The second part is focused on designing and evaluating components for a practical middlebox solution. Latency is incorporated as an important parameter in the design to support delay-sensitive applications. The third part combines the components and describes a real-world setup with a practical middlebox solution. In the solution we focus on achieving low latency (for delay-sensitive applications) during saturation of the upstream link, while proving full utilization of the downstream link.

The analysed effects of asymmetry also relevant to other asymmetric technologies like cable-broadband solutions. In this thesis, we will, however, only focus on the

[1] As of November 2003 [8], 44 percent of the private Danish Internet connections where based on broadband. Of those, 97% are based on a asymmetric technology; ADSL 68%, cable-broadband 29%.

[2] 8 Mbit/512 kbit ADSL connection.

asymmetric effects and solutions on ADSL.

1.1 Motivation

This thesis is motivated by the need for a practical solution to the problems experienced in a real-world network of about 200 autonomous users connected to the Internet by a single, shared 8 Mbit/768 kbit ADSL connection. The problems observed were a permanently saturated upstream line, underutilization of the downstream line and high latency. This effectively rendered the connection useless for interactive and other delay-sensitive applications and made web browsing a tedious task. Even bulk downstream TCP transfers didn't get the expected throughput.

We expect the problems to exist in smaller networks as well, since the bandwidth of most ADSL connections is small compared to the bandwidth of local connected machines. A single machine with a standard 100 Mbit/s network card can saturate an ADSL upstream line with ease.

The degradation of TCP performance over asymmetric links is a well-known phenomenon. It has been documented in several articles [19, 20, 39, 50, 60, 67] and most of the techniques have found their way into RFC3449[18] – "TCP performance Implications of Network Path Asymmetry". The main finding is that: "performance often degrades significantly because of imperfection and variability in the ACK feedback". The articles within the field of asymmetry are primarily concerned with optimizing downstream throughput on links with a high level of asymmetry, often with insufficient upstream ACK capacity.

Our focus on ADSL connections concerns links with a fairly low level of asymmetry, with sufficient upstream ACK capacity (at least for one-way transfers). Thus, we are motivated by and interested in determining how, and to what extent, links with this level of asymmetry are affected by their asymmetric nature. Another parameter of interest, that has not been addressed in the articles, is latency. We are motivated by finding a practical solution where supporting delay-sensitive applications is an important design parameter. Supporting delay-sensitive as well as other types of applications, requires allocating and sharing the link resources between network applications according to their service requirements.

1.2 Network Environment

This section illustrates a general view of our network environment. Our ADSL networking environment is illustrated in Figure 1.1 (with Cisco compliant diagrams). The individual components are numbered 1 to 10.

We have a Local Area Network (LAN) (1) which shares a single ADSL connection for Internet access. We choose to view the LAN (1) as an autonomous network, where we have no control of the individual machines. The components (2), (3), and (4) can be combined into a single component, but we choose to illustrate their functionality as separate components.

The gateway (2) and the router (3) both perform IP routing, but (3) changes the transport mechanism from IP to ATM. It should be noted, that data is transported as ATM all the way to the Broad Band Remote Access Router (BBRAS) (9), from where it is routed as IP packets again (10).

The ADSL modem (4) and the Digital Subscriber Line Access Multiplexer (DSLAM)

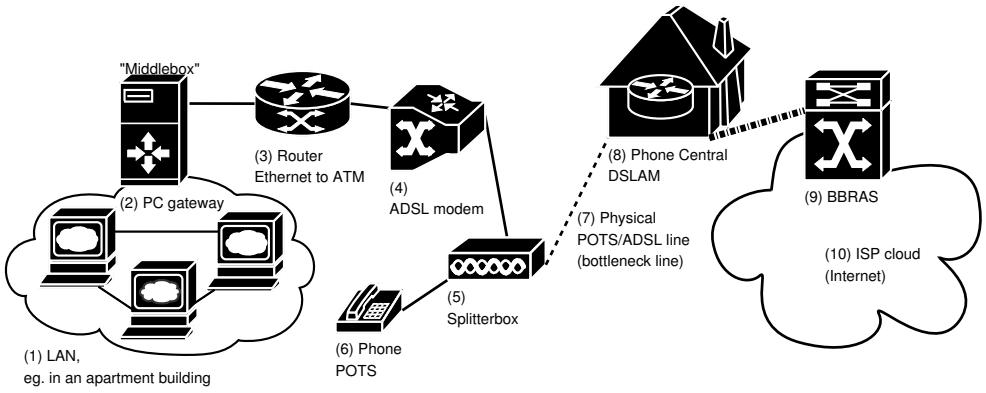


Figure 1.1: ADSL network environment.

(8) performs the ADSL signal encoding and decoding, over the physical copper wires (7). The components (5) and (6) are not active network components, but perform the task of allowing the Plain Old Telephone Service (POTS) to share the same wires as ADSL.

1.3 Goals

Our goal is to create a practical solution that optimizes an ADSL connection shared by a busy autonomous network with respect to both interactive comfort and maximum link utilization.

The following elaborates on the specific components of the goal:

- We are focused on a *practical solution*, where the network is *shared* and *autonomous*, which limits us from general protocol modifications. The autonomous property is a restriction to our environment and solution. The sharing property is a parameter for our solution, implying that each individual user should receive a fair share of the link.
- *Interactive comfort* puts an emphasis on making the connection useful for different types of network applications at the same time. This involves supporting delay-sensitive applications with special service requirements. Supporting this requires allocating and sharing the link resources between network applications according to their service requirements.
- *Maximum link utilization* is focused on finding a solution for full downstream utilization, as we assume upstream is saturated or at least is saturated with ease. Maximum link utilization also implies that the solution should avoid unnecessary waste link capacity to achieve other goals like low latency.

1.4 Why a Middlebox Solution

Our goal is to find a practical solution, which we also view as an easy deployable solution from the ADSL subscribers' point of view. Our goal also identifies the need for sharing

the link resource. Doing so requires control of the packets parsing through the link, as this gives us the ability to control and modify the timing and drop probability of the packets. As this allows us to enforce link resource sharing.

With the restrictions from our goal we choose to abandon the idea of modifying the TCP/IP stack, as we cannot deploy our changes to every host on the Internet (10) or on the autonomous LAN (1), since we have no control over the individual machines. Taking the subscribers' point of view we want our solution to be independent of the Internet Service Provider (ISP). Thus changing the equipment at the ISP, the DSLAM (8) and the BBRAS (9), is not an option for our solution.

With these restrictions the choices left are the PC gateway (2), the (ATM) router (3), and the ADSL modem(4). Due to hardware restrictions, we choose to introduce or replace the "PC gateway" (2), with a packet scheduling router. We refer to this router as a *middlebox*[27] solution[3]. This middlebox solution should fulfill the restrictions stated in our goal of being a practical solution, which can enforce resource allocation and sharing on the link of a shared autonomous network.

We would have preferred to introduce a middlebox which performed the combined functionality of the gateway (2), the (ATM) router (3), and the ADSL modem(4). Acting as the ADSL modem (4) we would have the advantage of getting the exact line speed from the modem. Acting as the ATM router (3) we would act on the same type of link layer and have knowledge of the specific encapsulation method. However, replacing the modem would require specialized hardware, which is hard to get hold of and thus not easily deployable. ATM network equipment is often expensive and not easy to come by, making it an impractical solution.

Thus, we choose to use standard PC hardware (with two network interfaces) running Linux and acting as a PC gateway (2). This serves our objectives of a practical and easily deployable solution for the home ADSL owner.

1.5 Challenges

This section will discuss the challenges to achieving our goal. The overall challenge is to achieve low latency, while at the same time achieving a high throughput on a loaded network, through the means of a single middlebox placed between the ADSL modem and the local (autonomous) network.

On our busy autonomous network, we assume that the resources on the link cannot meet all the traffic demands most of the time, thus we are faced with a congested network. Congested network connections experience high latency and packet loss rate, due to fully loaded packet queues. This makes the link unusable for delay and loss sensitive network applications. TCP does adapt to congestion when (data) packets are dropped, but a problem arises when ACK packets experience high delays. TCP uses the reverse path to estimate the link capacity via ACK packets, which serves as a self-clocking mechanism. Thus, delays on the reverse (ACK) path give a false capacity estimation of the forward (data) path. This can result in under-utilization of the forward path, when the reverse path is congested. We need a solution to mitigate this ACK delay in order to achieve full link utilization in both directions at the same time.

To be able to use the link for different types of services, at the same time, we need some kind of link resource sharing. To share the resources, we need to define a set

[3]Others also refer to this as a Performance Enhancing Proxy (PEP) [24], but a PEP is also a type of middlebox.

of service classes and (assured) service levels for each class. This is solved through link sharing with some kind of fair-queueing packet scheduling algorithm. This poses another problem, namely classifying traffic into the correct service classes. Users might try to evade classification due to the nature of our autonomous network, where users compete for bandwidth. This, unfortunately, turns out to be the norm rather than the exception. This is due to the recent development in peer-to-peer (P2P) file sharing systems, which perform a lot of evasive techniques in an attempt to avoid firewalling, and thus classification.

Link sharing has been done before but with ADSL we are faced with some additional problems. ADSL is, as indicated by its name, asymmetric by nature. This amplifies the negative effect on the TCP ACK clocking mechanism, as described above. Another issue is the overhead introduced by the ADSL link layer, which results in a variable link capacity. The link sharing mechanism (obviously) needs to know the capacity of the link it is sharing. Thus, we need to instrument the packet scheduler to account for this link layer overhead.

We also see it as an important challenge to establish how, and to what extent, a unmodified ADSL is affected by different traffic loads, especially related to its asymmetric nature. This is important for evaluating the effect of our solution and to document that ADSL does suffer from its asymmetric nature.

1.6 Approach and Thesis Outline

Our approach to attack these challenges has been done in three steps, which is why the thesis is partitioned into three parts, which contain the following:

Part I where we perform a preliminary analysis to establish how and to what extent ADSL is affected by its asymmetric nature. When analyzing the asymmetric effects, a special focus is put on the interaction between TCP and the asymmetric nature of ADSL.

Part II where we identify and evaluate components in order to design a middlebox solution. The components should mitigate the observed problems and seek to fulfill parts of our goal.

Part III where we combine the identified components into a practical and functional solution. The solution is evaluated on a real-world network to document whether we have achieved our overall goal.

The thesis also contains a list of acronyms on page 122 and an index on page 125.

The reader is assumed to have a good knowledge of the TCP/IP protocol, if not we recommend the book series "TCP/IP illustrated" by Richard Stevens[72, 73, 74]. We also expect the reader to have some basic knowledge about QoS and basic knowledge about token bucket theory, if not we recommend an excellent book about QoS[78].

1.7 Contributions

In this thesis we present the following main contributions:

- A detailed analysis and documentation of how TCP traffic across ADSL is affected by the asymmetric nature of ADSL when utilizing the upstream capacity.

- An overview of the different types of encapsulation methods on ADSL and their associated overhead.
- A practical evaluation and analysis of components needed to mitigate the observed problems on ADSL.
- A real-world evaluation of a practical solution based on a combination of the components.
- A patch for the Linux Traffic Control system for accurate link layer overhead modeling, which makes the packet scheduling algorithms work on ADSL.
- A software package called the *ADSL-optimizer*, which makes it easier for others to make use of our work in practice.

In addition to the primary contributions, we have contributed with patches, bug fixes, and feedback to the RRDtool frontend `drraw`, the Netfilter packet logging daemon `Specter`, and the token bucket queueing discipline `TBF`.

Part I

Preliminary Analysis of Asymmetric Effects

Chapter 2

Theory of Asymmetry and Effects on TCP

In this part, Part I, we perform a preliminary analysis of asymmetric effects with special focus on TCP and ADSL. This chapter is concerned with TCP performance problems related to various types of asymmetry. Chapter 3 relates our findings in this chapter to a real ADSL connection and documents the extent of the practical problems on ADSL.

The chapter is structured as follows. We briefly introduce the different types of asymmetries. We continue by exemplifying some technologies that exhibit the different types of asymmetry in order to show that a technology is often affected by several types of asymmetry.

This is followed by an analysis of how TCP is affected; Firstly by describing the TCP flow-control algorithm, which is the main cause of TCPs problems regarding asymmetry. Secondly, we describe how TCP is affected by each type of asymmetry. Here we describe and identify the *normalized bandwidth ratio, k*, which defines whether the upstream capacity is sufficient for transporting the ACK packets.

At last, we describe the effects of ACK queueing, because we find that ACK packets play a central role in the performance of TCP in asymmetric networks. Two queueing situations are described; sufficient and insufficient upstream ACK buffers.

2.1 Types of Asymmetry

There are various types of asymmetry, identified by [18, 19, 20], which can be related to:

Bandwidth The bandwidth is different in the two directions (downstream and upstream).

Latency The latency[1] is different in the two directions. Normally caused by data being sent on two different transmission media.

Quality The error rate is different in the two directions, also normally caused by two different transmission media.

[1]The latency in this case is viewed as the delay of physical medium, i.e., the combination of propagation and processing delay.

Media access The access to the media might be different according to the role in the network (e.g., for the base station and clients).

2.2 Asymmetric Technologies

In this section we describe some known asymmetric technologies to exemplify different types of asymmetry. A given asymmetric technology is often affected by a combination of different types of asymmetry.

We have chosen to describe categories of asymmetric technologies and only use specific standards and implementations where necessary.

Cable modems :

Cable TV networks have wide downstream bandwidth, but often there is no return channel. Some solutions use ISDN modems as return channel, which introduces a significant bandwidth asymmetry. Others use a limited return channel via cable net, which is shared with everyone else [30].

The technology clearly exhibits *bandwidth asymmetry*. For example in the Data-Over-Cable Service Interface Specification (DOCSIS) v1.0 [3], the downstream rate is either 27 or 52 Mbit/s and the upstream rate can be dynamically selected in a range between 166 kbit/s to 9 Mbit/s. The operator may assign several upstream channels per downstream channel. On the upstream channel DOCSIS uses a contention/reservation Media Access Control (MAC) protocol, which further limits the upstream capacity as each node must first request permission to send.

The technology also exhibits a degree of *latency asymmetry*, which is dependent on the return channel. If ISDN is used, the two medias differ and thus the latency will probably also differ. The latency of the return channel via cable net is often also affected, due to guard times between transmissions and contention intervals on the shared media.

The overhead per packet is different in the two directions (verified for DOCSIS [3, 6, 9]). A further look at the DOCSIS v1.0 MAC specification [3] reveals that, it is also affected by *media access asymmetry*. Each node is restricted to sending at most a single packet in each upstream time-division interval, which introduces a significant cost per packet and unfair access to the media.

Asymmetric Digital Subscriber Line (ADSL) :

As the name implies, the ADSL technology is based on asymmetric lines, where *bandwidth asymmetry* is the primary factor. In ADSL the downstream capacity is larger than the upstream capacity.

The ADSL standard G.992.1 [4] provides bandwidth in 32 kbit/s steps; Downstream from 32 kbit/s to 6.144 Mbit/s, and upstream 32 kbit/s to 640 kbit/s. The standard also supports some extra optional bearer channels, which can provide some extra capacity. A few ISPs in Denmark sell ADSL connections with 8 Mbit/s downstream and 768 kbit/s upstream, which is a factor of 10 between downstream and upstream.

The ADSL standard G.992.2 [5] is often called ADSL Lite or Splitterless ADSL (as it does not require a box for splitting the signal into phone lines and ADSL). It also provides bandwidth in 32 kbit/s steps; Downstream from 64 kbit/s to 1,536 kbit/s, and upstream from 32 kbit/s to 512 kbit/s.

ADSL uses ATM as link layer, which introduces a packet overhead in form of ATM Adaption Layer type 5 (AAL5) [1] encapsulation and an ATM packet alignment overhead.

Satellite links :

There is a vast number of satellite links with different kinds of feedback channels. Due to the many different types of feedback channels, satellite links can exhibit all types of asymmetry.

The two major asymmetries for satellite links are *bandwidth and latency asymmetry* [39, 45, 46]. The physical distance to the satellite often introduces a large propagation latency (roughly 300 ms one-way [45]).

The error rate on the satellite link may also be significant, but [45, 46] (in 1999) state that future satellite links will improve the bit error rate (quality) by using new modulation and coding techniques, which implies *quality asymmetry* when using two different mediums with different bit error rate.

Wireless/radio technologies :

Some wireless/radio technologies suffer from media access asymmetry[19]. Especially technologies using a central base station and several mobile clients as this requires a RTC/CTS (Ready To Send/Clear To Send) protocol[18, 20], with a large turn-around time caused by the radio. The MAC protocol makes it more expensive to switch the direction of transmission than to transmit in one direction continuously. The drop rate is also significantly higher in wireless networks, thus some kind of *quality asymmetry* is also likely to occur.

2.3 Effects on TCP

In this section we will look at the effects of the different kinds of asymmetry on TCP. Our primary focus is placed on how asymmetry affects TCP, because TCP is the dominating protocol on the Internet.

As a basis for understanding why TCP is affected by asymmetric lines we introduce and explain TCP's flow-control algorithm. The algorithm is the main reason that TCP experiences problems with asymmetric lines, because it is based upon an assumption of symmetric lines.

2.3.1 TCP Flow-control

The flow control algorithm used in TCP is described in Van Jacobson's classic article [49]. The basic idea is to achieve flow-control by using the reverse path to estimate the forward (data) path link capacity via ACK packets, which serve as a self-clocking mechanism. The algorithm tries to achieve balance or equilibrium by not sending new packets into the network until old packets have left the network. The principle of the algorithm is that the receiver cannot generate an ACK packet before receiving the corresponding data packet. The sender uses this information to deduce the bottleneck between sender and receiver. The information is given in form of an ACK feedback, which can be used directly by the sender to "clock" out data packets according to the incoming ACK rate. Hence, the algorithm has a self-clocking property[2].

[2]The description of the algorithm is simplified with regards to *congestion window* and *receiver's advertised window*.

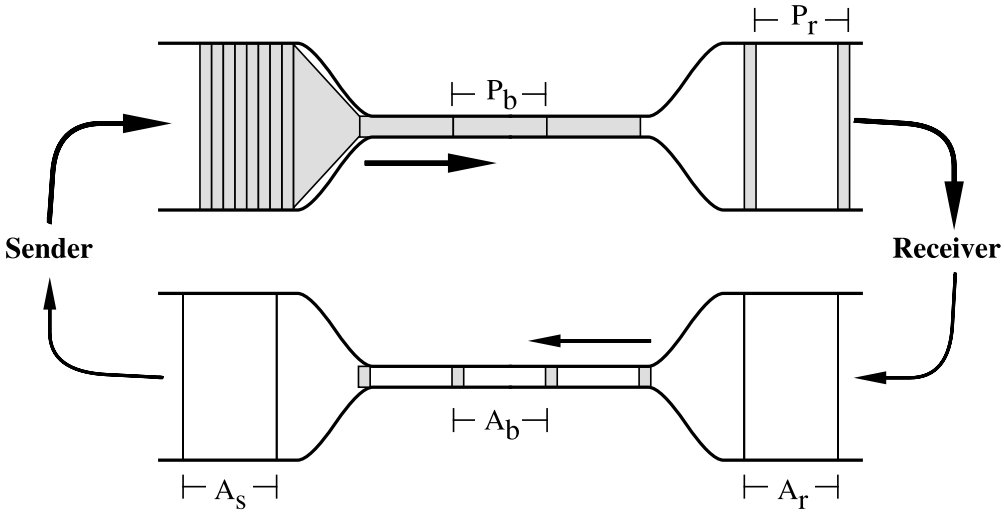


Figure 2.1: Van Jacobson's: Self-clocking Window Flow Control

How the algorithm deduces the bottleneck and achieves its "self-clocking" ability, is illustrated in Figure 2.1. The vertical dimension is bandwidth and the horizontal dimension is time. The shaded boxes are packets and the area is the packet size (as $Bandwidth * Time = Size$). Since the size of a packet is constant it is stretched out in time when it passes a line of less capacity, represented by the time P_b. Once the packets leave the bottleneck, the time interval between the packets, P_r, will still be P_b, which means that $P_b = P_r$. Assuming the receiver's processing time is the same for all packets, we can deduce that the ACK sending rate A_r equals P_b and P_r (knowing that the receiver cannot send an ACK before a data packet is received).

The **symmetry assumption**: If the bottleneck time slot P_b was big enough for the data packet then it is also big enough for the ACK packet, thus the spacing is preserved along the return path. Therefore, the ACK packets arrive at the sender with a time interval matching the data packets' bottleneck ($P_b = A_b = A_s$). Thus the effect of the bottleneck is communicated back to the sender.

The symmetry assumption introduces problems on asymmetric lines since data packets and ACK packets travel over lines with different performance characteristics. However, the problem is reduced a bit as (pure) ACK packets are significantly smaller than data packets. One might say that there is a built-in asymmetry between data and ACK packets. However, this is not enought to compensate for an asymmetric bandwidth.

The ACK vs. data ratio is further discussed in Section 2.3.2 below and the queueing implications are discussed in Section 2.4 on page 13. We will now describe how the individual types of asymmetry affect TCP, bearing in mind the TCP flow control algorithm.

2.3.2 Bandwidth Asymmetry

Bandwidth asymmetry affects the achieved downstream throughput due to a limited or congested upstream link.

As TCP relies upon data packets being acknowledged by the receiver, the return path carrying the ACK packets needs sufficient capacity. In an asymmetric setup,

where the upstream is the smallest link, a throughput problem arises if the upstream capacity is insufficient to transport the ACK packets needed for the downstream data packets.

Normalized Bandwidth Ratio

Whether the upstream capacity is sufficient, depends on the ratio between the downstream and upstream bandwidth divided by the ratio between the size of data packets and ACK packets.

For one-way transfers, [18, 20, 54] this is defined as the *normalized bandwidth ratio,* k:

$$k = \frac{Downstream/Upstream}{Data_{size}/ACK_{size}}$$

The value k indicates, how many data packets need to be acknowledged per ACK packet, to avoid saturation of the upstream by ACK packets. In other words, if there is more than one ACK packet for every k data packets, the upstream will get saturated (with ACK packets) before the downstream.

The TCP connection's *delayed ACK factor,* d defines the number of TCP data segments acknowledged by an ACK packet. Therefore, the k value must be evaluated in relation to the d factor. We can conclude if $k > d$ the upstream will be saturated by ACK packets. The delayed ACK algorithm [15, 36] specifies that TCP should send an ACK packet (at least) every second data packet. This means that 2 is an upper bound on d. Accordingly, it is a minimum requirement that $k \leq d \leq 2$.

However, delayed ACKs are not used when the TCP protocol are in certain states. Which situations depend on the TCP implementation, but it is generally recommended [61] that delayed ACKs are not used during *slow start* and *fast retransmit* mode[3]. A TCP push option set on the data packet generally also result in a instant ACK packet. This means that $k \leq 2$ is not a tight upper bound on k as we migth still have situations where the upstream link can be saturated by ACK packets.

Therefore, we wish to determine the lower bound on d and use this as a tigth upper bound on k. The TCP protocol states [15, 29, 36] that at most one ACK packet should be sent for each received data packet making $d \geq 1$ a tight lower bound.

Thus $k \leq d \geq 1 \Rightarrow k \leq 1$ is a tigth upper bound on k.

For example an ADSL line with 2 Mbit/s downstream and 500 kbit/s upstream, the raw capacity ratio is 4. With 1696-byte data packets and 106-byte ACK packets (two ATM frames), the packet size ratio is 16. This results in a k of approximately is $4/16 = 0.25$. This indicates that the ADSL line has sufficient upstream capacity to support full utilization of the downstream capacity.

Bidirectional Traffic

The above scenario only holds for one-way transfers, but most real-life traffic is bidirectional. The upstream link might be used for data packets, while at the same time delivering ACK packets. The competing reverse traffic consumes a part of the upstream capacity, effectively increasing the degree of bandwidth asymmetry [18, 54].

[3] Linux for example avoid using delayed ACKs for the first few ACKs of a new connection.

The effect of competing reverse traffic and the k value is discussed further in Section 2.4 concerning queueing, as the achieved throughput is a function of buffering, round-trip times, and k (a detailed description is found in [54]).

2.3.3 Latency Asymmetry

Variable latency affects the smoothness of a TCP data flow. TCP measures the Round-Trip Time (RTT) on the path to estimate the path Retransmission TimeOut (RTO) [62] (calculated from a smoothed RTT estimate and a linear deviation). In the event of (multiple) packet loss, the RTO needs to be accurate in order to respond in time to avoid unnecessary idle periods.

An unsteady ACK flow interferes with the flow of the data packets. A TCP flow is caused to pause if the latency of the ACK packets exceeds the time to transmit the TCP window thus causing underutilization of the available capacity.

2.3.4 Quality Asymmetry

The line quality and packet losses generally affect TCP and is not only an asymmetric problem. Loss of ACK packets is less significant than loss of data packets, as ACK packets are cumulative (ACK'ing all outstanding data), which generally results in stretched ACKs [18, 30, 39].

An example of an asymmetric path being intensified by data packet losses; Dropping data packets on the downstream will generate back-to-back ACK packets for each correctly received data packet after a loss, assuming that delayed ACKs are not used during the fast retransmit mode. The ACK packet for the retransmitted data packet may then be delayed by other duplicate ACK packet still in the upstream queue. This can result in a failure of the fast retransmit functionality and cause a TCP retransmit timeout. This scenario is more profound on a long delay path when the end host uses a large TCP window to maximize the throughput, thus having many data packets in flight each generating a duplicate ACK packet [18, 54].

2.3.5 Media Access Asymmetry

Media access asymmetry can be manifested in several ways (which we will not describe in detail). It is generally defined in [19] an uneven access to a shared medium by a distributed set of nodes.

We believe that the effect of this uneven media access can result in all of the above types of asymmetry. In some cases the base station has higher bandwidth to clients (than the other way around), thus a type of bandwidth asymmetry. Latency and quality for different wireless clients can differ due to the radio signal, which leads to latency and quality asymmetry.

2.4 ACK Queueing

This section is concerned with the effects of ACK packet queueing. When ACK packets begin to form queues on the upstream link, one of two scenarios may arise. Each is considered below.

1. The queue has sufficient buffers to prevent (ACK) packet loss.
2. The queue has a small buffer resulting in (ACK) packet drops.

Sufficient upstream ACK buffers

If the queue is sufficient, the downstream performance behaves according to the normalized bandwidth ratio k, which is described in Section 2.3.2. Sufficient implies enough buffers to avoid packet losses, that is larger than all the collective window sizes.

When $k > d$, indicating that the upstream capacity is insufficient to support ACK packets for the downstream capacity, the ACK packets are delayed or queued behind each other at the bottleneck router resulting in a breakdown in the ACK feedback mechanism. The ACK packet spacing no longer reflects the arrival time of the data packets, as assumed by the TCP flow-control algorithm. The sender now clocks out new data packets at a slower rate, which is now dependent on the queueing and capacity of the upstream link. Thus the downstream throughput is now limited by the upstream link.

An ACK queueing phenomenon called *ACK-compression* (first recognized by [81]) might occur when $k \leq 1$ (which indicates that the upstream is sufficient to carry the ACK packets generated in response to the data packets). As mentioned in Section 2.3.2, bidirectional traffic consumes a part of the upstream capacity, effectively increasing the degree of bandwidth asymmetry. In situations where $k > d$ a queue of ACK packets start to build. After a queueing delay, caused by data packet, the ACK packets are likely to be sent back-to-back resulting in bursty data traffic, which increases the risk of data packet loss and congestion on the data path.

Insufficient upstream ACK buffers, ACK drops

Lakshman et al. [54] have a detailed analysis of throughput with insufficient ACK buffers when $k > 1$. The analysis applies to a single TCP connection.

When k is large, upstream capacity is insufficient (to support all ACK packets for the downstream capacity) and ACK packets will start to form a queue. When the queue of the upstream link Q_{up} is filled, (ACK) packets will start to get dropped.

As mentioned in Section 2.3.4, ACK packets are cumulative and loss of ACK packets is less significant as the next ACK packet also acknowledges the previously received data. Unfortunately, this results in bursty data traffic as the ACK packet getting through acknowledges several data packets thus opening the sender window in a large chunk.

To avoid data packet loss, the *downstream queue* Q_{down} needs to be big enough to accommodate these bursts. Results from [54] show that the average burst size into the downstream buffer is k packets, because ACK packets which get through ACKs k packets. Thus $Q_{down} \geq k$ is needed to avoid data packet loss. This is not a guarantee to avoid loss as not all bursts are of equal duration.

The *window size* of the downstream path also needs to be big enough to accommodate for the less frequently arriving ACKs and the resulting bursty traffic. The mathematically derived model in [54] for the maximum window size W_{max} incorporates both the upstream Q_{up} and downstream Q_{down} queue sizes. The window size for full utilization in an ideal system is determined by the bandwidth-delay product ($W \geq Rate * RTT$). The model incorporates the normal window size plus the number of packets on the downstream path plus the surviving ACKs on the upstream path multiplied by the

number of data packets each ACK represent: $W_{max} = Rate * RTT + Q_{down} + k * Q_{up}$. Using a window size higher that W_{max} will result in (data) packet loss, as the window size would exceed the total calculated buffer and bandwidth capacity.

An anomaly was detected by Lakshman et al. [54] where the performance gets worse when the downstream queue $Q_{down} > 3k$. The number 3 matches the number of duplicate ACK packets needed to signal a packet loss. The anomaly is due to tail drops in the Q_{up} queue where dropping duplicate ACK packets results in a failure of the fast retransmit functionality. The problem occurs when the system reaches $W_{max} + 1$, resulting in a data packet drop. This tells us that the queues are fully loaded. The packet drop should be communicated back by duplicate ACKs but due to the large amount of packets in the Q_{down} queue and the resulting ACKs, which are sent on a slow link with high drop rate, the duplicate ACK will not get through in time. The anomaly is a TCP timeout which could have been solved by a fast retransmit[4]. The anomaly can be solved by using a drop-from-front method on the ACK queue. For a data transfer on the upstream path (with ACK packets traveling on the downstream path) ACK packets should/must not be dropped as the smaller upstream link is more vulnerable to the resulting bursts.

2.5 Summary

In this chapter we have described different types of asymmetry and how they affect TCP. We have exemplified asymmetry with some technologies, which exhibit the different types of asymmetry, and find that technologies often exhibit a combination of different types of asymmetry.

We have explained the TCP flow-control algorithm, as it is the main cause of TCPs problems regarding asymmetry, because it is based upon an assumption of symmetric lines. The algorithm uses ACK packets to clock out data packets. The assumption is that the spacing of data packets is preserved and communicated back by the corresponding ACK packets, which is only achievable if the reverse channel has sufficient capacity. This depends upon the normalized bandwidth ratio, k and competing reverse traffic (due to bidirectional traffic patterns).

When the reverse channel does not have sufficient capacity, the data throughput depends on the queueing of ACK packets. We have described the effects of queueing of ACK packet with sufficient and insufficient buffers, using a simple FIFO tail-drop queueing mechanism. Queueing of ACK packets generally results in a breakdown in the ACK feedback mechanism as the inter-ACK spacing is lost. A queueing phenomenon called ACK-compression, can occur even when the reverse channel should have sufficient capacity according to the k value, because competing reverse traffic can create periods of insufficient reverse capacity.

From this, we have identified the following performance-related factors:

1. The normalized bandwidth ratio k.
2. The delayed ACK factor d.
3. The competing reverse traffic flow.
4. The queue size of the bottleneck routers.

These factors are evaluated in practice in the next chapter.

[4]We refer to [54] for a detailed description as it also depends on the type of TCP/IP implementation (TCP-Tahoe and TCP-Reno are described)

Chapter 3

Practical Evaluation of Asymmetric Effects on ADSL

In this chapter we will establish and analyze how, and to what extent, TCP is affected by ADSL's asymmetric nature. We investigate performance related problems by performing various tests in order to document how a real physical ADSL connection is affected by different traffic patterns related to its asymmetric nature.

From Chapter 2, we found four factors which influence performance: The normalized bandwidth ratio k, the delayed ACK factor d, competing reverse traffic (bidirectional traffic), and queue size of the bottleneck routers. In this chapter, we evaluate these factors in practice.

As we focus on real-life ADSL connections, we relate the normalized bandwidth ratio k, to ADSL configurations sold by commercial ADSL providers. We show how bidirectional traffic significantly changes the achievable downstream throughput, as the upstream traffic disturbs the ACK feedback mechanism. We then analyze the results and find that the increased latency is caused by a (large) queue in the upstream router/modem. We also demonstrate a direct correlation between the TCP window size and the queue size. At last, we document the existence of the ACK-compression phenomenon[81] on ADSL and demonstrate the bursty behavior.

3.1 ADSL Products

All the commercial ADSL configurations shown in table 3.2 have sufficient upstream capacity to carry ACK packets generated in response to data packets This is indicated by the normalized bandwidth ratio k, which is below one ($k < 1$). In respect to unidirectional transfers, the level of asymmetry in commercial ADSL configurations should, therefore, not reduce the downstream throughput.

It is worth noting the ACK and data packet sizes in table 3.2. The packet sizes are caused by ATM link layer overhead of ADSL, which is described in detail in Chapter 5. The 106-byte ACK packet represent a 40-byte TCP ACK packet, which consumes two ATM frames on a ADSL connection. The 1696-byte data packet correspond to 1500-byte IP packet including ATM overhead.

ADSL products

on the Danish market July 2004, the four dominant providers.

ISP	$downstream$	$upstream$	$ratio$	k
A,B,C	256	128	2.00	0.13
A,B,C,D	512	128	4.00	0.25
C	1024	128	8.00	0.50
C	1536	128	12.00	0.75
A,C,D	2048	128	16.00	1.00
A	512	256	2.00	0.13
B,C	1024	256	4.00	0.25
A	2048	256	8.00	0.50
C	1024	512	2.00	0.13
A,B,C,D	2048	512	4.00	0.25
C	2560	768	3.33	0.21
A,C	4000	768	5.21	0.33
A,C	8000	768	10.42	0.65
A (mistake)	8000	512	15.63	0.98
A (sync rate)	8000	668	11.98	0.75

ISP-list:
A=TDC
B=Tele2
C=CyberCity
D=Tiscali

$Datapacketsize$:	1696.00
$ACKpacketsize$:	106.00
$PacketRatio$:	16.00

Table 3.2: The k value for different commercial ADSL products.

3.2 Test Setup

The tests performed in this chapter have been performed on two different ADSL connection. An 8 Mbit/512 kbit ADSL connection from TDC and a 2 Mbit/512 kbit ADSL connection from Tele2. Most of the tests have been performed on the Tele2 ADSL, because the TDC ADSL is a production ADSL which we were only allowed to take down for a short period of time.

The 8 Mbit/512 kbit TDC connection is later used to demonstrate our solution in a real-world scenario. After the tests where performed, the upstream capacity has changed because the connection had a mis-configured upstream capacity, which has since been corrected to 768 kbit/s by TDC[1]. When performing the tests for this chapter the upstream connection was configured to 512 kbit/s.

All experiments are performed between a local host on the network, connected to the Internet via an ADSL, and a couple of well connected external Internet hosts[2]. All hosts used the standard TCP/IP implementation of the Linux kernel 2.4 series.

The level of asymmetry in commercial ADSL configurations, indicate that it is reasonable to expect that a single downstream TCP transfer is able to get a throughput equal to the downstream capacity. Our theory from Chapter 2, tell us that competing reverse traffic consumes part of the upstream and thus influences the effective k values. This makes it interesting to investigate what happens when the upstream link carries

[1] At one time we were limited by the line's max (ADSL sync) rate of 668 kbit/s, this has also been corrected.

[2] The well connected external hosts have a 100 Mbit/s connection through "Forskningsnettet" and are 4-5 hops from the Danish Internet eXhange point. They are located at the Department of Computer Science, University of Copenhagen.

data packets, while at the same time delivering ACK packets (a.k.a. bidirectional traffic).

To facilitate bidirectional traffic tests a simple test setup has been constructed. The basic functionality is to start a number of concurrent upstream and downstream TCP transfers within a given time interval and record a packet dump and latency data. Information is pulled from the packet dump and latency data in order to generate different types of graphs. The test script (*bandwidth-tester*) and a description of the type of configuration files used can be seen in Appendix B.1 on page 134. There is one configuration file for each test, but we have only included a single configuration file to give an example.

Our *downstream* TCP transfer is performed with `wget`, which is a simple HTTP GET of a large file. Our *upstream* TCP transfer is performed with `scp`, which is a (secure shell) copy of a local file to a remote host. The traffic types was simple chosen because they each provide an easy way of performing bulk transfers in each direction.

Test precautions

Some precautions have been taken to assure that the tests measures the properties of the ADSL connection and not local interface queueing or disturbing traffic flows.

All the traffic experiments have been conducted so that no other TCP flow was present on the ADSL connection. Some random, and negligible, Internet packets might have been present during our tests, but we have verified that only our TCP flows were established.

There is a potential problem using `scp` and Linux' default queuing discipline (qdisc) "pfifo_fast" as it has 3 "bands" which get allocated according to the Type Of Service (TOS) bits. First In First Out (FIFO) queueing applies for packets in each band. Between bands all waiting packets in band 0 are processed before band 1, and band 1 before band 2. When copying with `scp` the TOS field is set to "maximum-throughput" (0x8) which puts it in band 2. The (ACK) packets from the `wget` are put in band 1 ("best-effort"). This could result in packet re-ordering, where the (ACK) packets from the `wget` are processed before the `scp` packets. The packet re-ordering by "pfifo_fast" is only a potential problem, as it should not happen in practice. The local Linux box is connected to the ADSL modem with a 100Mbit/s network card, thus no queue should be able to build up on the Linux box. To rule out any potential packet re-ordering we change the queuing discipline (qdisc) to use "pfifo", which is a strict FIFO queue.[3]

3.3 Bidirectional Traffic

We expect bidirectional traffic to have a negative impact on the throughput achieved by the downstream transfer as the reverse ACK traffic is competing with the upstream transfer, which effectively increases the k value (Section 2.3.2). The upstream throughput should not be affected in the same way as the downstream capacity, because the downstream capacity should be sufficient to handle an ACK feedback indication from an (upstream) line with less capacity (even when carrying data packets at the same time).

To illustrate and verify that the downstream TCP traffic is affected by upstream traffic a simple test is performed with a single continuous downstream transfer and two

[3] We have done some basic tests, which indicated no packet re-ordering, but to rule out this factor we change the queuing discipline.

short non-overlapping upstream transfers. The downstream transfer is allowed to run alone for 60 seconds after which an upstream transfer is started. This is referred to as *the first upstream period*. After the first upstream transfer completes, the downstream transfer runs alone again for 60 seconds after which a new upstream transfer is started, this is referred to as *the second upstream period*. The downstream transfer is allowed to complete after the second upstream transfer is completed. This test is illustrated with two figures showing two different ADSL lines, Figure 3.1 an 8 Mbit/512 kbit and Figure 3.2 on the following page a 2 Mbit/512 kbit connection.

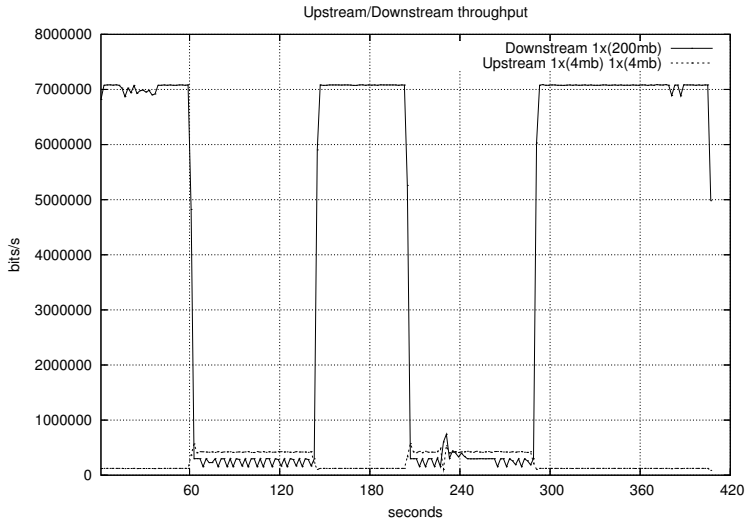


Figure 3.1: Illustrating the downstream throughput problem on a clean 8Mbit/512kbit ADSL line from TDC. One continuous downstream bulk TCP transfer (200Mb), and two upstream bulk TCP transfers (4MB each).

The upstream transfers have a significant impact on the downstream throughput, as can be seen in Figure 3.1 and 3.2. During the upstream transfers in Figure 3.1 (the 8 Mbit/512 kbit line), the downstream throughput is reduced from approximately 7 Mbit/s to 300 kbit/s (jumping between 150 kbit/s and 300 kbit/s). Figure 3.2 shows the same type of test on a 2 Mbit/512 kbit ADSL line, which is less asymmetric (see table 3.2). The downstream throughput is reduced from approximately 1800 kbit/s to around 440 kbit/s.

Test analysis

We will show that ACK packets from the downstream transfer get queued together with the upstream packets and thus experience the same delay and queuing, which return a false indication of the line capacity back to the (downstream) sender due to the ACK clocking/feedback mechanism (described in Section 2.3.1).

As the ACK packets are significantly delayed, the downstream sender uses/fills up the TCP window and has to wait until it is opened up by the reception of new ACK packets. Now the downstream throughput is dependent on the delay imposed on the ACK packets. Figure 3.3 on the next page shows the ping latency/RTT during the throughput test shown in Figure 3.2, on a 2 Mbit/512 kbit ADSL line from Tele2. During the two upstream transfers, the latency increases from a maximum of 185 ms (during the downstream-only part) to around 1150 ms. We have verified that the RTT experienced by the TCP packets of the downstream transfer (analysing the packet

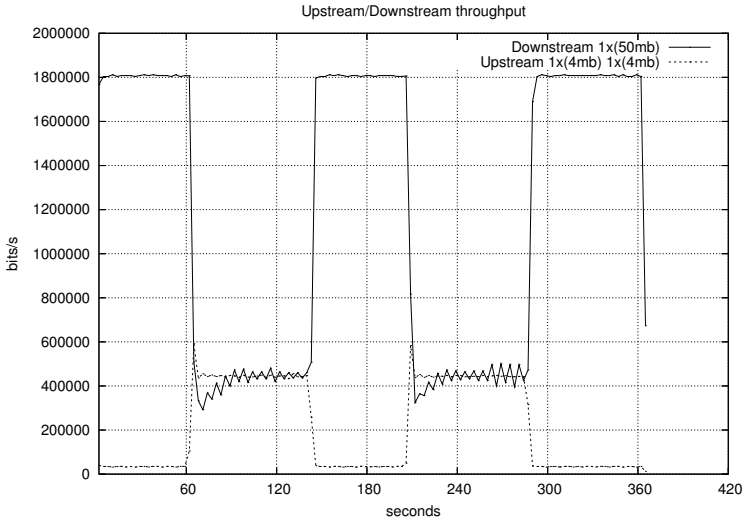


Figure 3.2: Illustrating the downstream throughput problem on a clean 2Mbit/512kbit ADSL line from Tele2.

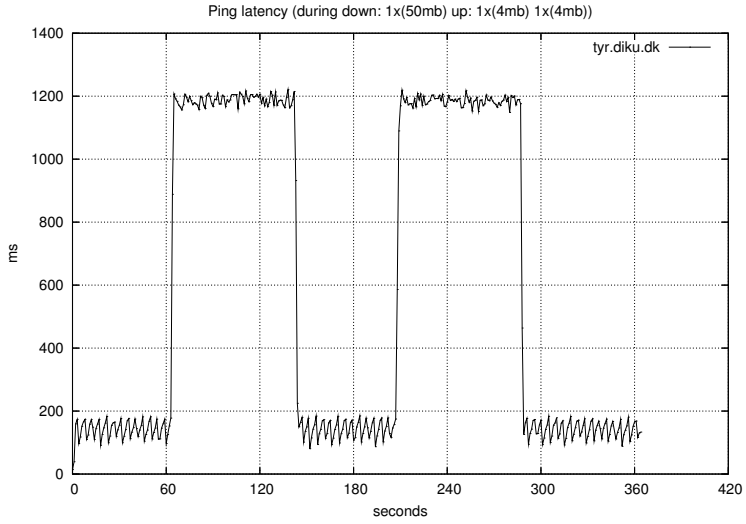


Figure 3.3: Ping latency during the throughput test in Figure 3.2 on a clean 2 Mbit/512 kbit ADSL line from Tele2.

dump) is equal to the RTT of the ping/ICMP packets, illustrated in Figure 3.4. We prefer to use the ping data because it is far easier to obtain and synchronize with other test data obtained from the test machine (the TCP latency graph requires a packet dump on the remote machine providing the downstream data).

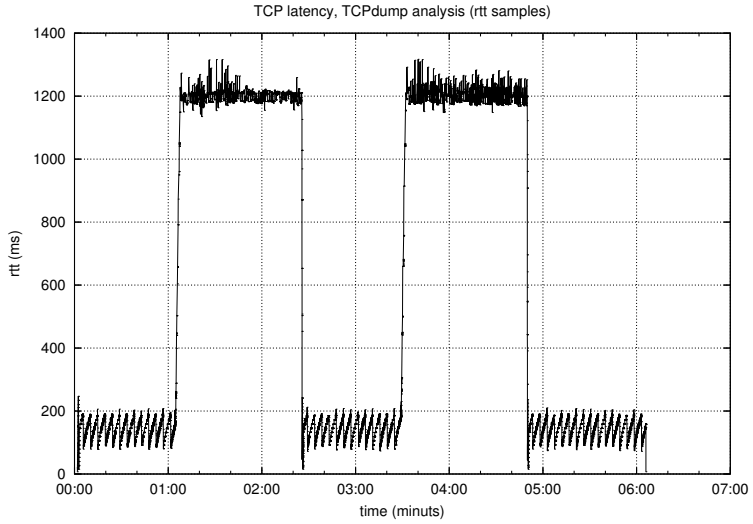


Figure 3.4: TCP latency of the downstream transfer seen from the downstream test host (which delivers data to our downstream transfers) during the throughput test in Figure 3.2 on the preceding page. (Analysis done on packet dump)

From the packet dump we can see that the advertised TCP window size is increased to 63712 bytes. The achievable throughput of a TCP connection can be roughly calculated from the (congestion) window size and RTT [53]:

Formula 3.1

$$Throughput = \frac{WindowSize}{RTT}$$

A RTT of 1150 ms imposed on a TCP connection with a window of 63712 bytes corresponds to a bandwidth of around 443 kbits/s. This matches the downstream throughput achieved in Figure 3.2 during the upstream transfers. We also notice that the downstream throughput is not achieved at once, the throughput slowly increases during the two upstream transfers and levels off. The reason for this is that the throughput is limited by the congestion window. As the downstream transfer is unlikely to experience packet drops during the upstream transfers (as it is under-utilizing the capacity) the congestion window can still grow, but will at some point be limited by the maximum allowed window.

To verify that the throughput is limited by the congestion window we show a correlation between the throughput and congestion window in Figure 3.5. The figure is a closer look of the first upstream period of Figure 3.2. The (estimated) congestion window is illustrated in Figure 3.5(b), which is the instantaneous outstanding data of the downstream transfer, measured as outstanding unacknowledged data at the sender. This is a good estimate of the congestion window at the sender. The congestion window (Figure 3.5(b)) is increased until it reaches the maximum advertised TCP window of 63712 bytes at around 92 seconds. This corresponds with the throughput leveling off

in Figure 3.5(a). It clearly shows that our throughput is limited by the TCP window size and that the TCP behavior is to utilize the total window.

The spikes in the congestion window Figure 3.5(b) before and after the upstream transfer (a period of unidirectional downstream traffic) is a perfectly normal behaviour of TCP/IP, which is caused by a packet loss. The congestion window is increased until a packet loss is experienced. The sharp spikes are caused by the use of SACK packets, which allows the TCP protocol to clock-out new data packet. The spikes do not reach the maximum window size, which indicate that the connection recovers (by a fast retransmit) before coming to a halt. An example of a packet loss and the resulting fast retransmit with SACK packet is described and illustrated in further detail in Section 3.7 on page 29.

We can calculate the maximum allowed RTT needed to achieve a certain throughput with a given window size, by changing Formula 3.1:

Formula 3.2

$$RTT = \frac{WindowSize}{Throughput}$$

Hence, to achieve full throughput on our 2 Mbit downstream line, the RTT has to be below 254 ms ($63712bytes/2000kbits = 254ms$). During the downstream-only transfer the RTT was well below that, thus we assume that the achieved throughput of 1800 kbit/s is the lines maximum capacity. This corresponds well to the overhead imposed by Asynchronous Transfer Mode (ATM) headers, which reduces the 2 Mbit/s to 1811 kbit/s ($2000kbit/53bytes \cdot 48bytes$ [4]). The ATM overhead is described in detail in Chapter 5.

To calculate the needed window size to achieve full utilization at a given RTT, the formula is transformed one last time:

Formula 3.3

$$WindowSize = Throughput \cdot RTT$$

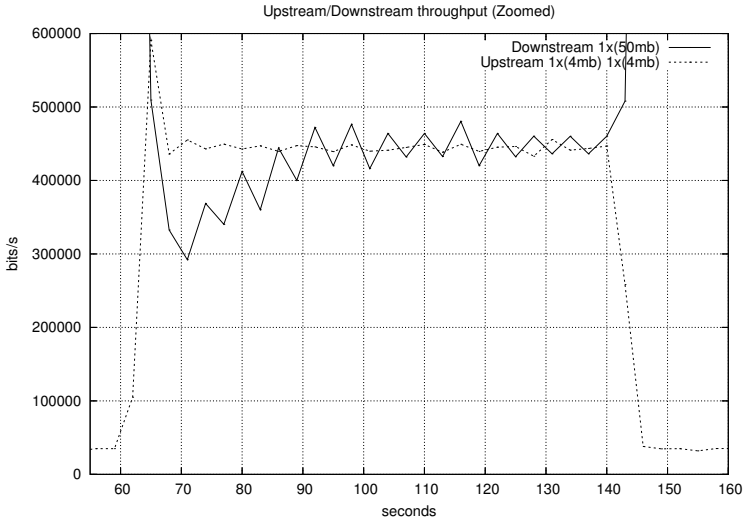
This expresses the bandwidth-delay product, which indicates the amount of unacknowledged bytes in transit in order to keep the line fully utilized, thus the needed window size. To achieve full utilization of the 2 Mbit/s or rather 1800kbit/s line, with a RTT of 1150 ms, a window size of 258750 bytes is needed ($1800kbit/1150ms = 258750bytes$). Increasing the window size might not be the correct action, as increasing the window size is likely to increase the latency according to Formula 3.2. Instead we will take a closer look at why we experience the high latency.

3.4 Queueing Delay

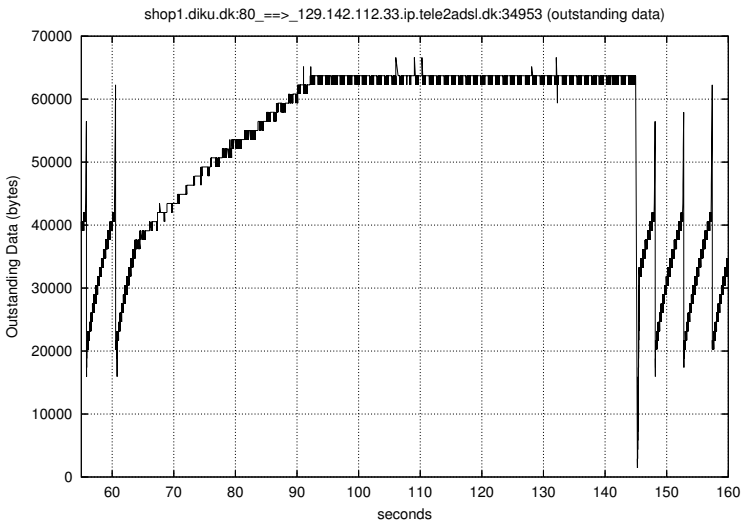
A RTT delay of 1.2 seconds indicates that something is wrong and throwing more resources after the problem in the form a larger window size might not be the answer. Comparing the delay components with the characteristics of the specific line should give us an insight into whether increasing the window size is a good solution.

The following is a quick introduction to the different components of the delay. Formula 3.4 shows the total delay experienced by a single packet travelling on a given line through one router to the next. The individual components will be discussed below, for a detailed explanation see [53].

[4] The 53 bytes is the size of an ATM cell, and the 48 bytes is the ATM payload size.



(a) Throughput



(b) Outstanding data / TCP congestion window

Figure 3.5: Illustrating the window size influence on throughput. Zoomed in on Figure 3.2 on page 20.

Formula 3.4

$$d_{total} = d_{transmit} + d_{queue} + d_{process} + d_{propagation}$$

Using the specific 2 Mbit/512 kbit ADSL from Tele2 as the target, it is possible to determine some of the delay components. As the reader might have noticed (from the throughput graphs) the full upstream throughput of 512 kbit/s is not achieved at the (measured) IP level. This is due to different types of overhead (ATM), which will be described in greater detail in Chapter 5 on page 48. The median (and the most frequent number) of the measured upstream throughput is 454 kbits/s from the test illustrated in Figure 3.6.

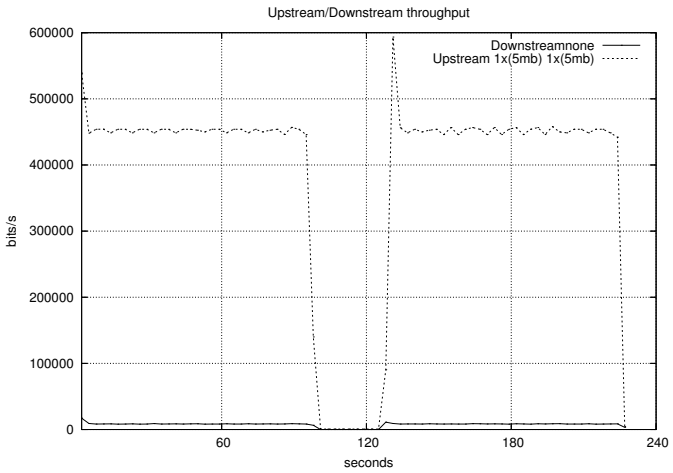


Figure 3.6: Effective/measured TCP/IP throughput on the 2 Mbit/s/512 kbit/s ADSL line from Tele2. Only upstream connections; two periods with one upstream connection each. A median of 454 kbit/s.

The transmission delay $d_{transmit}$ is the time required to push or transmit all of the bits onto the line, thus the packet size divided by the transmission rate. The transmission time of a maximum sized Ethernet frame of 1500 bytes with a (measured) line speed of 454 kbit/s corresponds to a $d_{transmit}$ of 26.43 ms ($1500bytes/454kbit/s$)[5].

The propagation delay $d_{propagation}$ is the time required for the electric signal (or a single bit) to propagate from the beginning of the link to the end. This is dependent on the physical medium, which in this case is copper wire. It is generally in the range 2 - $3 \cdot 10^8 m/s$, which is a little less than or equal to the speed of light [53]. To achieve 2 Mbit/s with the ADSL technology the maximum cable length is around 5 km [40], which gives an upper bound on $d_{propagation}$ of 0.025 ms ($5km/2 \cdot 10^8 m/s$).

The processing delay $d_{process}$ introduced by the ADSL modem is dependent on the line coding technique and especially the interleaving depth of the error correction scheme, which can introduce a delay in the order of 60 ms (usually configured lower). With the interleaver turned off, the residual latency of standard ADSL is 2 ms [4, 5, 33]. We have measured $d_{process}$ to between 8 ms to 9 ms with a ping test where we have accounted for the transmission delay (although on a different ADSL connection than used in this chapter). The test is shown in Appendix A.1.

[5]Can also be calculated taking the ATM overhead into account which is $1696bytes/512kbit/s$ = 26.5 ms

It should be clear that the above delay components ($d_{transmit}+d_{process}+d_{propagation}$) do not sum up to a delay of 1.2 second. Thus the main delay contributor is the queueing delay d_{queue}. The outstanding data of a TCP connection is limited by the window size, which in this case also determines our queue usage. With a queue usage of 63712 bytes (our window size) and a packet size of 1500 bytes, there are 42.5 packets in the queue. A new packet arriving has to wait for (at least) 42 packets to be transmitted $d_{transmit}$ (plus itself). This imposes a delay of 1136 ms ($43 \cdot 26.43ms$), which corresponds perfectly with the measured average delay of 1136 ms in the test (in Figure 3.6).

Thus it is unwise to increase the window size, as the queue is likely to increase the delay further. Reducing the TCP window size might be preferable as we will illustrate in the next section.

3.5 Queue Size

In this section we show that the delay is reduced when lowering the window size and that the window size affects the queue size directly. We will also show that adding more TCP connections has the same effect as increasing the window size, which affects the queue usage linearly, until it is limited by the queue size of the (upstream) router.

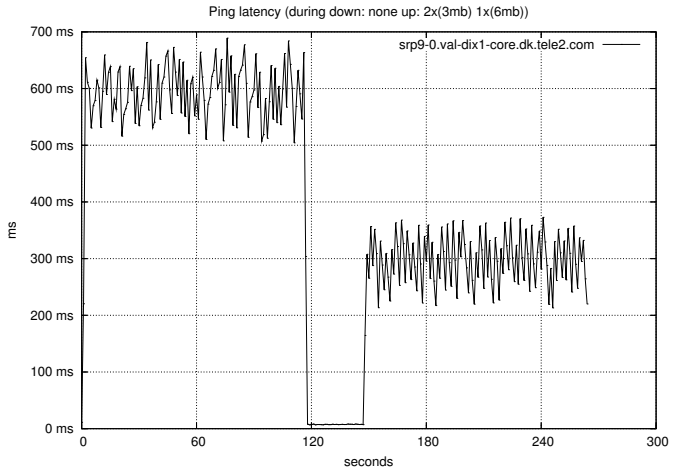


Figure 3.7: Ping RTT with a window size of 18750 bytes. Only upstream connections; First 2x 3 Mbyte and then 1x 6 Mbyte.

To demonstrate the effects of reducing the window size, the window size is lowered on our test host and a simple upstream test is performed, see Figure 3.7. The test consists of two periods of upstream traffic, in the first period there is two upstream transfers and in the second period only one upstream transfer.

It is sufficient only to lower the window size on our test host, as the lowest window size is determined by either the adverticed window by the reciever or the window size used by sender [72][6]. When lowering the window size on our local test machine, the upstream window is limited by the machines sending window size and the downstream window size is limited by the machines adverticed window.

[6]There is an excellent explanation and illustration of the TCP sliding window protocol in [72] Section 20.3 on page 280.

Under Linux, the TCP window size is changed by reducing the allowed buffer space for a single TCP socket (through the proc filesystem[7]). The default setting is to use 3/4 of the TCP socket size for the TCP window [16, 70]. In the tests the socket size is set to 25000 bytes, which results in a window size of 18750 bytes. For the test illustrated in Figure 3.7, the upstream throughput was approximately 450 kbit/s and the maximum outstanding data window was measured to 20273 bytes, which is approximately 18750 plus (one full MTU packet) 1500 bytes (20250 bytes). This shows that we have successfully decreased the window size by only changing the window size of the test host.

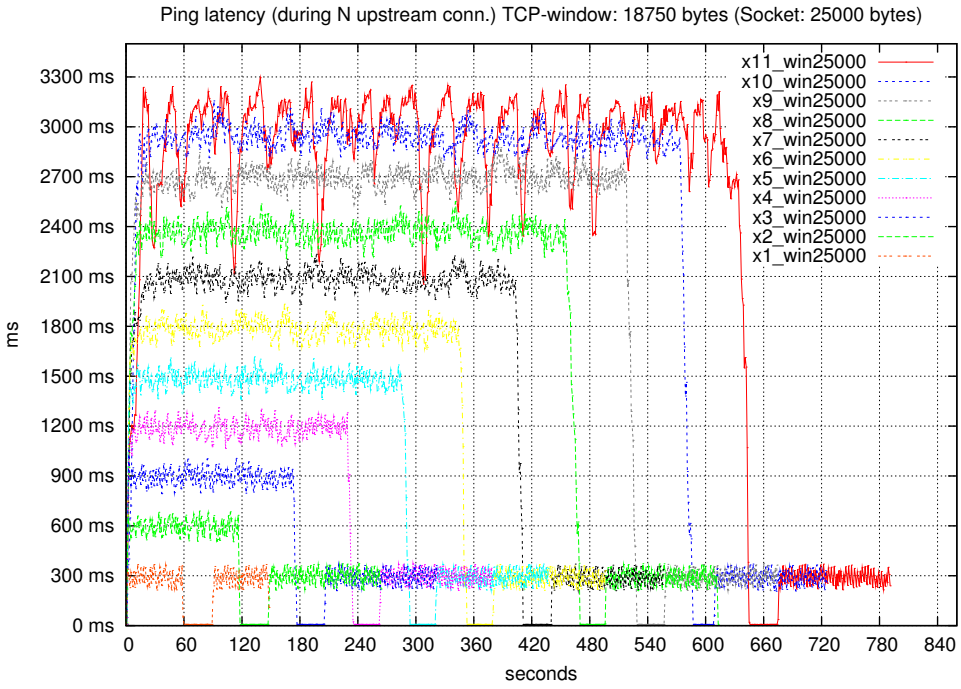
We will explain the second upstream period of Figure 3.7 first, as it only contains a single upstream transfer. First we estimate the expected delay from the window sizes and packet bursts observed, and then relate this to the measured latency data. The average outstanding window was 16583 bytes, giving an average delay of 294 ms ($16583 bytes / 450 kbit/s$). We expect some variation as a closer look at the packet flow reveals that packets are sent in bursts of 3-5 packets. Thus as a result we expect the queue length to vary between 12773 and 20273 bytes, which corresponds to queueing delays between 227 ms and 360 ms. The graph shows an average of 300 ms, and a delay variation between the lowest 213 ms and the highest 372 ms. This corresponds well with the expected results from the measured window sizes.

The first upstream period of Figure 3.7, illustrate how two concurrent TCP connections have the same effect as doubling the window size. With the double window size we expect the double latency as the queue size is expected to increase accordingly. This is verified in Figure 3.7, where first upstream period, with two upstream TCP bulk transfers, resulted in an average latency of 600 ms and the second period, with only one TCP bulk transfer, resulted in an average of 300 ms. Thus verifying our expectations of the double window size affecting the queue usage linearly.

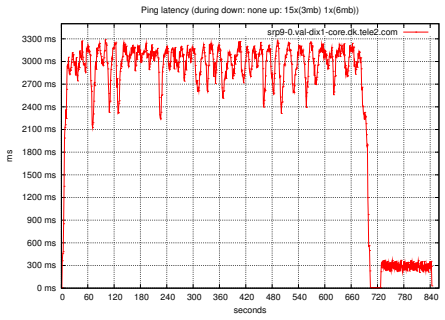
To determine the maximum upstream buffer size, we perform a number of tests where more and more upstream connections are added. Each connection has a window size of 18750 bytes, like the above test. Figure 3.8(a) on the next page illustrates the test results plotted on the same graph with upstream connections from one to eleven. At 11 concurrent upstream connections the maximum upstream buffer size is reached. Figure 3.8(b) with 15 connections and Figure 3.8(c) with 19 connections verify that the maximum RTT is reached. Thus, we estimate the upstream router buffer size to be approximately 11 times 18750 bytes which is around 206 kbytes.

This large upstream buffer is too high compared to the upstream capacity, as it introduces a latency up till 3300 ms. Therefore, we recommend the ISP to lower the upstream buffer on the ADSL modem, to avoid and limit this high delay. We also recommend the user to lower the machines (upstream) TCP window size, as we have demonstrated that the default window size introduce a delay of around 1200 ms, and that lowering the window size also lowers the delay without reducing the throughput.

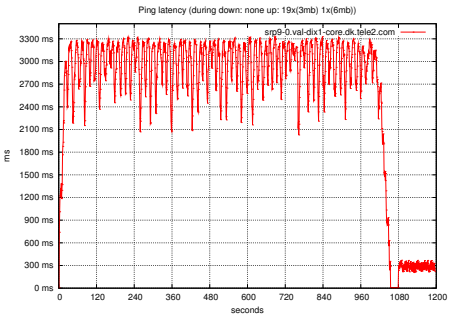
[7] `/proc/sys/net/ipv4/tcp_rmem` and `/proc/sys/net/ipv4/tcp_wmem`.



(a) Latency: N upstream connections



(b) 15 upstream connections

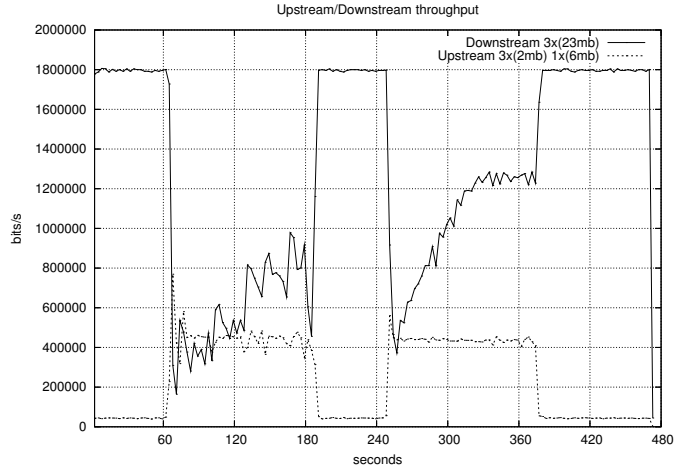


(c) 19 upstream connections

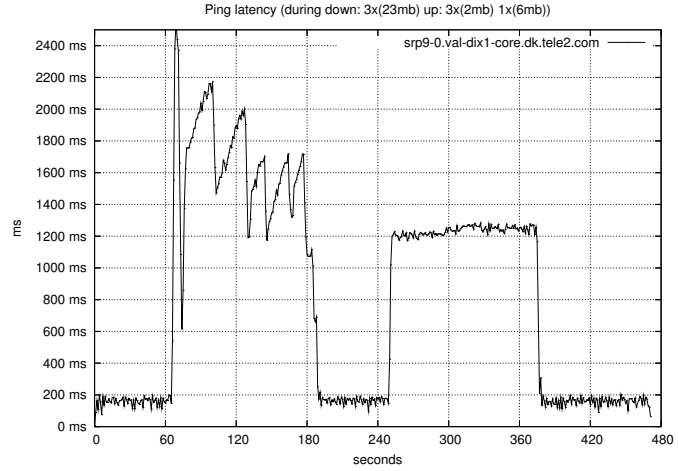
Figure 3.8: Ping RTT with increasing number of upstream connections. Each connection have a window size of 18750 bytes. (2 Mbit/512 kbit ADSL line from Tele2).

3.6 Several TCP Flows

In this section we show what happens with bidirectional traffic and several TCP connections. Opening several TCP connections gives a more realistic workload and thus a better indication of the effects in practice. We have shown above that having several concurrent TCP connections has the same effect as multiplying the window size by the number of connections. We found (in Section 3.3) that a larger window size was needed to compensate for the large latency. Thus using several downstream TCP connections, the total throughput should increase. We also found that the queue length/delay increased together with the total window size (in the data direction).



(a) Throughput



(b) Latency

Figure 3.9: Both throughput and latency during: 3 continuous downstream TCP connections, 3 upstream TCP connections in the first upstream period and only 1 upstream TCP connection in the last upstream period.

Using this knowledge we construct a test with 3 continuous downstream TCP connections; during the first upstream period, 3 upstream TCP connections are introduced; during the second upstream period only a single upstream TCP connection is running. The results are shown in Figure 3.9 on the preceding page.

During the second upstream period (with only one upstream connection) the expected result is achieved. The latency (Figure 3.9(b)) is stable around 1250 ms. Using 3 downstream TCP connections with a window of 63712 bytes equals a total window size of 191136 bytes. Using formula 3.1 on page 21 we would expect a throughput of 1223 kbit/s ($(3 \cdot 63712) bytes / 1.250 s$). The throughput climbs to 1260 kbit/s, the slow increase is expected due to the increase of the congestion window (an example is given in Figure 3.5 on page 23).

During the first upstream period with 3 upstream connections with the default window size of 63712 bytes we would expect a latency of 3397 ms ($3 \cdot 63712 bytes / 450 kbit$). The figure shows that the latency is not very stable, which is due to the flows competing for the limited upstream resource, causing packet drops and TCP back-offs. Even though the latency does not increase as expected, the total downstream throughput is still reduced significantly. This tells us that competing traffic flows disturb each other and more realistic traffic flows are thus harder to predict precisely.

We have illustrated and shown that we cannot achieve full downstream throughput, due to queueing delays on the upstream connection. In the next section we will take a closer look at what actually happens when queueing occurs on the upstream connection and how TCP reacts to this.

3.7 Bursty Traffic and ACK-compression

In this section we will take a closer look at the downstream transfer (on our Tele2 2 Mbit/512 kbit ADSL) to verify that ACK-compression (described in Section 2.4 on page 14) occurs and to illustrate the resulting bursty traffic patterns. We will also show that increasing the window size is not a good solution to increase throughput as it introduces bigger bursts resulting in packet loss.

The TCP sender is connected to a 100 Mbit/s network[8] and can saturate the 2 Mbit/s downstream capacity of the ADSL with ease. The ACK clocking mechanism ensures that the TCP sender does not send data too fast (as described in Section 2.3.1 on page 10), but with bidirectional traffic flows we have shown that queues start to build — thus we expect the ACK clocking to break down (as the inter ACK spacing is removed due to queuing). The packet dumps used/analyzed in this section are captured on the TCP (data) sender.

Figure 3.10 is a TCP time sequence graph (as are most of the graphs in this section), which plots the sequence number in a packet versus the time it was transmitted. For a steady progress in data (without retransmissions) the points should move up and to the right, where *the slope of the points represents transfer rate (throughput).* In addition to plotting the sequence numbers of the data segments and ACK packets, the retransmitted data segments are labeled with an *R*. The data segments and ACK packets look like they are one line, but that is only because of the time scale and resolution (other graphs will illustrate the difference with a zoom).

Figure 3.10 is based on the test in Figure 3.2 on page 20 (on our Tele2 2 Mbit/512 kbit ADSL), with a single continuous downstream transfer (of 52.5 MB) and one upstream connection during each upstream period. The graph shows only the downstream TCP

[8] and probably having 100 Mbit/s as the smallest link all the way to the DSLAM.

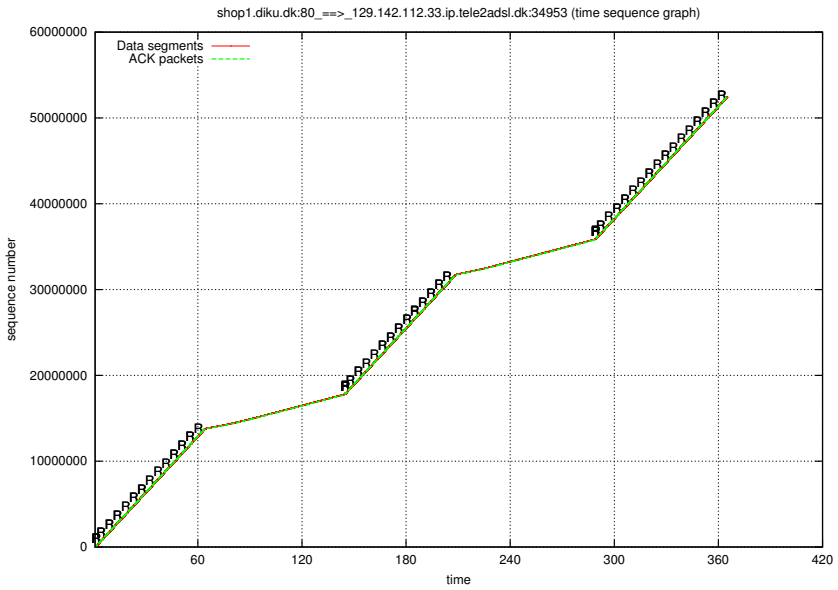


Figure 3.10: Time Sequence Graph: Illustrating TCP retransmissions. Based on the test in Figure 3.2 on page 20. Default TCP window size (max of 63712 bytes observed).

flow (the impact of the two upstream transfers should be easy to spot).

During the downstream-only parts the slope indicates a throughput of around 1800 kbit/s (see Figure 3.2 on page 20 for reference). There is also a clear indication of retransmitted data packets *R*, which actually is a normal TCP behavior as TCP will increase its congestion window until it experiences a packet loss, as demonstrated earlier in Figure 3.5.

Taking a close look at one of the retransmits in Figure 3.11 on the next page we see that a fast retransmit is triggered and the use of Selective ACKnowledgement (SACK) packets helps us clock out more data packets (instead of coming to a halt) although at a slower rate. With this zoom it is possible to see each individual data segment (the red squares) and the precise retransmitted segments (the two red stars). The *S* labels represent SACKs and purple line shows how much the SACK packet acknowledges. The figure also illustrate that during the fast retransmit phase each SACK packet only "releases" a single data packet, implying a delayed ACK factor of 1.

ACK-compression

Figure 3.12 on the following page illustrates how ACK-compression results in bursty behavior of the TCP sender. The figure is a zoom of Figure 3.10, where it is possible to see how data segments and ACK packets interact. The red lines (connected dots) represent data segments and the green line represents ACK packets for the data segments. The figure shows the transition between unidirectional and bidirectional traffic, in the form of a upstream TCP transfer being started (while a downstream TCP transfer is already running).

By showing the transition phase, it is possible to show both (1) the optimal use the ADSL capacity achieved through a perfect ACK clocking (the steady linear slope at the beginning), and (2) the result of infrequent and bursty ACK feedback. The bursty ACK feedback is a clear indication of ACK-compression as it shows that ACK packets

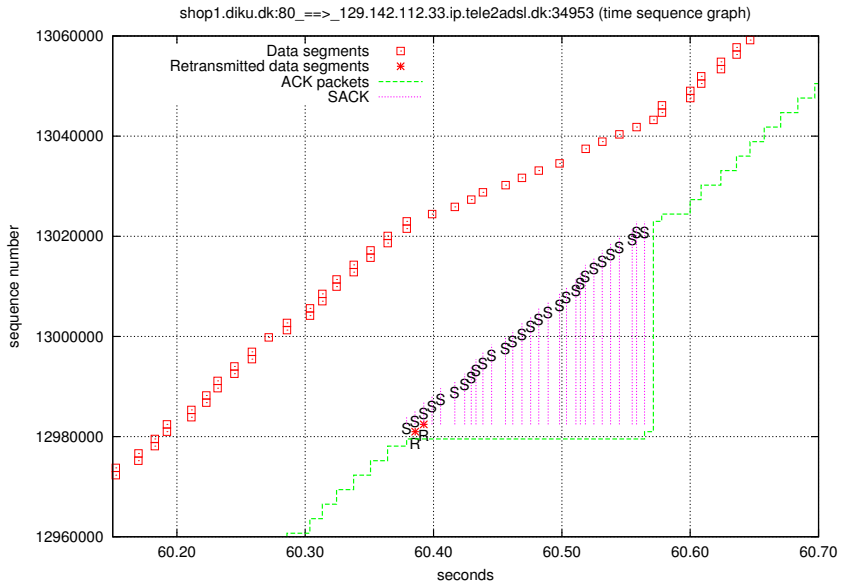


Figure 3.11: Time Sequence Graph. Zoom of a retransmission and selective ACKs. (Zoom of Figure 3.10 on the preceding page).

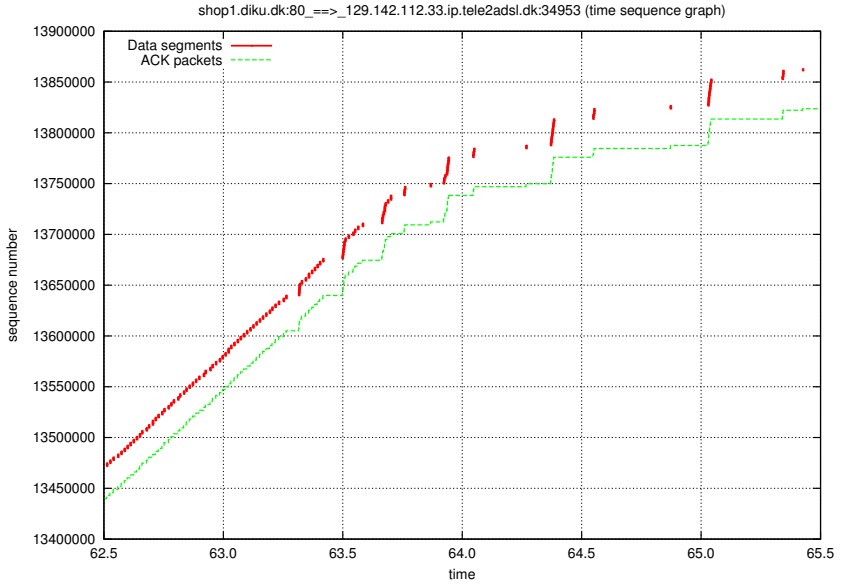


Figure 3.12: Time Sequence Graph: Illustrating bursts and ACK-compression. (Zoom of Figure 3.10 on the page before).

have lost their inter-ACK spacing (through queuing). Data is being sent too fast (for the downstream ADSL capacity to handle) if the sending rate gets steeper than the slope of the steady state (1). It is clear from the graph that data (after the upstream transfer is started) is being sent too fast in bursts. As can be seen from Figure 3.10 on page 30 this did not result in packet drops, because the downstream buffer is sufficient to handle these bursts.

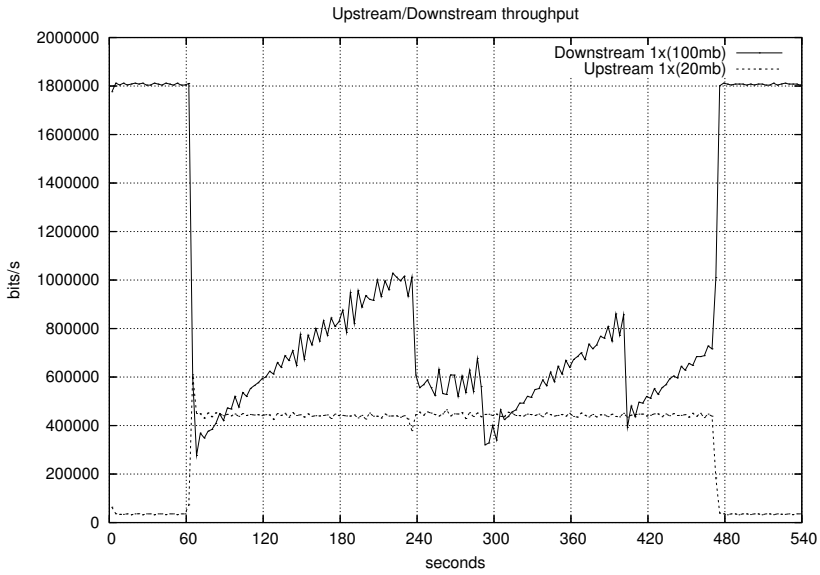
Large TCP window

In Section 3.3 we found that the window size needed to be larger to compensate for the large latency (and used the rest of the sections to argue that it probably was not a viable solution). Let us take a look at what really happens when using a very large window size.

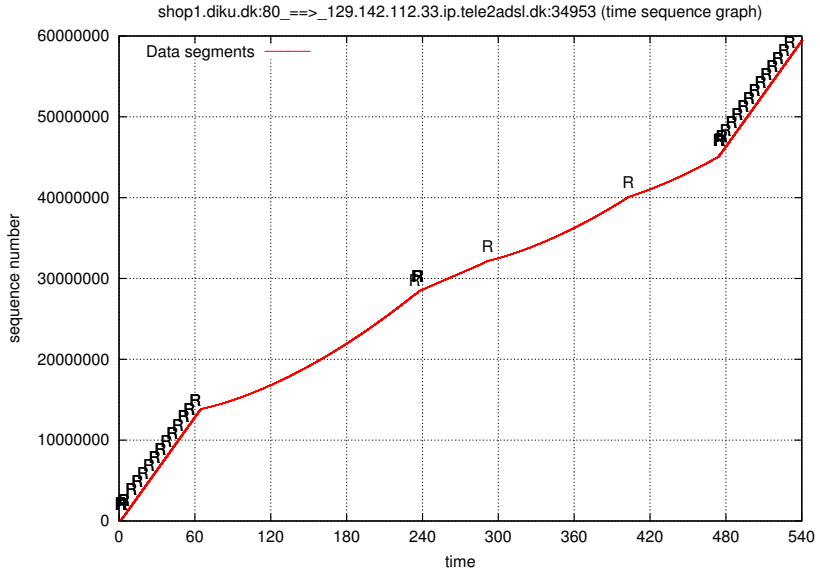
To increase the TCP window size both parties have to increase the memory allocated to the TCP socket/window buffers. The window size is changed to 750 kbytes both upstream and downstream. With full usage of the upstream router buffer we expect a delay of around 3 seconds. The bandwidth-delay product of an 2 Mbit/s line with 3 seconds is 750 kbytes ($3s \cdot 2Mbit/s$).

Using only a single upstream and downstream TCP connection we illustrate how TCP reacts in Figure 3.13. The downstream TCP connection is not able to utilize the full downstream capacity. The throughput increases in Figure 3.13(a) as expected until around 240 seconds where a sudden drop in throughput is experienced. From the time sequence graph Figure 3.13(b) there is a clear indication of packet drops. The packet drops are assumed to be caused by bursty traffic as the downstream line is not fully utilized at the given time (1 Mbit/s). The downstream connection continues to experience packet drops due to its bursty nature and is unable to achieve the expected throughput. Packet drops during the downstream only part occurs more frequently, but is not as severe, because they are handled by a fast retransmit, whereas the packet drops during the period with high latency is not able to recover by a fast retransmit.

This shows us that it is not a viable solution to increase the TCP window size to compensate for the large queueing delay of ACK packet on the upstream link.



(a) Throughput



(b) Time Sequence Graph

Figure 3.13: Test with a large TCP window size of approximately 750 kbytes, on a clean 2Mbit/512kbit ADSL line from Tele2. A single continuous downstream transfer and a single upstream transfer. (The graph has been cut off of a 540 seconds)

3.8 Summary

In this chapter we have documented and verified that ADSL is indeed affected by its asymmetric nature. The tests have been performed on real physical ADSL lines.

We have shown that full utilization of the downstream capacity is not possible when utilizing the upstream link. The downstream throughput is crippled, already, in case of a single upstream (data) transfer. On a 2 Mbit/512 kbit ADSL the downstream throughput was reduced from 1800 kbit/s to around 450 kbit/s, and on an 8 Mbit/512 kbit ADSL the throughput is reduced from 7200 kbit/s to around 300 kbit/s.

We have analyzed, in detail, how a physical 2 Mbit/512 kbit ADSL line reacts to different traffic patterns. An interesting and alarming result is how easily a single TCP connection can introduce a very high latency. A single upstream TCP connection introduced a RTT latency of around 1200 ms. The increased latency is caused by a (large) queue in the upstream router/modem. We show a direct correlation between the TCP window size and the queue size (which increased latency). We show that lowering the TCP window size is beneficial, as latency is lowered, and that the achievable TCP throughput is not lowered.

We have observed upstream delays up till around 3300 ms. This maximum latency were caused or limited by the upstream buffers at the ADSL modem. Based on this, we recommend the ISP to lower the upstream buffer on the ADSL modem, to avoid and limit this high delay. We would also recommend the user to lower the machines (upstream) TCP window size, as we have demonstrated that the default window size is too large compared to the upstream capacity.

We document the existence of the ACK-compression phenomenon[81] on ADSL with the default window size and only a single transfer in each direction. It is worth noticing that the resulting (data) bursts did not result in drops as the line was under-utilized and the downstream buffer could handle the bursts. We also demonstrate that bursty traffic can lead to packet drops, even on a under-utilized line. This is shown through increasing the TCP window significantly, which in addition to an increased latency, resulted in a bursty traffic pattern, which again resulted in (data) packet drops.

Our experiments also show, that when reaching the limits of the queue size on the router, several competing TCP flows start to disturb each other. Thus, we expect that more realistic traffic flows will get harder to predict precisely. We still recommend lowering the upstream buffer size as it determines the maximum delay and that TCP is build to adapt in case of competing flows and limited buffers.

As a side effect of our detailed analysis we also show how, during a uni-directional downstream transfer, that Selective ACKnowledgement (SACK) packets helps us to sustain a high throughput when data packets were dropped. The data packet drops are a natural TCP behavior and was handled by a fast retransmit. The interesting part was to see how SACK packets allowed the TCP connection to transmit new data while the retransmitted packet was in transit.

We have documented that ADSL performance is affected negatively by traffic on the upstream link. Beside under-utilization of the downstream link, the high latency observed is clearly unacceptable for our goal of low latency for delay-sensitive applications. We found that the main cause of increased latency to be large queues forming in the upstream router/modem. Thus we need to control this upstream queue in order to provide low latency for interactive applications.

Part II

Middlebox Considerations and Components

Chapter 4

Designing a Packet Scheduling Middlebox

Our goal, as stated in Chapter 1, is to create a practical solution that optimizes an ADSL connection shared by a busy autonomous network with respect to both interactive comfort and maximum link utilization.

As a step in achieving this goal, Part I, represented by Chapter 2 and 3, contained the preliminary analysis. Chapter 2 analyzed asymmetric technologies and their effects on TCP. Chapter 3 explored and documented the extent of the practical problems with a real ADSL connection.

In this part, Part II, we will describe the considerations and components for designing a middlebox. This chapter focuses on identifying the components needed to achieve our overall goal. The following chapters describe some of the identified components in further detail. The evaluation of the components are performed as an ongoing process in the individual chapters. Whether a combination of the components solve the overall goal in our real-world setup is described in Part III.

The constraints for our middlebox is described in Section 1.4. We choose to be independent of the ADSL Internet Service Provider (ISP), thus changing the equipment at the ISP is not an option for our solution. Our focus is put on finding a practical and deployable solution from the ADSL subscribers' point of view.

4.1 Service Differentiation

Part of our goal is to be able to use the ADSL connection for different types of services at the same time. Thus, we will look at *service differentiation*.

Our goal of optimizing latency and throughput at the same time conflicts with the current service model of the Internet. The Internet is based upon a datagram model where all packets are treated alike, resulting in a single level of service, often called *best effort*. The goal is conflicting as we cannot provide low latency and high throughput at the same time with a single level of service. Low latency requires that packets are not delayed in any queues on the path, while high throughput is best achieved by keeping the packet queues loaded to fill every time slot.

A new service model is necessary, in order to use the link for different types of services at the same time. To determine the required resource sharing model, we

have to define the needed *service classes*, in the QoS literature referred to as *service differentiation*[78]. The service classes needed for different network applications are described later in Section 4.4.

Another task in service differentiation is *traffic classification*, the task of classifying traffic into (the correct) service classes. In the autonomous network, correct classification might not be straightforward. Users might try to evade classification because they compete for the available bandwidth. Due to P2P traffic, this unfortunately turns out to be the norm rather than the exception. P2P file sharing systems have recently developed a lot of evasive techniques in an attempt to avoid firewalling, and thus also classification. The challenges of traffic classification are discussed in Section 4.6.

4.2 QoS Architecture

With our middlebox solution we have chosen to be independent of other network elements, so changing the network to support a QoS architecture is out of reach. Instead we will look at ideas and techniques from Differentiated and Integrated Services QoS architectures that can be used in our ADSL middlebox environment.

To find a resource sharing model, suitable for our needs, we need to determine how we can perform *resource assurance and allocation*. *Resource assurance* defines a required service level. *Resource allocation* performs the task of sharing or allocating the available link resources in accordance with the assured service levels. The ability to provide *service differentiation* and *resource assurance* is often referred to as Quality of Service (QoS).

Integrated Services (IS)[25] and Differentiated Services (DS)[23] are two resource allocation architectures designed for the Internet, which represent two different approaches. Resource assurance in IS is based on *per-flow resource reservation*. Before an application can transmit data into the network it must make a resource reservation, which involves several steps. Per-flow reservation makes sense for long lasting connections, but it is not appropriate for short lived connections like HTTP traffic. The advantage is the ability to provide deterministic worst-case delay bound. This is done through strict admission control and fair queue scheduling algorithms. A scalability issue arises as every node in the network needs to support IS and implement per-flow classification and scheduling. This might not scale with a very large number of flows at high speeds.

DS is a simpler and more scalable approach to offer a *better than best effort* service. The approach is to use a combination of edge policing, provisioning and traffic prioritization to achieve resource assurance. Traffic is divided into a small number of forwarding *classes* and resources are allocated on a per-class basis. To adjust the resource provisioning, the amount of traffic allowed into the network (DS domain) is limited at the edge of the network for each class. The class (DS) marking is performed at the edge node and is encoded directly into the packet header. Thus, the complex process of classification is limited to the edge node. Interior nodes can use the DS marking to differentiate the treatment of the packets. Resource assurance in DS is based upon provisioning. This makes providing deterministic guarantees difficult compared to reservation method of the IS architecture.

In the context of our environment we are closer related to the DS than the IS architecture. Our middlebox resembles a DS edge node, that performs classification of every packet as no external DS marking is performed. We are most likely to deploy resource allocation and assurance through aggregating traffic into a number of service *classes* (like DS), because the individual flows do not specify their resource requirements

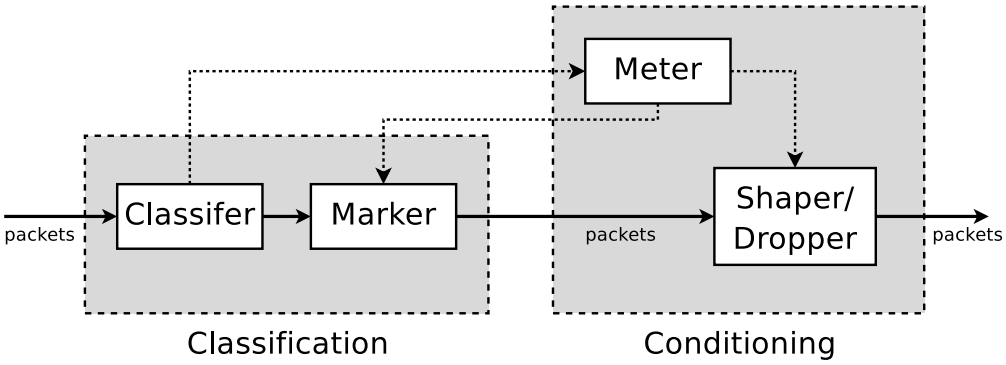


Figure 4.1: Logical view of Differentiated Services: Classification and conditioning.

like IS, thus, there are no parameters to base our resource assurance upon. Thus, we will primarily base our solution on the DS architecture, but we will (most likely) also use packet scheduling techniques developed for the IS architecture. The logical view of the DS architecture, in Figure 4.1, will be used as a reference base for our solution. Once the packets have been classified into a certain service class we need to perform some allocation or sharing of the link resources, this is referred to as *link sharing*. In DS terminology this is, under a broader term, called traffic *conditioning* [23] which is illustrated in Figure 4.1. As a component of the link sharing, we have the *packet scheduler*, which performs the central task of selecting a packet to transmit when the outgoing link is ready. The link sharing mechanism or packet scheduler also performs the task of shaping, through delaying or dropping packets to make a traffic flow conform to the configured traffic profile. Mechanisms for link sharing and packet scheduling is described in Section 4.7.

4.3 Queue Control and Link Layer Overhead

Implementing link sharing using a middlebox requires *queue control*. Our middlebox solution is independent and thus has no cooperation with other network nodes. Thus, we are left with packet scheduling and traffic conditioning, while the packets reside in our local packet queues. In other words, we must ensure that any packet queue builds up in our local queue. If packet queues build up elsewhere on the path, the link sharing mechanism will not have any effect, as queues elsewhere will determine the queueing delay and drop property of packets (as we saw in Chapter 3).

In order to achieve queue control we need to emulate the "bottleneck" link, as packet queues form naturally in front of the bottleneck link. This can be achieved by controlling the sending rate and adjusting it to slightly less than the rate of the smallest link. The smallest link on the path might not use the same type of link layer as our middlebox. Thus, when scheduling packets, we need to account for the specific link layer overhead (of the smallest link) in order to match the exact rate. We assume that the ADSL connection is our bottleneck link. In the case of ADSL, the link layer overhead varies (a lot) due to protocol overhead and packet aligning at different layers. This complicates the task of matching the exact rate at a different link layer. The ADSL link layer overhead is described in further detail in Chapter 5 and Chapter 6, where we show a worst-case overhead of 62% (for small TCP packets).

Another issue when achieving queue control is our placement in the network, as

it influences the effect of the link sharing mechanisms. Due to the placement of our middlebox, it is only possible to achieve direct queue control on the upstream link. Our middlebox is positioned before (packets are transmitted onto) the upstream link and after (packets have traveled) the downstream link. That is, between the ADSL modem and the local network. This allows us to intercept upstream packets *before* they travel the upstream link, thus we have the ability to control the packets transmission rate directly by simply delaying packets. We cannot control the downstream transmission rate directly, because downstream packets reach our middlebox *after* they have travelled the downstream link. It is possible to control the downstream traffic indirectly by delaying or dropping packets and relying on protocols to have some kind of flow control, like TCP, but doing this in a controlled and smooth manner is a project of its own.

Although it is a problem that we cannot achieve direct downstream queue control, we believe it to be more important to achieve upstream queue control. The upstream link is the limited resource compared to the downstream capacity in the case of ADSL. In Chapter 3 we demonstrated that the queueing delay was significantly larger on the upstream that on the downstream link. Latency caused by downstream traffic was below 200 ms while latency resulting from upstream traffic was measured to a maximum of approximately 3300 ms.

Thus, we choose to focus on upstream queue control. How we achieve upstream queue control, using our middlebox, is described and evaluated in detail in chapter 6.

4.4 Service Classes

We need to determine some classes or types of service, when wanting to offer service differentiation to different network applications. Thus, we need to determine the requirements of the different kinds of applications.

Differentiated Services (DS) is vague in its definitions of service types and requirements. It defines a set of different marking classes or marking schemes, but service and forwarding treatment (DS terminology) is defined as a site policy in form of a Service Level Agreement (SLA) and a Traffic Conditioning Agreement (TCA)[1]. The SLA is normally defined between a customer and the ISP and specifies the services to be provided. The TCA is part of the SLA and specifies the practical details of the service parameters for traffic profiles and policing actions. This may include; rate parameters for each class, actions for non-conforming packets, etc.

With Integrated Services (IS) the applications themselves have knowledge of the service requirements, as they explicitly request a specific service level.

We look at the telestandard Y.1541[7] – "Network performance objectives for IP-based services" for some more specific guide-lines for different types of services and their requirements. Y.1541[7] defines 6 IP service classes (numbered from 0 to 5) shown in table 4.1.

The delays specified are one-way end-to-end[2] delays. The *delay* is the average delay and the *delay variation* or delay jitter is defined as the maximum delay minus the minimum delay. Where the maximum delay is defined as the upper bound on the 99.9th percentile of the delay. This implies that we should perform at least 1000

[1] The terms SLA and TCA, have been revised in RFC3260[42] to be called respectably Service Level Specification (SLS) and Traffic Conditioning Specification (TCS).

[2] In ITU-T terminology UNI-to-UNI, that is to the User Network Interface (UNI) or sometimes mouth-to-ear.

Service Class	Node mechanisms	Application (examples)	Delay	Delay variation
0	Separate queue with preferential servicing, traffic grooming	Real-time, high interaction, jitter sensitive (VoIP)	100 ms	50 ms
1	(see above)	Real-time, interactive, jitter sensitive	400 ms	50 ms
2	Separate queue	Transaction data, highly interactive	100 ms	U
3	Separate queue	Transaction data, interactive	400 ms	U
4	Long queue	Low loss only (bulk data, short transactions, video streaming)	1 sec	U
5	Separate queue (lowest priority)	Traditional applications of default IP networks	U	U

Table 4.1: Based on Y.1541[7] – Guidance for IP service classes ("U" means Unspecified)

samples when evaluating these delay bounds. The delays in the table should be viewed as upper bounds and the network provider should in general try to offer shorter delays if possible.

The classes can be divided into two types of traffic: variation-sensitive (class 0 and 1) and non-variation-sensitive traffic (class 2 to 5). The variation-sensitive applications are in general real-time applications, like Voice over IP (VoIP) or Video conferencing. In the following we will describe and exemplify the individual classes:

Class 0 is ment for highly interactive applications that are sensitive to delay jitter. Applications in this class are assumed to have some real-time requirements. An example could be telephone conversations like Voice over IP (VoIP), as it is very sensitive to both delay and jitter. The telestandard G.114[12] – "One-way transmission time", talks about the effect on speech quality and specifies that the (one-way) delay should be kept below 150 ms, but it also notes that highly interactive tasks may be affected by delays below 100 ms [3]. That latency for a telephone conversation should be kept below 100 ms, is also noted by [28] that states that the normal unconscious etiquette of human conversation breaks down if latency exceeds 100 ms.

There are other applications that have real-time demands and requires low latency, but do not have a need for a continuous stream of low latency packets, like voice and video. An example is (highly) interactive multi-player games. It is of course dependent on how interactive the game is e.g., turn based games do not have these requirements (and could be put in class 3). When the player does not interact directly with other players, the latency requirements can be relaxed, but when two players "meet" then the latency can play a crucial role in who "wins".

Class 1 allows a larger delay but still requires a small delay jitter, an example of this could be one-way video streaming with little interaction, as a video frame arriving too late has exceeded its deadline and must be discarded (real-time requirement). There exists a lot of latency hiding techniques, which can be used if the delay bound cannot be met. In case of video streaming with no or little interaction

[3] G.114[12] shows a users satisfaction to delay graph based upon the E-model from G.107[10] and G.109[11] which shows a breaking point at 200 ms, between user being "very satisfied" and being "satisfied".

a simple buffer could be used, another trick is to keep displaying the old frame if the next exceeded its deadline. Video streaming with no interaction could be assigned to class 4 by applying a large enough buffer. We also assign ACK packets to class 1. How ACK packets fit into this class is explained in Section 4.5.

Class 2 is for highly interactive applications, it has the same delay bound as class 0 but the jitter constraint is removed. An example could be a remote command prompt (like SSH) that requires low latency for keystrokes but delay jitter is not critical for the service, especially command results are not very delay sensitive.

It might be wise to use class 2 for establishment of TCP connections. Establishing a TCP connection involves a three-way handshake, which involves three packets, thus three times the one-way delay. This proposal is based on the idea that the delay bounds should apply to the data transfers and new connections should not be penalized by the TCP handshake.

Class 3 is defined as interactive but not highly interactive. We view web-browsing (HTTP traffic) as interactive, but not highly interactive. Web-browsing is also a good example of an application, which could take advantage of TCP handshakes being put in class 2, as it involves a lot of short lived TCP connections.

Chat applications, which are line based, can be placed in class 3, they are interactive but not highly interactive. A person receives a chat message, needs time to read, understand and type an answer, before transmitting a new packet.

Class 4 allow long queues, and should be used for bulk transfer applications that do not require direct interaction. File Transfer Protocol (FTP) is a good example. The FTP data transfer connection could be placed in this class, while the FTP control channel could be placed in a interactive class (e.g. class 2). Another example for this class is email delivery; it require low loss but is delay-insensitive as no interaction is required once the email data is transmitted.

Class 5 has the lowest priority. Either this could be used for the default traffic which have not been classified into a class or it could be used as a penalty class for traffic which have been defined as "bad" according to the specific site policy.

Above we have given some examples of applications we believe maps to a specific service class. However, we will not define a strict setup of how to use service classes in general. The service classes are intended to be used as a basis for an agreement like a SLA. Thus, we leave the specific use as a site policy. In Chapter 9, we will show our site policy and define how and which of these service classes we choose to use in our real-world setup. In Chapter 7 we evaluate a specific packet scheduler and relate this to the delay bound for the service classes.

4.5 ACK-handling

The effect and importance of ACK packets, related to the performance of TCP, has been described in Chapter 2 and 3. It is clear from Part I that ACK packets require special attention and handling, especially on asymmetric links as ADSL. In this section we try to derive the properties and requirements for ACK packet and define them in the form of a special service class for ACK packets.

We see the ACK service class as being closely related to service class 1, with some extra constraints and properties, which can be exploited. ACK packets can handle fairly large delays, which is compensated through, and dependent on, the TCP window

size, as we saw in Chapter 3. We see ACK packets as jitter sensitive, because the inter-ACK spacing should be preserved and delay jitter disturb this spacing.

ACK packets also have some constraints, that makes them differ from service class 1. ACK packets are not very sensitive to drops, which is a relaxation of the requirements of service class 1. ACK drops are allowed because ACK packets are cumulative, meaning that the next ACK packet also acknowledges the previously received data (thus all ACK packets sent before itself). However, this can result in more bursty data traffic, as described in Section 2.4, because an ACK packet acknowledging several data packets opens the sender window in a larger chunk, this is known as stretched ACKs [18].

There exists a number of TCP-aware techniques that tries to exploit the properties and constrains of ACK packets. Most of them have found their way into RFC3449[18] – "TCP Performance Implications of Network Path Asymmetry". We will only describe techniques relevant to our middlebox scenario.

For our middlebox solution we identify four categories or techniques:

ACK-filtering [18, 20, 34]

ACK-filtering involves dropping ACK packets to reduce the amount of bandwidth consumed by ACK packet on a limited connection. A naive algorithm for dropping ACK packets could be performed when enqueueing a packet. The enqueue mechanism checks the queues for older ACK packets belonging to the same TCP connection. If ACK packets are found, some or all of them are removed from the queue.

A little more consideration should be applied when selecting ACK packets for dropping, because ACKs also performs other functions in TCP. It should be avoided to drop 3 duplicate ACK, which indicate a fast retransmit [15] and ACKs with the Selective ACK option (SACK)[58]. Special treatment of Explicit Congestion Notification (ECN)[65] feedback should also be considered[18].

ACK-pacing [13, 75, 80]

ACK-pacing is a method for controlling or smoothing the data sending rate by transmitting ACKs at a fixed rate without bursts. The method sort of recreates the inter-ACK spacing. While avoiding ACK bursts it will also influence and reduce the burstiness of the data packets. As ACK packets are a feedback mechanism, the size of the ACK queue should also be reduced and stabilized.

ACK-mangling [18, 51, 71, 79, 80]

ACK-mangling involves modifying and changing the content or size of ACK packets. We view compression as a form of ACK-mangling. But compression cannot be done on our middlebox as both ends of the connection have to implement compression and decompression.

A more advanced form of ACK-mangling, which is possible on our middlebox, is to control the data sending rate by modifying the TCP window size and the ACK sequence numbers [51, 79, 80].

ACK-prioritizing [18, 20, 50]

The idea is to give ACK packets highest priority. If ACK packets get the highest priority, the problems of ACK queueing and inter-ACK spacing should be solved as ACKs should not experience any delay. We will later in Chapter 7 show that it is difficult not to introduce any delay, because a (non-preemptive) packet scheduler has to wait for completion of at most a full packet before being able to transmit the high-priority packet.

We choose to implement ACK-prioritizing, as it is the simplest method. We have chosen not to design and implement any advanced ACK mechanisms, firstly because we believe that implementing an effective ACK technique is a project of its own, and secondly because time did not permit it. We have experimented with ACK-filtering, by emulating a simple ACK-filtering mechanisms with tail dropping ACK packets using a simple FIFO queue. This was done to reduce the amount of upstream bandwidth consumed by the ACK packet (on an 8 Mbit/512 kbit line). The bandwidth consumed by the ACK packets was reduced significantly with 40% and the downstream throughput was only reduced with 17% [26]. Unfortunately, we have not had enough time to investigate this.

When prioritizing ACK packets in a high priority service class, we hope to avoid delaying packets and hopefully avoid changing the inter-ACK spacing. We will later in Section 8.3 argue that the delay jitter introduced by upstream data packets can still disturb the inter-ACK spacing, but at a significantly smaller scale than without any packet scheduling. The effects of ACK-prioritizing are shown in Chapter 8.

4.6 Traffic Classification

Traffic classification is the task of classifying traffic into a given service class based on some specific rules. This section discusses our options for identifying and classifying the individual packets.

We consider three methods as options to identify and categorize traffic:

- Packet header fields.
- Data payload analysis.
- Traffic behavior.

Well-known service categorization

Identifying packets based on *packet header fields* is the classical method for categorizing packets into service classes. This is also referred to as *packet filtering*, which is a widely used technique for packet classification and deployed by packet filtering firewalls.

A set of port numbers has been assigned by IANA and listed in RFC1700[66]. The basic idea is to define "well-known" network services, so that clients know which port number to contact for a given service on a server. For example port number 80 is assigned to HTTP/web-traffic. RFC1700 state it as follows:

[66] **RFC1700:**"For the purpose of providing services to unknown callers, a service contact port is defined. This list specifies the port used by the server process as its contact port. The contact port is sometimes called the *well-known port*."

Defining rules to match specific header fields is therefore straightforward. When matching a specific network service the TCP or UDP port numbers are often used.

Failure of well-known service categorization

This simple service categorization by port number can be evaded with ease by a malicious user or program, simply by choosing to use a port number assigned to another service. Thus packets will be categorized into a different service class than intended.

This kind of misuse is normally not a problem, because servers that try to supply a service on a non-standard port must inform all clients that the service has changed port number. This limits the problem on a large scale, as it's hard to inform everybody on the Internet that the service has been reallocated.

Due to site policies and legal implications firewall administrators often try to block P2P traffic. This has lead to P2P protocols evolving evasive techniques in an attempt to avoid firewalling and thus classification by well-known port numbers [4]. The recent development in protocols for P2P overlay networks, has changed the scenario and imposes a serious threat to service categorization by well-known port numbers. The overlay functionality provides new service or resource mapping methods, which makes service port number reallocation possible.

One of the evasive techniques is to change port numbers dynamically, whenever the port is firewalled. The port number can also be changed manually by the user with ease. Another evasive technique, which we consider malicious, is to disguise itself by using port numbers of well-known services.

New methods for categorization

When the well-known service categorization fails, we are faced with a problem of separating evasive from legitimate traffic by other means than header fields. We see two options: *data payload* and *traffic behavior* analysis. These two techniques can be related to network security techniques.

Security techniques used in a Network Intrusion Detection System (NIDS) perform pattern matching on *data payload*, with a set of static rules to identify known hacker or worm attacks. Advanced NIDS also use statistical analysis on *traffic behavior* to detect anomalies and intrusion attacks, which have not been seen before or cannot be matched by a static rule.

With data payload analysis we are basically looking at the application layer. This expands the possible rules or patterns needed as each new application introduces a new pattern. False matching might also occur when having "weak" matching patterns, e.g, when the application data does not have peculiar strings or patterns which distinguish it easily from other applications. Random strings in binary data might also cause false matching. Resource wise, it is also expensive to perform pattern matching with a large number of rules on the payload of every packet. On ADSL we can probably manage the extra load, as we are operating with fairly small amounts of traffic due to the limited link capacity.

Behavior analysis

Detecting different traffic behavior for our needs do not necessary require advanced statistical analysis, but interpreting the data might still be hard and cause false matching.

We could simply count the amount of traffic that a connection has transferred. This could be interpreted as a bulk transfer, but looking at a larger time scale this might just be a long-lived connection. We might instead measure the amount of data over a smaller time scale to measure the rate consumed by each traffic flow. This would give a better identification of bulk transfers. The time scale is important as small bursts of conforming traffic might be categorized as bulk transfers. Care should be taken to avoid oscillation between traffic classes as it might cause reordering of

[4] P2P networks have understood "the value of not being seen" (cite: Monty Python)

packets. Counting the number of connections from a single local host to external hosts might also be a indication of bad behavior.

Instead of detecting bad traffic behavior we might as well try to detect traffic, which conforms to a given service class, e.g., interactive traffic. In the security literature, some efford has been done for detecting backdoors[82] and stepping stones[83], which is basically detecting interactive behavior of traffic flows.

The specific site-policy setup for our real-world setup is described in Chapter 9.

4.7 Packet Scheduling

In this section we will discuss some of the techniques for implementing link-sharing and how to combine these. Sharing link resources is done through packet scheduling or traffic conditioning (DS terminology, Figure 4.1), which is responsible for enforcing the configured resource allocation. Implementing a new special scheduling mechanisms is a project of its own. We note, that our practical implementation will be limited by what has been implemented on the system we have chosen for our middlebox (in this case Linux).

We have some basic requirements for our scheduling scheme: isolation and sharing. Traffic flows or aggregates should be isolated from each other, but on the other hand they should also be able to share the available bandwidth. Complete isolation through strict resource allocation results in wasted bandwidth if some of the resource allocations are not fully used. Thus, we need to find a balance between isolation and sharing.

A set of algorithms called Fair Queueing (FQ)[31, 52] maintains a good balance between the two conflicting forces of isolation and sharing. A lot of different implementations and variations of FQ algorithms exists. The Generalized Processor Sharing (GPS) model is an ideal FQ algorithm, and Weighted Fair Queueing (WFQ) is a well known model with many variations, like Worst-case Fair Weighted Fair Queueing (WF2Q)[22], Self-Clocking Fair Queueing (SCFQ), Weighted Round Robin (WRR), Stochastic Fairness Queuing (SFQ)[59] and Deficit Round Robin (DRR)[68]. The basic idea is to provide each flow with a fair share of the resources. The different algorithms allow resource allocation through priority or weights. The FQ algorithms were primarily developed for the Integrated Services (IS) architecture, which is why it is focused on individual flows.

We are closer related to the Differentiated Services (DS) architecture, as we categorize and aggregate traffic into a number of service classes. Thus, we need scheduling between service classes and within each service class. The FQ algorithms are focused on individual flows and is not suited for dividing traffic into classes. Instead we look at the Class-Based Queueing (CBQ) model, which is described in Sally Floyd's classic article [38], which describes a Class-Based Queueing (CBQ) model which is called "Hierarchical Link-Sharing". We do not need the hierarchical property of the model, but are more interested in the scheduling and isolation between service classes. A key feature is that the model gives us some tools to avoid starvation of a low priority class, as the framework has the ability to share bandwidth between classes with different priorities. The model also allows a packet scheduler to be assigned to each class, which determines which packet is selected when the class is allowed to transmit a packet. This implies that we can assign a FQ packet scheduler to each class and thus achieve fair sharing of resources within each traffic class.

Thus supports our goal of sharing the link resources between different groups of network applications by service classes and at the same time assuring fair sharing

within each service class with a FQ packet scheduler.

The specific setup implemented on our middlebox is described Section 9.3.1.

4.8 Summary

In this chapter we have discussed and identified the components needed for our middlebox solution.

To fulfill our goal of using the ADSL connection for different types of services at the same time, we need service differentiation and resource sharing, as the current "best effort" service model cannot fulfill our goal. Service differentiation involves defining a set of *service classes* and service levels for each class. Here ACK traffic is identified as a separate service class, to accommodate good utilization of the downstream link. Service differentiation also involves assigning traffic to a given service class through *traffic classification*. Resource sharing involves *packet scheduling* techniques and algorithms to achieve traffic isolation and fair resource sharing.

We describe how our middlebox is mostly related to the Differentiated Services (DS) QoS architecture, but without any cooperation with other network elements. With no cooperation we find a need for local traffic conditioning which involves *achieving queue control*, for our packet scheduling to have any effect. With our ADSL network environment we identify the need for knowing the specific *link layer overhead* of ADSL, before we can achieve queue control.

This involves the following components:

Service classes: We describe 6 service classes and their requirements in Section 4.4, which are based on the definitions in Y.1541[7]. The choice and usage of service classes for our real-world setup are described in Chapter 9.

Traffic classification: Challenges and methods for traffic classification are described in Section 4.6. Again the specific setup is described in Chapter 9.

ADSL link layer overhead: The types of link layer overhead on ADSL is described in Chapter 5.

Queue control: How we achieve local queue control using our knowledge of the ADSL link layer overhead is described in Chapter 6.

Packet scheduling: In Section 4.7 we describe, how we choose to use the Class-Based Queueing link-sharing model to perform traffic isolation and sharing between traffic aggregates or classes and choose to use a Fair Queueing (FQ) algorithm within each class. In Chapter 7 we evaluate the delay bounds of a specific packet scheduler (chosen due to our queue control implementation). In Chapter 9 we describe the specific setup of the scheduler.

ACK-prioritizing: Chapter 8 describes, how full utilization of the downstream capacity is achieve by prioritizing ACK packets.

Site-policy: How we choose to combine and configure the specific components for a given site is expressed as a site-policy. The site-policy for our real-world ADSL connection is described in Chapter 9.

Packet scheduling, queue control and ADSL link layer overhead modeling are closely related and together perform the role of traffic conditioning (DS). Service classes also

relate to packet scheduling, but the configuration and usage is up to the specific site-policy. The site-policy is primarily concerned with the usage of service classes and traffic classification. Traffic classification is closely related to the DS classification module (Figure 4.1 on page 38).

Part II is concerned with the details these components which are the basis for our middlebox. Part III is concerned with the practical use through combining of the components. Chapter 9, which is mentioned above represents the start of part III.

Chapter 5

ADSL Link Layer Overhead

In Section 4.3, we described a need for queue control and that achieving queue control involved scheduling packet at the same rate (or slightly less) as the rate of the bottleneck link. In our case the ADSL connection is the bottleneck link, as documented in Chapter 3. Thus, we need to know the link layer overhead introduced by ADSL, as our packet scheduling middlebox resides on another type of link layer and needs to model this link layer in order to achieve queue control.

In this chapter we describe the ADSL protocol stack with focus on the link layer overhead imposed by the different kinds of encapsulation. We only consider the overhead on the ADSL line, not the overhead on the path through the ISP's ATM network, as we assume they have sufficient capacity to handle the overhead.

In the case of ADSL, the overhead varies (a lot) due to protocol overhead and packet aligning at different layers. This complicates the task of matching the exact rate at a different link layer. In this chapter we use the IP layer as the reference layer and the overhead is calculated relative to IP, because our packet scheduling middlebox functions at the IP layer.

The ADSL standard[4, 5] is dependent on ATM. ADSL provides a data connection at (maximum) possible link (sync) rate of the physical copper wires, while ATM is used to provide different speed and service levels.

5.1 Encapsulation Layers of IP over ADSL

Overview: Encapsulation layers of IP over ADSL		
Layer	AAL5 coupling	Notes
IP		The perceived payload
PPP	SSCS (optional)	
MAC	SSCS (optional)	Bridged mode (RFC2684B)
PPPoE or PPPoA	SSCS (optional)	RFC2516 and RFC2364
LLC or VC	SSCS	Encapsulation mode
AAL5	CPCS + SAR	Padding
ATM		
ADSL		Physical link

Table 5.1: Overview of Encapsulation layers of IP over ADSL

Transporting IP over ADSL can be done using different types of encapsulation. Starting from the bottom of Table 5.1 the common lower layers (beside ADSL) are Asynchronous Transfer Mode (ATM) and ATM Adaption Layer type 5 (AAL5) [1].

The basic idea behind ATM is to transmit data in small fixed-size cells. The ATM cells are 53 bytes long consisting of a 5-byte header and 48 bytes of payload. This introduces a *fixed overhead* of 9.4% ($\frac{5}{53}$), which is often refered to as the ATM tax.

The ATM Adaption Layer (AAL) have two functions or sublayers:

1. Segmentation And Reassembly (SAR)
2. Convergence Sublayer (CS),

as illustrated in Figure 5.1 for AAL5. The purpose of Segmentation And Reassembly (SAR) is to segment a data packet from the higher layers into the payload part of the ATM cells (and the reverse operation of reassembly to the higher layers). The Convergence Sublayer (CS) is divided into a Common Part Convergence Sublayer (CPCS) and a Service Specific Convergence Sublayer (SSCS). The CPCS (of AAL5) adds an *8-byte trailer* and assures that the CPCS data packet is aligned for the SAR sublayer (a multiple of 48 bytes) through 0 to 47 bytes of padding. This type of padding result in a *variable overhead* (non-linear to the packet size of the higher layers). There are often several *optional* SSCS layers, which is specific for a given service. Each SSCS layer generally adds an extra header. The coupling between AAL5 and the different types of encapsulation on ADSL are shown in Table 5.1.

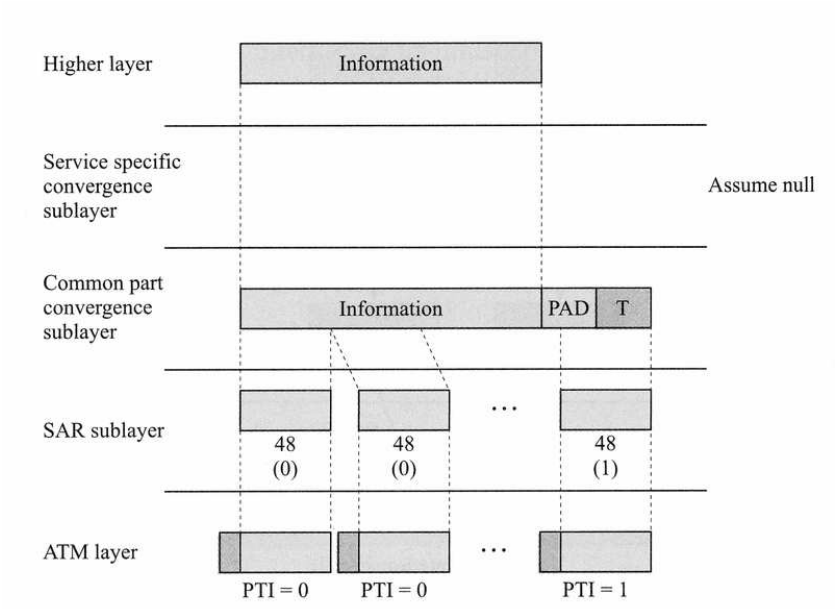


Figure 5.1: The AAL5 process (figure scanned from [56]).

Our goal is to determine the per packet overhead at the IP level. Determining the overhead and in particular the padding overhead is complicated further as each SSCS adds extra overhead in form of headers. This shifts and changes the ATM cell alignment done in the CPCS layer.

It is worth noticing that with the smallest TCP/IP packet (40 bytes) and the AAL5 tail (8 bytes) the payload limit is reached for a single ATM cell, implying that a single

byte of extra overhead will result in two ATM cells (53+53=106 bytes) to carry the smallest TCP/IP packet. The TCP ACK packet is a good example of the smallest TCP/IP packet, which is used very frequently.

Thus we need to determine which kind of SSCS services are used and the associated overhead. In the following we will describe the individual SSCS services, or rather encapsulation protocols, from Table 5.1. Most of the encapsulation protocols can be combined to provide a given SSCS service. The descriptions are ordered according to how they can be combined.

5.1.1 AAL5 - LLC or VC

The encapsulation mode of AAL5 is not optional, as can be seen from Table 5.1. Using AAL5 there is a basic choice between using Logic Link Control (LLC) encapsulation or Virtual Circuit (VC) multiplexing mode [41, 43, 55]. LLC [17] is the default[55] and most often used technique even though VC multiplexing has no overhead. VC multiplexing has no header as there is a separate VC for each protocol. The VC information or number is an integrated part of the (fixed) cell ATM header. LLC supports multiple protocols on a single VC. The *LLC header size is 3 bytes*[1]. To avoid confusion, description of the header content of LLC is dropped as it is not relevant for our overhead calculation, for a further description see [17, 41, 43].

5.1.2 PPP

The Point-to-Point Protocol (PPP) is an encapsulation protocol which provides a standard method for transporting multi-protocol datagrams over point-to-point links. PPP also offers a set of management mechanisms like authentication, network and link control protocols and more.

PPP packet:

Protocol ID 8/16 bits	data	Padding (not used)

The PPP packet includes the possibility to include PPP padding, however most (if not all) applications using PPP never make use of padding [14]. The primary overhead is thus the protocol ID, which is 2 bytes for IP as the network layer. The PPP protocol ID can be negotiated to one byte via the Protocol-Field-Compression PPP option but RFC2364[41] – "PPP over AAL5" recommends not to use this option. Thus we assume that the *PPP overhead is 2 bytes*.

5.1.3 PPPoA

PPP over AAL5 (PPPoA) is an ADSL stack or SSCS service, which is a combination of the above encapsulation methods. It combines LLC/VC and PPP and is defined in RFC2364[41] – "PPP over AAL5".

PPPoA with LLC:

3 bytes LLC header	1 byte NLPID (PPP = 0xCF)	2 bytes PPP	IP	0-47 bytes ATM padding	8 bytes AAL5 tail

[1]The LLC header size can exceed 3 bytes, but only if the specified format includes sequence numbering, which is not the case of ADSL encapsulation protocols [17].

It supports both VC multiplexing and LLC encapsulation. When using VC multiplexing, no extra headers are added and the PPP packet is simply put directly into the AAL5 (CPCS-PDU). In *LLC mode 1 extra byte is added* (the Network Layer Protocol IDentifier (NLPID)), which is set to PPP as illustrated above. Resulting in a total overhead with LLC of 14 bytes $(3+1+2+8)$ and VC of 10 bytes $(2+8)$.

5.1.4 Bridged Mode

In bridged mode the Ethernet MAC header is encapsulated and transported across the ADSL connection. Bridged mode for IP/Ethernet over AAL5 is defined in RFC2684[43] – "Multiprotocol Encapsulation over ATM Adaptation Layer 5". The RFC defines encapsulation for several protocols, besides Ethernet, as the title reveals. There is generally two modes: routed or bridged mode. We refer to the two modes by adding a "R" or a "B" to the RFC number. The focus here is on bridged mode (routed mode is described later). The RFC is often referred to as RFC1483 in older documentation (and when configuring Cisco IOS), but RFC2684 obsoletes RFC1483.

RFC2684B[43] allows both VC or LLC mode. We describe the LLC mode first. The encapsulation of an Ethernet/802.3 frame is shown below:

3 bytes	3 bytes	2 bytes	2 bytes	14 bytes			4 bytes	8 bytes
LLC	OUI	PID	PAD	MAC	IP	MAC	FCS	AAL5
	SNAP					pad	optional	+pad

The LLC is known, but the next header (OUI+PID) is an IEEE 802.1a SubNetwork Attachment Point (SNAP) header. The SNAP header consists of a 3-byte Organizationally Unique Identifier (OUI) and a 2-byte Protocol IDentifier (PID). The "PAD" field is padding to align the MAC frame to begin at a four byte boundary, in case of Ethernet this field is 2 bytes long. The Ethernet MAC header is 14 bytes long and consists of 6 bytes destination address, a 6 bytes source address and a 2 bytes type. The Frame Check Sequence (FCS) is the MAC checksum – it is optional and occupies 4 bytes.

The MAC padding is done to ensure a minimum packet size as an Ethernet frame must contain at least 64 bytes (payload + MAC overhead) because of the Ethernet collision (CSMA/CS) protocol. That is, the padding is performed on frames with a payload less than 46 bytes $(64-18)$. With a TCP/IP ACK packet of 40 bytes the MAC padding is 6 bytes. If the FCS field is dropped, the 64-byte minimum packet size can also be dropped, because it is then required to process the packet at a higher layer (e.g. an IP router) before transmitting the packet on the new link layer where the checksum is (re)calculated anyway (the PID field specifies whether to include the FCS). The FCS checksum is redundant as the AAL5 tail also contains a 4-byte checksum.

In VC multiplexed mode everything before the PAD field is dropped, that is, the LLC and SNAP (OUI+PID) headers. This results in a 2-byte overhead plus of course the MAC overhead.

Overhead elements RFC2684B	
VC	+ 2 bytes
LLC	+ 10 bytes
MAC FCS not preserved	+ 14 bytes
MAC FCS preserved	+ 18 bytes
Minimum size with FCS	64 bytes
AAL5 tail	+ 8 bytes

Total overhead with RFC2684B		
	Using LLC	Using VC
without FCS	32 bytes	24 bytes
with FCS	36 bytes	28 bytes

Due to the Ethernet MAC padding of small packets, the largest overhead imposed (on the smallest TCP/IP packet) is 42 bytes. That is with a minimum Ethernet packet size of 64 bytes (when preserving FCS and using LLC) a TCP/IP (ACK) packet of 40 bytes gets an extra 6-byte MAC padding overhead ($36 + 6 = 42$), giving a total packet size of 82 bytes ($40 + 42$). Although this seems extreme, it is worth noticing that this is still contained within two ATM cells (which can contain 96 bytes of payload).

5.1.5 Routed Mode

Routed mode is also part of RFC2684[43] and we referrer to it as RFC2684R.

RFC2684R with LLC:

3 bytes	3 bytes	2 bytes		8 bytes
LLC	OUI	Type	IP	AAL5 tail
	SNAP			

In LLC mode RFC2684[43] states that RFC2684R *MUST* use SNAP mode, and that it *MUST NOT* use the NLPID format (see the PPPoA packet figure), even though there is a ISO NLPID value (0xCC) that indicates IP. The SNAP format uses 5 bytes whereas the NLPID format uses 1 byte. Thus the LLC/SNAP overhead is 8 bytes, giving a total *overhead of 16 bytes*. Relating this to PPPoA shows that PPPoA with LLC introduces less overhead even though it carries more information than RFC2684R with LLC.

In VC multiplexed mode we have the optimal situation of no additional header information. This implies no SSCS sublayer, thus only the CPCS sublayer with the AAL5 tail of 8 bytes. This is the only IP over ADSL mode where the TCP/IP (ACK) 40 bytes packet fits exactly into one ATM cell. Unfortunately, this mode is hardly ever used in commercial ADSL products.

5.1.6 PPPoE

When adding PPP functionality to the Bridged mode (RFC2684B[43]), the PPP over Ethernet (PPPoE) protocol specification is used on top of RFC2684B[43]. PPPoE is defined in RFC2516[57] – "A Method for Transmitting PPP Over Ethernet (PPPoE)". This is the ADSL over IP encapsulation stack with the most overhead. The header size of PPPoE is 6 bytes. The specific content of the PPPoE header can be seen in [57]. The PPP protocol is encapsulated in PPPoE, thus another 2-byte overhead.

The total PPPoE encapsulation stack:

3 bytes	5 bytes	2 bytes	14 bytes	6 bytes	2 bytes			4 bytes	8 bytes
LLC	SNAP	PAD	MAC	PPPoE	PPP	IP	MAC pad	FCS optional	AAL5 +pad

PPPoE (with LLC) has a stunning total overhead of 44 bytes. The overhead alone almost uses an entire ATM cell except for 4 bytes ($48 - 44$), which is left for the IP level.

5.1.7 OAM Overhead

A special kind of overhead which influences ATM links is Operations Administration and Management (OAM) cells. At the physical layer one out of every 27 ATM cells is an OAM cell (or idle cell) [2, 35, 37] [2]. ADSL also has some OAM control packets at the physical layer [4, 5], which consumes bandwidth, but these ADSL-OAM packets can (fortunately) be ignored in our calculations because the bandwidth offered to the ATM layer is not affected/reduced by the ADSL-OAM packets.

5.2 Overview of the Encapsulation Methods

We have identified 12 combinations of encapsulation methods or SSCS services which are shown in Table 5.3. The SSCS overhead column shows the overhead imposed at AAL5 SSCS layer and the AAL5 tail column shows the total overhead imposed on an IP packet. The TCP/IP column is an example with a 40-byte TCP/IP packet (e.g. a TCP/IP ACK packet) and the last column shows the number of ATM cells needed to transport the TCP/IP packet.

Name	**Type**	**VC or LLC**	**SSCS overhead**	**AAL5 tail** + 8 bytes	**TCP/IP** + 40 bytes	**ATM cells**
PPPoA	Routed	VC	2	*10*	50	2
PPPoA	Routed	LLC/NLPID	6	*14*	54	2
RFC2684R	Routed	VC	0	*8*	48	1
RFC2684R	Routed	LLC/SNAP	8	*16*	56	2
RFC2684B	Bridged w/o FCS	VC	16	*24*	64	2
RFC2684B	Bridged w/o FCS	LLC/SNAP	24	*32*	72	2
RFC2684B	Bridged with FCS	VC	20	*28*	68	2
RFC2684B	Bridged with FCS	LLC/SNAP	28	*36*	76	2
PPPoE	Bridged w/o FCS	VC	24	*32*	72	2
PPPoE	Bridged w/o FCS	LLC/SNAP	32	*40*	80	2
PPPoE	Bridged with FCS	VC	28	*36*	76	2
PPPoE	Bridged with FCS	LLC/SNAP	36	*44*	84	2

Table 5.3: Overhead summary

It is easy to account for a fixed overhead per IP packet when scheduling packets. Accounting for the packet alignment cost is a bit more difficult. First the number of ATM cells, for a given IP packet, is determined through dividing the IP packet size plus overhead (AAL5:SSCS+CPCS) with 48 bytes (the payload size of an ATM cell), and aligning it to a multiplum of 48 bytes. Then the actual bytes used on the ADSL/ATM link is determined by multiplying with the size of an ATM cell (including header): 53 bytes. The calculation is shown in formula 5.1.

[2] We have actually experienced this in a misconfigured ADSL setup, where the bottleneck was on the wire between the Cisco1401-ADSL router and ADSL-modem which was running ATM at 512 Kbit/s.

Formula 5.1

$$ATM\_Link_{Size} = \left\lceil \frac{IP_{PacketSize} + Overhead}{48} \right\rceil \cdot 53$$

5.3 Summary

In this chapter we have described the variable overhead on ADSL connections, which is caused by the ATM/AAL5 link layer and the different encapsulation methods used on ADSL. We have identifyed 12 types or combinations of encapsulation overheads, which is shown in Table 5.3.

An important finding is that, for all but one encapsulation method *two ATM cells* are needed to transport a TCP/IP control packet (of 40 bytes). That is, to transport a packet of 40 bytes, 106 bytes of ADSL capacity is used giving a overhead of 66 bytes, which is an overhead of 62%. For larger packets the overhead has less impact as it constitutes a smaller percentage of the total packet size. The *overhead cost per packet* makes smaller packets more expensive to send.

Chapter 6

Achieving Queue Control

As concluded in Section 4.3, we need to achieve queue control for the packet scheduler to have any effect on our middlebox solution. We can achieve queue control by controlling the sending rate and adjusting it to slightly less than the rate of the bottleneck link. In Chapter 3 we found that latency was related primarily to upstream queueing. Therefore, we are focused on upstream queue control. Upstream queue control is achieved directly by simply delaying or dropping packets to make the traffic conform to a configured rate.

This chapter contains our approach and solution to upstream queue control with ADSL link layer overhead modeling and the implementation details of the required modifications to the Linux Traffic Control system.

6.1 Link layer overhead modeling

In this section we describe how to achieve queue control using a middlebox based on Linux. The Linux Traffic Control system does not have functions to model or compensate for the ADSL/ATM link layer overhead (described in Chapter 5). Firstly, we describe and exemplify the naive approach of reducing the rate to handle different cases of overhead. This is done to demonstrate the effect and magnitude of the overhead. Secondly, we describe how the Linux kernel and the userspace program is modified to perform accurate overhead modeling.

6.1.1 Naive Approach

The naive approach, which does not require any modification of the QoS system, is simply to fix the sending rate at a reduced level in order to compensate for the overhead on the physical ADSL link. The question is how much the rate should be reduced to compensate for the worst-case scenario. We will calculate the fixed overhead and try to estimate the varying overhead using our knowledge from Chapter 5. This exercise also gives an indication of the magnitude and impact of the ADSL overhead.

The example is based upon our primary test ADSL. It is a 2 Mbit/512 kbit ADSL line with a per-packet overhead of 28 byte — this represents an ADSL with RFC2684B encapsulation in bridge mode (using VC, including FCS), see Table 5.3 on page 53 for reference. Please note, that this ADSL connection does not represent the worst case per-packet overhead (the worst case is 44 bytes).

The ATM header overhead is fixed and can be calculated with ease. For the 512000 bits/s upstream line the ATM cell header overhead is 48300 bits/s ($\frac{512kbit/s}{53bytes} \cdot 5bytes$). Thus, the rate is reduced to 463700 bits/s.

Estimating the rate reduction due to per-packet overhead is harder. A rate measurement is a time measurement. Thus, it is possible to calculate the overhead (reduction) rate knowing the packet rate. It depends on the average packet size on the line: a large packet size gives a small number of packets per second and thus a smaller overhead. Knowing the bandwidth and the average packet size, the packet rate is calculated as follows:

$$bandwidth/packetsize = packetrate$$

Packet rate estimates

With 1500-byte packets the resulting ATM link layer size is 1696 bytes including overhead ($\lceil(1500+28)/48\rceil \cdot 53$). A packet size of 1696 bytes on a 512 kbit/s link results in only 38 packets/s. The lower bound on packet size on ATM is a single cell of 53 bytes, which gives a maximum of 1208 packets/s. As packets often use two ATM cells (as described in Section 5.2), a more likely maximum is 604 packets/s.

On a busy ADSL connection we have observed an average of 200 packets/s and likely bursts of 300 packets/s. The upstream link should also be able to transport ACK packets generated from downstream traffic. To support an ACK packet for every (1696-byte) data packet on a 2 Mbit/s (downstream) link gives an (upstream) ACK load of 148 packets/s ($2Mbit/1696bytes$).

Per-packet overhead to rate estimate	
Case:	Rate calculation
Best-case :	$38packets/s \cdot 28bytes = 8512bits/s$
ACK support :	$148packets/s \cdot 28bytes = 33152bits/s$
Average-case:	$200packets/s \cdot 28bytes = 44800bits/s$
Bursty-case :	$300packets/s \cdot 28bytes = 67200bits/s$
Worst-case :	$600packets/s \cdot 28bytes = 134400bits/s$

Although, the worst-case packets/s is unlikely to occur, queue control should still be maintained in this situation. If the worst-case packet rate is not handled, a sudden packet burst in form of a port-scan or worm attack can easily result in loss of queue control.

Ignoring worst-case anyway, and only focusing on the more likely bursty-case, the rate should be reduced to approximately 396 Kbit/s. The difference between the bursty and the best-case is 58.6 kbit/s, which is thus wasted in good conditions. In normal/average conditions 23 kbit/s is wasted. For handling the worst-case, we would be wasting 126 kbit/s when the line is operating in the best-case. This is not an optimal utilization of the bandwidth.

Alignment estimate

Moreover, the rate reduction needed is actually larger due to the packet alignment overhead. The packet alignment overhead is even harder to predict. Viewing the packet alignment as a per-packet overhead, we try to estimate the overhead. The worst case packet alignment overhead is 47 bytes per packet. Using the bursty-case (300 packets/s)

gives an alignment overhead of 112800 bits/s. A corresponding adjustment results in a rate reduction to approximately 45% to 284 kbit/s. With this rate reduction it is possible to handle the bursty-case with a very bad packet alignment, but we can still loose queue control.

Instead, assuming an average case packet alignment overhead of 23 bytes per packet, the bursty-case packet rate gives an alignment overhead of 55200 bits/s, thus the rate is reduced to approximately 341 kbit/s, which is a reduction of 33%.

In the worst-case scenario with 603 packet/s and the line filled with 40-byte IP packets, a packet consumes two ATM cells (106 bytes)(due to the encapsulation overhead), thus a per packet overhead of 66 bytes ($106-40$). Thus, to handle the worst-case packets per second scenario the rate needs to be reduced to 194 kbit/s a reduction of 62%.

It is clear that reducing the rate to compensate for the overhead is not a viable solution, because of wasted bandwidth to handle unlikely situations. This violates our goal of maximum link utilization, which states that we should avoid wasting link capacity. This implies a need for modifying the QoS system to handle the ADSL/ATM overhead dynamically.

In the calculations above we ignore that the packet alignment changes/lowers the packet rate as it consumes part of the bandwidth. We can do this because we base our calculations on the observed packet rates and not the calculated ones.

6.1.2 Accurate Overhead Modeling

In this section, we describe how to achieve queue control using a middlebox based on Linux. We describe the shortcomings of the Linux *Traffic Control* system and how to turn this to our advantage, when modifying the system to perform accurate link layer overhead modeling. We prefer to be as non-intrusive as possible and avoid extensive modifications in our modifications of the Linux kernel.

The Linux Traffic Control system[1] consists of a kernel part and a userspace program (called `tc` for Traffic Control). In the kernel the packet *transmission times* or the cost of transmitting a packet are based upon a lookup table, this table is called the *rate table*. A rate table is associated with each token bucket. The rate tables are pre-calculated in the userspace program and passed on to the kernel[2]. The scheduling mechanism is not very precise due to this rate table, because a rate table only contains 256 elements, which obviously is not enough to represent all packet sizes.

The table (packet size) lookup function is a simple shift right, which means that the resolution or packet intervals are in powers of two. The packet intervals or resolution of the table is determined by the Maximum Transfer Unit (MTU). To determine the smallest power of two necessary for packet size lookups to be within the table, the MTU is simply shifted right until it is below 256. With an MTU of 1500 bytes (and up to 2047 bytes) three shift right is required, thus 2^3 which result in intervals of 8 bytes. The number of shifts is termed `cell_log`, and the rate table lookup function is shown below:

```
rtab[ pkt_len >> cell_log ] = pkt_xmit_time;
```

We can avoid extensive kernel modifications by modifying the calculation of the rate

[1]The description is based upon the Linux kernel 2.4.27.

[2]The pre-calculated rate table is used in the token bucket based schedulers, HTB, CBQ, police, and TBF, that is, almost all.

table in the userspace program. Our overhead calculation involves ATM cell alignment, which corresponds to packet size intervals. Thus as long as our cell alignment align with the table interval we achieve an accurate rate table. The IP packet needs to be aligned for 48-byte cells (the ATM cell payload size), which can be aligned to a table interval up to 16 bytes (or 2^4).

Unfortunately, with the current "hash" mapping or table lookup method, the table does not align. The table mapping intervals look like this, with the default shift of 3 (inteval $2^3 = 8$):

```
Outputting the tc rate table (with cell_log=3)
 entry[0](maps: 0- 7)=xmit_size: 0 atm_size:0
 entry[1](maps: 8-15)=xmit_size: 8 atm_size:53
 entry[2](maps:16-23)=xmit_size:16 atm_size:53
 entry[3](maps:24-31)=xmit_size:24 atm_size:53
 entry[4](maps:32-39)=xmit_size:32 atm_size:53
 entry[5](maps:40-47)=xmit_size:40 atm_size:53
 entry[6](maps:48-55)=xmit_size:48 atm_size:53
 entry[7](maps:56-63)=xmit_size:56 atm_size:106
 entry[8](maps:64-71)=xmit_size:64 atm_size:106
```

The current rate table implementation uses the lower boundary for calculating the `pkt_xmit_time` (here shown as `xmit_size`). The `atm_size` is the number of ATM cells calculated from the `xmit_size`. The mapping is not perfect, and does not align for our 48-byte ATM cell payloads. This mapping actually implies that transmitting a 7-byte packet "costs" 0 bytes to transmit. It also implies that for packet sizes which do not align, the cost reported to the Traffic Control system is less than the actual size. This poses a potential problem, as we might loose queue control by transmitting faster than the system was configured for.

To make the table align, we chose to make a small kernel modification. We change the kernel lookup method and make the userspace program use the upper boundary for calculating the `pkt_xmit_time` (`xmit_size`). The new lookup method:

```
rtab[ (pkt_len-1) >> cell_log] = pkt_xmit_time;
```

This changes the mapping to:

```
entry[0](maps: 1- 8)=xmit_size:8
entry[1](maps: 9-16)=xmit_size:16
entry[2](maps:15-24)=xmit_size:24
entry[3](maps:24-32)=xmit_size:32
entry[4](maps:33-40)=xmit_size:40
entry[5](maps:41-48)=xmit_size:48
entry[6](maps:49-56)=xmit_size:56
```

For the purpose of the ATM/ADSL mapping, it is now possible to do a perfect mapping to ATM cells, even with a `cell_log` of 4 (inverval $2^4 = 16$). The new table mapping looks like this (the `pkt_xmit_time`'s are then calculated from the `atm_size`):

```
Outputting the tc rate table (with cell_log=4)
 entry[0](maps:  1- 16)=xmit_size: 16 atm_size: 53 cells:1
 entry[1](maps: 17- 32)=xmit_size: 32 atm_size: 53 cells:1
 entry[2](maps: 33- 48)=xmit_size: 48 atm_size: 53 cells:1
 entry[3](maps: 49- 64)=xmit_size: 64 atm_size:106 cells:2
```

```
entry[4](maps: 65- 80)=xmit_size: 80 atm_size:106 cells:2
entry[5](maps: 81- 96)=xmit_size: 96 atm_size:106 cells:2
entry[6](maps: 97-112)=xmit_size:112 atm_size:159 cells:3
entry[7](maps:113-128)=xmit_size:128 atm_size:159 cells:3
entry[8](maps:129-144)=xmit_size:144 atm_size:159 cells:3
entry[9](maps:145-160)=xmit_size:160 atm_size:212 cells:4
```

A new problem arises, when a packet size lookup actually ought to be a lookup of packet size plus an overhead. Resorting to only modifying the userspace rate table, this corresponds to shifting the table interval according to the size of the overhead, thus the overhead size also needs to align with the table intervals (to achieve an accurate rate table). Assuming a table interval of 8 bytes (2^3) only 6 out of the 12 overheads (identified in table 5.3) can be aligned. To solve this, we choose to make a kernel modification where the overhead is added when doing table lookups (the overheads are the ones described in Chapter 5). Our kernel table lookup method is thus:

```
rtab[ (pkt_len - 1 + overhead) >> cell_log] = pkt_xmit_time;
```

Our overhead patch is primarily based on Linux's Hierarchical Token Bucket (HTB) scheduler, because there was an existing overhead patch for HTB, which implemented a per-packet overhead functionality by (only) modifying the userspace rate table. The existing HTB patch was used as an early prototype where we estimated the aligning overhead per packet. Thus, it was natural to continue to use HTB when implementing the accurate overhead modeling. In our kernel patch, we modify all the schedulers which use the rate table lookup system, but we have only evaluated the HTB implementation, because we have not implemented the (overhead) command line option for all schedulers.

The modifications to the kernel and userspace program can be seen in appendix B.2 on page 138 including a more detailed description of the specific modifications.

6.2 Evaluation

In this section, we demonstrate the effectiveness of our overhead patch. We will show that the naive approach cannot handle all packet distributions and show that the overhead patch does maintain queue control in the same as well as more extreme situations. Note that we are limited to upstream queue control as described in Section 4.3. Thus, this evaluation focuses on upstream queue control.

It should be clear that, due to different types of overhead, it is not possible to achieve a throughput equal to the sold or published capacity of the ADSL connection. Thus, showing that we cannot achieve queue control in this situation should be unnecessary. Instead we show that lowering the upstream rate helps gain queue control and that it is lost again when the overhead changes.

The ADSL line used in these tests is a 2 Mbit/512 kbit ADSL from Tele2.

6.2.1 Queue Test Setup

For the purpose of this queue control test, traffic is organized into three traffic classes or queues.

- (1:10) A queue for interactive traffic.

- (1:30) A queue for upstream bulk transfers.
- (1:50) A (default) queue for "disturbing" traffic.

A simple priority scheduler would have been sufficient in this setup, but the class-based HTB scheduler was chosen because our overhead patch/implementation is based on HTB. The HTB class names are shown in parenthesis.

A simple FIFO queue is used for packet scheduling within each class. The FIFO queue buffer size is specifically set to 28125 bytes, to give a maximum queueing delay of 500 ms at 450 kbit/s ($500ms \cdot 450Kbit/s$).

The class setup is shown in the table below; The *ceil* is set to 100% for every class, which implies that classes can borrow from each other up to 100% of the available bandwidth (the ceil rate). Class *1:10* has the highest priority (lower is better), then comes *1:30*, and *1:50* is served last with a priority of 5. The *rate* is the guaranteed bandwidth for the class.

parent:	1:1		
class:	1:10	1:30	1:50
prio	0	4	5
rate	20%	40%	40%
ceil	100%	100%	100%

For traffic classification; SSH packets are classified in to the interactive class *1:10*, SCP packet are classified in to the bulk class *1:30*[3] and the rest goes into the default class *1:50*.

To measure the latency in each class, ping (ICMP) packets (to different IP-addresses) are classified into each class. The ping IP-addresses are chosen to be as close as possible to the ADSL connection. We have chosen three IP-addresses, which are 2 IP hops away, their mappings are shown in the table below. We have been informed by Tele2 that the three IP-addresses are three outgoing interfaces on the same (RedBack) Broad Band Remote Access Router (BBRAS) (in Valby, Copenhagen). This implies that the packet has traveled through a fairly long ATM network.

Class	DNS	IP
1:10	atm0-1–0.val10-core.dk.tele2.com	130.227.0.81
1:30	atm0-2–0.val10-core.dk.tele2.com	130.227.255.137
1:50	atm0-3–0.val10-core.dk.tele2.com	130.227.0.69

The upstream bulk TCP transfers (SCP in class 1:30) and the ping tests are performed on the middlebox or packet scheduler itself. The disturbing packet-stream (UDP packets in class 1:50), is performed from a machine behind the middlebox. The packet-stream is performed using the Linux packet generator kernel module `pktgen`. It functions at a level lower than the packet scheduler and thus, cannot be performed from the middlebox itself. In Chapter 7 we show that performing the ping tests from the middlebox itself introduces a timer granularity abnormality.

The classification scheme and class setup are shown in Appendix B.3 on page 146.

[3]The distinction between SSH and SCP is done using the Type Of Service (TOS) field, as both services uses the same TCP port number.

6.2.2 The Naive Approach

For our naive approach, where the rate reduction is fixed, we want to avoid wasting to much upstream bandwidth. It is unrealistic to handle the worst-case overhead, as this requires reserving 62% of the bandwidth for overhead (see Section 6.1.1). We are optimistic and choose only to handle the situation with upstream bulk traffic, that is large upstream packets.

The rate reduction is estimated in the following. First the fixed ATM cell header overhead is subtracted; The 512 kbit/s upstream capacity is reduced to approximately 463 kbit/s ($512Kbit/53 \cdot 48$)[4]. Then the per-packet overhead and padding overhead is estimated. The ADSL line in question uses RFC2684B (bridge mode including FCS) over a Virtual Circuit (VC) giving a overhead per packet of 28 bytes (see Table 5.3 on page 53). An upstream bulk transfer with 1500-byte packets on a 463 kbit/s line gives around 38 packets/s ($463kbit/1500bytes$). The padding overhead for the 1500-byte packet plus the 28 bytes of overhead (1528 bytes) is 8 bytes ($\lceil 1528/48 \rceil * 48 - 1528$). Thus, an overhead of 36 bytes per 1500-byte packet, giving a rate reduction of 10944 bits/s ($38pkt/s \cdot 36bytes$). For the interactive traffic we assume a packet every 150 ms, giving 7 packets/s. Our latency test uses 64-byte ICMP ping packet, as it consumes two ATM cells giving a payload overhead of 32 bytes ($(2 \cdot 48) - 64$), thus a rate overhead of 1792 bits/s ($7pkt/s \cdot 32bytes$). The total rate reduction needed is approximately 13 kbit/s ($10944 + 1792 = 12736bits/s$) to accommodate for expected the overhead. **The bottom line bandwidth is *450 kbit/s***.

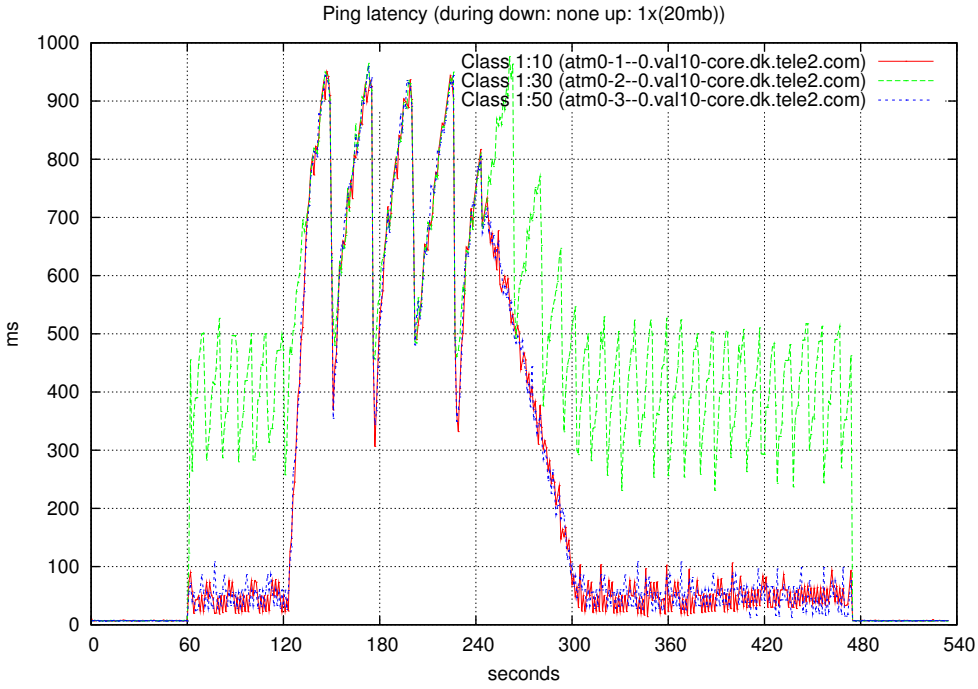
The class setup script and output from the script (which shows the rate left and the different overhead reductions) are listed in Appendix B.3.2 on page 147.

The graphs, in Figure 6.1 on the following page, demonstrate that queue control is achieved and lost again due to a change in traffic pattern. The test shows a single upstream (SCP) transfer of 20 MB, in class *1:30* started at time 60 seconds; 60 seconds into the upstream transfer (at time 120 seconds), a stream of 100 packets/s of 40-byte UDP packets[5] is generated into class *1:50* (to simulate the smallest TCP/IP packet). A stream of 100 packets/s is not very extreme, as noted in Section 6.1.1, 148 ACK packet/s are required for a 2 Mbit/s download.

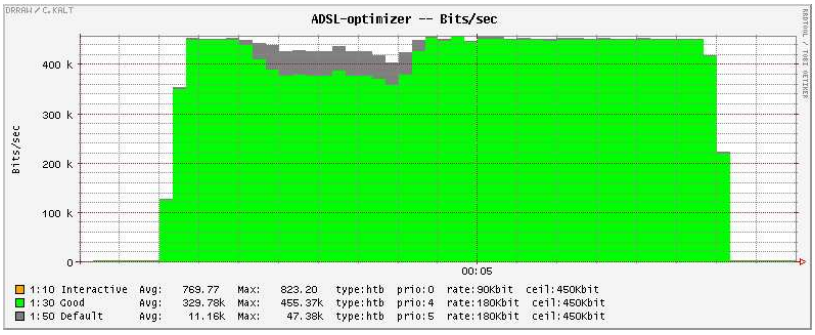
From the latency graph 6.1(a) it is clear that queue control is obtained, before time 120 seconds, as latency in different classes are clearly separated. At time 120 seconds the 100 packet/s UDP stream is introduced. The UDP stream is running for 120 seconds and thus finishes at 240 seconds. The graph 6.1(a) shows that queue control is first attained again at 300 seconds, thus it takes 60 seconds to gain queue control again. During queue control, we see that the latency in class *1:30* is bounded to around 500 ms, as we would expect from a full FIFO buffer size of 28125 bytes at 450 kbit/s.

The UDP stream only uses a small part of the bandwidth illustrated in Figure 6.1(b) showing the bandwidth used in the different classes. As expected with our overhead theory, there is a small drop in the achieved/total throughput during the UDP stream, because the payload bandwidth is reduced by the overhead introduced by the UDP stream. We have verified that our middlebox queueing system does not drop any packets.

[4]The ATM overhead is calculated and subtracted automatically by our scripts.
[5]Generated with pktgen kernel module; 12000 packet spaced by 10ms; 1s/10ms = 100 pkt/s.



(a) Latency in each class



(b) Throughput for each class

Figure 6.1: Naive overhead approach, queue control is achieved through reducing the rate to 450 Kbit/s. The queue control is lost when the traffic pattern changes.

6.2.3 The Accurate Overhead Modeling

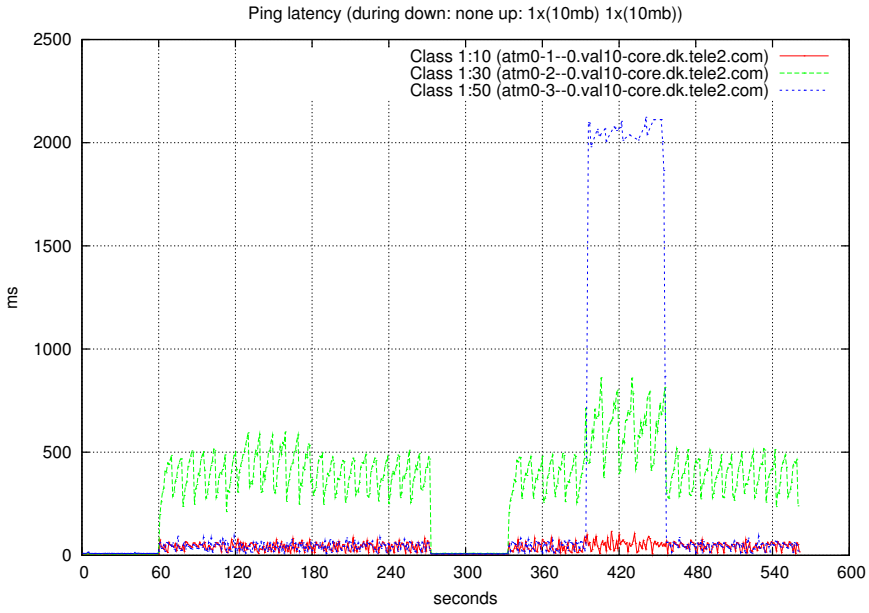
The queue control test in Figure 6.2 shows the effectiveness of our accurate overhead modeling, which included our kernel and userspace (tc) program modifications. The graphs show continuous queue control even in extreme situations with up to 1000 packets/s.

The tests consists of two phases, both with a 10 MB upstream bulk transfer in class *1:30*: (1) In phase one, a test (similar to the one shown in Figure 6.1) is performed with a 60 seconds disturbing traffic stream of 100 packets/s in class *1:50*. (2) In the second phase an excessive traffic stream of 1000 packets/s for 60 seconds is introduced into class *1:50*.

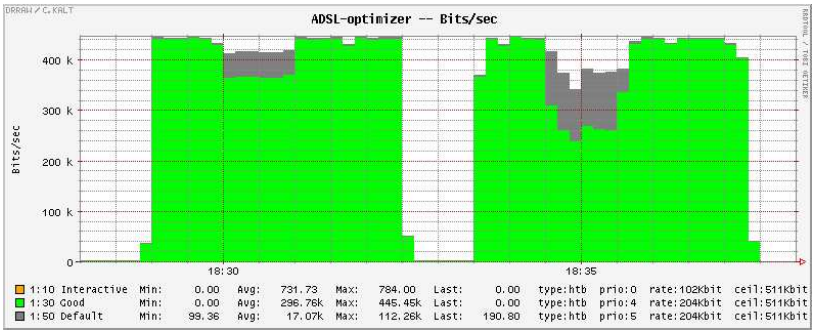
The latency graph 6.2(a) clearly shows that continuous queue control is achieved, as the latency of each class is kept separated. During phase one, in the period of the 100 packets/s (120 to 180) the latency of class *1:30* is slightly raised. This is expected as the lowest throughput achieved for the upstream transfer was 356 kbit/s, thus a FIFO buffer size of 28125 bytes can introduce a delay of 632 ms ($28125 bytes / 356 kbit/s$), the highest measured was 598.2 ms. During phase two, the latency in class *1:30* is raised further due to the same phenomenon. The lowest throughput of the upstream transfer was 252 kbit/s thus a maximum delay of 893 ms can be expected, the highest measured was 860 ms. The maximum expected delay at a given speed is not exceeded at any time, which shows that we are in control of the queue.

The latency in class *1:50* is low in phase one, because the class is not backlogged as 100 packets/s is not excessive and does not violate the assured rate of the class (including overhead 84.8 kbit/s). The 1000 packets/s, during phase two, are clearly excessive with 848 kbit/s including overhead ($106 bytes \cdot 1000 packet/s$). Thus class *1:50* experiences a very high latency and a high drop rate. We have verified that our middlebox queueing system only drops packets in class *1:50* during phase two. From the packet/s graph in Figure 6.2(c) we see that only 241 packets/s are allowed through class *1:50*, which, including overhead, corresponds to 204 kbit/s ($241 packet/s \cdot 106 bytes$) at the ATM layer. This matches the assured rate of 204 kbit/s in class *1:50* (see rate on Figure 6.2(b)), indicating that our overhead calculations and implementation are correct and accurate. Thus we have achieved our goal of avoiding unnecessary waste link capacity when achieving queue control.

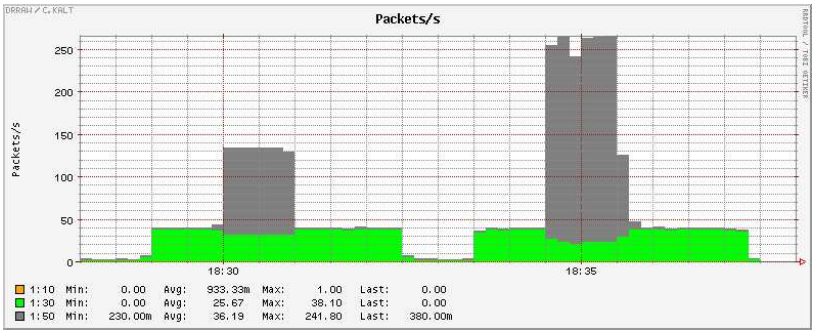
A closer look at the latency for our high-priority packets (class 1:10) show a average latency of 44 ms, which satisfies the average delay requirements of our service classes (described in Section 4.4). However, the delay varies between 7.9 ms and 116 ms giving a delay variance of 108 ms. The delay variance exceeds the delay jitter requirement of the variation-sensitive service classes (50 ms). This delay variance is not satisfying.



(a) Latency in each class



(b) Throughput for each class



(c) Packet/s for each class

Figure 6.2: The accurate overhead modeling. Full queue control even with extreme traffic patterns. First phase 100 packets/s, second phase 1000 packets/s (maintaining queue control, and drops the correct packets)

6.3 Summary

We have demonstrated the effect and magnitude of the ADSL link layer overhead. We show a naive approach for achieving queue control, where the bandwidth is simply reduced a fixed amount to compensate for the overhead. The bandwidth had to be reduced by 62% to handle the worst-case situation. Choosing only to handle an average case packet alignment overhead together with a maximum of 300 packet/s, would require a bandwidth reduction of 33%. The naive approach violates our goal of avoiding to waste link capacity.

To achieve queue control, we modify the Linux Traffic Control system to perform accurate link layer overhead modeling. We demonstrate that our implementation is accurate and attains continuous queue control during all traffic patterns. Furthermore, the solution achieves our goal of avoiding unnecessary waste link capacity when achieving queue control

Although, we achieve a satisfying average delay of 44 ms for our high-priority packets, we are not satisfied with the delay variance or delay jitter of 108 ms, as it exceeds the delay jitter requirement of the variation-sensitive service classes. The following chapter investigates the achievable delay bounds and delay variation for high-priority packets.

Chapter 7

Packet Scheduling and Delay Bounds

The entire focus of the industry is on bandwidth, but the true killer is latency.
— Professor M. Satyanarayanan,
keynote address to ACM Mobicom 1996 [28]

In Chapter 6 we demonstrate that queue control was achieved and that we achieved a satisfying average delay. However, we also observed a delay variance that exceeded the delay jitter requirement of the variation-sensitive service classes. In this chapter we evaluate the achievable delay bounds and delay variation for high-priority packets using the HTB packet scheduler. The aim is to determine, how close to an optimal packet scheduling we can get, and what delay the chosen scheduler introduces. As in Chapter 6 the focus is on upstream queue control.

7.1 Queue Test Setup

The queue setup used is the same as in Section 6.2.1, except for one important difference. The ping tests are not performed from the middlebox, but performed from a machine behind the middlebox. The traffic generator is still performed on a seperate physical machine also behind the middlebox.

7.2 Expected Delay Bounds

As basis for comparison when evaluating the quality and precision of our modifications to the packet scheduler, we need to determine the achievable delay bounds with an optimal scheduler.

An important property of the packet scheduler is it cannot preempt an (IP) packet transmission by another (e.g., high-priority) packet. This makes the transmission delay interesting, as a high-priority packet is likely to be delayed by the transmission time of another (low-priority) packet. Packet preemption is possible in ATM networks at the cell level, but this would require several ATM Virtual Circuit (VC)s (as ATM

Transmission delay on ADSL and upstream delay bound				
Downstream capacity:		$2,000,000$	Kbit/s	
Upstream capacity:		$512,000$	Kbit/s	
Baseline delay:		8.5	ms	
Packet size		Transmission delay (ms)		Upstream delay bound
IP	+ATM	Downstream	Upstream	(Incl.baseline)
36	53	0.21 ms	0.83 ms	9.33 ms
84	106	0.42 ms	1.66 ms	10.16 ms
500	583	2.33 ms	9.11 ms	17.61 ms
1000	1113	4.45 ms	17.39 ms	25.89 ms
1500	1696	6.78 ms	26.50 ms	35.00 ms

Table 7.2: Transmit delay on a 2 Mbit/512 Kbit line.

guarantees in-order delivery of cells per VC). Unfortunately, we cannot change the number of ATM VCs in the ADSL modem and on the DSLAM, as it is determined by the ISP.

We have performed a number of tests to evaluate the delay bound on an unused ADSL connection, these tests are shown in Appendix A.1 on page 128. On an unused ADSL connection, we find that the access link (the ADSL connection plus equipment) dominates the delay bound for Internet access. The findings are; (1) that the processing delay in the ADSL modem determines the lower delay bound, and (2) the upstream packet transmission delay is the main contributor to the delay and is dependent on the packet size as described in Section 3.4. (3) The delay between our testhost and the BBRAS is small compared to the rest of the delay. The transmission delay for different packet sizes on a generic 2 Mbit/512 Kbit ADSL connection, are shown in table 7.2[1].

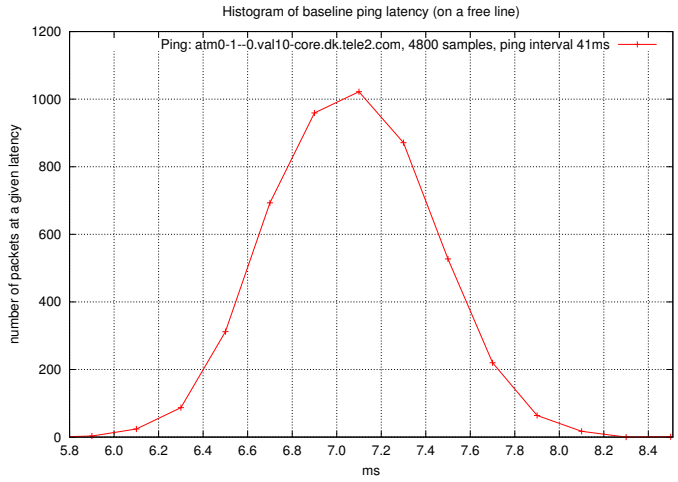


Figure 7.1: Histogram of the ping latency between the testhost behind middlebox and the BBRAS on a clean ADSL line. (4800 samples)

With an optimal packet scheduler without packet preemption, we expect that transmission of a high-priority packet must wait for a single full sized MTU packet of 1500

[1]The packet sizes are adjusted to account for the ATM header overhead and frame alignment, but not for the AAL5 overhead.

bytes in the worst case. This is a transmission delay of 26.5 ms on 512 kbit/s connection according to table 7.2.

We also need to incorporate two other delay components in our measurements; the processing delay in the ADSL modem and the path to the measurement point. To account for these two delay components, we perform ping latency test on the unloaded system, from which we determine our *baseline* RTT delay. We have chosen the closest IP measurement point, which is an IP-address on the BBRAS router. The distribution of the ping latency is illustrated in form of a histogram, in Figure 7.1. The ping packet size was 84 bytes. The average delay is 7.06 ms, and there is only a variation of 2.59 ms between the minimum delay of 5.89 ms and the maximum delay of 8.48 ms. Using the maximum ping delay, we get a delay bound of approximately *35 ms* ($26.5ms + 8.5ms$) for our given test environment.

We will in the next section show, that the granularity and allowed bursts of the specific scheduler increases this delay bound.

7.3 Real Delay Bounds

In this section we look at the delay bounds for the high-priority packets (class 1:10). We evaluate the achievable delay bound by performing some practical latency tests through our middlebox. The purpose is to determine how close to an optimal packet scheduling we can get, with the practical implementation of the HTB scheduler under the given Linux operating system.

This evaluation is (only) performed with the HTB scheduler, because our overhead patch is only fully implemented for HTB. We show that the HTB implementation introduces a large delay jitter due to some optimizations in the code and that removing this optimization feature helps to achieve a better jitter control. We also show that the granularity of the timer mechanisms in Linux influences the accuracy of our delay bound.

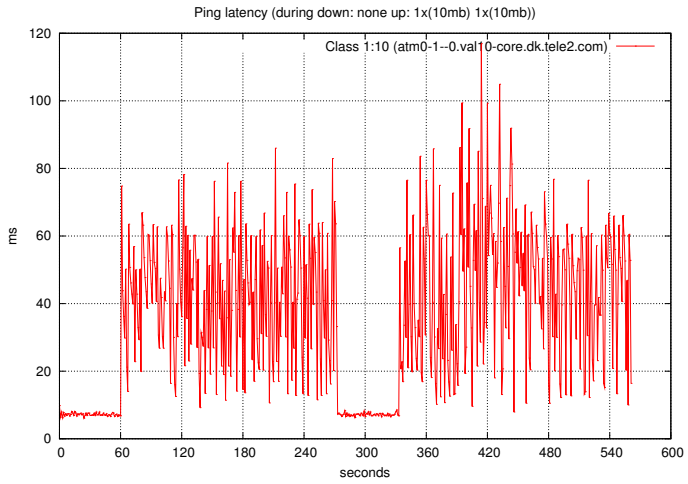


Figure 7.2: Latency in class 1:10, from the test shown in Figure 6.2(a).

We are not satisfied with the delay variation for the high priority class *1:10*, during the tests performed in Chapter 6. Figure 7.2 shows a zoom of the latency in the high priority class *1:10*, during the test shown in Figure 6.2. The graph clearly shows that

we exceed the expected (upstream) delay bound of 35 ms. The average ping latency was 44 ms, during the two upstream data transfers, which is only 9 ms larger than the expected delay bound. The problem is mainly the large delay variation or delay jitter, that varies between 7.9 ms and 116.9 ms, which is too high to satisfy the jitter requirements of the variation-sensitive service classes.

In the rest of this section we describe our improvement of this delay bound to an acceptable level still using the HTB scheduler.

7.3.1 Hysteresis

As part of our experimental work, we found a problem in the HTB implementation. To explain this problem, some basic knowledge of token buckets and the dual token bucket algorithm is needed. We assume that the reader is familiar with the token bucket algorithm[2], and choose to explain the color scheme of the dual token bucket algorithm briefly below, as it is the source of the problem.

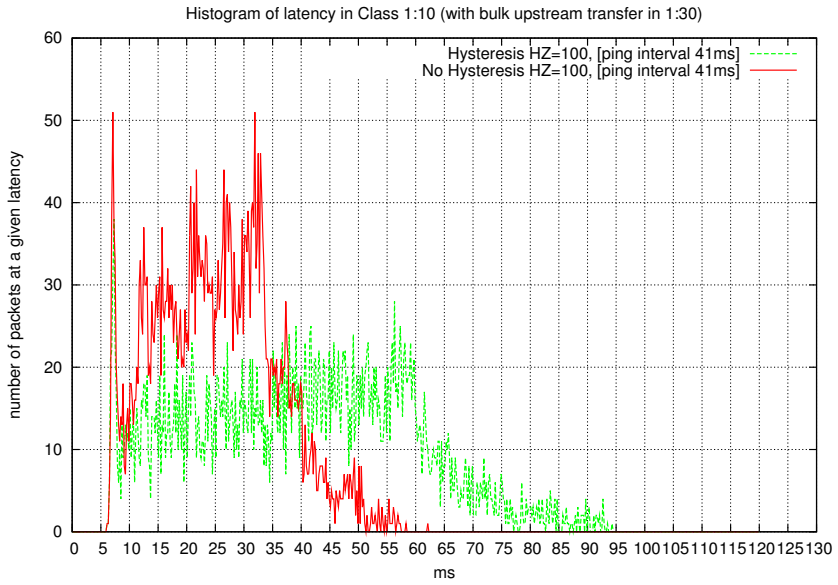


Figure 7.3: Histogram of the latency achieved with and without the hysteresis code in HTB. The ping test is performed during a upstream bulk transfer in class 1:30. (4800 ping samples for both).

Each class in HTB is defined in terms of a token bucket regulator as implied by the name of the scheduler HTB – Hierarchical Token Bucket. To be more precise the algorithm used is a dual token bucket algorithm [44, 78], which is often referred to as Two Rate Three Color Marker (trTCM) [44][3]. The trTCM is based on two token bucket regulators, a *peak* and *committed* regulator, which is used to mark a traffic flow with three colors (green, yellow, or red). The token bucket regulators are defined as two rates and their associated burst sizes (token bucket depths). This corresponds to HTB's class parameters; `rate` and `burst` for the committed token bucket and `ceil` and `cburst` for the peak token bucket. A traffic flow is marked red if it exceeds the

[2]If the reader is not familiar with the token bucket algorithm, we refer to some excellent explanations in [76, 78] and the original leaky bucket article [77].

[3]We refer to the book [78] for a good description of the trTCM algorithm.

peak regulator, else it is marked either yellow or green depending on whether it exceeds or does not exceed the committed regulator. The colors are called modes in the HTB implementation and used as; red = "cannot send", yellow = "may borrow", green = "can send".

The HTB implementation uses an "optimization" that reduces the number of mode (or color) recalculations and thus lowers the CPU usage of the scheduler. According to the author Martin Devera this gives a (CPU) speedup of 15% due to fewer mode changes[32]. The "optimization" adds a hysteresis of (the token bucket) burst size to the mode computations, and thus postpones the mode change. We refer to this as the *hysteresis* code. This means that the precision of the token bucket regulator is lost, which then affects the latency. This speed optimization is not relevant to our setup as we are scheduling packet on a rather slow link. We have not investigated (and will not comment on) whether this lower CPU usage is necessary on high speed links.

The latency achieved with and without the hysteresis code is shown in Figure 7.3. The latency histogram clearly shows a significant improvement in the achieved latency, without the hysteresis code. The tests are performed during a upstream bulk transfer in class 1:30 and 4800 ping samples in class 1:10. During this bulk upstream delay test, with the hysteresis code enabled we see an average delay of 39.7 ms, and a delay variance between 6.10 ms and 94.2 ms. Without the HTB hysteresis code we see an average delay of 25.2 ms, and a maximum delay of 62.0 ms. In Figure 7.3 we also see a spike at the lowest achievable delay and at around 35 ms, and the graph slowly decreases from this point. The spike around 35 ms is promising as it corresponds to the estimated delay bound for our test environment. We have not quite reached the estimated delay bound; 82.6% of the samples are below or equal to the delay bound of 35 ms. We also note that 96.3% are below or equal to 45 ms (A possible cause is an OS timer granularity of 10 ms).

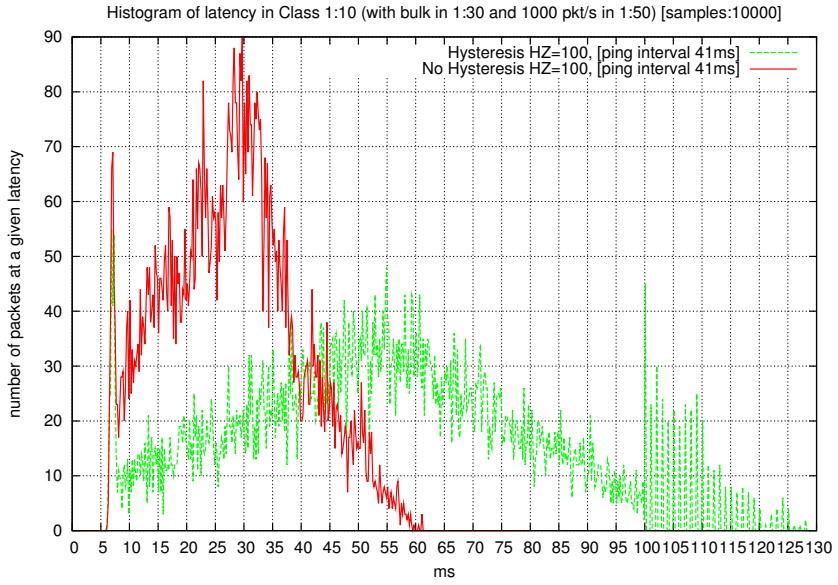


Figure 7.4: Histogram of the latency achieved with and without the hysteresis code in HTB. The ping test is performed with "full load", that is with an upstream bulk transfer in class 1:30 and 1000 packet/s in class 1:50. (10000 ping samples in both tests).

In Figure 7.4 we show the latency during "full load" in all classes. The latency with the hysteresis code is increased while the delay variance are maintained without

hysteresis. The load is 1000 packet/s in class 1:50 and an upstream bulk transfer in class 1:30. Without the HTB hysteresis code we see a maximum delay of 61.0 ms, which show that we have maintained the delay variance.

With hysteresis the largest delay variance is increased to 128 ms. This indicates that the delay variance or jitter introduced by the hysteresis code increases as the number of saturated classes increase. This is consistent with the imprecise mode changes, which allow each class to exceed their limits with the (token bucket) burst size. A burst of 2000 bytes correspond to 2226 bytes on the ATM layer, which equals a delay of approximately 34 ms. These 34 ms correspond to the increased delay, with the hysteresis code, between Figure 7.3 with a maximum on 94 ms and Figure 7.4 with a maximum delay on 128 ms ($128 - 94 = 34ms$). There is a difference of 32 ms between the 94 ms and our maximum delay of 62 ms without hysteresis, which is fairly close to the 34 ms allowed extra burst of one class. This indicate that the hysteresis code allows each saturated class to add an extra delay of 34 ms, because each class can exceed their limits with their burst size.

The strange spikes, in Figure 7.4, starting at 100 ms are due to the resolution of the measurement tool (no decimals are reported above 100 ms and we use a histogram resolution of 0.2 ms).

7.3.2 Timer Granularity

The precision of HTB scheduler is affected by the granularity of the Linux timers. The timers in Linux (kernel 2.4.x) are triggered every 10 ms and is defined through the kernel HZ value, which is set to 100 in 2.4.x kernel and 1000 in 2.6.x kernels. As we are using 2.4.x kernels we have a granularity of 1/HZ seconds, that is 10 ms. The Linux timers are used for a lot of different functions in the operating system. It is for example used by the CPU scheduler to preempt processes which have been running for more than 1/HZ seconds. A granularity of 10 ms might seem coarse, but preemptive hardware interrupts, like the ones generated from the keyboard and network packets, makes the system respond much more fine grained. When shaping a packet flow the packet scheduler needs some kind of timer, since packets need to be delayed to conform to the configured traffic profile. It cannot rely on another packet generating a timely hardware interrupt. Although a granularity of 10 ms is not optimal for a packet scheduler, the Linux packet schedulers (HTB, CBQ, police and TBF) still makes use of them, as there is no alternative in the official kernel.

Some kernel patches implementing high-resolution timers in Linux[4] exist, but they are not part of the official kernel. We have not experimented with these patches, because they only provide a framework for better timers and we would have to change the implementation of the packet schedulers to make them fit and use this framework.

We illustrate in Figure 7.5, how HTB is affected by this timer granularity, by performing a ping test from the middlebox during a stream of small packets in another class (1:50). The idea behind running the ping program on the packet scheduling middlebox is, that the ping application uses the sleep system call, which is also affected by the timer granularity. The ping application will wake up at exactly the start of a timer tick, thus the ping packet will be delayed by the HTB timer a full timer period. Figure 7.5 displays a very distinct spike at 17 ms (most occuring number), which corresponds to the baseline delay (average of 7.06 ms) plus 10 ms. This is a clear indication, that the HTB scheduler is affected by the timer granularity problem.

[4]KURT: http://www.ittc.ku.edu/kurt/, and The high resolution timers project: http://high-res-timers.sourceforge.net/.

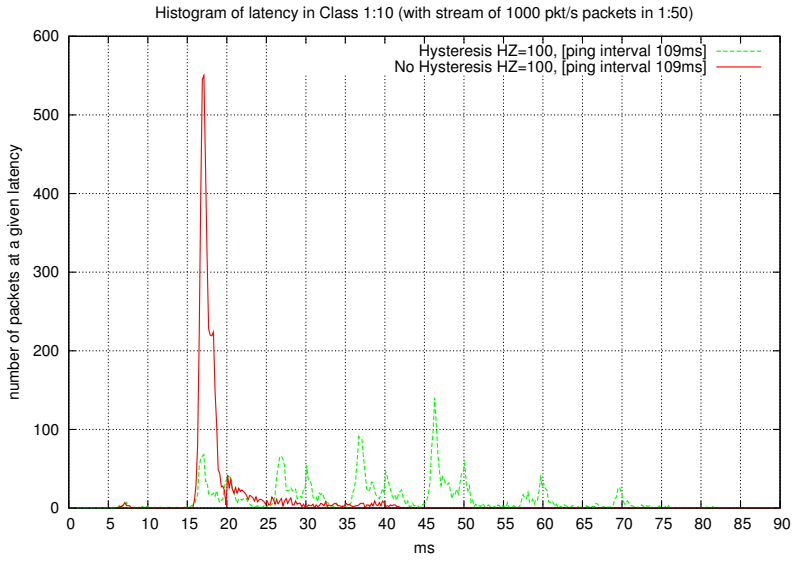


Figure 7.5: Histogram of ping latency test performed from the middlebox, to illustrate Linux' timer granularity. Latency in class 1:10 during a stream of 1000 packets/s in class 1:50. (HZ=100 which equals a 10 ms granularity) (4800 samples).

A stream of 1000 packets/s are transmitted in class 1:50 and thus a hardware interrupt is generated every 1 ms. It seems that HTB does not consider hardware interrupts to clock out the high-priority packets waiting for a timer. This seems like an obvious optimization to be made to the HTB code. Thus, when scheduling packets with HTB we can expect an added delay of 0 to 10 ms, as packets from remote hosts do not hit the exact start of the timer ticks (like the local ping packets). This situation only occurs when HTB is loaded, since obviously no delay is added if the packet can be transmitted at once avoiding the use of the timer mechanism.

To illustrate the bad latency performance achieved using the hysteresis code when scheduling small packets, the histogram in Figure 7.5 also include the latency achieved with the hysteresis code enabled. With the hysteresis code we also see spikes at 10 ms intervals. The maximum delay was 88 ms and 38 ms on average. This is clearly not satisfying.

The histogram in Figure 7.5 also show some delays up to 42.8 ms (max delay) and down to 6.22 ms. To explain these delays, we show the latency vs. time graph in Figure 7.6. The graph shows two very distinct spikes, which are exactly 5 minutes apart. The reason for these spikes was a cronjob, which was run every 5 minutes, on the host generating the packet stream (1000 packet/s). This cronjob caused the packet generator (pktgen) to make small pauses. These small pauses was enough for the token bucket system to allow a burst of packets after the pause. During the disturbance of the packet generator some of the high priority packets are allowed directly through (low delay), while others where delayed up to 42.8 ms. The interesting thing about this maximum delay is that it corresponds to the allowed burst size of the token bucket. We have configured the token bucket burst size (bucket depth) to be 2000 bytes, which due to our overhead patch correspond to 2226 bytes "on the wire" ($\lceil(2000)/48\rceil * 53$). The 2226 bytes represent a transmission delay of 34.78 ms at 512 kbit/s. This delay and the upper bound baseline latency delay (approximately 8 ms) gives 42.78 ms which correspond to the maximum delay of 42.8 ms. This shows that our delay bound is affected by the token bucket burst size and not only the 1500-byte MTU packet size.

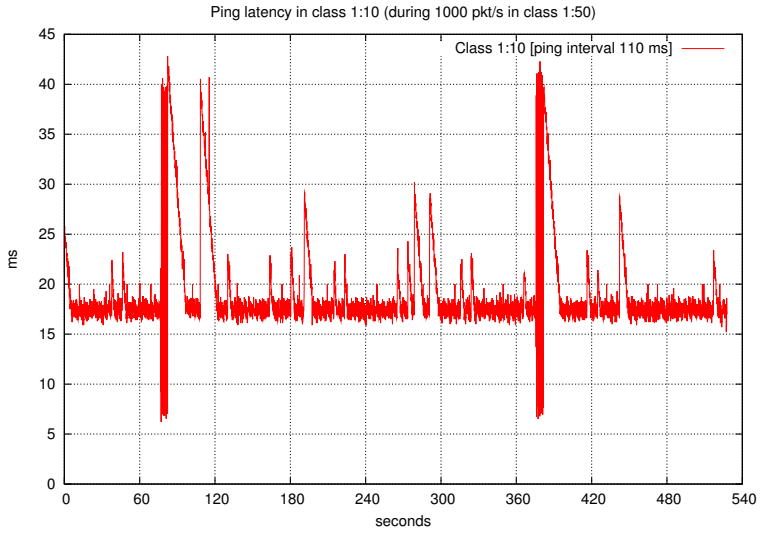


Figure 7.6: Ping latency test performed from the middlebox, to illustrate Linux'es timer granularity. HTB no hysteresis. Latency in class 1:10 during a stream of 1000 packets/s in class 1:50. (4800 samples)

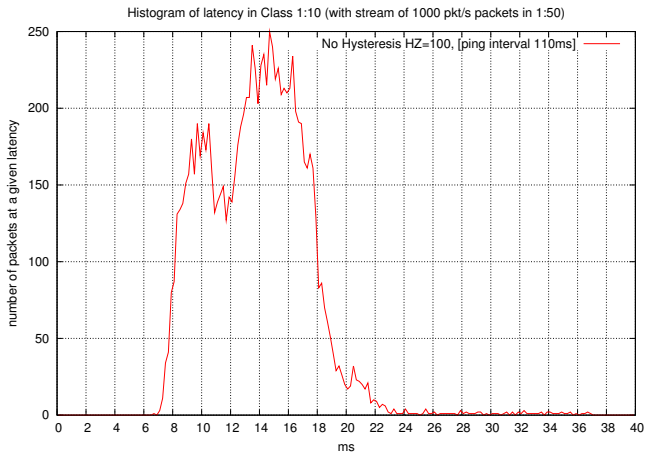
When the ping test is performed from a host behind the middlebox, the histogram of Figure 7.5 should look like Figure 7.7(a). As expected a delay between 0 to 10 ms delay is added as packets from the remote host do not hit the exact start of the timer ticks, like the ping packets on the middlebox from the previous test. How the clock/timer of the two Linux boxes drift can be seen in Figure 7.7(b). The timers are clearly not in sync. The test is performed with 10000 samples to catch more timer drift periods. We have tried to start the ping test and the traffic generator at the same time, to illustrate the first burst allowed by the traffic generator. The cronjob on the machine running the traffic generator has been removed. It should be noted, that running the ping test and traffic generator on the same machine is not recommended, as the traffic generator affects the granularity of network transmissions.

7.3.3 Improving Granularity

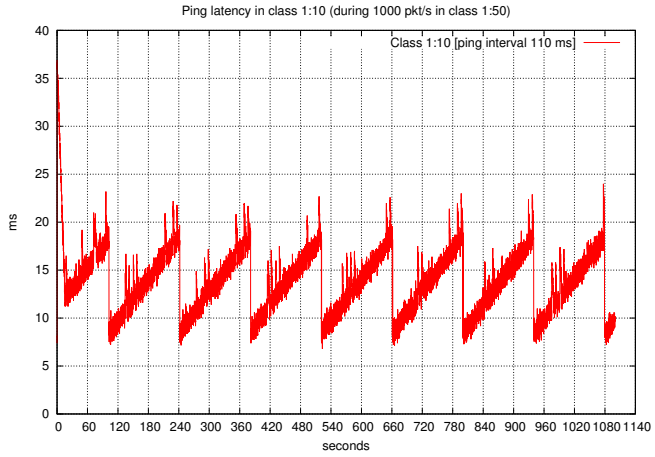
An easy way of achieving a better timer granularity is simply to change the kernel HZ value and recompile the kernel. We have used the same kernel and increased the HZ value to 1500, which gives a granularity of 0.666 ms (1/1500 seconds). The HZ increase was limited by some restrictions in the kernel code. The maximum value would have been 1532, as the HZ value was required to be below 1536 (see `include/linux/timex.h`) and divisible by 4 (see `net/sched/estimator.c`[5]). We choose 1500 somewhat arbitrary.

The effect of increasing the timer granularity is shown in Figure 7.8. The test is performed with a stream of 1000 packets/s in class 1:50. The result is plotted together with the data from Figure 7.7(a). The graph clearly show an improvement in the delay bounds. With HZ=1500 the average delay was 10.92 ms and 96% of the ping delays were below 15 ms. For larger packets we do not see the same profound improvement in latency when increasing timer granularity. With "full load" the two measurements (with the HZ value set to 100 and 1500) more or less overlay each other, which can be

[5]This restriction have been removed by Thomas Graf in kernel 2.4.28 (we used kernel 2.4.27).



(a) Histogram



(b) The periodic changes in the slope shows how the 10 ms timers drift between the two Linux boxes.

Figure 7.7: Ping latency performed from a host behind the middlebox (HZ=100, no hysteresis). Latency in class 1:10 during a stream of 1000 small packets/s in class 1:50. Due to the timer granularity a 0 to 10 ms delay is added.

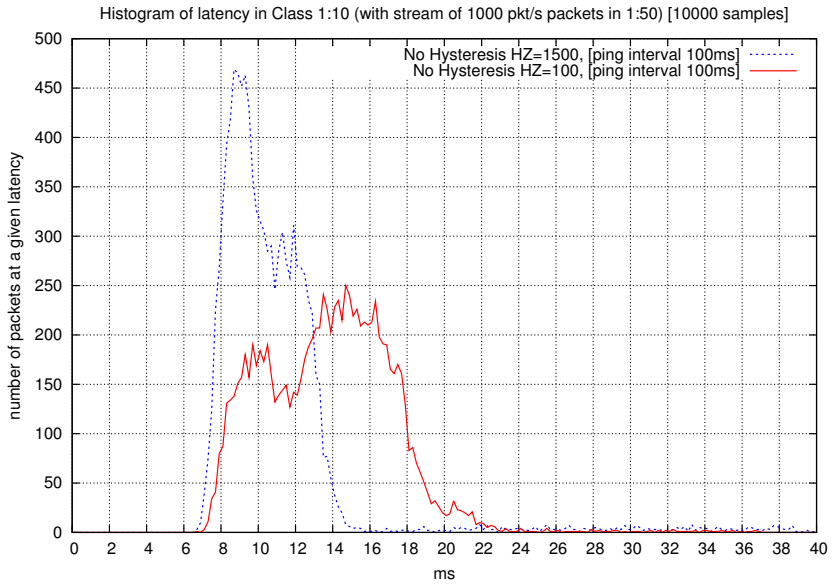


Figure 7.8: Demonstrating the effect of increased timer granularity, 100 vs. 1500 HZ. Histogram of the latency achieved with HZ=100 and HZ=1500 (without the hysteresis code). The ping test is performed during a stream of 1000 packet/s in class 1:50. (10000 ping samples in both tests).

seen from Figure 7.9.

The maximum delay bound observed was 62 ms, which is larger than the expected delay bound of approximately 35 ms for our specific environment. We have not been able to determine if this delay bound is due to an extra delay in HTB scheduler or due to an extra packet being processed in the ADSL modem. It could be that we actually "lose" queue control for one single packet due to bursts. The difference between the expected delay of 35 ms and the measured maximum of 62 ms equals 27 ms, which actually correspond to the transmission delay of one full sized MTU packet (see the transmission delays in Table 7.2).

7.4 Summary

With our tuning of the HTB packet scheduler we have achieved a delay bound, where high-priority packets have to wait for at most two full sized 1500-byte MTU packets. This is close to the optimal packet scheduler, as 75% during "full load" is scheduled within 35 ms, which was the expected delay bound of an optimal packet scheduler. An additional 10 ms can be caused by the OS timer granularity giving an expected delay bound of 45 ms, which constituted 92% of latency measurements during "full load" at 100 HZ (Figure 7.9 with 10000 samples).

We believe that we have achieved an excellent delay bound result, especially taking into account, that our packet scheduler achieves queue control by modeling the ADSL link layer overhead. A maximum delay of 62 ms for our high-priority packets is an excellent result, compared to the maximum upstream delay of approximately 3300 ms observed in Chapter 3 on an unmodified ADSL connection.

With these delay bounds we can fulfill the average delay bound of all service classes

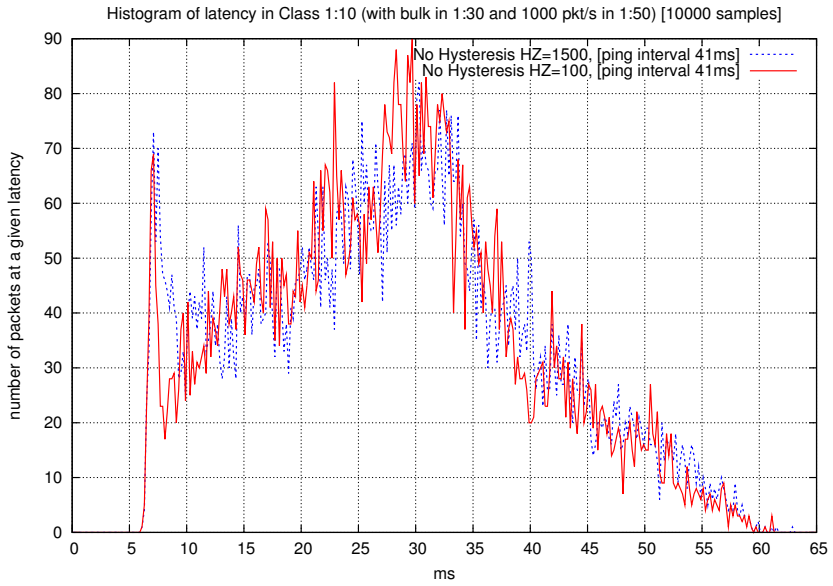


Figure 7.9: "Full load" 100 vs. 1500 HZ. Histogram of the latency achieved with HZ=100 and HZ=1500 (without the hysteresis code). The ping test is performed during "full load"; a stream of 1000 packet/s in class 1:50 and a bulk transfer in class 1:30.

and thus achieves our goal of supporting delay-sensitive applications. We cannot completely support the jitter requirements of the two variation-sensitive or real-time service classes. We have achieved upstream delay jitter of approximately 56 ms ($62ms - 6ms$). This is close to the requirements of the real-time service classes which had one-way end-to-end delay jitter demand of 50 ms. It should be noted, that the delays shown in this chapter is not end-to-end but to the closest IP address (on the BBRAS) and upstream only. Thus, other links on the path are likely to contribute further to this delay bound.

In this chapter we have focused on upstream delay. We also expect the downstream to introduce some additional delay, as noted and illustrated in several figures in Chapter 3. Thus, we expect that full utilization of the downstream connection is likely to increase the delay further. This will be illustrated in the next chapter.

Chapter 8

ACK-prioritizing and Full Utilization

This chapter demonstrates that we are able to achieve full link utilization of both upstream and downstream traffic by prioritizing ACK packets.

We will also illustrate that full utilization of the downstream link introduces an extra queueing delay on the downstream path. This delay was expected and has been noted and illustrated in several figures in Chapter 3. Solving this downstream delay problem requires methods for indirect downstream queue control, which we have chosen not to focus on in this thesis. However, we will show some simple methods for reducing this delay.

8.1 Queue and Filter Setup

The purpose of this chapter is to evaluate the effects of ACK-prioritizing. We want to determine whether ACK-prioritizing on the upstream link helps achieve full utilization of the downstream capacity, while the upstream connection is saturated. To demonstrate the effect of ACK-prioritizing the traffic is organized into four traffic classes or queues:

- (1:10) A queue for interactive traffic.
- (1:20) A queue for ACK packets.
- (1:30) A queue for upstream bulk transfers.
- (1:50) A default queue/class.

The interactive class (1:10) is included to evaluate the delay imposed on delay-sensitive applications during full utilization of the connection. The default class (1:50) is used for determining the delay imposed on a low priority class.

The setup resembles the queue setup described in Section 6.2.1, which was used for evaluating our link layer overhead modeling. The main difference is the extension of an ACK service class. For simplicity all classes use a simple FIFO queue for packet scheduling. The FIFO queue buffer size is (specifically) set to 28125 bytes, like Section 6.2.1 on page 59.

The tests analyzed in this chapter have been performed with the `HTB_HYSTERESIS` option disabled and with a timer granularity of 1500 HZ on the middlebox. The tests have been run from a host behind the middlebox.

ACK rate calculations

Bandwidth allocation for ACK packets is handled in a special way. Basically, bandwidth is allocated to the ACK class first and then the remaining bandwidth is allocated between the remaining classes. Our scripts automatically calculate the bandwidth needed for ACK packets, if the user chooses to specify this as a parameter. The information needed is the downstream rate from which we calculate the amount of bandwidth required for ACK packets.

We should avoid reserving too much bandwidth for ACK packets, because there should be a reasonable amount left to the rest of the classes. We incorporate the delayed ACK factor into our calculations, to get a more realistic estimate of the bandwidth needed for ACK packets. We assume that most TCP implementations use a delayed ACK factor of two as recommended in [15, 36]. However, we also expect some of the situations described in Chapter 2 to arise where the effective delayed ACK factor is lower than two during a TCP connection. Therefore, we choose an ACK delay factor of 1.5, when calculating the ACK rate.

Formula 8.1

$$ACK_{rate} = \left(\frac{Downstream_{rate}}{Packet_{size}} \cdot ACK_{size} \right) / ACK_{DelayFactor}$$

The formula calculates the number of data packets on the downstream and multiply with the size of an ACK packet to get the rate used by ACK packets and then compensate for the delayed ACK factor. In our implementation we set the $Packet_{size} = 1500$ bytes, and thus assume that the $Downstream_{rate}$ is without ATM overhead, that is the maximum achievable IP throughput. The ACK_{rate}, is expressed including the overhead. Thus, the ACK_{size} is set to 106 bytes, which represents two ATM cells including ATM headers. The shell script function can be seen in Appendix B.6.1 on page 161.

It should be noted that the tests performed in this chapter have been performed with earlier version of the ACK rate calculation, which reserved more bandwidth to the ACK class than describe above (115 kbit/s). This higher reservation does not influence the test results, as the there is sufficient bandwidth left for the other classes. We have verified from the test data, that the ACK rate consumed in these controlled tests is less than with the above calculations. The above calculations would reserve 85 kbit/s (with a downstream of 1800 kbit/s) and the actual ACK rate used was only 69 kbit/s at maximum and 63.8 kbit/s on average. This real ACK rate shows that we were close to a delayed ACK factor of two, which should have resulted in an ACK rate of 63.6 kbit/s ($1800/1500 * 106/2$).

HTB class setup

The ACK rate is subtracted from the upstream ceil rate and the remaining bandwidth is distributed between the other classes by a percentage. All classes are allowed to utilize 100% of the ceil rate. This can lead to starvation, but we will ignore this as this setup is simpler.

parent:	1:1			
class:	1:10	1:20	1:30	1:50
class:	Interactive	ACKs	Bulk	Default
prio	0	1	4	5
rate	20%	special	40%	40%
ceil	100%	100%	100%	100%

Listing 8.1: Output from the HTB setup script

```
./htb_overhead_ack_test1.sh -i eth1 -o 28 -u 511 -d 2000

Setup information:
--------------------
The ATM/AAL5 overhead calculations are done by tc and kernel
This shows what throughput can be expected

Device                  : eth1
Link bandwidth          : 511 Kbit/s
Max payload bandwidth : 462 Kbit/s (subtracted fixed ATM overhead)
ATM fixed overhead      : 49 Kbit/s
Overhead per packet     : 28 bytes
Downstream bandwidth : 2000 Kbit/s (only calc ACK rate)
Reserved for ACK packets: 115 Kbit/s
Rate left for Classes   : 396 Kbit/s
```

The output from the HTB setup script displayed in Listings 8.1 shows the result of the ACK rate calculation and what is left for remaining traffic classes. As described above, the ACK rate calculations used by this script was different (as an earlier version was used). The ACK_{rate} is calculated to 115 kbit/s from a downstream capacity of 2,000 kbit/s, which was correct according to old calculations. The upstream rate is set to 511 kbit/s, which is 1 kbit/s less than the actual upstream rate. The rate left for other classes is 396 kbit/s which is allocated according to the above percentage. The HTB setup script itself is shown in Appendix B.4.2 on page 153.

Traffic Classification

For packet classification; SSH packets are classified into the interactive class *1:10*, ACK packets are classified into class *1:20*, SCP packets are classified into the bulk class *1:30*, and the rest goes into the default class *1:50*.

To measure the latency in each class, ping (ICMP) packets (to different IP-addresses) are classified into each class. The ping IP-addresses are chosen to be as close as possible to the ADSL connection (on the BBRAS). The ping table is based upon the table in Section 6.2.1 (for a explanations of the IP-addresses we refer to this section). The ping table has only been expanded by one entry, which is used for class 1:20. The IP-address is almost as close as the other IP-addresses. It is a router connected to the BBRAS.

Class	Name	DNS	IP
1:10	Interactive	atm0-1–0.val10-core.dk.tele2.com	130.227.0.81
1:20	ACKs	li53.adsl.tele2.cust.dk.tele2.com	130.227.255.138
1:30	Bulk	atm0-2–0.val10-core.dk.tele2.com	130.227.255.137
1:50	Default	atm0-3–0.val10-core.dk.tele2.com	130.227.0.69

We perform ACK packet classification based on the TCP ACK flag and packet size. We try to match pure ACK packet, but the categorization can potentially result in mis-categorization of very small data packets with the ACK flag set. When classifying ACK packets it is important also to match SACK packet, as they help clocking out new data as illustrated in Chapter 3. SACK packets is a TCP option which implies

that the size of the ACK packet increases. Thus, when matching ACK packets, this increased size should be taken into considerations. In these tests the standard SACK size was 72 bytes of which 12 bytes was due to a TCP timestamp option.

The filter rules are shown in Appendix B.4.1 on page 153 and the ACK filter rules in Appendix B.7.1 on page 170.

8.2 Basic ACK-prioritizing

The test performed consists of one continuous bulk downstream transfer of 90 Mb and two periods with one bulk upstream transfer of 4 Mb each. The result is illustrated in Figure 8.1 and show how full link utilization is achieved for upstream and downstream at the same time. This clearly shows that prioritizing ACK packet has the presumed and expected result of allowing us to utilize the downstream capacity even when the upstream is loaded. The graph shows an average downstream throughput of 1803 kbit/s for the complete period.

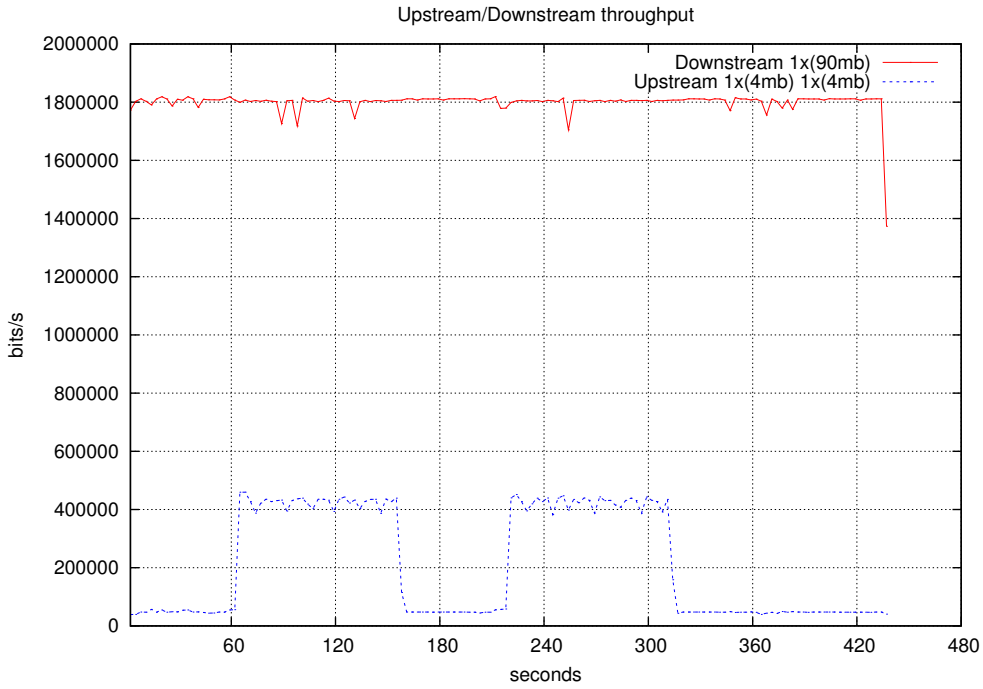


Figure 8.1: **Basic ACK-prioritizing**: Illustrating the effect of prioritizing ACK packet on the upstream link. We achieve full utilization of downstream throughput while utilizing upstream. On a 2Mbit/512kbit ADSL line from Tele2.

Full utilization of the downstream bandwidth introduces an extra delay, due to a queueing delay on the downstream path. This is expected from our evaluations in Chapter 3. We will refer to this delay phenomenon as the *downstream delay problem*. This is illustrated in Figure 8.2, which corresponds to the throughput graph in Figure 8.1. The latency graph clearly shows how all classes gets an added delay. We assume this added delay is caused by queueing on the downstream link, as we have documented, that we have upstream queue control. The latency in the bulk class is as expected, except for the added delay. The average latency in class 1:10 is approximately 134

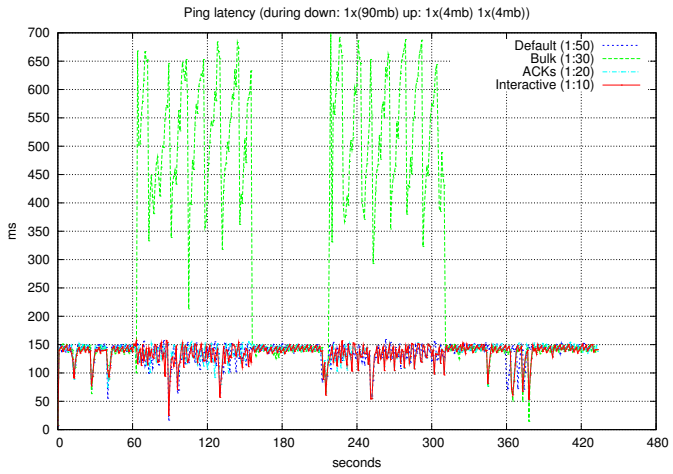


Figure 8.2: **Basic ACK-prioritizing**: Latency during test in Figure 8.1. This latency graph illustrates the downstream delay problem.

ms during the complete test run. This is not satisfying for the requirements of the interactive service class, which had a one-way average delay requirement of 100 ms. We can argue that we actually do satisfy the delay requirements of our interactive service class, because our measured latency is the round trip time delay. As this implies an allowed 100 ms delay in each direction giving a delay bound of 200 ms. Even though the delay bound theoretically satisfied the requirements of the service classes, we are not satisfied, because we know this can be done better.

The downstream delay problem does introduce an added delay, but it is worth noticing that this delay is significantly less than the delays observed due to the upstream queue in Chapter 3. This was one of the reasons, why we chose to focus on upstream queue control. Controlling the downstream queue requires techniques for indirect queue control. We believe that doing this in an effective and smoothed manor is a project on its own.

However, we will not completely ignore the downstream queueing delay. We will in the next sections show two simple methods to reduce the downstream queue.

8.3 Ingress Filtering

The first method for reducing the downstream queue is simply to drop incoming data packet on the downstream link and rely on TCP to back-off and reduce its sending rate. This is achieved using *ingress* filtering or policing. The ingress filter is implemented by simple token bucket meter, that drop packets if they exceed the configured rate. All packets on the downstream link is filtered through the ingress filter. With the ingress filter under Linux the packets are dropped before it enters the IP stack [48].

Our ingress setup script is given the rate including ATM overhead and subtracts the fixed ATM overhead before setting the ingress rate. This was done for ease of configuration. The ingress script can be seen in Appendix B.4.3 on page 156.

Listing 8.2: Output from the ingress setup script

```
./downstream_limit.sh -i eth1 -u 1950
```

```
Setup information:
--------------------
Device                  : eth1
Link bandwidth          : 1950 Kbit/s
Payload bandwidth       : 1766 Kbit/s (subtracted fixed ATM overhead)
ATM fixed overhead      : 184 Kbit/s
```

The downstream throughput is on average 1755 kbit/s during the entire test period, this is 50 kbit/s or 2.8% less than the throughput achieved in the first throughput test in Figure 8.1 (1805 kbit/s). The downstream throughput during the upstream transfer periods is not optimal and contains small spikes and drops. (The average throughput outside the upstream periods was 1775 kbit/s.)

As can be seen from the latency graph with ingress filtering in Figure 8.3(b) we have reduced the downstream delay problem in some situations. The latency is still higher than expected during the upstream transfer periods. This is probably caused by the downstream flow being more bursty during these periods, which causes downstream queues. These bursts can be introduced by the delay variance of our non-preemptive (upstream) scheduler, as ACK packets can be delayed behind data packets, after which the ACK packets can be sent in a burst.

8.4 Downstream Packet Scheduling

As mentioned in Section 4.3 we cannot achieve direct queue control on the downstream connection. This is because downstream packets reach our middlebox *after* they have travelled the downstream link. Thus delaying the data packets does not directly affect the sending rate of the downstream connection.

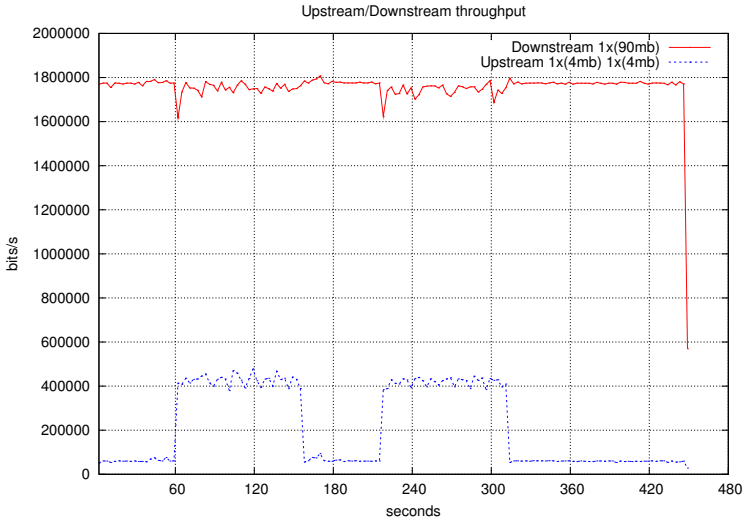
As a middlebox we can indirectly affect the TCP sending rate on the downstream connection, by delaying the data packets for the LAN machines. This will delay the generation of ACK packets, because the LAN machine does not generate an ACK packet before receiving the corresponding data packet. This implies that the rate, at which data packets is sent to the LAN machines, is indirectly communicated back to the data sender via ACK packets. This makes the sender back-off without dropping packets, like the ingress mechanism did.

The downstream packet scheduling is simply performed by using the same HTB setup script on the interface that transmits packets from the middlebox to the LAN. The packets are thus scheduled at the configured rate towards the host located on the LAN on which the tests is run.

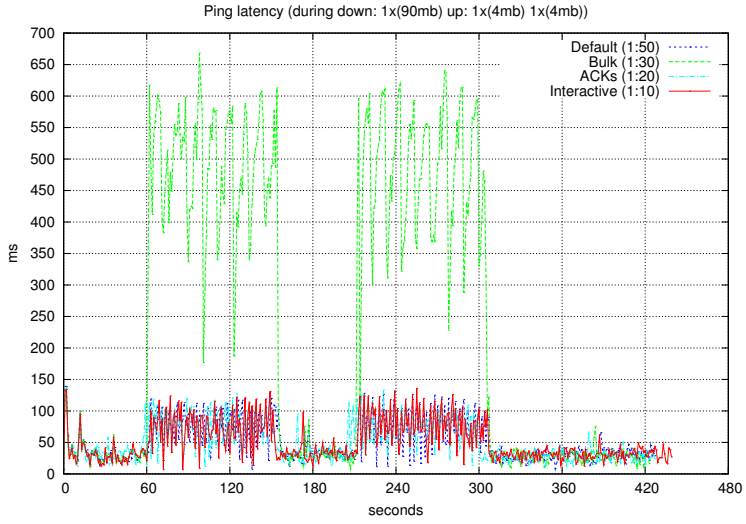
Listing 8.3: Output from the HTB setup script

```
./htb_overhead_ack_test1.sh -i eth3 -o 28 -u 1999 -d 512

Setup information:
--------------------
The ATM/AAL5 overhead calculations are done by tc and kernel
This shows what throughput can be expected

Device                  : eth3
Link bandwidth          : 1999 Kbit/s
Max payload bandwidth : 1810 Kbit/s (subtracted fixed ATM overhead)
ATM fixed overhead      : 189 Kbit/s
Overhead per packet     : 28 bytes
Downstream bandwidth : 512 Kbit/s (only calc ACK rate)
Reserved for ACK packets: 30 Kbit/s
Rate left for Classes   : 1969 Kbit/s
```

The output from the HTB setup script displayed in Listings 8.3 show the result of using the script for the downstream direction. The link bandwidth is set to 1999 kbit/s



(a) **Ingress filtering:** Throughput



(b) **Ingress filtering:** Latency

Figure 8.3: **Ingress filtering:** Ingress filtering 1950 Kbit/s, reduced to 1766 Kbit/s without ATM header overhead.

to emulate a bottleneck. The ACK rate is configured to carry the ACK packets from the upstream link.

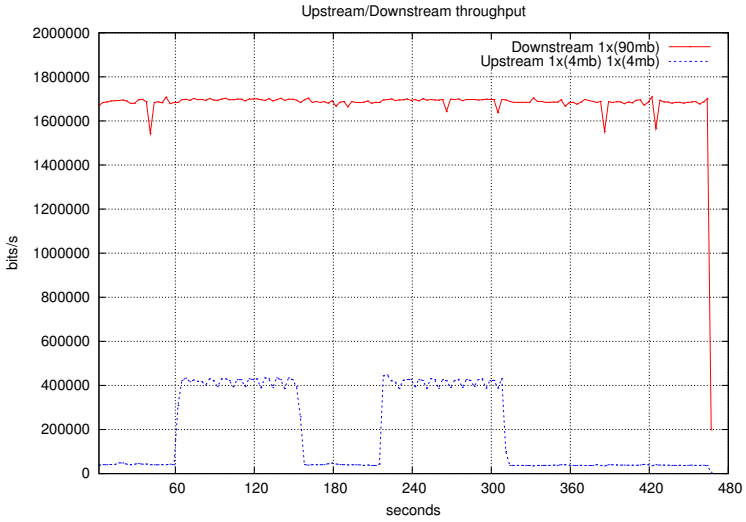


Figure 8.4: **Downstream Packet Scheduling:** Throughput.

The throughput graph is shown in Figure 8.4. The average downstream throughput for the complete period was 1687 kbit/s, which was lower than expected. We have not made further investigations into why the throughput was lower than expected. The 1687 kbit/s is 118 kbit/s or 6.5% lower than the throughput achieved in the first throughput test in Figure 8.1 (1805 kbit/s). Thus, with this downstream packet scheduling mechanism we waste more bandwidth than with the ingress method. This does not serve our goal of maximum link utilization. We believe this can be done better, but it is out of the scope of this thesis to investigate the issue in further detail.

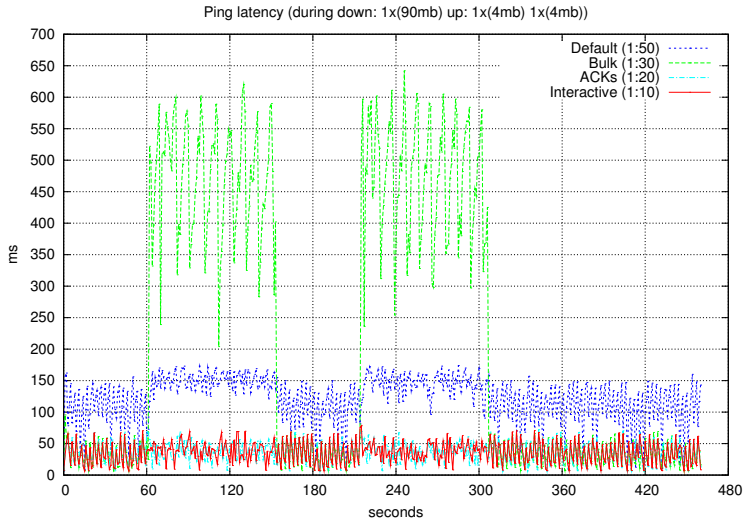


Figure 8.5: **Downstream Packet Scheduling:** Latency.

The latency graph in Figure 8.5 clearly show that we have maintained queue control during the entire test. The downstream transfer is HTTP traffic and is categorized into

the default class (1:50). The latency graph shows, how the bulk downstream delay is imposed on the default class (1:50). The graph also show that we maintain low latency for the interactive class (1:10) and for ACK packets (1:20) during the entire test. The average latency in the interactive class (1:10) is 35.74 ms and the highest is 78.4 ms. This is sufficient to satisfy the delay requirements of our interactive service class.

8.5 Summary

In this chapter we have shown that we are able to achieve full link utilization of both upstream and downstream traffic at the same time by prioritizing ACK packets. We have also shown that full utilization of the downstream link introduces an extra queueing delay, which was expected. With this extra downstream delay we can still satisfy the average delay requirements of our service classes, as they were defined as one-way delay bounds, as the added delay occurs on the downstream path. We know that it is possible to do better and show two simple methods for indirect downstream queue control, which reduce this delay and gives a better delay bound for our service classes, but also wastes downstream capacity.

For our high-priority class (1:10) we were able to achieve an average delay of 35.74 ms and a maximum delay of 78.4 ms, when using downstream packet scheduling to mitigate the downstream delay problem. This was achieved while the upstream connection was fully utilized and the downstream connection was 6.5% from maximum utilization. This is a promising result, but it does not completely satisfy our goal of avoiding to waste link capacity.

Solving the downstream delay problem in a controlled and smooth manner, and without wasting as much bandwidth as our simple solutions, is a project of its own. It requires more advanced methods for indirect downstream queue control, than presented in this chapter. Some of the methods and techniques have been discussed in Section 4.5.

Part III

Practical Solution

Chapter 9

Combining the Components

Part III concerns our practical solution. This chapter describes how the combination of components identified in Chapter 4 creates a practical and functional middlebox solution based on Linux. The choices made, when combining the components, are based upon and limited by the available tools under Linux and the restrictions of our real-world environment.

Our real-world setup, described in this Chapter, is based upon the dormitory Kollegiegården's production ADSL. An 8 Mbit/768 kbit ADSL connection shared by potentially 307 individuals. The network is directly connected to the Internet by a C-class of official IP-addresses, limiting the number of machines to 254[1]. Chapter 3 documented a number of problems occurring with a few upstream connection. In our real-world production environment we will experience thousand of concurrent TCP connections and a diversity of TCP protocol implementations. Solving the observed problems in an environment like this will just substantiates the effectiveness of our solution.

9.1 Components and goal

The components, identified in Chapter 4 were:

- Packet scheduling,
- Queue control,
- ADSL link layer modeling,
- Site-policy,
- Service classes,
- Traffic classification,
- ACK-prioritizing

Our components of *packet scheduling*, *queue control*, and *ADSL link layer modeling* are closely related and serves our goal of providing accurate sharing of link resources

[1] 220 IP-addresses are assigned via DHCP and use a fairly small lease time to accommodate the demand.

between aggregated traffic flows. We have changed and tuned the accuracy and precision of the packet scheduler when seeking to achieve *queue control* and when incorporating accurate *ADSL link layer modeling* into the scheduler. The tuning of the packet scheduler have been done to support the demands our service classes, which achieves our goal of supporting delay-sensitive applications.

How the components are combined and configured to fulfill the overall goal of a practical solution is defined in form of a *site-policy*. The site-policy determines the specific configuration of *service classes* and *traffic categorization*.

Some specific *service classes* are chosen, as a part of the site-policy, and define the associated (assured) service requirements of each class. The choice of classes are related to our goal of defining groups of network applications according to their service requirements. *Traffic classification* determines which traffic flows are mapped into a given service class. Choosing to classify ACK packet into a high-priority class constitutes our component of *ACK-prioritizing*, which has the purpose of achieving our goal of full downstream utilization.

The main interest in this chapter is the choice and usage of the *service class* component and the possibilities and setup of the *traffic classification* component. Or more specifically the site-policy of these components. The thesis does not aim at optimizing a specific site-policy. The purpose of the specific site-policy is to demonstrate that a combination of different components work in a real-world scenario.

We do not claim that our site-policy, described in this chapter, is optimal, but only that it is a functional solution that works in practice, and actually makes it possible to share a single ADSL effectively between a large number of users. The development of the site-policy has been an ongoing process of adjustments, and still is, in order to face new challenges of evasive and excessive[2] traffic patterns.

9.2 Queue Control, Overhead and Scheduling

In this section we describe how we perform *packet scheduling* and incorporate *queue control* and *ADSL link layer overhead* with our specific setup.

Packet Scheduling

Linux' HTB scheduler is used for packet scheduling, since our link layer overhead modeling is only fully implemented for HTB.

The `HTB_HYSTERESIS` option, causing the latency bound problem described in Chapter 7 has been disables for its ill effects. We choose not to change the Linux kernel `HZ` value to increase granularity, because the real-world ADSL is a production connection and we are uncertain of the side effects of changing this value under the 2.4 series Linux kernel.

HTB is an implementation of a hierarchical Class-Based Queueing (CBQ) model as defined in [38]. This matches our choice in Section 4.7 on page 45, of using a CBQ model to divide traffic into a number of service classes.

We also wanted to use a Fair Queueing (FQ) algorithm within each class, which is also possible using HTB (or any other class-based Linux sheduler). Different queueing

[2]Excessive is defined as: something being greater than seems reasonable or appropriate[47], which is a fitting description of the observed traffic patterns.

disciplines can be assigned to each HTB class. We choose to use the only FQ implementation offered by the official Linux kernel, which is based on the Stochastic Fairness Queuing (SFQ) algorithm.

The SFQ implementation is not perfect because its hash-bucket size is limited (on the compile time) and the hashing algorithm is not configurable. As an alternative an unofficial implementation of WRR for Linux, along with an unofficial *extended* version of the SFQ implementation exists, called Extended Stochastic Fairness Queuing (ESFQ). Regretfully WRR and ESFQ are not part of the official Linux kernel. The interesting property of both WRR and ESFQ, is that they offer fair queueing based on local originating IP-adresses, which is usefull for our *shared* setup. In our shared setup each user or machine is associated with a local IP-address. We have stated in our goal of sharing that "each user should receive a fair share of the link", making fair queueing between local machines an attractive option in order to achieve our goal. Unfortunately, we became aware of these FQ algorithms too late in the process to seriously consider using them in this thesis.

The Hierarchical Token Bucket (HTB) scheduler is based upon token buckets as implied by its name. HTB uses a dual token bucket algorithm called Two Rate Three Color Marker (trTCM) as described in Section 7.3.1 on page 69. This means that HTB has two rates defined per class; an assured `rate` and a `ceil` rate, and their associated token bucket/burst sizes, `burst` and `cburst` respectably. HTB uses Deficit Round Robin (DRR)[68] fair queueing between classes that exceed their rate. The specific HTB configuration parameters are specified as a site-policy for our service classes.

Queue Control

On this specific real-world setup we have chosen to apply upstream queue control only. This has worked reasonably well because sites, which are able to provide enough data at a high enough rate to continuously saturate the 8 Mbit/s downstream link are limited. Therefore, we estimated that the setup was unlikely to be affected by the downstream delay problem described in Chapter 8. In our real-world production setup, it turns out that some users have found sites with enough capacity and data to saturate the 8 Mbit/s downstream link (hint: Linux ISO images). This is illustrated in Chapter 10.

ADSL link layer overhead

To achieve upstream queue control, we configure HTB to be the upstream bottleneck. This is done using our overhead patch and knowing the type of encapsulation used on the specific ADSL connection, by which we deduct the encapsulation overhead size. The real-world ADSL connection uses PPPoA with VC. This implies an encapsulation overhead of 10 bytes per IP-packet (see table 5.3). The overhead size is simply used as a parameter when configuring the HTB scheduler. The upstream rate is set to 1 kbit/s below the maximum upstream rate, in order not to loose queue control in the event of bursts.

9.3 Site-policy: Service Classes

This section describes which of the service classes we choose to implement. The service classes are defined in Section 4.4 on page 39. We will use the term *traffic classes*, when referring to the implemented service classes.

9.3.1 Choice of Service Classes

In this section, we describe the choice and mapping of service classes to the traffic classes we choose to implement.

Supporting Real-time Classes

It is difficult to support the jitter requirement for the real-time service classes (0 and 1), which have an end-to-end delay jitter requirement of 50 ms. In Chapter 6 we showed, on a 512 kbit/s upstream ADSL connection, how we achieved an upstream delay jitter of approximately 56 ms ($62ms - 6ms$) to the closest IP-address (on the BBRAS). This jitter is too high for the real-time service classes even non end-to-end.

With the 786 kbit/s upstream link we should be able to lower the delay bound, since the delay bound is dependent on the transmission delay, as argued in Chapter 7. The transmission delay of 1696 bytes (corresponding to 1500 bytes including ATM overhead) is 17.66 ms, which should be multiplied by two and added to the baseline delay, according to our findings when evaluating the achievable delay bound of HTB. This gives an estimated delay bound of 43.33 ms ($2 \cdot 17.66ms + 8ms$). Even though the delay bound is lowered a more significant reduction is needed to meet the end-to-end jitter requirement. Therefore, we will not commit to supplying a real-time traffic class, as other hops on the path are likely to introduce additional delays.

Thus, we choose to abandon service class 0. This also implies that we choose not to support other highly interactive application with strict delay jitter demands like highly interactive gaming.

ACK-prioritizing

We choose to use service class 1 even though we cannot completely guarantee the delay jitter bound as described above. This is because we have defined ACK-prioritizing as belonging to service class 1. We would of course have preferred a better jitter bound or granularity for ACK packets, as accurate ACK scheduling would give a smoother data flow, but prioritizing ACKs should help us utilize the downstream capacity (although slightly more bursty).

As an example approximately 24 full sized data packets can be generated downstream with a 40 ms delay (Downstream $delay_{transmission} 1696 * 8/8000 = 1.696ms$). Assuming a delayed ACK factor of two this would result in 12 ACK packets. But it is important to realize that the burst of ACK packets will not be seen as back-to-back ACKs by the data sender, because sending an ACK packet also "takes" time, as the limited upstream sending rate introduces a minimum spacing. On our 768 kbit/s upstream link this represents a minimum ACK spacing of 1.1 ms ($106 * 8/768$) . The effect is dependent on capacity ratio, that is the k value and how much the ACK packet acknowledges, which is dependent on the delayed ACK factor, d. This can be expressed as $\frac{1}{k} \cdot d$, which expresses the factor by which we can acknowledges data packet faster than the downstream capacity, which on the current line is around 3 ($\frac{1}{0.65} \cdot 2$). This shows that we can send ACK packets fast enough to fully utilize the downstream capacity as indicated by the k value. We can thus argue that our limited upstream link actually helps limit the downstream burstiness.

Traffic classes

The traffic classes are based on the service classes, but adapted to our specific needs. The definition of the 6 traffic classes used in our site-policy are listed below. To make the mapping to HTB classes easy the HTB class number is shown parenthesis. The mapping to service classes are shown clearly and explained further in the description.

(1:10) Interactive — service class 2.

Our *interactive* traffic class maps to service class 2 and should be used for highly interactive traffic.

(1:20) ACK — service class 1.

Our *ACK* traffic class should be used for ACK packet only. As mentioned earlier the requirements of ACK packets are best mapped to service class 1.

(1:30) Good traffic — service class 3.

Our *"good"* traffic class should be used for services, which require good service and probably some interaction, but not high interaction. This class is best mapped to service class 3. We imagine the service being used for well-known services, which require a "good" or "better-than-best-efford" level of service.

(1:40) Bulk — service class 4.

Our *bulk* traffic class maps to service class 4, which allows long queues and delay bounds of up to 1 second. Mapping of bulk traffic could be done dynamically looking at the traffic behavior or mapped statically to well-known services which are used for bulk transfers.

(1:50) Default — service class 5.

Unclassified traffic is assigned to our *default* traffic class. The class maps to service class 5, but not quite because this class does not have the lowest priority. The class is our "best-effort" service class which get low service, but should not be starved completely.

(1:666) Bad — service class 5.

Our *bad* traffic class has the lowest priority, which maps in to service class 5, but we allow starvation, which should be avoided for service class 5. The class should be used for traffic, that should only receive service and bandwidth when the connection is not used for anything else. Traffic in this class is likely to exhibit some kind of "bad" or excessive behavior according to the site-policy. P2P traffic is a prime candidate for this class. Denial of Service (DoS) attacks should not be classified into this class, as traffic in this class should still have some purpose.

9.3.2 Setup of Service Classes

We do not use the hierarchical structure of HTB, but choose to define a flat class structure with our 6 traffic classes. The hierarchical structure is normally used to divide traffic between separate agencies.

As described in Chapter 8, the ACK rate is calculated from the downstream throughput, which is given as a parameter to the HTB setup script. The ACK rate is then subtracted from the upstream rate and the remaining bandwidth is distributed between the other classes by a percentage.

All classes should have sufficient capacity not to exceed their assured rate under normal usage. This is because once classes exceed their rate they are scheduled after other classes, which do not exceed their rate, and according to the DRR algorithm. As explained in 8, one should take care not to reserve too much bandwidth for ACK packets, because there should be a reasonable amount left to the rest of the classes. To allocate a more realistic ACK rate a delayed ACK factor of 1.5 has been incorporated into the ACK rate calculation, as described in Chapter 8.

Listing 9.1: Output from the HTB setup script

```
./htb_overhead_kernel_03.sh -i eth1 -o 10 -u 767 -d 7200

Setup information:
--------------------
The ATM/AAL5 overhead calculations are done by tc and kernel
This shows what throughput can be expected

Device                  : eth1
Link bandwidth          : 767 Kbit/s
Max payload bandwidth : 694 Kbit/s (subtracted fixed ATM overhead)
ATM fixed overhead      : 73 Kbit/s
Overhead per packet     : 10 bytes
Downstream bandwidth : 7200 Kbit/s (only calc ACK rate)
Reserved for ACK packets: 340 Kbit/s
Rate left for Classes   : 427 Kbit/s
```

The output from the HTB setup script displayed in Listings 9.1 shows the result of the ACK rate calculation and what is left for remaining traffic classes. The rates are including ATM overhead, because the overhead calculations are accounted for in the kernel. The ACK_{rate} is calculated to 340 kbit/s from a downstream capacity of 7,200 kbit/s (approximately 8 Mbit/s subtracted the fixed ATM headers). The rate left is 427 kbit/s as we set the upstream rate to 767 kbit/s, that is 1 kbit/s less than the actual upstream rate. The HTB setup script itself is shown in Appendix B.6.2 on page 166.

The remaining bandwidth after subtracting the ACK rate is allocated according to the table below. The *ceil* percentages are calculated from the configured/full upstream rate. It should be noted that this means that the percentage for a *rate* and a *ceil* in the table does not express the same number.

parent:	1:1					
class:	1:10	1:20	1:30	1:40	1:50	1:666
name:	Interactive	ACK	Good	Bulk	Default	Bad
prio	0	1	4	4	4	7
rate	20%	special	28%	22%	20%	10%
ceil	20%	100%	80%	80%	95%	100%
service	2	1	3	4	5	5

The *ceil* rate determines how much the class is allowed to borrow from other classes.

The interactive class (1:10) is given higher priority than the ACK class (1:20), because delays in our highly interactive traffic will be more noticeable than slightly higher ACK delays. Class 1:10 has the highest priority and could starve other classes (in situations when scheduling according to priority with the DRR algorithm). To avoid this the ceil of class 1:10 is limited to 20%. Class 1:20 has a ceil of 100% and care should be taken to only classify ACK packets into this class. Full usage of the ceil by ACK packets is unlikely, as the number of ACK packets depend on the rate of the downstream link.

Classes 1:30, 1:40, and 1:50 have the same priority, even though we stated earlier that they were significantly different. This is done to avoid starvation of class 1:50. It

should be noted that they have different rates: 1:30=28%, 1:40=22%, and 1:50=20%. In the case of all three classes exceeding their assured rate, the classes are scheduled according to their priority and then according to rate. Thus, because of the rates we get the desired effect and avoid starvation. The sharing between classes is actually done according to the DRR quantum, which is calculated from the accured rate. The quantum determines, how much is borrowed from ancestors. In our flat structure there is only one common ancestor.

Class 1:30 and 1:40 have a ceil of 80% which also is a mean of avoiding starvation of other classes. It might seem strange that e.g. the bulk class (1:40) is not allowed to use 100% if no other class requests any resource. We argue that this situation will not occur, because we are faced with a busy autonomous network. The default class is allowed a ceil of 95% as it is more likely, that all traffic falls into this class at times.

Class 1:10 and 1:20 must not exceed their rate, as they have low delay requirements. Class 1:30 should hopefully not exceed its rate too often, as the traffic categorized into this class should be well-known traffic/applications, which behave "nicely". The bulk class (1:40) is likely to try to exceed its rate because of the behavior of bulk transfers. The default class (1:50) probably always contains some traffic and will also fairly often experience excessive traffic in form of DoS attacks or mis-categorized "bad" traffic.

The "bad" traffic class (1:666) has the lowest priority and is only guaranteed 10% of the remaining rate. This is done to avoid the service and usage of the class to come to a complete halt. The class is allowed to borrow 100% of the rate, but due to its priority, this is only possible when the capacity is not requested by any other class.

9.4 Site-policy: Traffic Classification

Our site-policy for traffic classification determines the assignment of traffic into different traffic classes.

We use Linux' packet filter framework called Netfilter/iptables for traffic classification. The individual packets are mapped to HTB classes based on filter rules that "mark" packets for the Linux traffic control system. Netfilter is a very flexible and modular framework, which supports the three types of traffic classification, discussed in Section 4.6:

- Header fields
- Traffic behavior
- Data payload analysis

Netfilter is primarily a firewall framework, thus most of the functionality (or modules) are based on matching *header fields*. Classification based on *traffic behavior* or *data payload analysis* is limited and often experimental features, that are not part of the official kernels. The kernel running on our Middlebox has been patched with the experimental Netfilter patches called "patch-o-magic".

To facilitate easy construction and configuration of a site-policy for traffic classification, we have implemented a fairly advanced filtering and "marking" framework. This is part of our software package: *The ADSL-optimizer*, which is described later in Section 9.5. The construction and design of this filtering framework is not relevant to our goal. Thus, we will not describe how it is constructed and designed, eventhough we have spent a lot of time designing and implementing it. It merely serves as a tool

for easy setup of our site-policy. We exemplify our setup using configuration files from the filtering framework, as it eases our description of our site-policy setup.

9.4.1 Specific Classification Setup

The reasoning behind our site-policy is not relevant to demonstrate that we can achieve our overall goal and has thus been omitted from the thesis. Our specific filter setup is simply an example, that demonstrates some of the possibilities for classifying traffic.

Our specific site configuration is listed in Listings 9.2. The listing shows the actual "scheme" configuration file used on our real-world production site (character **#** denotes a comment).

The *first column* represent a "rules" configuration file, which contains the specific filter rules (in a Netfilter like notation). The file names are fairly descriptive. The *second column* defines, which traffic class the filter rules is mapped or marked into. The mark number corresponds to the HTB class numbers, e.g., 0x10 correspond to 1:10. The filter framework supports different marking of upstream and downstream packets, this is shown as the *last column*. This could actually be ignored, because we only perform upstream packet scheduling in our real-world setup.

Listing 9.2: File: kollegie03.scheme

```
#
# Kollegiegaarden - Netfilter mark scheme 03
#
scp                0x40    upstream
ssh                0x10    upstream
dns-server         0x10    upstream downstream
http               0x30    upstream downstream
webserver          0x30    upstream downstream

#
mail_client        0x30    upstream
mail_server        0x30    upstream downstream
chat               0x30    upstream downstream
#
ftp_upload         0x40    upstream
#
# Simple P2P detection via port numbers
bad-simple         0x666   upstream
#
# Advanced P2P detection via payload analysis
layer7-p2p         0x666   upstream downstream

# Local add-on's
simple_vpn         0x30    upstream downstream

# TCP handshake matching
#tcp_handshake      0x10    upstream
#tcp_syn_limit      0x10    upstream
tcp_syn_dstlimit   0x10    upstream

# Prio ACK packet high, this ensures high downstream traffic
ack                0x20    upstream
#
# Latency test rule: icmp traffic
ping-ask           0x10    upstream downstream
ping-munin         0x20    upstream downstream
ping-hugin         0x30    upstream downstream
ping-brok          0x40    upstream downstream
ping-www.diku.dk 0x666 upstream downstream
```

The description of the filter rules is structured after the three types of traffic classification. We describe the function of the filter rules and will not explain the specific content of the filter "rules" configuration files. The contents of these "rules" files are located in Appendix B.7.1 on page 170.

When a filter rule matches a given packet the classical firewall behavior is to stop further matching/processing and make a decision to i.e. accept or drop the packet. The behavior of the packet "marking" scheme is to allow further processing as this allows remarking of the packet. This implies that the order of the filter rules is significant. The idea of remarking is also used by the Differentiated Services (DS) architecture. DS uses rate metering to determine if a class does not conform to the configured traffic profile and thus needs to be remarked (as illustrated in Figure 4.1 by the "meter" box). We define this metering in general as traffic behavior and also allow remarking based on header fields and data payload analysis.

In the next sections we describe the considerations behind the individual filter rules. Special considerations are necessarily when classifying traffic on our real-world autonomous network. The network exhibits evasive and excessive traffic patterns. We see the traffic patterns as excessive, because the traffic flows observed are often greater than seems reasonable or appropriate. Therefore, creation of filter rules should be conservative in order to avoid classifying excessive traffic flows.

9.4.2 Header Fields

The correct mapping of `ssh` and `scp` is an example of a simple remarking scheme, which is done using header field matching only. Secure SHell (SSH) is an interactive remote shell application (class 1:10) and Secure CoPy (SCP) is used for bulk file transfers (class 1:40) both applications are part of the same software package (sshd) and use the same port number (port 22). This poses a problem for well-known port number classification, as the same port number is used for an application with two different traffic behaviors. Fortunately, the application sets the Type Of Service (TOS) header field in accordance with its traffic behavior. We utilize this, as illustrated in the listing, to mark all packet using the port number to `scp` and then remark packets matching interactive `ssh` packets with the TOS field set to: "Minimize-Delay". Matching in this way gives a more strict admission control to the interactive class.

Our matching of DNS traffic (`dns-server`) only includes a limited number of DNS servers, which are the DNS servers announced via DHCP. The local DNS servers out-bound requests are also marked into class 1:10.

The HTTP/Web traffic port 80 is often used by evasive applications. In an attempt to limit this, only local out-bound HTTP requests get prioritized (`http`) and only a limited number of local machines get prioritized as web-servers (`webserver`). This is not a bullet-proof solution, as bulk traffic from local clients to external machines on port 80 would be classified as HTTP/Web traffic.

The matching of mail clients and servers have been separated and categorized into two different classes (1:30 and 1:40). The mail server traffic is mapped into the bulk traffic class (1:40), as discussed in Section 4.4. The mail-clients (`mail_client`) are mapped into the good traffic class (1:30), because a user is most likely interacting with the program, but probably not in a highly interactive way seen from a network perspective. The filter rule `mail_server` matches the mail-server protocol SMTP, but explicitly for the local mail server, because other local machines could be used as spam relays.

Line-based `chat` protocols are mapped to the good traffic class (1:30) as discussed in Section 4.4. The filter rule `simple-vpn` prioritizes a specific users' VPN connection.

Prioritizing ACK packets is done as late as possible, because ACK packets are most likely to already have been marked by some of the above filter rules. ACK packets have special status and thus should be remarked into the ACK traffic class (1:20).

The last filter rules `ping-xxx` is simply used for ping latency tests, which serves as an easy way to determine the latency in each of the traffic classes. Care should be taken when interpreting the latency, because the queue of the traffic classes uses as SFQ queueing mechanism. The measurement is a measurement of the level of service the traffic class achieves related to other traffic classes, but is not a good indication of how loaded the specific queue is.

9.4.3 Traffic Behavior

The reason for placing TCP handshake under traffic behavior classification is a result of the excessive network behavior exhibited by our real-world network. The configuration shows three attempts to classify TCP handshakes (two have been commented out): `tcp_handshake`, `tcp_syn_limit`, and `tcp_syn_dstlimit`.

Our TCP handshake classification only matches the first two packets of the three-way TCP handshake. That is, the SYN and SYN+ACK packets, because the last ACK packet, which completes the handshake, is matched as a normal ACK and thus already gets preferential service.

Initial experiments showed that this simple matching of the SYN and SYN+ACK packets, performed by `tcp_handshake`, misbehaved, because it overloaded the traffic class with too many matches. This is due the the excessive behavior on the network, which is primarily caused by P2P traffic. On the network we have experienced that the number of connections in the Netfilter connection tracking table exceeded 200,000 and that a single IP consumed around 40,000 connections alone. Measurements on the network have revealed a SYN packet rate of around 40 packet/s (over a period of 20 minuts).

The filter rules `tcp_syn_limit` tries to solve this overloading by performing traffic behavior matching using Netfilters packet rate limiting module. The *limit* module matches packets under a given packet rate. Although the limiting worked and only allowed a given number of packets/s into the interactive class it did not fulfill our need. The SYN packets getting through were not necessarily the ones we wanted. The packet rate was most likely consumed by P2P connections. The problem with the limit module is that it only has a single rate meter (per filter rule).

The filter rules `tcp_syn_dstlimit` which is used now is not perfect, but suits our needs better. It is based upon the experimental module *dstlimit*. It solves the problem with the single rate meter of limit module, by having a rate limit counter for every destination IP-address, which is implemented using a hash-table. We would have preferred to have a limit counter for each source IP-address, as we want to achieve fair sharing based on local IP-addresses, but it is closer to our needs.

We have experimented with the experimental *connrate* Netfilter module, which can be used for matching bulk transfers. The module records the rate for every connection (using the connection tracking table) and allows matching against the current transfer rate of connections. Thus, this makes it possible to mark all connections exceeding a given rate. The module is only available for the 2.6 kernel series, but we have ported it to the kernel 2.4 series. After porting the module we found that the precision of the rate estimator was dependent on the timer granularity. The imprecision of the rate estimator made the rate "jump" too much, which could result in packets jumping or oscillating between traffic classes resulting in packet re-ordering. For this reason we have chosen not to use the modules in production. However, this would have been a useful and effective module for controlling bulk traffic flows.

9.4.4 Data Payload Analysis

FTP-data traffic is actually hard to detect, because the FTP protocol [64] uses a secondary TCP connection for the actual transmission of files [21] (the FTP-data connection). Information about the secondary TCP connection is contained in the FTP-command connection. Thus, to identify the FTP-data connection data payload analysis is necessary. The Netfilter NAT framework actually performs FTP payload analysis to make FTP connections work under Network Address Translation (NAT). The real-world setup has official IP-addresses and does not need the NAT functionality. We trick Netfilter into using the FTP NAT module anyway by loading the correct NAT modules. To access the information contained in the NAT module the connection tracking *helper* module is used, which is kind of a trick. The filter rule `ftp_upload` uses this trick.

It would have been an idea to categorize the FTP-command connection into the interactive class and the FTP-data connection into the bulk class as discussed in Section 4.4. Unfortunately, we have observed FTP clients using the FTP-command connection for file transfers. Thus, we have chosen not to split up the categorization of FTP traffic.

In order to detect P2P traffic we initially try to match on port numbers with the filter rules `bad-simple`. Since users or P2P programs themselves change the default port number, this is not effective. It is worth noting that even if all local users were using the default port numbers, this matching would still fail. This is because external users can change their port numbers and will therefore, be contacted on their changed port number by our local users. This renders matching by port number almost useless. We still use the filter rules `bad-simple`, because users applying the default port will be matched.

A couple of categorization modules for Netfilter exists, which perform data payload analysis to detect P2P traffic. We have tested a couple of them and have chosen to use the module *layer7*[3]. The advantage of this module is that it is based upon configuration files, which contain pattern matching rules (with regular expressions). The other modules we have seen have hardcoded the protocol matching in the source code resulting in modules, that are unable to adjust to new protocols. The module is not part of the "patch-o-magic" patch set, and we have patched the kernel explicit to support this modules.

It requires a lot of processing power to perform regular expression pattern matching with a large number of rules on the payload of every data packet. The *layer7* module reduced the amount of processing power required by only looking at the first 8 data packets after which the connection is marked for the duration of the connection. The module works quite well and matches almost all of the P2P traffic on our real-world environment. Updating of P2P filters and manual detection of evasive users is sometimes still necessary.

9.5 Software Package: The ADSL-optimizer

The *ADSL-optimizer* is the practical solution in form of a software package, which include installations instructions for easy deployment on a Linux based router (see appendix B.5 on page 158). The ADSL-optimizer has been used for all the setups throughout the report. It is currently operational on 3 sites and has proven its worth on our real-world production network. The specific setup of the ADSL-optimizer for our real-world network has been described in this chapter.

[3] `l7-filter.sourceforge.net`

The ADSL-optimizer consists of three elements or modules, which have been designed to be as independent as possible of each other:

Queues This module contains the different setup scripts for the Linux Traffic Control system, e.g, the HTB and ingress scripts. All scripts have the same parameter parsing and have access to some common functions which e.g, can calculate the rate consumed by ACK packet from the downstream rate, as described in Section 8.1. In the HTB setup scripts the rates are calculated from a percentages to make it easy to use the script on other ADSL connections with a different rate.

Filter This module performs the traffic classification and is based on the Netfilter framework. The module has several configuration files, which makes it more flexible to design and develop a site-policy. An example of a specific site-policy for our real-world setup is described in this chapter in Section 9.4.1, which shows the use of *scheme* and filter *rules* configuration files. When changing a filter rule the hole *scheme* configuration does not need to be reloaded. It is possible to load and unload individual *rules* configuration files.

Graph This module collects statistics from the Linux Traffic Control system. It currently supports gathering statistics from HTB and HFSC. The collected data from the traffic classes are stored in some RRDtool data files. This makes our *graph* module a RRDtool backend. We recommend the RRDtool frontend `drraw`[4] for displaying the RRDtool graphs. Graphs created with this graph module and `drraw` is used in Chapter 6 and Chapter 10. The module almost detects everything itself and even creates missing (RRDtool) data collection files. The *graph* module is fairly independent and could just as well be distributed as a separate tool for collecting traffic statistics from the Linux Traffic Control system.

The software package also include UNIX init-scripts (start and stop scripts) for each module and a common base configuration file. In the appendix, we have only included the necessary parts of the ADSL-optimizer, which are relevant to the thesis. From the *filter* module, only the necessary configuration files used are included, e.g. for different test setups and the final site-policy setup for the real-world production ADSL. None of the *graph* module code has been included. Most of the code and setup scripts from the *queues* module has been included in the appendix, as we refer to most of the HTB setup scripts and the code for the ACK rate calculations.

The full software package for the ADSL-optimizer will be available at:

http://www.ADSL-optimizer.dk

9.6 Summary

In this chapter we have combined the different available components under Linux to create a practical and functional middlebox solution. The Hierarchical Token Bucket (HTB) scheduler is used to provide isolation and sharing between the traffic classes. The Stochastic Fairness Queuing (SFQ) algorithm is used to provide a fair sharing between traffic flows within each class. For traffic classification the Linux Netfilter framework is used.

We have describe and created a specific site-policy setup for our real-world production ADSL connections. We define 6 traffic classes which is related to the service

[4] http://web.taranis.org/drraw/

classes 1 to 5 of of Y.1541. We will not to commit to supporting the real-time service class 0 of Y.1541. We have two traffic classes which relates to service class 5. The 6 traffic classes are named after their function and are called: *interactive*, *ACK*, *good*, *bulk*, *default*, and *bad*.

For traffic classification we use all three types of classification. We use *header field* as the general classification method, *traffic behavior* for limiting and marking the correct TCP-handshake packets, and *payload analysis* for detecting P2P traffic.

As part of constructing a practical setup, we have created at software package, which we call the ADSL-optimizer, which consists of a set of tools or modules that ease the implementation and setup of our specific site-policy. This software package, should make it easier for others to make use of our work and make it easy to design and develop a specific site-policy.

We have already described some of the experiences with traffic from our real-world network when describing the setup considerations of the traffic classification. In the next chapter we evaluate the solution and describe our experiences by showing some real-life traffic graphs taken from our real-world production ADSL.

Chapter 10

Evaluating the practical solution

In this chapter we show some examples from our real-world setup in the form of live traffic graphs taken from our real-world production ADSL connection. Real live traffic graphs means that we have not mangled the data or introduced artificial traffic loads.

The aim of this chapter is to demonstrate that we have created a practical solution, which fulfills our goal and is running in production with the setup described in Chapter 9. All graphs in this chapter are generated with the RRDtool frontend tool `drraw` and the data has been collected with our own tool; the graph module/part of the ADSL-optimizer. The graphs are available via a web interface to the system administrator and possibly to the users.

10.1 The Project History Illustrated over 9 Months

The project history is illustrated in Figure 10.1 in form of a upstream throughput graph over the last 9 months, the graph shows data from 16th of April 2004 to 16th of January 2005.

For a long time the real-world production ADSL connection, at Kollegiegården, used a simple setup of a packet scheduler based on the naive approach. With the naive approach the bandwidth was simply reduced to compensate for the ADSL overhead (as described in Chapter 6). Due to the diverse nature of the real-world network the bandwidth had to be reduced significantly to obtain queue control. This was not a viable setup or situation as too much bandwidth was wasted.

The real work on this thesis started around March 2004[1]. During March and April the ADSL-optimizer came alive. As can be seen from Figure 10.1 the graph module was deployed in production on April 20th, 2004.

Our first solution to compensate or model the ADSL link layer overhead was based on an existing overhead patch for HTB. The overhead patch implemented an overhead per packet. The patch only modified the user space program (the rate table as described in Section 6.1.2). This allowed us to compensate for the overhead per packet, but did not allow accurate calculation of ATM alignment overhead. This was compensated through over-estimating the alignment overhead per packet. This worked fairly well

[1]The official start time was in December 2003, but an interesting 10 Gbit/s project came along ;-)

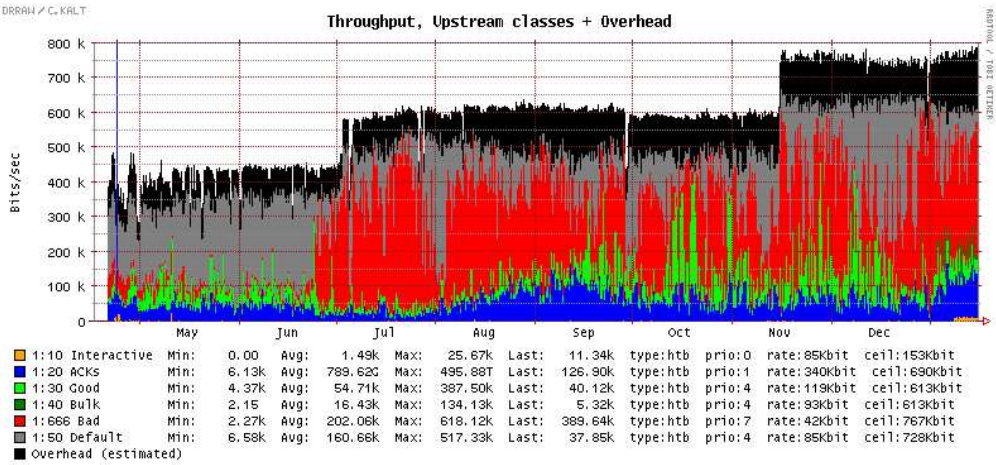


Figure 10.1: The project history illustrated over 9 months. The graph shows the upstream throughput for each class, from April 2004 to January 2005.

and was not changed until November.

The black overhead in the graph is estimated from the packets per second, much like the original overhead patch. The graphs' overhead estimation is adjusted to our latest overhead patch. This is why the first part of the graph does not reach 512 kbit/s, as we over-estimated the alignment overhead during this period. This shows that the over-estimation of alignment overhead wasted bandwidth.

Around the 14th of May, we published the first results of our work in form of an article[26]. The article was submitted to ACM Internet Measurement Conference 2004, but was not accepted and was instead published as an technical report at the University of Copenhagen.

The tall blue line, that indicate several gigabit/s worth of data in the ACK class, is an overrun error in the graph module/script, which has been corrected. Unfortunately, this makes the average and max statistics for the ACK class unusable.

The graph shows how the P2P detection was improved significantly around the 22nd of June. In this period we tested several P2P detection methods and settled with the Netfilter *layer7* module (described in Chapter 9). It is clear from the rest of the graph, that categorizing traffic into class 1:666 has been effective from this point on.

Around the 2nd of July, TDC discovered an configuration error on the ADSL equipment which was limiting the upstream bandwidth to 512 kbit/s. This changed the capacity from 512 kbit/s to 668 kbit/s. The 668 kbit/s was due to the maximum ADSL (sync) rate over the copper wires. Around the 15th of November, another correction was made and the line could perform at 768 kbit/s. At that time we were informed that the sync rate of the line was 768 kbit/s, but we could not achieve that throughput. We bought the Cisco (1401) router equipment and found another configuration error which limited the burst rate to 700 kbit/s.

Also around the 15th of November, we finished the evaluation of the accurate overhead implementation on the Tele2 ADSL line (which is documented in Chapter 6), and chose to try the patch on the real-world production ADSL. It is difficult to evaluate the increased accuracy of the new overhead patch because the upstream line rate was changed around the same time as the overhead modeling was changed.

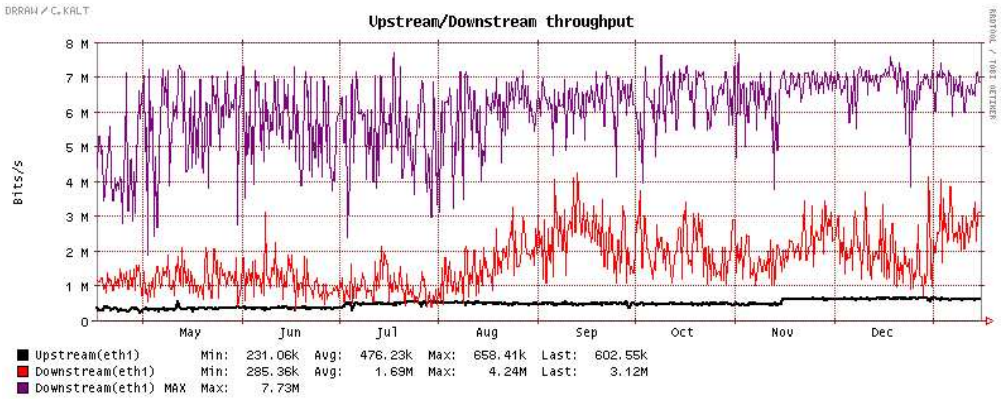


Figure 10.2: The average and maximum throughput illustrated over 9 months. The graph shows the throughput.

The downstream average and maximum throughput is illustrated in Figure 10.2 over the same 9 months period. The statistics in this graph is collected with another tool (`rrdcollector`), which is why the data collection starts before the graph in Figure 10.1. The graph clearly shows that we have been able to achieve high utilization of the downstream capacity by prioritizing ACK packets at least as peek throughput. The graph also shows that the downstream capacity is not saturated all the time, like the upstream throughput in Figure 10.1.

10.2 Evaluation Overview over 12 Hours

In this section we illustrate, that we have achieved our service class requirements over a typical 12 hour period. The 12 hour period is illustrated in Figure 10.3 and 10.4.

Figure 10.3 is structured as follows:

- the upstream throughput in each class in Figure 10.3(a)
- the (downstream) throughput in Figure 10.3(b)
- and the latency for the high-priority classes in Figure 10.3(c).

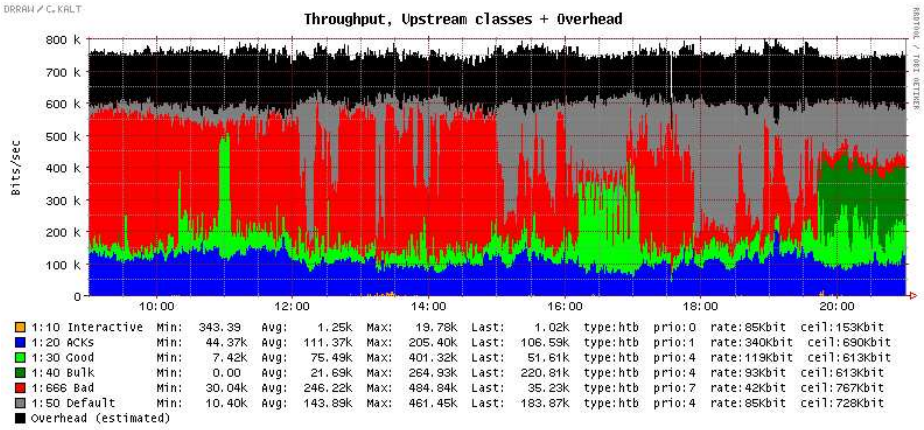
Figure 10.4 shows the latency in the other traffic classes. Two graphs are used with different latency scales to give a more detailed view of the latency. Figure 10.4(a) shows the latency in all but class 1:666 and Figure 10.4(b) includes class 1:666.

The delay bound for our two high-priority classes, the interactive class 1:10 and the ACK class 1:20 was 100 ms. Figure 10.3(c) clearly shows that we have fulfilled the requirement over a period of 12 hours. For both high-priority classes, the average delay is around 30 ms and only one spike around 19:00 o'clock reach approximately 93 ms. This spike is caused by the downstream delay problem, discussed in Chapter 8. A more detailed view of how the real-world ADSL is affected by this downstream delay problem is shown later in section 10.3. The 93 ms delay should be seen in context of the latency when the ADSL-optimizer is not used, which on this ADSL connection would normally be above 1000 ms.

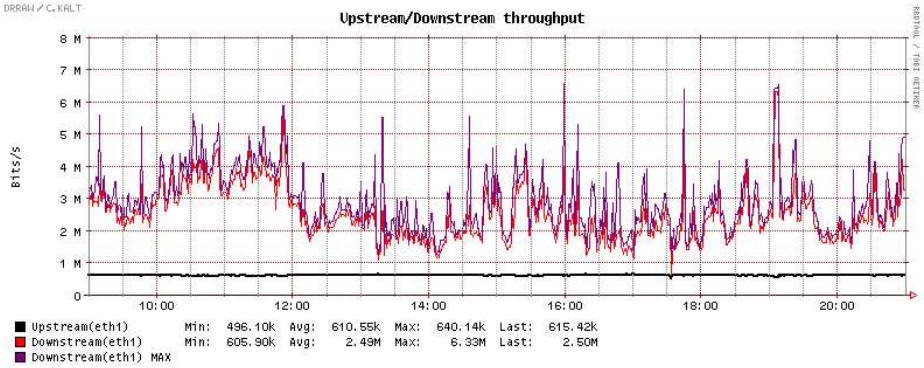
Around 20:00 o'clock there is traffic in all classes. The graph in Figure 10.3(a) shows that none of the classes starves, even the bad class (1:666) gets a minimum of bandwidth. This serves our requirement of avoiding starvation. (Note, that it might be difficult to see the difference between class 1:30 and 1:40, in the printout, we do apologize.)

The latency graphs, in Figure 10.4, show that we also have maintained the latency requirement in our other service classes. The good traffic class 1:30 has a maximum delay of 355 ms, which is below the delay bound requirement on 400 ms for this class. Unfortunately, we did not record the latency in class 1:40, due to a mis-configuration of the latency measurement tool (`smokeping`). The default class 1:50 has a maximum latency of 864 ms, this class did not have a delay bound, but it fulfills the 1 second delay requirement of class 1:40. Thus, we assume that the requirement of class 1:40 has been achieved, as class 1:40 was given more resources than class 1:50. The bad traffic class 1:666 received delays up till 3.17 seconds, which is okay for this class.

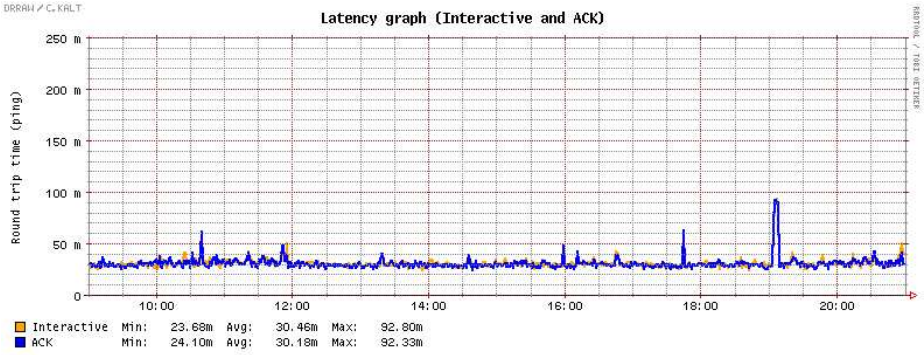
The latency achieved during this 12 hour period is well below the service class delay requirements. The delay bounds defined for the service classes (of Y.1541[7]) is one-way delay bound and our measured delays are the Round-Trip Time (RTT). Furthermore, the delay bound is defined as an average delay bound, which allow us to exceed the delay bound for a short period of time. Y.1541 suggests a 1 minute evaluation interval.



(a) Upstream throughput for each class

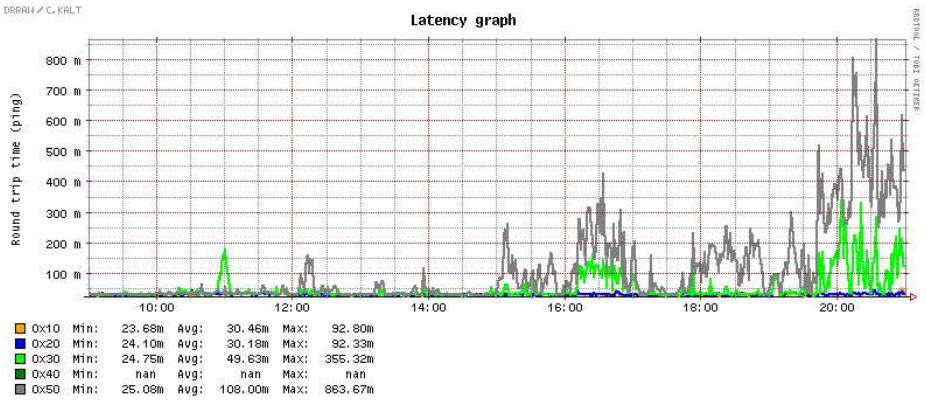


(b) Throughput

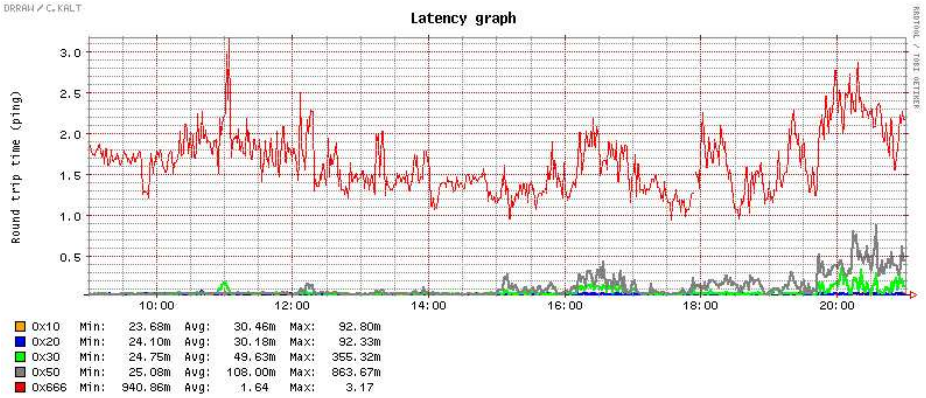


(c) Latency for high-priority classes

Figure 10.3: Period 12 hours. Throughput and latency.



(a) Latency in all classes except class 1:666



(b) Latency including "bad" traffic class 1:666

Figure 10.4: Period 12 hours. Latency in all classes.

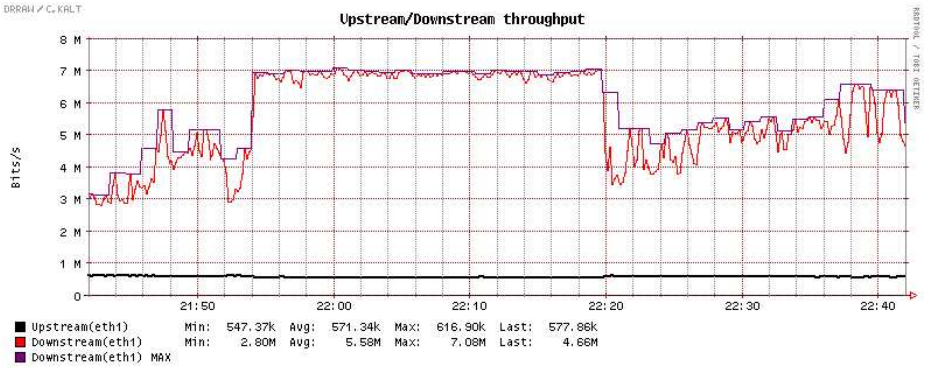
10.3 Downstream Delay Problem

We have chosen only to apply upstream queue control through upstream packet scheduling on the real-world ADSL connection, as mentioned in Chapter 9. This was done because we thought that the downstream capacity was large enough to avoid saturation and thus the downstream delay problem (identified in Chapter 8) was not an issue of interest.

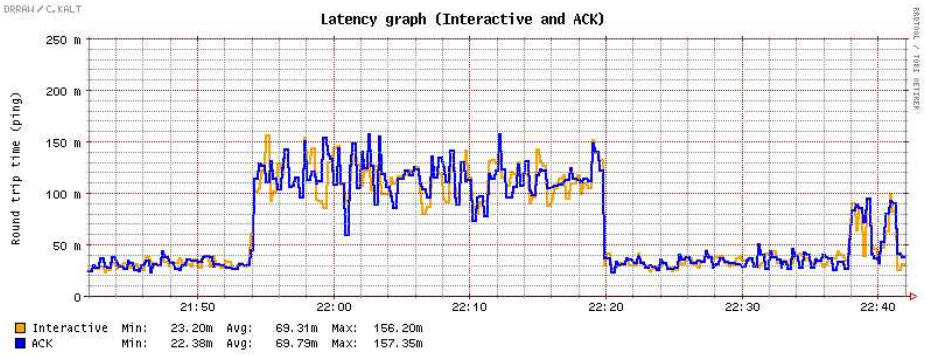
The longest incident we have observed of the downstream delay problem occurring on the real-world network is shown in Figure 10.5. The downstream connection was saturated over a period of 26 minutes. This was caused by a single user or IP-address downloading ISO images from `www.klid.dk`, presumable two as the download period corresponds to approximately 1400 Mb (assuming 700 Mb ISO images). This shows that users can utilize the full capacity of the 8 Mbit/s downstream link, as long as they find sites with enough capacity and data. However, this also shows a need for some kind of downstream queue control.

The latency effect is shown in Figure 10.5(b). The maximum delay for the high-priority classes were around 157 ms. The latency effect of the downstream delay problem is not as critical as the observed upstream delays in Chapter 3, which reached a maximum of 3300 ms.

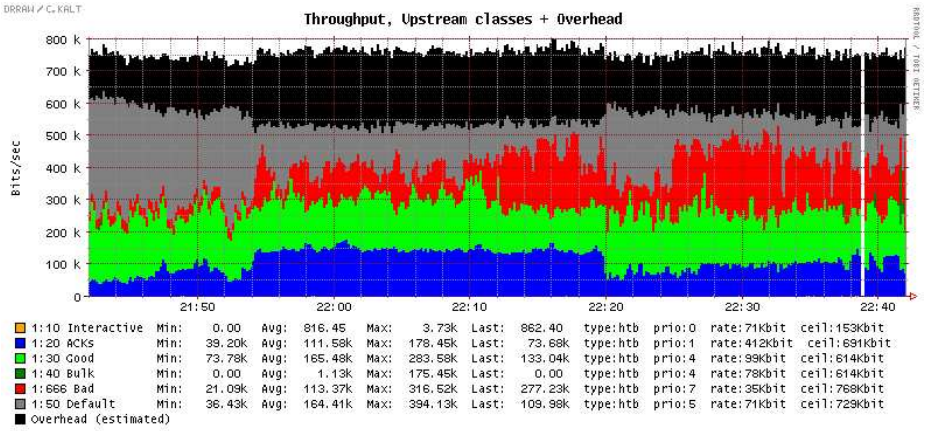
We still conclude that we can achieve our delay requirements, as our delay bounds were defined as one-way delay bounds. However, these situations can fairly easily be mitigated using some of the simple solutions described in Chapter 8, as long as we are willing to sacrifice some downstream bandwidth.



(a) Throughput



(b) Latency for high-priority classes



(c) Upstream throughput for each class

Figure 10.5: Real-world senario where the latency is affected by heavy usage of the downstream link.

10.4 Excessive P2P traffic

In this section we show the excessive nature of P2P traffic on our real-world production network. Figure 10.6 shows a one hour period where the upstream connection was primarily used for P2P traffic, as no other traffic classes had any demand.

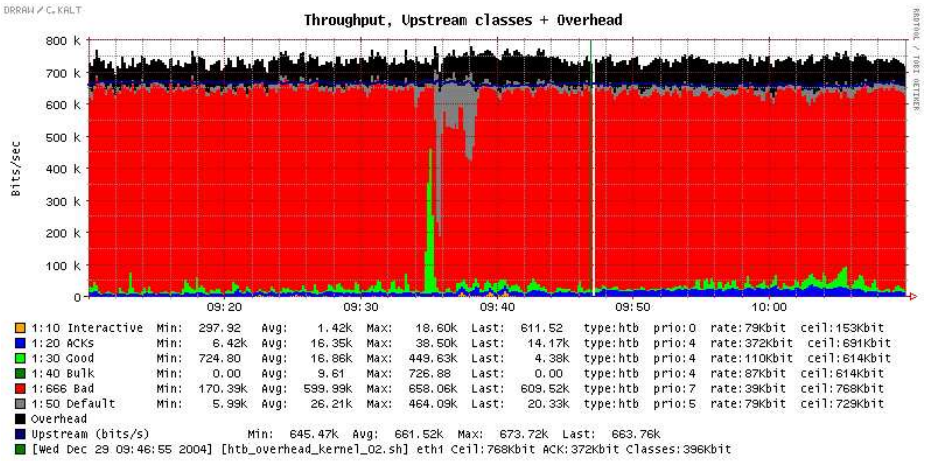


Figure 10.6: Excessive P2P traffic. Upstream throughput for each class.

At around 9:47 we changed the queue size of the bad traffic class 1:666. The queue size was changed from 64 packets to 128 packets. The effects of this change is most profound in Figure 10.7 on page 110 and illustrates the excessive behavior of P2P traffic. From the backlog graph in Figure 10.7(a) we clearly show that these extra 64 packets are consumed instantaneously. The number of dropped packets in the drop graph in Figure 10.7(b) is not significantly reduced. This together with the backlog graph, indicates that the demand for resources are excessive.

The latency illustrated in Figure 10.7(c) is more than doubled. This significantly higher latency has no effect on the achieved upstream throughput, which indicate a lot of connections that through their collective window sizes can utilize the bandwidth anyhow.

A closer look at the number of connections

From this period we have analyzed the Netfilter connection tracking table. The connection tracking table has some fairly long connection timeouts and garbage collection intervals. We will not go into the details of these intervals. The use of the table is simply used to illustrate the excessive behavior of the P2P connection.

The connection tracking table contained 40,254 connection. Of these, 29,423 connections were *unreplied*, which we assume is due to P2P hosts which do not respond.

This leaves 10,831 confirmed connections, which at some time have transfered some data packets. Of these 10,831 connections 9,971 connections had been marked by the *layer7* module as P2P traffic, that is 92% confirmed P2P connections, leaving us with only 860 maybe legitimate connections.

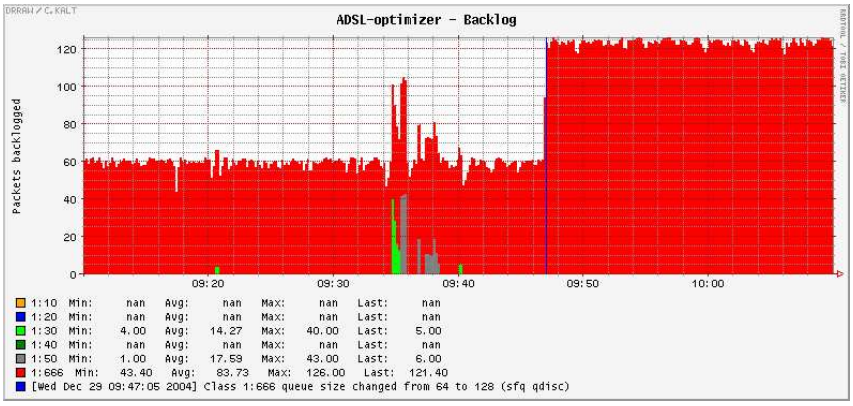
For these 860 connections, the distribution or use of port numbers, both source and destination port and UDP or TCP are shown in the table below. We have chosen only to look at port numbers, which occur five or more times.

Service name	Port	Connections	In timeout state
HTTP	80	297	141
P2P bittorrent	6881	97	2
?	12972	87	
?	6346	50	
Chat MSN	1863	50	1
DNS	53	42	
?	12449	35	
?	34588	29	
?	3531	23	
FTP	21	18	
?	3134	17	
?	2088	15	
SSH	22	11	
?	6348	8	
?	3077	8	
?	2004	8	1
P2P	5050	7	
Chat	5190	6	
P2P bittorrent	6883	5	
P2P eDonkey	4662	5	
Summary:			
Known P2P		114	
Other known		424	
Unknown		280	
Total		818	

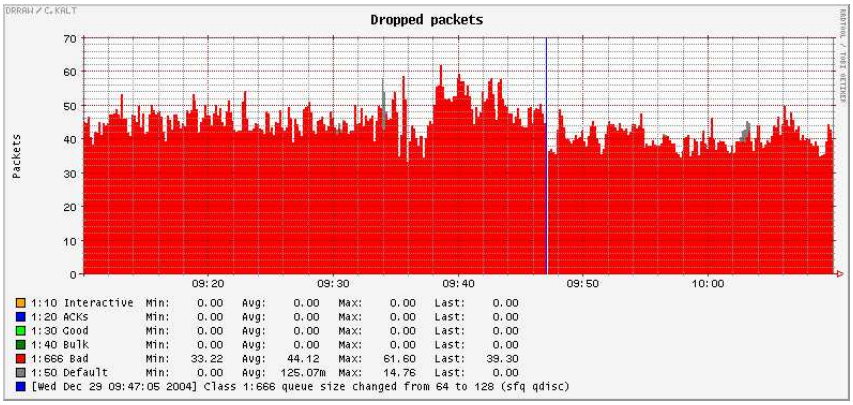
HTTP traffic is a clear winner, with 297 connections of which 141 connections is in a timeout situation. We only recognize 424 connections as normal known services out of the 818 connections. 114 connections were spotted as P2P connections and the rest 280 connection are likely also to be P2P connection. The 424 recognized connections only constitute 1% of the of the total 40,254 connections in the table.

This situation is not uncommon on the real-world network. We have observed situations with more that 200,000 connections in the Netfilter connection tracking table. For this reason we have been forced to increase the number of entries in the connection tracking table to 256,000 connection/entries[2]. Each entry consumes approximately 300 bytes (times 256,000) result in 76.8 Mb memory usage for the table alone. We categorize these types of P2P traffic patterns observed on the real-world network as excessive traffic patterns.

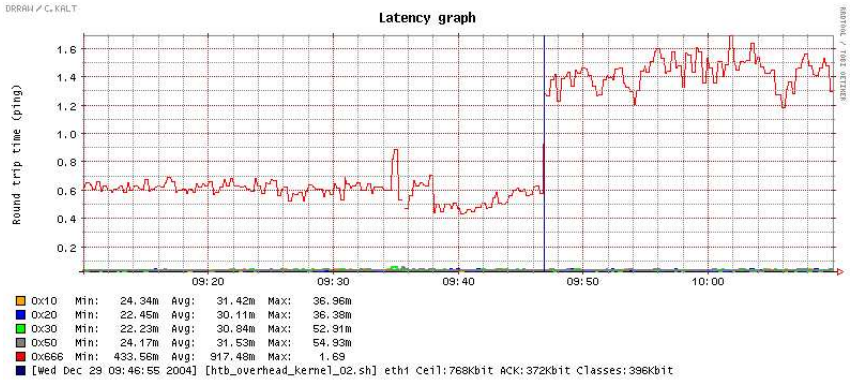
[2] /proc/sys/net/ipv4/ip_conntrack_max



(a) Upstream throughput for each class



(b) Upstream throughput for each class



(c) Upstream throughput for each class

Figure 10.7: Example of a excessive P2P traffic load.

10.5 Summary

We have demonstrated that we have achieved our goal to create a practical solution that optimizes a real-world production ADSL connection shared by a large number of users. Our solution has been improved over time and has been running for 2 months with the setup described in Chapter 9.

We have shown that the real-world production network sharing the ADSL connection is a busy network, which exhibits excessive P2P traffic patterns. Over a period of 9 month we have shown, how the upstream connection is constantly saturated, implying a constantly busy network.

We also demonstrate that we have achieved our goal of maximum link utilization. When the downstream capacity is fully utilized we experience a downstream queueing delay, but this was expected from our experiences in Chapter 8.

Our 12 hour real-world evaluation enables us to conclude that our setup of the traffic classes has fulfilled the promised resource assurance requirements of our service classes. All classes receive their guaranteed resources and enjoy delay bounds that are on average much better than promised. Furthermore, the classification of P2P traffic into the bad traffic class has been successful and the bad traffic class only receives a minimal service as intended, when other classes have demand. Therefore, we conclude that our practical real-world deployment has been a success and has achieved our overall goal.

Chapter 11

Conclusion

In this thesis we have implemented a practical solution that optimizes an ADSL connection shared by a busy autonomous network with respect to both interactive comfort and maximum link utilization.

We have documented and demonstrated the effectiveness of our solution by evaluating the solution in production on a shared real-world autonomous network, that exhibits evasive and excessive traffic patterns. This fulfills our goal requirements of the solution working in a *shared*, *busy*, and *autonomous* network.

Latency has been incorporated as an important parameter in our design and solution. We have improved the latency bounds significantly from a maximum upstream delay of approximately 3300 ms to a upstream delay bound of 62 ms for our high priority packets on the same physical ADSL (2 Mbit/512 kbit) connection. This has been achieved by tuning the precision of the packet scheduler and implementing accurate link layer overhead modeling into the packet scheduler. Therefore, we conclude to have achieved our goal of *interactive comfort* and the ablilty to support delay-sensitive applications, through controlled link-sharing.

We have demonstrated that we can achieve full link utilization both upstream and downstream at the same time by prioritizing ACK packet. Through accurate overhead scheduling, we achieve our goal of avoiding unnecessary waste of link capacity, which was part of our goal for maximum link utilization. Existing packet scheduling optimizers for ADSL use the naive approach of reducing the rate to achieve queue control and thus waste link capacity. Through this, we conclude that we have achieved our goal of *maximum link utilization*.

Findings for the unmodified ADSL connection

In our preliminary analysis of ADSL and its asymmetric nature, we find and investigate four performance-related factors:

1. The normalized bandwidth ratio k.
2. The delayed ACK factor d.
3. The competing reverse traffic flow.
4. The queue size of the bottleneck routers.

The normalized bandwidth ratio, k, together with the delayed ACK factor, d, determines whether the upstream connection carrying the ACK packets has sufficient capacity. If $k > d$ the upstream connection has insufficient capacity and will be saturated by ACK packets before achieving full downstream utilization, thus effectively limiting the downstream throughput.

It seems that all commercial ADSL connections have sufficient upstream capacity, $k < 1$, to avoid disturbing the ACK feedback mechanism. However, we demonstrate through our practical analysis and tests, that *competing reverse traffic flow* on the upstream connection, disturbs the ACK feedback mechanism, reducing the achieved downstream throughput significantly. We show how the achieved downstream throughput is directly dependent on the delay caused by a *queue on the upstream bottleneck router*. We also demonstrate a direct correlation between the TCP window size and the queue size. This phenomenon, we have documented occurring on ADSL, is called ACK-compression and occurs due to reverse data traffic causing periods of congestion and (ACK) queueing on the upstream connection.

We find upstream queueing and the resulting high latency to be the main cause of the problems observed on ADSL. We recommend the ISP to lower the upstream buffer on the ADSL modem, because the buffer size is too large compared to the upstream capacity, as this buffer size determines the maximum latency. The default window size (Linux 64 kbytes) introduced a queueing delay of 1200 ms on a 512 kbit/s upstream link. Therefore, we also recommend the user to lower the machines (upstream) TCP window size, as we have demonstrated that the default window size is too large compared to the upstream capacity.

Solution phase: The packet scheduling middlebox

Our packet scheduling middlebox solution is closely related to the Differentiated Services (DS) QoS architecture. The middlebox is positioned between the ADSL modem and the local network and resembles a DS edge node, that performs classification and conditioning of every packet as no external DS network elements are present. In this kind of setup, with no cooperation from other network elements, our middlebox solution need control of the queue for the packet scheduler to have any effect. We have achieved upstream queue control through modifying the Linux HTB packet scheduler to perform accurate link layer overhead modeling of the ADSL connection.

Modeling the link layer overhead is of special importance on ADSL, because the available bandwidth for IP traffic varies significantly. Depending on packet sizes the available bandwidth can be reduced up to 62%. The overhead is caused by protocol encapsulation overhead and packet/cell aligning at the ATM/AAL5 link layer. We provide a detailed description of the different encapsulation methods used on ADSL and their associated overhead.

Our solution supports different types of network applications at the same time, through traffic classes based on aggregated traffic flows, like DS. Our traffic classes are based upon the service classes defined in the telestandard Y.1541[7] – "Network performance objectives for IP-based services". We have been able to support the average delay requirements of all the service classes, by tuning the precision of the HTB packet scheduler. We have almost accomplished to support the delay jitter requirements of the variation-sensitive service classes, on our specific test ADSL connection (2 Mbit /512 kbit). However, almost is not sufficient to commit to support the real-time service class 0 of Y.1541.

By tuning the HTB scheduler we have achieved a delay bound, where high-priority packets have to wait for at most two full sized 1500-byte MTU packets. On a 2 Mbit

/512 kbit ADSL connection we have demonstrated that we are close to the optimal non-preemptive packet scheduler, as during full traffic load 75% of the packets are scheduled within 35 ms, which were the expected delay bound of the optimal non-preemptive packet scheduler for that specific ADSL connection. We have also demonstrated that an additional 10 ms can be caused by the OS timer granularity giving an expected delay bound of 45 ms, which constituted 92% of latency measurements during full traffic load (Figure 7.9 with 10000 samples). The maximum observed delay was 62 ms, which related to the minimum baseline delay of 6 ms, giving a delay jitter of 56 ms. The real-time service class 0 requires a delay jitter of 50 ms.

We have shown that we are able to achieve full link utilization of both upstream and downstream traffic at the same time by prioritizing ACK packets. We have also shown that full utilization of the downstream link introduces an extra downstream queueing delay. With this extra downstream delay we can strictly still satisfy the average delay requirements of the service classes in Y.1541, as they were defined as one-way delay bounds. We know that it is possible to do better, we show two simple methods for indirect downstream queue control, which reduce this delay and gives a better delay bound for our service classes, but also wastes downstream capacity. This does not satisfy our goal of avoiding to waste link capacity, and have thus not been applied in practice. Further studies should be made to solve this downstream delay problem in a smooth manor and without wasting downstream bandwidth.

The 12 hour real-world evaluation enables us to conclude that our setup of the traffic classes have fulfilled the promised resource assurance requirements of our service classes. All classes receive their guaranteed resources and enjoy delay bounds that are on average much better than promised. Furthermore, the classification of P2P traffic into the bad traffic class has been successful and the bad traffic class only receives a minimal service as intended, when other classes have demand.

From this we conclude that the project has been a success and that the overall goal has been fulfilled altogether.

11.1 Future Work

As we have mentioned before, we believe that future studies should look at solving the downstream delay problem in a smoothed manor. We have mentioned some of the techniques (in Section 4.5), which involve indirect control of the data packets through pacing or mangling ACK packets. We do see a potential problem when performing ACK pacing or smooth scheduling on ADSL. With our current non-preemptive packet schedulers a minimum delay variation is introduced, according to the transmission delay of data packets, which will limit the accuracy of the ACK pacing mechanism. This might pose a problem when trying to perform a smooth ACK scheduling.

On connections with a normalized bandwidth ratio k factor that approaches one or the delayed ACK factor d, implies that ACK packets from the downstream data packets can consume a large amount of the upstream bandwidth. It might be worth developing some ACK-filtering methods, to avoid ACK packets to consume too much of the upstream bandwidth.

With the experiences from our deployment on a real-world production network, we see a need for some more advanced or effective forms of traffic classification, primarily to avoid mis-categorization and to detect traffic anomalies like network viruses. We presume it should be based on some kind of traffic behavior analysis.

Beyond a Middlebox Solution

With our middlebox solution the focus has been put on solving problems without any modification of or cooperation with other network elements. If this restriction is removed a new set of interesting possibilities arise.

We have shown that the non-preemptive packet scheduler is affected by the Maximum Transfer Unit (MTU). It is possible to change the effective MTU on the ADSL connection by changing the PPP setup to use the PPP Multilink Protocol RFC1990[69]. This PPP protocol offers the ability to interleave packets by splitting and recombining packets. This naturally introduce a new level of overhead dependent on the chosen PPP fragmentation size. Using a smaller effective MTU size on the ADSL connection, can lower the delay bound of the non-preemptive packet scheduler although, at the expense of a higher overhead.

We could also utilize the true power of ATM, which has the ability to perform preemptive packet scheduling, by using several ATM Virtual Circuit (VC)s. With preemptive packet scheduling we could with ease support the required jitter delay bound for real-time and variation-sensitive application like Voice over IP (VoIP). A separate VC for ACK packet would also be beneficial as a more smooth clocking of ACK packets would help avoid bursts.

Performing header compression for small packets, e.g. ACK packet would also be beneficial. We have shown that ACK packet consume two ATM cells giving an overhead for an ACK packet of 62%. With ACK header compression this could with ease be reduced to one ATM cell. It would make sense to perform header compression for ACK packet as they are frequently used packets and can constitute a fairly large amount of a small upstream connection.

We have seen evidence that Internet Service Provider (ISP)s are in the progress of developing and deploying new Customer Premise Equipment (CPE) (much like our middlebox) for supporting VoIP on ADSL. The Voice over DSL (VoDSL) technology they want to deploy is, from our view point, basically the idea of using a separate ATM VC for the voice packets.

Bibliography

[1] I.363.5 – B-ISDN ATM adaptation layer specification:, Type 5 AAL. Standard, International Telecommunication Union (ITU-T), August 1996.

[2] I.432 B-ISDN user-network interface – physical layer specification. Standard, International Telecommunication Union (ITU-T), 1996.

[3] Data-Over-Cable Service Interface Specifications, radio frequency interface specification 1.0. Standard ANSI/SCTE 22-1, Cable Television Laboratories, Inc., November 1999. URL http://www.scte.org/documents/pdf/ANSISCTE2212002DSS0205.pdf.

[4] G.992.1: Asymmetric digital subscriber line (ADSL) transceivers. Standard, International Telecommunication Union (ITU-T), June 1999.

[5] G.992.2: Splitterless asymmetric digital subscriber line (ADSL) transceivers. Standard, International Telecommunication Union (ITU-T), June 1999.

[6] Data-Over-Cable Service Interface Specifications, radio frequency interface specification 1.1. Standard ANSI/SCTE 23-1, Cable Television Laboratories, Inc., 2000. URL http://www.scte.org/documents/pdf/ANSISCTE2312002DSS0209.pdf.

[7] Y.1541 - Network performance objective for ip-based services. Standard, International Telecommunication Union (ITU-T), May 2002.

[8] Befolkningens brug af internet 2. halvår 2003. Technical report, Danmarks Statistik, November 2003. URL http://www.dst.dk/Statistik/IT/Befolkningen.aspx.

[9] Data-Over-Cable Service Interface Specifications, radio frequency interface specification 2.0. Standard ANSI/SCTE 79-1, Cable Television Laboratories, Inc., 2003. URL http://www.scte.org/documents/pdf/ANSISCTE7912003DSS0201.pdf.

[10] G.107 – The E-model, a computational model for use in transmission planning. Standard, International Telecommunication Union (ITU-T), March 2003.

[11] G.109 – Definition of categories of speech transmission quality. Standard, International Telecommunication Union (ITU-T), March 2003.

[12] G.114 – One-way transmission time. Standard, International Telecommunication Union (ITU-T), May 2003.

[13] A. Aggarwal, S. Savage, and T. Anderson. Understanding the Performance of TCP Pacing. In *Proceedings of the 2000 IEEE Infocom Conference*, March 2000.

[14] Dick Van Aken and Sascha Peckelbeen. *Encapsulation Overhead(s) in ADSL access networks*. Thomson SpeedTouch, v1.0 edition, June 2003. URL www.speedtouch.com.

[15] M. Allman, V. Paxson, and W. Stevens. RFC2581 – TCP congestion control. Technical report, RFC Editor, April 1999.

[16] Oskar Andreasson. *Ipsysctl tutorial 1.0.4*, August 2004. URL http://ipsysctl-tutorial.frozentux.net/.

[17] *802.2, Logical Link Control*. ANSI/IEEE, iso/iec 8802-2:1998 edition, 1998.

[18] H. Balakrishnan, V. N. Padmanabhan, G. Fairhurst, and M. Sooriyabandara. RFC3449 – TCP performance implications of network path asymmetry. Technical report, RFC Editor, December 2002.

[19] Hari Balakrishnan and Venkata N. Padmanabhan. How network asymmetry affects TCP. *IEEE Communications Magazine*, April 2001.

[20] Hari Balakrishnan, Venkata N. Padmanabhan, and Randy H. Katz. The effects of asymmetry on TCP performance. In *MobiCom '97: Proceedings of the 3rd annual ACM/IEEE international conference on Mobile computing and networking*, pages 77–89. ACM Press, 1997. ISBN 0-89791-988-2.

[21] S. Bellovin. RFC1579 – Firewall-friendly FTP. Technical report, RFC Editor, February 1994.

[22] Jon C. R. Bennett and Hui Zhang. WF^2Q: Worst-case fair weighted fair queueing. In *INFOCOM (1)*, pages 120–128, 1996. URL citeseer.ist.psu.edu/zhang96wfq.html.

[23] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. RFC2475 – An architecture for differentiated service. Technical report, RFC Editor, December 1998.

[24] J. Border, M. Kojo, J. Griner, G. Montenegro, and Z. Shelb. RFC3135 – Performance enhancing proxies intended to mitigate link-related degradations. Technical report, RFC Editor, June 2001.

[25] R. Braden, D. Clark, and S. Shenker. RFC1633 – Integrated services in the Internet architecture: an overview. Technical report, RFC Editor, June 1994.

[26] Jesper Dangaard Brouer and Jørgen Sværke Hansen. Experiences with reducing TCP performance problems on ADSL. Technical Report Number 04/07, DIKU, May 2004.

[27] B. Carpenter and S. Brim. RFC3234 – Middleboxes: Taxonomy and issues. Technical report, RFC Editor, February 2002.

[28] Stuart Cheshire. Latency and the quest for interactivity. In *Synchronous Person-to-Person Interactive Computing Environments Meeting*. Volpe Welty Asset Management, L.L.C., November 1996. URL http://www.stuartcheshire.org/papers/LatencyQuest.ps.

[29] David D. Clark. RFC813 – Window and acknowledgement strategy in TCP. Technical report, RFC Editor, July 1982.

[30] R. Cohen and S. Ramanathan. TCP for high performance in hybrid fiber coaxial broad-band access networks. *IEEE/ACM Transactions on Networking*, 6:15–19, February 1998.

[31] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. In *SIGCOMM '89: Symposium proceedings on Communications architectures & protocols*, pages 1–12. ACM Press, 1989. ISBN 0-89791-332-9.

[32] Martin Devera. *Hierachical token bucket theory*, May 2002. URL http://luxik.cdi.cz/~devik/qos/htb/manual/theory.htm.

[33] *Technical – Frequently Asked Questions*. DSL forum, September 2001. URL http://www.dslforum.org/aboutdsl/tech_faqs.html.

[34] Robert C. Durst, Gregory J. Miller, and Eric J. Travis. TCP extensions for space communications. In *MobiCom '96: Proceedings of the 2nd annual international conference on Mobile computing and networking*, pages 15–26. ACM Press, 1996. ISBN 0-89791-872-X.

[35] Phil Dykstra. *Protocol Overhead*, April 2003. URL http://sd.wareonearth.com/~phil/net/overhead/.

[36] R. Braden (Editor). RFC1122 – Requirements for internet hosts - communication layers. Technical report, RFC Editor, October 1989.

[37] Ericsson, editor. *Understanding Telecommunications*. Ericsson, September 2003. URL http://www.ericsson.com/support/telecom/.

[38] Sally Floyd and Van Jacobson. Link-sharing and resource management models for packet networks. *IEEE/ACM Transactions on Networking*, 3(4):365–386, 1995. URL citeseer.ist.psu.edu/article/floyd95linksharing.html.

[39] Fairhurst G., Samaraweera N.K.G, Sooriyabandara, M. Harun, H. Hodson K., and R. Donardio. Performance issues in asymmetric service provision using broadband satellite. In *IEEE Proceedings on Communication*, volume Vol.148 no.2, pages 95–99, 2001.

[40] David J. Greggains. Adsl and high bandwidth over copper lines. *Int. Journal Network Management*, 7(5):277–287, 1997. ISSN 1099-1190.

[41] G. Gross, M. Kaycee, A. Lin, A. Malis, and J. Stephens. RFC2364 – PPP over AAL5. Technical report, RFC, July 1998.

[42] D. Grossman. RFC3260 – New ew terminology and clarifications for diffserv. Technical report, RFC Editor, April 2002.

[43] D. Grossman and J. Heinanen. RFC2684 – Multiprotocol encapsulation over atm adaptation layer 5. Technical report, RFC Editor, September 1999.

[44] J. Heinanen and R. Guerin. RFC2698 – A two rate three color marker. Technical report, RFC Editor, September 1999.

[45] T. Henderson. TCP performance over satellite channels. Technical report, University of California, Berkeley, 1999. URL citeseer.ist.psu.edu/henderson99tcp.html.

[46] T. Henderson and R. Katz. Transport protocols for internet-compatible satellite networks. *IEEE Journal of Selected Areas in Communications*, Vol.17 No.2:345–359, February 1999. URL citeseer.ist.psu.edu/henderson99transport.html.

[47] A S Hornby, editor. *Oxford Advanced Learner's Dictionary of Current English*. Oxford University Press, Walton Street, Oxford OX2 6DP, sixth edition, 2003.

[48] Bert Hubert, Thomas Graf, Greg Maxwell, Remco van Mook, Martijn van Oosterhout, Paul B Schroeder, Jasper Spaans, and Pedro Larroy. *Linux Advanced Routing & Traffic Control*. LARTC, rev.1.43 edition, October 2003. URL http://www.lartc.org/.

[49] Van Jacobson. Congestion avoidance and control. In *ACM SIGCOMM '88*, pages 314–329, Stanford, CA, August 1988. URL citeseer.ist.psu.edu/article/jacobson88congestion.html.

[50] Lampros Kalampoukas, Anujan Varma, and K. K. Ramakrishnan. Improving TCP throughput over two-way asymmetric links: analysis and solutions. In *Proceedings of the 1998 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, pages 78–89. ACM Press, 1998. ISBN 0-89791-982-3.

[51] Shrikrishna Karandikar, Shivkumar Kalyanaraman, Prasad Bagal, and Bob Packer. TCP rate control. *SIGCOMM Computer Communications Rev.*, 30(1): 45–58, 2000. ISSN 0146-4833.

[52] Srinivasan Keshav. On the efficient implementation of fair queueing. *Journal of Internetworking Research and Experience*, 1991.

[53] James F. Kurose and Keith Ross. *Computer Networking: A Top-Down Approach Featuring the Internet*. Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN 0201976994.

[54] T. V. Lakshman, Upamanyu Madhow, and Bernhard Suter. Window-based error recovery and flow control with a slow acknowledgement channel: A study of TCP/IP performance. In *INFOCOM (3)*, pages 1199–1209, 1997. URL citeseer.ist.psu.edu/lakshman97windowbased.html.

[55] M. Laubach and J. Halpern. RFC2225 – Classical IP and ARP over ATM. Technical report, RFC, April 1998.

[56] Alberto Leon-Garcia and Indra Widjaja. *Communication Networks: Fundamental Concepts and Key Architectures*. McGraw-Hill School Education Group, 1999. ISBN 0-07-242349-8.

[57] L. Mamakos, K. Lidl, J. Evarts, D. Carrel, D. Simone, and R. Wheeler. RFC2516 – a method for transmitting PPP over Ethernet (PPPoE). Technical report, RFC Editor, February 1999.

[58] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. RFC2018 – TCP selective acknowledgment options. Technical report, RFC Editor, October 1996.

[59] Paul E. McKenney. Stochastic fairness queuing. *Interworking: Research and Experience*, Vol.2, January 1991. URL citeseer.ist.psu.edu/mckenney91stochastic.html.

[60] Ivan Tam Ming-Chit, Du Jinsong, and Weiguo Wang. Improving TCP performance over asymmetric networks. *ACM SIGCOMM Computer Communication Review*, 30(3):45–54, 2000. ISSN 0146-4833.

[61] Wael Noureddine and Fouad Tobagi. The transmission control protocol. URL citeseer.ist.psu.edu/noureddine02transmission.html.

[62] V. Paxson and M. Allman. RFC2988 – computing TCP's retransmission timer. Technical report, RFC Editor, November 2000.

[63] Jon Postel. RFC793 – Transmission Control Protocol. Technical report, RFC Editor, September 1981.

[64] Jon Postel and J. Reynolds. RFC959 – File Transfer Protocol (FTP). Technical report, RFC Editor, October 1985.

[65] K. Ramakrishnan, S. Floyd, and D. Black. RFC3168 – The addition of explicit congestion notification (ecn) to ip. Technical report, RFC Editor, September 2001.

[66] J. Reynolds and Jon Postel. RFC1700 – assigned numbers. Technical report, RFC Editor, October 1995. URL http://www.iana.org/.

[67] N. K. G. Samaraweera. Return link optimization for internet service provision using dvb-s networks. *ACM Computer Communications Review (CCR)*, 29(3): 4–19, 1999.

[68] M. Shreedhar and George Varghese. Efficient fair queueing using deficit round robin. In *SIGCOMM '95: Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*, pages 231–242. ACM Press, 1995. ISBN 0-89791-711-1.

[69] K. Sklower, B. Lloyd, G. McGregor, D. Carr, and T. Coradetti. RFC1990 – The PPP multilink protocol (MP). Technical report, RFC Editor, August 1996.

[70] Michael Smith and Steve Bishop. *Flow Control in the Linux Network Stack*. Cambridge, August 2004. URL http://www.cl.cam.ac.uk/~pes20/Netsem/.

[71] Neil T. Spring, Maureen Chesire, Mark Berryman, Vivek Sahasranaman, Thomas Anderson, and Brian N. Bershad. Receiver based management of low bandwidth access links. In *INFOCOM*, pages 245–254, 2000. URL citeseer.ist.psu.edu/spring00receiver.html.

[72] W. Richard Stevens. *TCP/IP illustrated (vol. 1): the protocols*. Addison-Wesley Longman Publishing Co., Inc., 1993. ISBN 0-201-63346-9.

[73] W. Richard Stevens. *TCP/IP illustrated (vol. 3): TCP for transactions, HTTP, NNTP, and the Unix domain protocols*. Addison Wesley Longman Publishing Co., Inc., 1996. ISBN 0-201-63495-3.

[74] W. Richard Stevens and Gary R. Wright. *TCP/IP illustrated (vol. 2): The implementation*. Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN 0-201-63354-X.

[75] Yishen Sun, C. C. Lee, Randall Berry, and A. H. Haddad. A load-adaptive ACK pacer for TCP traffic control. URL citeseer.lcs.mit.edu/692887.html.

[76] Andrew S. Tanenbaum. *Computer Networks*. Prentice Hall PTR, 1985. ISBN 0-13-394248-1.

[77] Jonathan S. Turner. New directions in communications (or which way to the information age?). *IEEE Communications Magazine*, vol.24(10):8–15, October 1986. URL http://www.comsoc.org/livepubs/ci1/public/anniv/turner.html.

[78] Zheng Wang. *Internet QoS: Architectures and Mechanisms for Quality of Service*. Morgan Kaufmann Publishers Inc., 2001. ISBN 1-55860-608-4.

[79] Huan-Yun Wei and Ying-Dar Jason Lin. A survey and measurement-based comparison of bandwidth management techniques. *IEEE Communications Surveys and Tutorials*, 5(2), 2003.

[80] Huan-Yun Wei, Shih-Chiang Tsao, and Ying-Dar Lin. Assessing and improving TCP rate shaping over edge gateways. *IEEE Transactions on Computers*, 53(3): 259–275, march 2004.

[81] Lixia Zhang, Scott Shenker, and David D. Clark. Observations on the dynamics of a congestion control algorithm: the effects of two-way traffic. In *Proceedings of the conference on Communications architecture & protocols*, pages 133–147. ACM Press, 1991. ISBN 0-89791-444-9.

[82] Yin Zhang and Vern Paxson. Detecting backdoors. In *In Proc. of 9th USENIX Security Symposium*, 2000. URL citeseer.ist.psu.edu/article/zhang00detecting.html.

[83] Yin Zhang and Vern Paxson. Detecting stepping stones. In *In Proc. of 9th USENIX Security Symposium*, 2000. URL gunther.smeal.psu.edu/article/zhang00detecting.html.

Acronyms

AAL ATM Adaption Layer.

AAL5 ATM Adaption Layer type 5.

ADSL Asymmetric Digital Subscriber Line.

ANSI American National Standards Institute.

ATM Asynchronous Transfer Mode.

BBRAS Broad Band Remote Access Router.

CBQ Class-Based Queueing.

CPCS Common Part Convergence Sublayer.

CPE Customer Premise Equipment.

CS Convergence Sublayer.

DHCP Dynamic Host Configuration Protocol.

DNS Domain Name Service.

DOCSIS Data-Over-Cable Service Interface Specification.

DoS Denial of Service.

DRR Deficit Round Robin.

DS Differentiated Services.

DSL Digital Subscriber Line.

DSLAM Digital Subscriber Line Access Multiplexer.

ECN Explicit Congestion Notification.

ESFQ Extended Stochastic Fairness Queuing.

FCS Frame Check Sequence.

FIFO First In First Out.

FTP File Transfer Protocol.

FQ Fair Queueing.

GPS Generalized Processor Sharing.

HTB Hierarchical Token Bucket.

IANA Internet Assigned Numbers Authority.

IS Integrated Services.

ISDN Integrated Services Digital Network.

ISP Internet Service Provider.

LAN Local Area Network.

LLC Logic Link Control.

MAC Media Access Control.

MTU Maximum Transfer Unit.

NAT Network Address Translation.

NIDS Network Intrusion Detection System.

NLPID Network Layer Protocol IDentifier.

OAM Operations Administration and Management.

OUI Organizationally Unique Identifier.

P2P peer-to-peer.

PID Protocol IDentifier.

PEP Performance Enhancing Proxy.

POTS Plain Old Telephone Service.

PPP Point-to-Point Protocol.

PPPoA PPP over AAL5.

PPPoE PPP over Ethernet.

PSTN Public Switched Telephone Network.

RTO Retransmission TimeOut.

RTT Round-Trip Time.

SACK Selective ACKnowledgement.

SAR Segmentation And Reassembly.

SCFQ Self-Clocking Fair Queueing.

SFQ Stochastic Fairness Queuing.

SLA Service Level Agreement.

SLS Service Level Specification.

SMTP Simple Mail Transfer Protocol.

SNAP SubNetwork Attachment Point.

srTCM Single Rate Three Color Marker.

SSCS Service Specific Convergence Sublayer.

TCA Traffic Conditioning Agreement.

TCS Traffic Conditioning Specification.

TCP Transmission Control Protocol.

TOS Type Of Service.

trTCM Two Rate Three Color Marker.

UNI User Network Interface.

VC Virtual Circuit.

VoIP Voice over IP.

VoDSL Voice over DSL.

VPN Virtual Private Network.

WRR Weighted Round Robin.

WFQ Weighted Fair Queueing.

WF2Q Worst-case Fair Weighted Fair Queueing.

QoS Quality of Service.

Index

Appendix A

Appendix

A.1 Transmission Delay

This appendix show a practical test of the transmission and processing delay as it should also give us an indication of achievable delay bounds in evaluating our solution.

The tests in this appendix are performed on a unused 2 Mbit/512 Kbit ADSL line from Tiscali[1]. We use different sized ping packet to evaluate the delay. As we are using a unused ADSL we assume that there is no queuing delay. Thus our ping measurement include processing, transmission and propagation delay. We assume that the propagation delay negligible (0.025 ms) as demonstrated in Section 3.4 on page 22.

To get a feel of our test environment the tests are done between our testhost (which delivers data to our downstream transfers) and the ADSL. The Round-Trip Time (RTT) for different sized (ping) packets are shown in figure A.1. Apart from some minor spikes, the latency is quite stable, indicating no congestion along the network path.

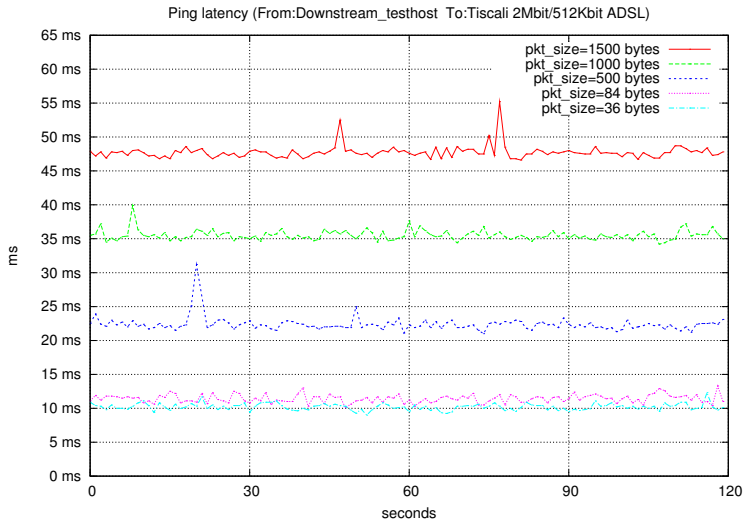


Figure A.1: Ping RTT with different packet sizes, from our downstream-testhost (which delivers data to our downstream transfers) to a clean 2 Mbit/512 Kbit ADSL line from Tiscali.

For the smallest packet allowed by ping 36 bytes (8 bytes payload, ICMP header 8 bytes, IP header 20 bytes) the average RTT observed is 10 ms. The 36 bytes is transmitted in a single ATM frame using 53 bytes. The transmission delay ($d_{transmit}$) of a single ATM frame only contributes with 1 ms (see transmit-delay Table on the following page). As we assume no queueing delay (d_{queue}), the main contribution to this delay must be the processing delay ($d_{process}$) of the ADSL modem. Thus, the processing delay ($d_{process}$) in the ADSL modem, can be seen from the figure A.1 as the lowest achievable delay bound. The ADSL line coding technique and especially the interleaving depth of the error correction scheme can introduce a delay in the order of 60 ms (usually configured lower). With the interleaver turned off the residual latency of standard ADSL is 2 ms.[2]

[1] The Tele2 ADSL from section 3.3 on page 19 were not used because the IP is used to run different services, and different clients try to reconnect to the service when it is shutdown.

[2] http://www.dslforum.org/aboutdsl/tech_faqs.html (August 2004)

Formula A.1

$$d_{transmit} = \frac{PacketSize}{LinkRate}$$

The transmission delay ($d_{transmit}$) in each direction can be calculated as specified in formula A.1 on the page before.

Transmit delay on ADSL		
Downstream capacity:	2,000,000	Kbit/s
Upstream capacity:	512,000	Kbit/s
Baseline delay:	9	ms

Packet size		**Transmit delay (ms)**			
IP	+ATM	Downstream	Upstream	Down+Up	Incl.baseline
36	53	0.21 ms	0.83 ms	1.04 ms	10.04 ms
86	106	0.42 ms	1.66 ms	2.08 ms	11.08 ms
500	583	2.33 ms	9.11 ms	11.44 ms	20.44 ms
1000	1113	4.45 ms	17.39 ms	21.84 ms	30.84 ms
1500	1696	6.78 ms	26.50 ms	33.28 ms	42.28 ms

The transmit-delay Table on this page shows that the transmission delay on the ADSL line is the major contributor to the RTT delay illustrated in figure A.1 on the page before. The table shows the calculated transmission delay for the packet sizes used in the test, on a 2 Mbit/512 Kbit line. The packet sizes are adjusted to account for the ATM header overhead and frame alignment.

Figure A.2 on the following page illustrate how much the rest of the network path contributes to the delay. The same ping test (as figure A.1 on the page before) is performed, but instead of pinging the ADSL modem, the DSLAM is ping'ed [3].

The graph in figure A.2 on the following page contains some spikes, which tell us that the networks path is active and carry packets other than ours ICMP ping packets. The average RTT of 1500 bytes between the testhost and DSLAM is 5.61 ms, and $d_{transmit}$ on a 2 Mbit/512 Kbit ADSL line is 42.28 ms, the sum is 47.89 ms which matches the average RTT of 47.80 ms from figure A.1 on the page before. This indicate that our measurements and calculations add up.

[3] We assume that the DSLAM is the network hop one hop before the ADSL modem.

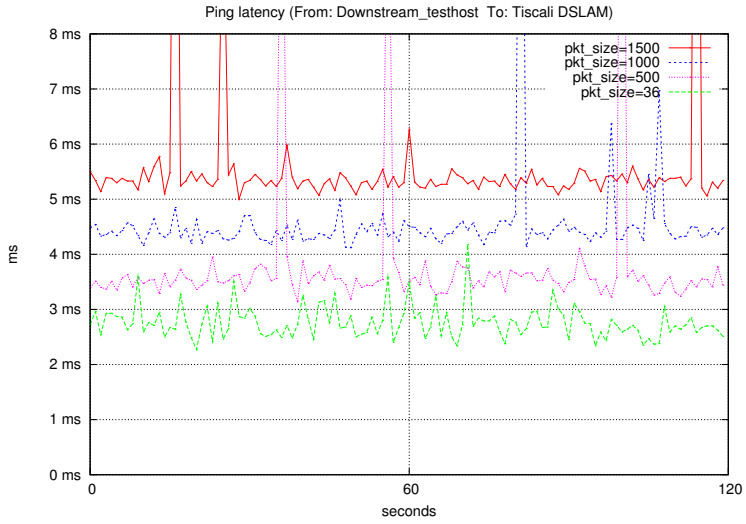
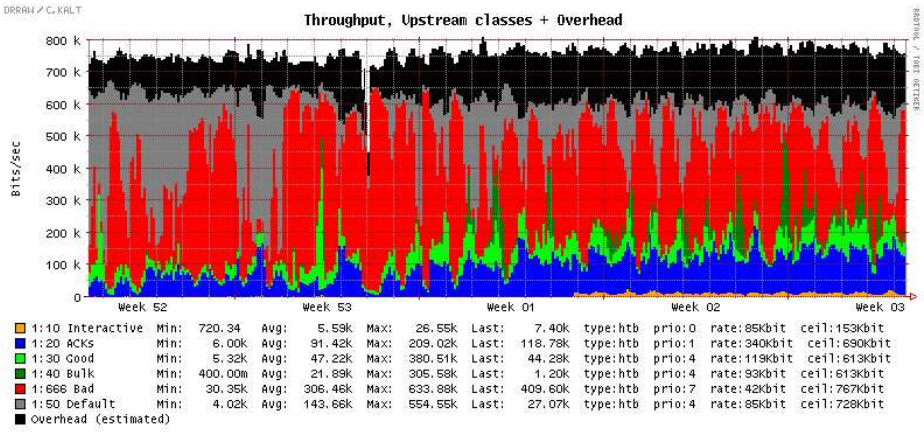


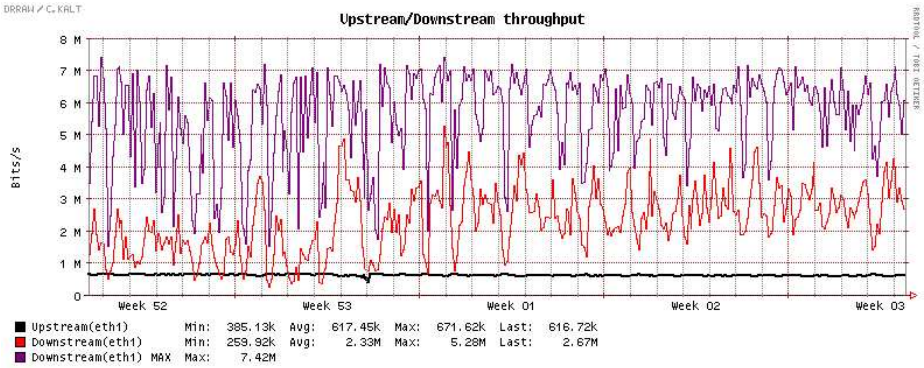
Figure A.2: Ping RTT with different packet sizes, from our downstream-testhost to the DSLAM located just before the ADSL line from Tiscali.

A.2 Extra Graphs: Real-world One Month Overview

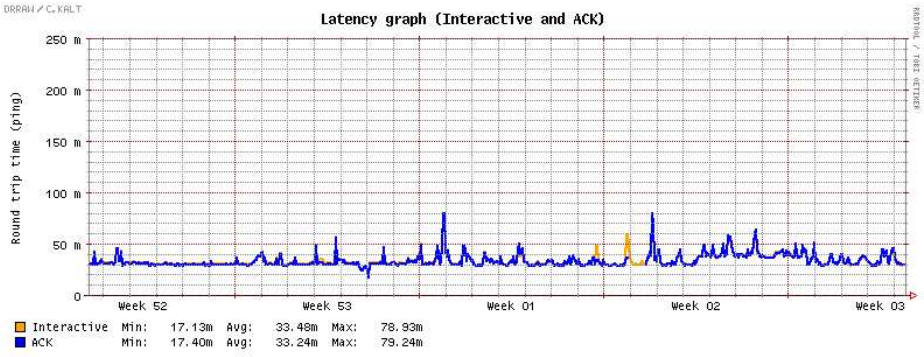
A one month overview from our real-world production ADSL connection, which shows that we have obtained our goal and latency requirements. Unfortunately, we have not have time to describe the details of these graphs.



(a) Upstream throughput for each class



(b) Throughput



(c) Latency for high-priority classes

Figure A.3: One month, from 21/12-2004 to 21/1-2005.

Appendix B

Code

B.1 Bandwidth-tester

The bandwidth-tester script uses a configuration file per test which specifies the test-setup. An example is given below.

```
File: test303.setup
-------------------
 testname=test303-noprio-pfifo
 download1=http://www.trykdenaf.dk/sletmig.23mb
 download1_times=3
 sleep1=60
 upload1=~/bandwidth-tests/upload.2mb
 upload1_times=3
 sleep2=60
 upload2=~/bandwidth-tests/upload.6mb
 upload2_times=1
 ping_host=tyr.diku.dk
```

The test-setup file above specifies that the packetdump filename will be named "test303-noprio-pfifo.dump" and the following actions; Three (wget) downloads are started (of http://www.trykdenaf.dk/sletmig.23mb). After a sleep period of 60 seconds, three (scp) uploads are started (of the file upload.2mb). After the three uploads finish a new sleep period of 60 seconds starts. After that a single (scp) upload is started (of upload.6mb). The download is allowed to finish before ending the testrun. Throughout the test a ping of the host "tyr.diku.dk" is performed to generate some latency statistics. [1]

B.1.1 Script

```
#!/bin/sh

e=echo
scp_dest=brouer@ask.diku.dk:/dev/null

if [ -z "$1" ]; then
    echo "[ERROR] Missing test number!"
    exit 2
fi
testnum="$1"
testsetup=test${testnum}.setup

function info() {
    perl -e 'print time." [",scalar localtime()."] "';
    echo "(test${testnum}) $@"
}


echo "Test number $testnum"
echo "--------------------"

if [ -f $testsetup ]; then
   info "Using setup file: $testsetup"
   source $testsetup
elif [ -f ../$testsetup ]; then
   info "Using setup file: ../$testsetup"
   source ../$testsetup
else
   info "ERROR: can not find testXXX.setup file"
   exit 1
fi

#Should contain something like:
# testname=gw-koll-test${testnum}-8Mbit
# sleep1=10
# upload1=upload.3mb
# download1=http://www.trykdenaf.dk/sletmig.50mb
# ping_host=tyr.diku.dk
```

[1] An example of the resulting graph can be seen in figure 3.9(a) on page 28.

```
if [ -z "$testname" ]; then
    echo "[ERROR] incorrect content of setup file: \"${testsetup}\""
    exit 3
fi

qos="qos.setup"
echo "Recording QoS setting to file: $qos"
# ---------------------------------------------
echo "" >> $qos
echo "Qdisc:"  >> $qos
echo "------"  >> $qos
tc -d -s qdisc >> $qos
echo "" >> $qos
netcards="eth0 eth1 eth2"
for eth in $netcards; do
    echo ""                     >> $qos
    echo "Class: $eth"          >> $qos
    echo "-----------"          >> $qos
    tc -d -s class ls dev $eth >> $qos
    echo ""                     >> $qos
done

filter="filter.setup"
echo "Recording Filter setting to file: $filter"
# ------------------------------------------------
echo "" >> $filter
echo "Filter:" >> $filter
echo "-------" >> $filter
iptables -t mangle -nL >> $filter

# exec > $testname.output 2>&1

function test_pid() {
    pid="$1"
    txt="$2"
#    ps -p $pid_wget > /dev/null 2>&1
    ps -p $pid > /dev/null 2>&1
    res=$?
#    if [ $res -ne 0 ]; then
# info "  WARNING -- Pid test failed for ($pid) $txt"
#    fi
}

info "tcpdump startet file: ${testname}.dump"
$e tcpdump -s 100 -i eth1 -w ${testname}.dump &
pid_tcpdump=$!

# Ping stats
# ---------------
if [ -n "$ping_host" ]; then
    info "Startet ping stat of $ping_host"
    $e ping $ping_host > ping__${ping_host} &
    pid_ping=$!
fi

# List of ping_hosts
if [ -n "$ping_hosts" ]; then
    for host in $ping_hosts; do
info "Startet ping stat of $host"
$e ping $host > ping__${host} &
    done
fi




# Wget
# ---------------
if [ -n "$download1" ]; then
    info "Wget startet"
    $e wget --output-document=/dev/null --proxy=off --progress=dot:binary $download1 -o ${testname}.log &
    pid_wget=$!
fi

# Wget repeat
#  Starts some extra parallel wgets
# ---------------
if [ -n "$download1_times" ]; then
    info "Extra $download1_times Wgets "
    a=1
```

```
    LIMIT=$download1_times
    while [ "$a" -lt $LIMIT ]; do
sleep 1
let "a+=1"
info " Wget#$a "
$e wget --output-document=/dev/null --proxy=off --progress=dot:binary $download1 -o ${testname}.log${a} &
    done
fi


# Sleep1 / delay
# ---------------
if [ -n "$sleep1" ]; then
    info "Sleeping for $sleep1 sec."
    sleep $sleep1
fi

test_pid "$pid_wget" "Wget-download1"

#
# Starts some extra parallel SCP's
# ---------------
if [ -n "$upload1_times" ]; then
    info "Extra $upload1_times SCP's "
    a=1
    LIMIT=$upload1_times
    while [ "$a" -lt $LIMIT ]; do
#sleep 1
let "a+=1"
info " scp#$a "
$e scp $upload1 $scp_dest &
    done
fi

# Scp1
# ---------------
if [ -n "$upload1" ]; then
    info "Scp of file $upload1 STARTED"
    $e scp $upload1 $scp_dest
    info "Scp of file $upload1 FINISHED"
fi

# Sleep2 / delay
# ---------------
if [ -n "$sleep2" ]; then
    info "Sleeping for $sleep2 sec."
    sleep $sleep2
fi

test_pid "$pid_wget" "Wget finished before scp/upload2 started..."

# Scp2
# ---------------
if [ -n "$upload2" ]; then
    info "Scp of file $upload2 STARTED"
    $e scp $upload2 $scp_dest
    info "Scp of file $upload2 FINISHED"
fi

test_pid "$pid_wget" "Wget finished before the scp('s) finished..."

if [ -n "$pid_wget" ]; then
    info "Waiting for wget to finish"
    wait $pid_wget
    info "Wget finished"
fi

# Sleep3 / delay
# ---------------
if [ -n "$sleep3" ]; then
    info "Sleeping for $sleep3 sec."
    sleep $sleep3
fi

if [ -n "$pid_ping" ]; then
    info "Killing/Stopping ping"
    $e kill $pid_ping
fi

if [ -n "$ping_hosts" ]; then
```

```
    info "Killing/Stopping ALL pings"
    $e killall ping
fi


info "Killing/Stopping tcpdump"
$e kill $pid_tcpdump

info "Deleting all sletmig files"
$e rm -f sletmig.*
```

B.2 Overhead Patch

The software package `iproute2` which contains the Traffic Control `tc` user space program has been unmaintained for quite a while. The official "current" release by Alexey Kuznetsov dates from October 2000, but the recommended/latest version to use is from January 2002, called `iproute2-2.4.7-now-ss020116-try` (we will refer to this as *iproute2-2.4.7-old*). This version requires several patches to use newer schedulers, which are included in the official Linux kernel, e.g. the HTB scheduler (which we use).

Stephen Hemminger (shemminger@osdl.org) has started to maintain the `iproute2` utility, starting in March 2004 and getting generally accepted around June 2004. The only distribution (which I know of) which uses the new version of `iproute2` is Gentoo.

The homepage for the new `iproute2`:
http://developer.osdl.org/dev/iproute2

I have modified an existing overhead patch for HTB, by Walter Karshat (wkarshat@yahoo.com). This patch (without my modifications) has been accepted into the new version of iproute2. For this reason, I include patch sets for both the new and the old iproute2.

B.2.1 iproute2-2.6.9: tc_core.c

Changes to (`tc/tc_core.c`) the rate table calculation in the Traffic Control `tc` user space program. Against `iproute2-2.6.9`.

Listing B.1: Patch: `tc_core.c`

```
Index: iproute2/iproute2-2.6.9/tc/tc_core.c
diff -u iproute2/iproute2-2.6.9/tc/tc_core.c:1.1 iproute2/iproute2-2.6.9/tc/tc_core.c:1.7
--- iproute2/iproute2-2.6.9/tc/tc_core.c:1.1    Wed Nov 3 15:09:08 2004
+++ iproute2/iproute2-2.6.9/tc/tc_core.c        Sun Nov 7 13:33:22 2004
@@ -42,6 +42,35 @@
        return tc_core_usec2tick(1000000*((double)size/rate));
 }

+#ifndef ATM_CELL_SIZE
+#define ATM_CELL_SIZE          53  /* ATM cell size incl. header */
+#endif
+#ifndef ATM_CELL_PAYLOAD
+#define ATM_CELL_PAYLOAD       48  /* ATM payload size */
+#endif
+
+#define ATM_ALIGN y
+
+/*
+ The align_to_cells is used for determining the (ATM) SAR alignment
+ overhead at the ATM layer. (SAR = Segmentation And Reassembly)
+ This is for example needed when scheduling packet on an ADSL
+ connection. The ATM-AAL overhead should preferably be added in the
+ kernel when doing table lookups (due to precision/alignment of the
+ table), if not the ATM-AAL overhead should be added to the size
+ before calling the function. --Hawk, d.7/11-2004. <hawk@diku.dk>
+ */
+unsigned tc_align_to_cells(unsigned size, int cell_size, int cell_payload)
+{
+  int linksize, cells;
+  cells = size / cell_payload;
+  if ((size % cell_payload) > 0) {
+     cells++;
+  }
+  linksize = cells * cell_size;
+  return linksize;
+}
+
 /*
    rtab[pkt_len>>cell_log] = pkt_xmit_time
  */
@@ -63,10 +92,15 @@
        }
        for (i=0; i<256; i++) {
                unsigned sz = (i<<cell_log);
-               if (overhead)
-                       sz += overhead;
+               if (overhead) {
+                       // Is now done in the kernel (eg. sch_htb.c, func L2T )
+                       // sz += overhead;
+               }
                if (sz < mpu)
                        sz = mpu;
+#ifdef ATM_ALIGN
+               sz = tc_align_to_cells(sz, ATM_CELL_SIZE, ATM_CELL_PAYLOAD);
+#endif
                rtab[i] = tc_core_usec2tick(1000000*((double)sz/bps));
        }
        return cell_log;
```

B.2.2 iproute2-2.4.7-old: tc_core.c + q_htb.c

Changes to the rate table calculation in the Traffic Control `tc` user space program. Against `iproute2-2.4.7-old`.

Listing B.2: Patch: `tc_core.c` (for old iproute2)

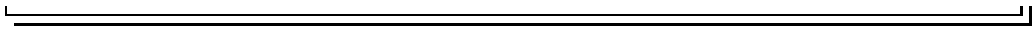
```
--- iproute2-2.4.7-now-ss020116-try.orig/tc/tc_core.c Sun Apr 16 19:42:55 2000
+++ iproute2-2.4.7-now-ss020116-try/tc/tc_core.c    Mon Nov 8 12:17:02 2004
@@ -42,6 +42,35 @@
        return tc_core_usec2tick(1000000*((double)size/rate));
 }

+#ifndef ATM_CELL_SIZE
+#define ATM_CELL_SIZE          53  /* ATM cell size incl. header */
+#endif
+#ifndef ATM_CELL_PAYLOAD
+#define ATM_CELL_PAYLOAD       48  /* ATM payload size */
+#endif
+
+#define ATM_ALIGN y
+
+/*
+ The align_to_cells is used for determining the (ATM) SAR alignment
+ overhead at the ATM layer. (SAR = Segmentation And Reassembly)
+ This is for example needed when scheduling packet on an ADSL
+ connection. The ATM-AAL overhead should preferably be added in the
+ kernel when doing table lookups (due to precision/alignment of the
+ table), if not the ATM-AAL overhead should be added to the size
+ before calling the function. --Hawk, d.7/11-2004. <hawk@diku.dk>
+ */
+unsigned tc_align_to_cells(unsigned size, int cell_size, int cell_payload)
+{
+  int linksize, cells;
+  cells = size / cell_payload;
+  if ((size % cell_payload) > 0) {
+     cells++;
+  }
+  linksize = cells * cell_size;
+  return linksize;
+}
+
 /*
    rtab[pkt_len>>cell_log] = pkt_xmit_time
  */
@@ -63,6 +92,9 @@
                unsigned sz = (i<<cell_log);
                if (sz < mpu)
                        sz = mpu;
+#ifdef ATM_ALIGN
+               sz = tc_align_to_cells(sz, ATM_CELL_SIZE, ATM_CELL_PAYLOAD);
+#endif
                rtab[i] = tc_core_usec2tick(1000000*((double)sz/bps));
        }
        return cell_log;
```

Changes to the HTB kode, this is mostly parameter parsing for the overhead parameter.

Listing B.3: Patch: `q_htb.c` (for old iproute2)

```
--- iproute2-2.4.7-now-ss020116-try+htb.orig/tc/q_htb.c Mon Nov 8 12:00:29 2004
+++ iproute2-2.4.7-now-ss020116-try/tc/q_htb.c Mon Nov 8 12:21:51 2004
@@ -34,10 +34,14 @@
                " default minor id of class to which unclassified packets are sent {0}\n"
                " r2q     DRR quantums are computed as rate in Bps/r2q {10}\n"
                " debug   string of 16 numbers each 0-3 {0}\n\n"
-               "... class add ... htb rate R1 burst B1 [prio P] [slot S] [pslot PS]\n"
+               "... class add ... htb rate R1 [burst B1] [mpu B] [overhead O]\n"
+               "                      [prio P] [slot S] [pslot PS]\n"
+               "                      [ceil R2] [cburst B2] [mtu MTU] [quantum Q]\n"
                " rate    rate allocated to this class (class can still borrow)\n"
                " burst   max bytes burst which can be accumulated during idle period\n"
+               " mpu     minimum packet size used in rate computations\n"
+               " overhead per-packet size overhead used in rate computations\n"
+
                " ceil    definite upper class rate (no borrows) {rate}\n"
```

```
                " cburst  burst but for ceil {computed}\n"
                " mtu     max packet size we create rate map for {1600}\n"
@@ -102,7 +106,9 @@
        struct tc_htb_opt opt;
        __u32 rtab[256], ctab[256];
        unsigned buffer=0,cbuffer=0;
-       int  cell_log =-1,ccell_log = -1,mtu;
+       int  cell_log =-1,ccell_log = -1;
+       unsigned mtu, mpu;
+       unsigned char mpu8 = 0, overhead = 0;
        struct rtattr *tail;

        memset(&opt, 0, sizeof(opt)); mtu = 1600; /* eth packet len */
@@ -119,6 +125,16 @@
                        if (get_u32(&mtu, *argv, 10)) {
                                explain1("mtu"); return -1;
                        }
+               } else if (matches(*argv, "mpu") == 0) {
+                       NEXT_ARG();
+                       if (get_u8(&mpu8, *argv, 10)) {
+                               explain1("mpu"); return -1;
+                       }
+               } else if (matches(*argv, "overhead") == 0) {
+                       NEXT_ARG();
+                       if (get_u8(&overhead, *argv, 10)) {
+                               explain1("overhead"); return -1;
+                       }
                } else if (matches(*argv, "quantum") == 0) {
                        NEXT_ARG();
                        if (get_u32(&opt.quantum, *argv, 10)) {
@@ -190,14 +206,18 @@
        if (!buffer) buffer = opt.rate.rate / HZ + mtu;
        if (!cbuffer) cbuffer = opt.ceil.rate / HZ + mtu;

-       if ((cell_log = tc_calc_rtable(opt.rate.rate, rtab, cell_log, mtu, 0)) < 0) {
+/* encode overhead and mpu, 8 bits each, into lower 16 bits */
+       mpu = (unsigned)mpu8 | (unsigned)overhead << 8;
+       opt.ceil.mpu = mpu; opt.rate.mpu = mpu;
+
+       if ((cell_log = tc_calc_rtable(opt.rate.rate, rtab, cell_log, mtu, mpu8)) < 0) {
                fprintf(stderr, "htb: failed to calculate rate table.\n");
                return -1;
        }
        opt.buffer = tc_calc_xmittime(opt.rate.rate, buffer);
        opt.rate.cell_log = cell_log;

-       if ((ccell_log = tc_calc_rtable(opt.ceil.rate, ctab, cell_log, mtu, 0)) < 0) {
+       if ((ccell_log = tc_calc_rtable(opt.ceil.rate, ctab, cell_log, mtu, mpu8)) < 0) {
                fprintf(stderr, "htb: failed to calculate ceil rate table.\n");
                return -1;
        }
@@ -221,6 +241,7 @@
        double buffer,cbuffer;
        SPRINT_BUF(b1);
        SPRINT_BUF(b2);
+       SPRINT_BUF(b3);

        if (opt == NULL)
                return 0;
@@ -243,10 +264,16 @@
            fprintf(f, "ceil %s ", sprint_rate(hopt->ceil.rate, b1));
            cbuffer = ((double)hopt->ceil.rate*tc_core_tick2usec(hopt->cbuffer))/1000000;
            if (show_details) {
-               fprintf(f, "burst %s/%u mpu %s ", sprint_size(buffer, b1),
-                       1<<hopt->rate.cell_log, sprint_size(hopt->rate.mpu, b2));
-               fprintf(f, "cburst %s/%u mpu %s ", sprint_size(cbuffer, b1),
-                       1<<hopt->ceil.cell_log, sprint_size(hopt->ceil.mpu, b2));
+               fprintf(f, "burst %s/%u mpu %s overhead %s ",
+                       sprint_size(buffer, b1),
+                       1<<hopt->rate.cell_log,
+                       sprint_size(hopt->rate.mpu&0xFF, b2),
+                       sprint_size((hopt->rate.mpu>>8)&0xFF, b3));
+               fprintf(f, "cburst %s/%u mpu %s overhead %s ",
+                       sprint_size(cbuffer, b1),
+                       1<<hopt->ceil.cell_log,
+                       sprint_size(hopt->ceil.mpu&0xFF, b2),
+                       sprint_size((hopt->ceil.mpu>>8)&0xFF, b3));
                fprintf(f, "level %d ", (int)hopt->level);
            } else {
                fprintf(f, "burst %s ", sprint_size(buffer, b1));
```



B.2.3 Kernel 2.4.27: sch_htb.c (non-intrusive)

This is the smallest and most non-intrusive kernel patch for the HTB scheduler. We take advantage of the overhead patch for the new version of iproute2, which hides the overhead parameter in the top 8 bits of the `mpu` (Minimum Packet Unit size).

Listing B.4: Patch: `sch_htb.c`

```
--- linux-2.4.27/net/sched/sch_htb.c Sun Aug 8 01:26:07 2004
+++ kernel/net/sched/sch_htb.c Mon Nov 8 18:36:08 2004
@@ -200,7 +200,8 @@
 static __inline__ long L2T(struct htb_class *cl,struct qdisc_rate_table *rate,
         int size)
 {
-    int slot = size >> rate->rate.cell_log;
+    int overhead = (rate->rate.mpu >> 8) & 0xFF;
+    int slot = (size-1+overhead) >> rate->rate.cell_log;
     if (slot > 255) {
         cl->xstats.giants++;
         slot = 255;
```

To give a context for the modifications, the *modified* function `L2T` from `sch_htb.c` is shown below.

Listing B.5: Function: `sch_htb.c:L2T`

```
200 static __inline__ long L2T(struct htb_class *cl,struct qdisc_rate_table *rate,
201         int size)
202 {
203     int overhead = (rate->rate.mpu >> 8) & 0xFF;
204     int slot = (size-1+overhead) >> rate->rate.cell_log;
205     if (slot > 255) {
206         cl->xstats.giants++;
207         slot = 255;
208     }
209     return rate->data[slot];
210 }
```

The table lookup part is the `rate->data[slot]`. The `rate` variable is a struct `tc_ratespec` which is described on the next page.

B.2.4 Kernel+iproute2 header: pkt_sched.h

It is probably an accident that the `overhead` parameter is transfered packed in the `mpu` value all the way to the kernel, because the overhead patch was only intended to modify the user space calculation of the rate table.

The real solution would be to change the `struct tc_ratespec` (which is part of the `qdisc_rate_table`). It is defined in `/usr/include/linux/pkt_sched.h`. But changing a header file, which are used by both the kernel and user space program can be problematic.

Listing B.6: Original: `pkt_sched.h`

```
79 struct tc_ratespec
80 {
81         unsigned char    cell_log ;
82         unsigned char    __reserved ;
83         unsigned short   feature;
84         short            addend;
85         unsigned short   mpu;
86         __u32            rate;
87 };
```

Listing B.7: Modified: `pkt_sched.h`

```
79 struct tc_ratespec
80 {
81         unsigned char    cell_log ;
82         unsigned char    __reserved ;
83         unsigned short   feature;
84         unsigned short   overhead;
85         unsigned short   mpu;
86         __u32            rate;
87 };
```

It is a very simple modification where the variable `addend` is replaced by an `overhead` variable. The variable `addend` is removed and replaced, to avoid changing the size of the struct. This is important because the struct is copied between user space and kernel. Changing the size of the struct would make the user space program iproute2/tc incompatible with older kernel versions. And the variable `addend` is not used any where.

B.2.5 Kernel: Overhead Patch, All Schedulers

These are the patches for all the schedulers, which uses the rate table. They are all based on a token bucket regulator. We have only tested the changes with the HTB scheduler (`sch_htb.c`). These patches assume that `pkt_sched.h` has been modified so that the struch `tc_ratespec` contain an `overhead` variable.

Listing B.8: Patch: `sch_htb.c`

```
--- linux-2.4.27/net/sched/sch_htb.c  Sun Aug 8 01:26:07 2004
+++ kernel/net/sched/sch_htb.c Mon Nov 8 19:15:06 2004
@@ -200,7 +200,8 @@
 static __inline__ long L2T(struct htb_class *cl,struct qdisc_rate_table *rate,
         int size)
 {
-    int slot = size >> rate->rate.cell_log;
+    int overhead = rate->rate.overhead;
+    int slot = (size+1+overhead) >> rate->rate.cell_log;
     if (slot > 255) {
         cl->xstats.giants++;
         slot = 255;
```

Listing B.9: Patch: `sch_cbq.c`

```
--- linux-2.4.27/net/sched/sch_cbq.c  Sun Aug 8 01:26:07 2004
+++ kernel/net/sched/sch_cbq.c Mon Nov 8 19:13:04 2004
@@ -190,7 +190,7 @@
 };


-#define L2T(cl,len)     ((cl)->R_tab->data[(len)>>(cl)->R_tab->rate.cell_log])
+#define L2T(cl,len)     ((cl)->R_tab->data[((len)+1+(cl)->R_tab->rate.overhead)>>(cl)->R_tab->rate.cell_log])


 static __inline__ unsigned cbq_hash(u32 h)
```

Listing B.10: Patch: `sch_tbf.c`

```
--- linux-2.4.27/net/sched/sch_tbf.c  Sun Aug 8 01:26:07 2004
+++ kernel/net/sched/sch_tbf.c Mon Nov 8 19:13:35 2004
@@ -132,8 +132,8 @@
         struct Qdisc    *qdisc;         /* Inner qdisc, default - bfifo queue */
 };

-#define L2T(q,L)  ((q)->R_tab->data[(L)>>(q)->R_tab->rate.cell_log])
-#define L2T_P(q,L) ((q)->P_tab->data[(L)>>(q)->P_tab->rate.cell_log])
+#define L2T(q,L)  ((q)->R_tab->data[((L)+1+(q)->R_tab->rate.overhead)>>(q)->R_tab->rate.cell_log])
+#define L2T_P(q,L) ((q)->P_tab->data[((L)+1+(q)->P_tab->rate.overhead)>>(q)->P_tab->rate.cell_log])

 static int tbf_enqueue(struct sk_buff *skb, struct Qdisc* sch)
 {
```

Listing B.11: Patch: `police.c`

```
--- linux-2.4.27/net/sched/police.c   Fri Dec 21 18:42:06 2001
+++ kernel/net/sched/police.c  Mon Nov 8 19:12:39 2004
@@ -31,8 +31,8 @@
 #include <net/sock.h>
 #include <net/pkt_sched.h>

-#define L2T(p,L)  ((p)->R_tab->data[(L)>>(p)->R_tab->rate.cell_log])
-#define L2T_P(p,L) ((p)->P_tab->data[(L)>>(p)->P_tab->rate.cell_log])
+#define L2T(p,L)  ((p)->R_tab->data[((L)+1+(p)->R_tab->rate.overhead)>>(p)->R_tab->rate.cell_log])
+#define L2T_P(p,L) ((p)->P_tab->data[((L)+1+(p)->P_tab->rate.overhead)>>(p)->P_tab->rate.cell_log])

 static u32 idx_gen;
 static struct tcf_police * tcf_police_ht [16];
```

B.3 Evaluation of Overhead Solution

B.3.1 Filter setup

Listing B.12: **Filter** *scheme* file: `tele2_evaluate_overhead_01.scheme`

```
#
# Tele2 - Scheme setup for
#         Evaluating the overhead patches
#
scp             0x30    upstream
ssh             0x10    upstream

# Tele2 special
# Latency test rule: icmp traffic
ping-tele2_01   0x10    upstream downstream
ping-tele2_02   0x30    upstream downstream
ping-tele2_03   0x50    upstream downstream

# Default latency rules
# Latency test rule: icmp traffic to ask
ping-ask        0x10    upstream downstream
ping-munin      0x20    upstream downstream
ping-hugin      0x30    upstream downstream
ping-www.diku.dk 0x666 upstream downstream
```

Listing B.13: **Filter** *rules* file: `ping_tele2_01.rules`

```
#
# Test rule, to measure the (ping) latency
#
# The one hop from the Tele2 ADSL
# atm0-1--0.val10-core.dk.tele2.com = 130.227.0.81
#
Append -p icmp -d 130.227.0.81
Append -p icmp -s 130.227.0.81
```

Listing B.14: **Filter** *rules* file: `ping_tele2_02.rules`

```
#
# Test rule, to measure the (ping) latency
#
# The closest hop to the Tele2 ADSL
# atm0-2--0.val10-core.dk.tele2.com = 130.227.255.137
Append -p icmp -d 130.227.255.137
Append -p icmp -s 130.227.255.137
```

Listing B.15: **Filter** *rules* file: `ping_tele2_03.rules`

```
#
# Test rule, to measure the (ping) latency
#
# The one hop from the Tele2 ADSL
# atm0-3--0.val10-core.dk.tele2.com = 130.227.0.69
#
Append -p icmp -d 130.227.0.69
Append -p icmp -s 130.227.0.69
```

Listing B.16: **Filter** *rules* file: `ping_tele2_04.rules`

```
#
# Test rule, to measure the (ping) latency
#
# li53.adsl.tele2.cust.dk.tele2.com == 130.227.255.138

#
Append -p icmp -d 130.227.255.138
Append -p icmp -s 130.227.255.138
```

B.3.2 HTB Script: Naive Overhead Solution

Scripts used for evaluating the naive rate reduction setup.

The output from the script, for the setup used i section 6.2.2 on page 61. Shows the rate left and the different overhead reductions.

```
./htb_03_nooverhead.sh -i eth1 -u 512 -x 13

Setup information:
------------------
Device                  : eth1
Link bandwidth          : 512 Kbit/s
Payload bandwidth       : 463 Kbit/s (subtracted fixed ATM overhead)
ATM fixed overhead      : 49 Kbit/s
Extra rate reserved     : 13 Kbit/s
Rate left for Classes   : 450 Kbit/s
```

HTB eval script: The naive solution

Listing B.17: **Queues** simple HTB queue setup: `htb_03_nooverhead.sh`

```
#!/bin/sh

# Hierarchical HTB class prio script
#  with FIXED overhead calculations
#  without overhead HTB patch usage
# ------------------------------
# Simple with FIFO queues
#
# Author: Jesper Dangaard Brouer <hawk@diku.dk>

 configfile =/usr/local/etc/ADSL-optimizer.conf
if [ ! -f $configfile ]; then
    echo "ERROR missing configuration file: \"${configfile}\""
    exit 1
fi
source ${ configfile }

# Parameters are parsed here...
source ${QUEUE_INC}/parameters.inc
# Functions loaded there ...
source ${QUEUE_INC}/functions.inc

# Fixed ATM Overhead calc...
# ~~~~~~~~~~~~~~~~~~~~~~~~~
# The ATM headers are subtracted the CEIL
# (will set the variable $CEIL)
fixed_ATM_overhead $LINE_CEIL

# Subtract EXTRA reserved bandwidth (migth be zero)
RATE=$[${CEIL}-${EXTRA}]

# Print setup information:
 print_info

# Also subtract EXTRA reserved bandwidth from CEIL
# (done after the print_info to get a nice output)
CEIL=$[${CEIL}-${EXTRA}]

# Update the event file ${EVENTDIR}/${QUEUE_EVENT}
event_startup

# Deletes previous classification (if any)
# ---------------------------------------
# Suppress output, because no classes/qdiscs result
# in an error message, which is expected ...
fun_tc qdisc del dev $DEV root > /dev/null 2>&1

# Create the root of tree
# -----------------------
# (default class: 1:50)
```

```
#
fun_tc qdisc add dev ${DEV} root handle 1: htb default 50 r2q 1
fun_tc class add dev ${DEV} parent 1: classid 1:1 htb rate ${CEIL}kbit ceil ${CEIL}kbit

#####
# FIFO buffer size
# ----------------
# 450 Kbit/s * 500 ms = 28125 bytes
# (450000/8) * 0.500 = 28125 bytes

##############################################
# Class: 1:10
# Mark : 10
# Description: Interactive traffic
##############################################
procent=20
ceil_procent=100
fun_tc class add dev ${DEV} parent 1:1 classid 1:10 htb \
    rate $[${procent}*${RATE}/100]kbit \
    ceil $[${ceil_procent}*${CEIL}/100]kbit prio 0

fun_tc qdisc add dev ${DEV} parent 1:10 handle 4210: \
    bfifo limit 28125

fun_tc filter add dev ${DEV} parent 1:0 protocol ip \
    prio 10 handle 0x10 fw classid 1:10


##############################################
# Class: 1:30
# Mark : 30
# Description: Good traffic
##############################################
procent=40
ceil_procent=100
fun_tc class add dev ${DEV} parent 1:1 classid 1:30 htb \
    rate $[${procent}*${RATE}/100]kbit \
    ceil $[${ceil_procent}*${CEIL}/100]kbit prio 4

fun_tc qdisc add dev ${DEV} parent 1:30 handle 4230: \
    bfifo limit 28125

fun_tc filter add dev ${DEV} parent 1:0 protocol ip \
    prio 30 handle 0x30 fw classid 1:30


##############################################
# Class: 1:50
# Mark : 50
# Description: Default fallthrough traffic
##############################################
procent=40
ceil_procent=100
fun_tc class add dev ${DEV} parent 1:1 classid 1:50 htb \
    rate $[${procent}*${RATE}/100]kbit \
    ceil $[${ceil_procent}*${CEIL}/100]kbit prio 5

fun_tc qdisc add dev ${DEV} parent 1:50 handle 4250: \
    bfifo limit 28125

fun_tc filter add dev ${DEV} parent 1:0 protocol ip \
    prio 50 handle 0x50 fw classid 1:50

###
# List
list
```

Pktgen: 100 pkt/s

Listing B.18: Pktgen script: Generates 100 pkt/s for 120 sec

```
#! /bin/sh

modprobe pktgen

PGDEV=/proc/net/pktgen/pg0
```

```
function pgset() {
    local result
    echo $1 > $PGDEV
    result=`cat $PGDEV | fgrep "Result: OK:"`
    if [ "$result" = "" ]; then
        cat $PGDEV | fgrep Result:
    fi
}

function pg() {
    echo inject > $PGDEV
    cat $PGDEV
}

# Packet size is incl. 14 bytes MAC header
# 40 + 14 = 54 (to emulate a 40 bytes TCP/IP packet)
#
pgset "pkt_size 54"

pgset "odev eth1"
#
# 83.73.0.1 = Our gateway
pgset "dst 83.73.0.1"
pgset "src_min 10.0.0.42"
pgset "dstmac 00:D0:B7:9E:F9:16"
pgset "srcmac 00:50:BA:C4:FF:01"

pgset "ipg 10000000" # 10ms = 1000ms/10ms = 100 pkt/s

# 100 pkt/s * 120 sec = 12000 packets
pgset "count 12000"

# Start
pg
```

B.3.3 HTB Script: Real Overhead Solution

The output from the script, for the setup used i section 6.2.3 on page 63:

```
./htb_overhead_kernel_queue_test.sh -i 512 -o 28

Setup information:
------------------
The ATM/AAL5 overhead calculations are done by tc and kernel
This shows what throughput can be expected

Device                  : eth1
Link bandwidth          : 512 Kbit/s
Max payload bandwidth   : 463 Kbit/s (subtracted fixed ATM overhead)
ATM fixed overhead      : 49 Kbit/s
Overhead per packet     : 28 bytes
```

HTB eval script: The real solution

Listing B.19: **Queues** HTB queue setup with overhead solution: `htb_overhead_kernel_queue_test3.sh`

```
#!/bin/sh

# ADSL-optimizer HTB script
# ------------------------------------
# Author: Jesper Dangaard Brouer <hawk@diku.dk>
#
# HTB class prio script with overhead setup for ADSL
# This script requires my kernel and iproute2 patch.
#
# Note:
# HTB needs to support the overhead parameter/option.

configfile =/usr/local/etc/ADSL-optimizer.conf
if [ ! -f $configfile ]; then
    echo "ERROR missing configuration file: \"${configfile}\""
    exit 1
fi
source ${ configfile }

# Use specific "tc" util
if [ -z "$tc" ]; then
   bin=/usr/local/ADSL-optimizer/bin
   tc=${bin}/tc.atm_overhead_kernel
fi

source ${QUEUE_INC}/parameters.inc
source ${QUEUE_INC}/functions.inc

CEIL=$LINE_CEIL
RATE=$LINE_CEIL

burst=2000b
#burst=1696b
#burst=1802b
cburst=2000b
#cburst=1696b
#cburst=1802b

# ======================================================
# Packet overhead per TCP/IP packet due to ATM/AAL5
# ======================================================
#
# Overhead per AAL5 packet
# AAL5 tail    : 8 bytes per packet (incl. 4 bytes checksum)
#
aal5_tail =8

# AAL5 SSCS headers (SSCS = Service Specific Common Sublayer)
# -------------------
#
# VC (Virtual Circuit) vs. LLC (Logical Link Control)
```

```
# There is a basic choice between LLC and VC when configuring
# the encapsulation mode on ADSL. LLC gives more overhead.

# Routed modes:
overhead_PPPoA_VC=2
overhead_PPPoA_LLC=6
#
overhead_rfc2684R_VC=0
overhead_rfc2684R_LLC=8

# Bridged modes:
# There is a special feature bridged mode which some equipment
# support where the MAC-checksum (FCS) can be dropped, which
# reduces the overhead by 4 bytes.
#
overhead_rfc2684B_VC=20
overhead_rfc2684B_VC_noFCS=16
overhead_rfc2684B_LCC=28
overhead_rfc2684B_LCC_noFCS=24
#
overhead_PPPoE_VC=28
overhead_PPPoE_VC_noFCS=24
overhead_PPPoE_LLC=36
overhead_PPPoE_LLC_noFCS=32

# ************************************************
# *** Choose the SSCS overhead encapsulation mode ***
# ************************************************
#overhead_SSCS=${overhead_PPPoE_VC}
#overhead_SSCS=${overhead_PPPoE_LLC}
#overhead_SSCS=${overhead_PPPoA_VC}
overhead_SSCS=8

# Encapsulation overhead:
# ----------------------
if [ -z "$overhead" ]; then
    overhead=$[ ${overhead_SSCS} + ${aal5_tail} ]
fi

# ATM cell padding/aligning
# -------------------------
# The ATM padding/aligning cost is accounted for by
# modifying tc, thus applying my tc (tc_core.c) patch ;-)


# Print setup information:
#
print_info2

# Update the event file ${EVENTDIR}/${QUEUE_EVENT}
event_startup


# Deletes previous classification (if any)
# ---------------------------------------
# Suppress output, because no classes/qdiscs result
# in an error message, which is expected ...
fun_tc qdisc del dev $DEV root > /dev/null 2>&1


# Create the root of tree
# -----------------------
# (default class: 1:50)
#
fun_tc qdisc add dev ${DEV} root       handle 1: htb default 50 r2q 1

fun_tc class add dev ${DEV} parent 1: classid 1:1 htb \
    burst $burst cburst $cburst \
    rate ${CEIL}kbit ceil ${CEIL}kbit \
    overhead $overhead


##############################################
# Class: 1:10
# Mark : 10
# Description: Interactive traffic
##############################################
procent=20
ceil_procent=100
fun_tc class add dev ${DEV} parent 1:1 classid 1:10 htb \
```

```
    rate $[${procent}*${RATE}/100]kbit \
    ceil $[${ceil_procent}*${CEIL}/100]kbit \
    burst $burst cburst $cburst \
    prio 0 \
    overhead $overhead

fun_tc qdisc add dev ${DEV} parent 1:10 handle 4210: \
    bfifo limit 28125

fun_tc filter add dev ${DEV} parent 1:0 protocol ip \
    prio 10 handle 0x10 fw classid 1:10


############################################
# Class: 1:30
# Mark : 30
# Description: Good traffic
############################################
procent=40
ceil_procent=100
fun_tc class add dev ${DEV} parent 1:1 classid 1:30 htb \
    rate $[${procent}*${RATE}/100]kbit \
    ceil $[${ceil_procent}*${CEIL}/100]kbit \
    prio 4\
    overhead $overhead \
    burst $burst cburst $cburst \
    #burst 10000 cburst 500 \

fun_tc qdisc add dev ${DEV} parent 1:30 handle 4230: \
    bfifo limit 28125

fun_tc filter add dev ${DEV} parent 1:0 protocol ip \
    prio 30 handle 0x30 fw classid 1:30



############################################
# Class: 1:50
# Mark : 50
# Description: Default fallthrough traffic
############################################
procent=40
ceil_procent=100
fun_tc class add dev ${DEV} parent 1:1 classid 1:50 htb \
    rate $[${procent}*${RATE}/100]kbit \
    ceil $[${ceil_procent}*${CEIL}/100]kbit \
    burst $burst cburst $cburst \
    prio 5 \
    overhead $overhead

fun_tc qdisc add dev ${DEV} parent 1:50 handle 4250: \
    bfifo limit 28125
    #pfifo limit 500

fun_tc filter add dev ${DEV} parent 1:0 protocol ip \
    prio 50 handle 0x50 fw classid 1:50


###
# List
list
```

B.4 Evaluation of ACK-prioritizing

B.4.1 Filter setup: ACK-prioritizing

Listing B.20: **Filter** *scheme* file: `tele2_evaluate_ack_prio_03.scheme`

```
#
# Tele2 - Scheme setup for
#         Evaluating the overhead patches
#
scp               0x30    upstream downstream
scp_extranet      0x30    upstream downstream
ssh               0x10    upstream downstream

# Prio ACK packet high, this ensures high downstream traffic
#ack              0x20    upstream downstream
ack_extra         0x20    upstream downstream

# TCP handshake packets
tcp_handshake     0x10    upstream downstream

# Tele2 special
# Latency test rule: icmp traffic
ping-tele2_01     0x10    upstream downstream
ping-tele2_02     0x30    upstream downstream
ping-tele2_03     0x50    upstream downstream
ping-tele2_04     0x20    upstream downstream

# Default latency rules
# Latency test rule: icmp traffic to ask
ping-ask          0x10    upstream downstream
ping-munin        0x20    upstream downstream
ping-hugin        0x30    upstream downstream
ping-www.diku.dk 0x666 upstream downstream
```

B.4.2 HTB Script: ACK-prioritizing

Listing B.21: **Queues** HTB queue setup with overhead solution and ACK prioritizing: `htb_overhead_ack_test1.sh`

```
#!/bin/sh

# ADSL-optimizer HTB script
# ------------------------------------
# Author: Jesper Dangaard Brouer <hawk@diku.dk>
#
# HTB class prio script with overhead setup for ADSL
# This script requires my kernel and iproute2 patch.
#
# Including the ACK queue
#
# Note:
# HTB needs to support the overhead parameter/option.

 configfile =/usr/local/etc/ADSL-optimizer.conf
if [ ! -f $configfile ]; then
    echo "ERROR missing configuration file: \"${configfile}\""
    exit 1
fi
source ${ configfile }

# Use specific "tc" util
if [ -z "$tc" ]; then
   bin=/usr/local/ADSL-optimizer/bin
   tc=${bin}/tc.atm_overhead_kernel
fi

source ${QUEUE_INC}/parameters.inc
source ${QUEUE_INC}/functions.inc


# Calculate the ACK_rate from the $LINE_DOWNSTREAM
#
```

```
if [ -z "$ACK_rate" -a -n "$LINE_DOWNSTREAM" ]; then
    # Will set the variable $ACK_rate
    ACK_rate_calc $LINE_DOWNSTREAM
    #echo "ACK rate calculated to: $ACK_rate "
    #echo "(for a $LINE_DOWNSTREAM Kbit downstream link)"
fi

# Set ACK_rate to zero if not set
if [ -z "$ACK_rate" ]; then
    ACK_rate=0
fi

ACK_ceil=$LINE_CEIL

# Rate bandwidth left for Classes
#
RATE=$[${LINE_CEIL}-${ACK_rate}]

#RATE=$LINE_CEIL
CEIL=$LINE_CEIL


burst=2000b
#burst=1696b
#burst=1802b
cburst=2000b
#cburst=1696b
#cburst=1802b

# ======================================================
# Packet overhead per TCP/IP packet due to ATM/AAL5
# ======================================================
#
# Overhead per AAL5 packet
# AAL5 tail    : 8 bytes per packet (incl. 4 bytes checksum)
#
aal5_tail=8

# AAL5 SSCS headers (SSCS = Service Specific Common Sublayer)
# ------------------
#
# VC (Virtual Circuit) vs. LLC (Logical Link Control)
# There is a basic choice between LLC and VC when configuring
# the encapsulation mode on ADSL. LLC gives more overhead.

# Routed modes:
overhead_PPPoA_VC=2
overhead_PPPoA_LLC=6
#
overhead_rfc2684R_VC=0
overhead_rfc2684R_LLC=8

# Bridged modes:
# There is a special feature bridged mode which some equipment
# support where the MAC-checksum (FCS) can be dropped, which
# reduces the overhead by 4 bytes.
#
overhead_rfc2684B_VC=20
overhead_rfc2684B_VC_noFCS=16
overhead_rfc2684B_LCC=28
overhead_rfc2684B_LCC_noFCS=24
#
overhead_PPPoE_VC=28
overhead_PPPoE_VC_noFCS=24
overhead_PPPoE_LLC=36
overhead_PPPoE_LLC_noFCS=32

# ************************************************
# *** Choose the SSCS overhead encapsulation mode ***
# ************************************************
#overhead_SSCS=${overhead_PPPoE_VC}
#overhead_SSCS=${overhead_PPPoE_LLC}
#overhead_SSCS=${overhead_PPPoA_VC}
overhead_SSCS=8

# Encapsulation overhead:
# ----------------------
if [ -z "$overhead" ]; then
    overhead=$[ ${overhead_SSCS} + ${aal5_tail} ]
fi
```

```
# ATM cell padding/aligning
# ------------------------
# The ATM padding/aligning cost is accounted for by
# modifying tc, thus applying my tc (tc_core.c) patch ;-)


# Print setup information:
#
print_info2

# Update the event file ${EVENTDIR}/${QUEUE_EVENT}
event_startup


# Deletes previous classification (if any)
# ---------------------------------------
# Suppress output, because no classes/qdiscs result
# in an error message, which is expected ...
fun_tc qdisc del dev $DEV root > /dev/null 2>&1


# Create the root of tree
# -----------------------
# (default class: 1:50)
#
fun_tc qdisc add dev ${DEV} root       handle 1: htb default 50 r2q 1

fun_tc class add dev ${DEV} parent 1: classid 1:1 htb \
    burst $burst cburst $cburst \
    rate ${CEIL}kbit ceil ${CEIL}kbit \
    overhead $overhead


##############################################
# Class: 1:10
# Mark : 10
# Description: Interactive traffic
##############################################
procent=20
ceil_procent=100
fun_tc class add dev ${DEV} parent 1:1 classid 1:10 htb \
    rate $[${procent}*${RATE}/100]kbit \
    ceil $[${ceil_procent}*${CEIL}/100]kbit \
    burst $burst cburst $cburst \
    prio 0 \
    overhead $overhead

fun_tc qdisc add dev ${DEV} parent 1:10 handle 4210: \
    bfifo limit 28125

fun_tc filter add dev ${DEV} parent 1:0 protocol ip \
    prio 10 handle 0x10 fw classid 1:10


##############################################
# Class: 1:20
# Mark : 20
# Description: ACK "class"
##############################################
fun_tc class add dev ${DEV} parent 1:1 classid 1:20 htb \
    rate ${ACK_rate}kbit \
    ceil ${ACK_ceil}kbit \
    prio 1 \
    burst $burst cburst $cburst \
    overhead $overhead

fun_tc qdisc add dev ${DEV} parent 1:20 handle 4220: \
    bfifo limit 28125

fun_tc filter add dev ${DEV} parent 1:0 protocol ip \
    prio 20 handle 0x20 fw classid 1:20

##############################################
# Class: 1:30
# Mark : 30
# Description: Good traffic
##############################################
procent=40
ceil_procent=100
```

```
fun_tc  class  add dev ${DEV} parent 1:1 classid 1:30 htb \
     rate  $[${procent}*${RATE}/100]kbit \
     ceil  $[${ceil_procent}*${CEIL}/100]kbit \
     prio  4\
     overhead $overhead \
     burst $burst cburst $cburst \
     #burst 10000 cburst 500 \

fun_tc  qdisc  add dev ${DEV} parent 1:30 handle 4230: \
     bfifo  limit  28125

fun_tc   filter   add dev ${DEV} parent 1:0 protocol ip \
     prio  30 handle 0x30 fw classid  1:30



##############################################
# Class:   1:50
# Mark : 50
# Description: Default  fallthrough   traffic
##############################################
procent=40
ceil_procent=100
fun_tc  class  add dev ${DEV} parent 1:1 classid 1:50 htb \
     rate  $[${procent}*${RATE}/100]kbit \
     ceil  $[${ceil_procent}*${CEIL}/100]kbit \
     burst $burst cburst $cburst \
     prio  5 \
     overhead $overhead

fun_tc  qdisc  add dev ${DEV} parent 1:50 handle 4250: \
     bfifo  limit  28125
     #pfifo  limit  500

fun_tc   filter   add dev ${DEV} parent 1:0 protocol ip \
     prio  50 handle 0x50 fw classid  1:50


###
# List
list
```

B.4.3 Ingress filtering

Listing B.22: **Queues** Limiting the downstream capacity using ingress policing: `downstream_limit.sh`

```
#!/bin/sh

# Simple Ingress  policer
# (ingress  =  incomming traffic)
# ----------------------------
# Author: Jesper Dangaard Brouer <hawk@diku.dk>, June 2004
#

configfile =/usr/local/etc/ADSL-optimizer.conf
if  [ ! -f  $configfile  ]; then
    echo "ERROR missing configuration file: \"${configfile}\""
    exit  1
fi
source ${ configfile }

echo
echo "Use: -u for the line capacity (even if this is the downsteam)"
source ${QUEUE_INC}/parameters.inc
source ${QUEUE_INC}/functions.inc


# Fixed ATM Overhead calc...
# ~~~~~~~~~~~~~~~~~~~~~~~~~~
# The ATM headers are subtracted the CEIL
# (will  set  the  variable  $CEIL)
fixed_ATM_overhead $LINE_CEIL

# Print setup information:
#
```

```
print_info

# Update the event file ${EVENTDIR}/${QUEUE_EVENT}
event_startup

# Remove any existing ingress with this handle
#  also deletes the filters
fun_tc qdisc del dev $DEV handle ffff: ingress
# > /dev/null 2>&1


# Attach the ingress policer to the device
fun_tc qdisc add dev $DEV handle ffff: ingress

#
# Filter everything to the ingress filter
fun_tc filter add dev ${DEV} parent ffff: protocol ip prio 42 \
    u32 match ip src 0.0.0.0/0 \
    police rate ${CEIL}kbit burst 10k mtu 1500 drop flowid :1
#    police rate ${CEIL}kbit burst 20k drop flowid :1
#    police rate ${CEIL}kbit burst 10k drop flowid :1

#
# Listing the filter
#
LIST=yes
if [ -n "$LIST" ]; then
    echo
    echo "FILTERS:"
    fun_tc -s filter ls dev $DEV parent ffff:
fi
```

B.5 ADSL-optimizer

B.5.1 Install and Config

Listing B.23: Install.txt: Installation instructions

```
~~ $Id: INSTALL.txt,v 1.5 2004/12/22 16:07:43 hawk Exp $

                -----------------------------------------
                     Installation of the ADSL-optimizer
                -----------------------------------------
                  Jesper Dangaard Brouer (hawk@diku.dk)
                -----------------------------------------
                        $Date: 2004/12/22 16:07:43 $
                              $Revision: 1.5 $
                -----------------------------------------

Introduction:
~~~~~~~~~~~~~
 The ADSL-optimizer is targeted towards optimizing ADSL lines,
 including calculating/accounting for the ATM overhead associated with
 ADSL lines.

 This file only describes the basic Install recipe.

 For configuration see the file: {{design/config.apt}}

Requires:
~~~~~~~~~
 The software package have the following requirements. The module
 which the requirement applies for is specified in brackets.

 * The "tc" program need to be patched with the overhead patch (queues)
 * The kernel need to be patched with the overhead patch (queues)
 * The "iptables" software package ( filter )
 * The "RRDtool" software package including perl RRDs module (graph)

 For graph viewing, some RRDtool frontend is needed. Our graph module
 only collects "tc" data into RRDtool files.

 We highly recomment "drraw" and supply some example files for "drraw"
 (in directory "saved") (which is hardcoded to use files from
 '/var/spool/rrdqueues'). The example files also uses information
 about the physical interface, which is collected with the program
 package "rrdcollect" ( files stored in '/var/spool/rrdcollect').

 The latency graph depends on "smokeping", but you will need to adapt
 that to your specific setup.


Install instructions step-by-step:
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 1.)
 Unpack the ADSL-optimizer in a directory, preferable:
   "/usr/local/ADSL-optimizer"
 (we will refer to this directory as "$BASEDIR")

 2.)
 Copy the $BASEDIR/sample.conf file to /usr/local/etc/ADSL-optimizer.conf.
 (Modify the first line of ADSL-optimizer.conf if not installed in the
  default location "/usr/local/ADSL-optimizer")

 3.)
 Copy $BASEDIR/filter/expansions.conf.sample to expansions.conf in the
 same directory. Modify "expansions.conf" for your setup, the files
 eg. contains the definition of your "LOCALNET" IP-range.

 Read the {{design/config.apt}} for modifying the filter setup.

* Init scripts
~~~~~~~~~~~~~~
 The init-scripts are located in $BASEDIR/init.d/.

 4.) (/etc/init.d/ADSL-filter)
 Modify the init-script: "ADSL-filter.init". It specifies which filter
 <base-script> and <scheme-script> to use.
 ( See under: $BASEDIR/filter/base/ and scheme/ )
```

```
5.) (/etc/init.d/ADSL-queues)
Modify the init-script: "ADSL-queues.init".
It specifies the following parameters:

  * INTERFACE, which is the upstream interface.
  * UPSTREAM, what is the upstream link speed.
  * DOWNSTREAM (optional), only used for ACK rate calc.
  * ACK_RATE (optional), not used if DOWNSTREAM specified.
  * SCRIPT, which queueing script to use.

6.)
Install the init-scripts by hand or use the "install.sh" script in
the $BASEDIR/init.d directory.

* Other
~~~~~~~
7.)
Change the device variabel in the script "tc-collector.pl"
(in $BASEDIR/graph).

Detecting the correct devices is on the TO-DO list...
```

Listing B.24: Config.apt: Configuration instructions

```
~~ -*-text-*-

                ------------------------------------------
                          ADSL optimizer
                        Configuration setup
                ------------------------------------------
                  Jesper Dangaard Brouer (hawk@diku.dk)
                ------------------------------------------
                          $Revision: 1.5 $
                      $Date: 2004/12/22 16:20:20 $
                ------------------------------------------


Intro
~~~~~

 The ADSL-optimizer consists of three elements:

  * Queues - Setting up the priority classes.

  * Filter - Classification and "marking" of the traffic.

  * Graph - Gathering of "tc" queue statistics into RRD data files.

  []


 The active part of the ADSL-optimizer is "Queues" and "Filter". The
 "Graph" element is not important for the "optimizing" function and
 can be views as a seperate element.

 "Queues" and "Filter" are both shell scripts and share a common
 configuration file :

      <<< /usr/local/etc/ADSL-optimizer.conf >>>

 You will need to create this configuration fil.
 You can use the sample.conf file.


Filter and queues common
~~~~~~~~~~~~~~~~~~~~~~~~

 Information in the "Filter" and "Queues" common config file:

      <<< /usr/local/etc/ADSL-optimizer.conf >>>

 You will need to modify the first line of this configuration file to
 specify the install location. The default is to install in
 /usr/local/ADSL-optimizer/.


------
 BASEDIR=/usr/local/ADSL-optimizer/
 EVENTDIR=/var/spool/rrdqueues
```

```
QUEUEDIR=${BASEDIR}/queues
QUEUE_INC=${QUEUEDIR}/include/
QUEUE_EVENT=changes_queue.evt

FILTERDIR=${BASEDIR}/filter
FILTER_INC=${FILTERDIR}/include/
FILTER_BASE=${FILTERDIR}/base/
FILTER_SCHEME=${FILTERDIR}/scheme/
FILTER_RULES=${FILTERDIR}/rules/
FILTER_EVENT=changes_filter.evt

GRAPHDIR=${BASEDIR}/graph
------

The basic information needed is the shell script include files.

The "Filter" also need to know the different configurations file
directories

The "GRAPHDIR" variabel is used by the "Graph" elements init-script,
to locate the data collector and set the perl (PERL5LIB) include dir.


Filter (shell-script)
~~~~~~~~~~~~~~~~~~~~~

The filters are where you will need to make the most modifications.
You will need to modify the filters to fit the needs of your own
network.

Manual setup (see init-script for "auto" setup):
~~~~~~~~~~~~~~
First load the base setup: <<< "load_base.sh -f filename" >>>

Second load the scheme setup: <<< "load_scheme.sh -f filename" >>>


Queues (shell-script)
~~~~~~~~~~~~~~~~~~~~~

The scripts have functions to calculate the overhead and estimate the
required rate for ACK packets.

The ACK rate can either be specified
  1) directly with "-a" or
  2) be calculated based on the downstream bandwidth with "-d".

Interface "-i" and Link (upstream) capacity "-u" are required
parameters.

------
ADSL-optimizer - Class Queueing

Usage:
------
Script     : ./htb_01_overhead.sh
Parameters: [-v] -i interface -u SPEED-CEIL
              [-a ACK-reserved-rate] [-d Downstream_calc_ACK_rate]

  -v : verbose
  -i : Interface/device
  -u : Link capacity (upstream)
  -a : Rate reserved for ACK packets (can be calculated automatic)
  -d : ACK rate calculation by Downstream capacity
------


graph (perl-script)
~~~~~~~~~~~~~~~~~~~

The tc-graph tool is fairly easy to use, as it basically tries to
detect everything and creates the RRDtool files if they do not exist.

init scripts
~~~~~~~~~~~~
Some small changes are necessary for the init scripts, see INSTALL.txt.

"Names":
  ADSL-filter
```

B.6 ADSL-optimizer: Queues

B.6.1 Queues: Common Functions and Parameters

Listing B.25: Queues module: Common Functions

```
#!/bin/bash
#
# Author: Jesper Dangaard Brouer <hawk@diku.dk>


# Determining the 'tc' command location, if not specified
# and verify the location if the $tc variable is set.
#
if [ -z "$tc" ]; then
    which tc > /dev/null 2>&1
    if [ $? -eq 0 ]; then
        tc=tc
    elif [ -x /sbin/tc ]; then
        tc=/sbin/tc
    else
        echo ""
        echo "ERROR: Linux traffic control command 'tc' not found!"
        exit 1
    fi
else
    # Verify $tc exists and is executable
    if [ ! -x $tc ]; then
        echo ""
        echo "ERROR: Linux traffic control command '$tc' not found/executable!"
        exit 1
    fi
fi


function fun_tc() {
    $tc $@
    result=$?
    if [ $result -gt 0 ]; then
        echo "WARNING -- Error ($result) when executing the tc command:"
        echo " \"tc $@\""
    else
        if [ -n "$VERBOSE" ]; then
            echo "tc $@"
        fi
    fi
}

function list () {
    if [ -n "$LIST" ]; then
        echo "CLASSES:"
        fun_tc -d class ls dev $DEV
        echo
        echo "MARK FILTERS:"
        #fun_tc -d filter ls dev $DEV | grep handle
        fun_tc -d filter ls dev $DEV parent 1: | grep handle
        fun_tc -d filter ls dev $DEV parent 100: | grep handle
        echo
        echo "QDISCs:"
        fun_tc -d qdisc ls dev $DEV
        echo
    fi
}



#
# ACK_rate
# ---------
# To utilize the DOWNSTREAM bandwidth, it is very important to reserve
# enough bandwidth to ACK packets.
```

```
#
# The latency of the ACK packets also influences the downstream
# utilization (due to the bandwidth-delay product and default window
# size). The assure low latency using HTB, the ACK class must never
# be backlogged. This is achived by reserving a big enough RATE to
# the class and assigning a "high" priority.
#
# The ACK_rate can be (calculated or) estimated based on the
# downstream capacity.
#
# downstream_capacity / data_packet_size = num_data_packets
#
# num_data_packets / ACK_packets_per_data_packet = Num_ACK_packets
#
# num_ACK_packets * ACK_packet_size = Required_ACK_rate
#
#
function ACK_rate_calc() {
    if [ -z "$1" ]; then
        echo "[$FUNCNAME] ERROR need to know downstream capacity (in Kbit/s)."
        exit 1;
    fi
    local DOWNSTREAM_=$1

    # Delayed ACK factor, ACK_factor:
    # ------------------------------
    # The minimum delayed ACK factor is 1, since at most one ACK should be
    # sent for each data packet [rfc1122, rfc2581]. "Normal" behavior of
    # TCP bulk transfers is to ACK every second data packet.
    # (notice: ACK_scale=10 => 10=1 16=1.6)
    #ACK_factor=12
    ACK_factor=15
    ACK_scale=10

    # ACK_size
    # --------
    # The ACK size on ADSL with ATM link-layer is two ATM cells (2*53)
    #
    # With out cell headers, only payload 2*48
    #ACK_size=96
    # Complete ATM cells with headers 2*53
    ACK_size=106

    # Data packet size
    # ----------------
    # The (average) data packet size.
    #
    Data_size=1500

    # Precision factor, doing integer part divisions we lose the
    # fractional part of the number. To avoid this we "move" the decimal
    # point, when doing fixed point arithmetic
    scale=100000

    # Round off loss
    round_off_bonus=1

    # Old calc:
    # ACK_rate=$[ ($DOWNSTREAM_*$scale)/$Data_size/$ACK_factor * $ACK_size \
    #           * $ACK_scale / $scale ]

    # New calc: (multiply first, then divide, avoiding div loss)
    ACK_rate=$[ $DOWNSTREAM_ * $ACK_size * $ACK_scale \
                / $Data_size / $ACK_factor ]

    if [ $ACK_rate -lt 0 ]; then
        echo "ERROR negative ACK rate \"$ACK_rate\""
        exit 2
    fi

    if [ -n "${CEIL}" ]; then
        local RATE_=$[${CEIL}-${ACK_rate}]
        if [ ${RATE_} -lt 0 ]; then
            echo "ERROR ACK rate ($ACK_rate) greater that ceil ($CEIL)"
            echo "      Resulting in negative rate ($RATE_) to classes!"
            exit 2
        fi
    fi

    ACK_rate=$[ $ACK_rate + $round_off_bonus ]
```

```
}

# Fixed ATM Overhead calc...
# ~~~~~~~~~~~~~~~~~~~~~~~~~
#  The ATM frame header overhead can be calculated static
#  and be subtracted from the line bandwidth/ceil ...
#
#  (bandwidth/53/8) = Number of ATM frames
#  (bandwidth/53)*5 = ATM cell header overhead
#  (bandwidth/53)*48 = Payload bandwidth

# ATM also uses control messages/frames OAM one out of every 27 frame.
#  (bandwidth/53)/27 * 48 = OAM bandwidth overhead
#
# fx. bandwidth = 512000
#  (512000/53/8)       =   1207.547169811321 Number of ATM frames
#  (512000/53)*5       =  48301.88679245283 ATM cell header overhead
#  (512000/53)*48      = 463698.1132075472 Payload bandwidth
#  (512000/53)/27 * 48 = 17174.00419287212 OAM bandwidth overhead
#
#  Payload bandwidth = 446524.1090146751 bits/s
#                      ========================
#
# ((bandwidth/53)*48) - ((bandwidth/53)/27 * 48) = payload_bandwidth
#  (bandwidth*48)/53 -(bandwidth*48)/53*27 = payload_bandwidth
#  (bandwidth*48*27)/53*27 -(bandwidth*48)/53*27 = payload_bandwidth
#  (bandwidth*48*27) -(bandwidth*48) =53*27* payload_bandwidth
#  (bandwidth*48*26) =53*27* payload_bandwidth
#  bandwidth =((53*27)/(48*26))* payload_bandwidth
#  bandwidth = 1.14663 * payload_bandwidth
#
# Measurements of the real line, with at scp upload _only_
#  Highest value: 426492 Kbit/s
#  (assuming) measurement includes TCP/IP headers
#
# Adding padding for a 1500 (see later)
# (426492 / 1500 / 8)      = 35.541 packets per sec
# 20 bytes padding for a 1500 bytes packet (see later)
# (426492 / 1500) * 20 = 5686.559999999999 bits/s extra
#
# 426492 + 5686.559999999999 = 432178 line Payload bandwidth
#
# ((53*27)/(48*26))* payload_bandwidth = bandwidth
# ((53*27)/(48*26))* 432178            = 495550 ADSL bandwidth
#
# Which corresponds to approx: 500000 Kbit/s
#
# (500000/53)*48     = 452830.1886792453
# (500000/53)/27 * 48 = 16771.48846960168
# ((500000/53)*48)-((500000/53)/27 * 48)
#        436058.7002096436

function fixed_ATM_overhead() {
    if [ -z "$1" ]; then
        echo "[$FUNCNAME] ERROR need to know line capacity (in Kbit/s)."
        exit 1;
    fi
    line_capacity=$1

    #if [ -n "$CEIL" ]; then
    #   echo ""
    #   echo "[WARNING] The CEIL variable is already set (to: \"$CEIL\")"
    #   echo "          This function will overwrite the CEIL variable"
    #   echo ""
    #fi

    # Precision factor, doing integer part divisions we lose the
    # fractional part of the number. To avoid this we "move" the decimal
    # point, when doing fixed point arithmetic
    scale=100000

    # (bandwidth/53)*48 = Payload bandwidth
    payload_bandwidth=$[ ($line_capacity*$scale)/53*48 / $scale]
    #echo "payload_bandwidth: $payload_bandwidth"

    # ATM also uses control messages/frames OAM one out of every 27 frame.
    #  (bandwidth/53)/27 * 48 = OAM bandwidth overhead
#    OAM_overhead=$[ ($line_capacity*$scale) /53*48 / 27 / $scale]
    OAM_overhead=0
```

```
    # The availably payload bandwidth
    # after subtracting the Fixed ATM overhead
    CEIL=$[ $payload_bandwidth - $OAM_overhead ]
}

function print_info () {

    echo ""
    echo "Setup information:"
    echo "------------------"

        echo "Device                : $DEV"
        echo "Link bandwidth        : $LINE_CEIL Kbit/s"

    if [ -n "$CEIL" ]; then
        echo "Payload bandwidth     : $CEIL Kbit/s (subtracted fixed ATM overhead)"
        fixed_overhead=$[ $LINE_CEIL - $CEIL ]
        echo "ATM fixed overhead    : $fixed_overhead Kbit/s"
    fi

    if [ -n "$LINE_DOWNSTREAM" ]; then
        echo "Downstream bandwidth  : $LINE_DOWNSTREAM Kbit/s (only calc ACK rate)"
    fi

    if [ -n "$ACK_rate" ]; then
        echo "Reserved for ACK packets: $ACK_rate Kbit/s"
        echo "Rate left for Classes : $RATE Kbit/s"
    fi

    if [ -n "$EXTRA" ]; then
        echo "Extra rate reserved   : $EXTRA Kbit/s"
        echo "Rate left for Classes : $RATE Kbit/s"
    fi
    #exit 42
}

function print_info2 () {

    echo ""
    echo "Setup information:"
    echo "------------------"
    echo "The ATM/AAL5 overhead calculations are done by tc and kernel"
    echo "This shows what throughput can be expected"
    echo ""

    ceil_push=$CEIL
    fixed_ATM_overhead $CEIL
    ceil_minus_atm=${CEIL}
    CEIL=${ceil_push}

        echo "Device                : $DEV"
        echo "Link bandwidth        : $LINE_CEIL Kbit/s"

    if [ -n "$ceil_minus_atm" ]; then
        echo "Max payload bandwidth : $ceil_minus_atm Kbit/s (subtracted fixed ATM overhead)"
        fixed_overhead=$[ $LINE_CEIL - $ceil_minus_atm ]
        echo "ATM fixed overhead    : $fixed_overhead Kbit/s"
    fi

    if [ -n "$overhead" ]; then
        echo "Overhead per packet   : $overhead bytes"
    fi

    if [ -n "$LINE_DOWNSTREAM" ]; then
        echo "Downstream bandwidth  : $LINE_DOWNSTREAM Kbit/s (only calc ACK rate)"
    fi

    if [ -n "$ACK_rate" ]; then
        echo "Reserved for ACK packets: $ACK_rate Kbit/s"
        echo "Rate left for Classes : $RATE Kbit/s"
    fi

    if [ "$EXTRA" -ne 0 ]; then
        echo "Extra rate reserved   : $EXTRA Kbit/s"
        echo "Rate left for Classes : $RATE Kbit/s"
    fi
    #exit 42
}

function event_startup() {
```

```
    # Getting the UNIX timestamp
    local time=`perl -e 'print time;'`
    local base=${0##*/}
    local event="$time [$base] ${DEV} Ceil:${CEIL}Kbit"
    if [ -n "$ACK_rate" ]; then
        event="$event ACK:${ACK_rate}Kbit Classes:${RATE}Kbit"
    fi
    event_file ="${EVENTDIR}/${QUEUE_EVENT}"
    if [ ! -f $event_file -a -w ${EVENTDIR} ]; then
        touch $event_file
    fi
    if [ -w $event_file ]; then
        echo $event >> ${EVENTDIR}/${QUEUE_EVENT}
    else
        echo "WARNING: can not write to event file \"$event_file\""
    fi
}

function event_stop() {
    # Getting the UNIX timestamp
    local time=`perl -e 'print time;'`
    local base=${0##*/}
    local event="$time [$base] stop script"
    event_file ="${EVENTDIR}/${QUEUE_EVENT}"
    if [ ! -f $event_file -a -w ${EVENTDIR} ]; then
        touch $event_file
    fi
    if [ -w $event_file ]; then
        echo $event >> ${EVENTDIR}/${QUEUE_EVENT}
    else
        echo "WARNING: can not write to event file \"$event_file\""
    fi
}
```

Listing B.26: Queues module: Common Parameter Parsing

```
#!/bin/bash
#
# Author: Jesper Dangaard Brouer <hawk@diku.dk>

function usage() {
    echo ""
    echo "ADSL-optimizer - Class Queueing"
    echo ""
    echo "Usage:"
    echo "------"
    echo " Script   : $0"
    echo " Parameters: [-v] -i interface -u SPEED-CEIL"
    echo "             [-a ACK-reserved-rate] [-d Downstream_calc_ACK_rate]"
    echo ""
    echo "  -v : verbose"
    echo "  -i : Interface/device"
    echo "  -u : Link capacity (upstream)"
    echo "  -a : Rate reserved for ACK packets (can be calculated automatic)"
    echo "  -d : ACK rate calculation, by Downstream capacity"
    echo "  -x : Subtract eXtra fixed overhead (after ATM fixed overhead)"
    echo ""
}

if [ -z "$1" ]; then
    usage
    exit 1
fi

## --- Parse command line arguments ---
while getopts ":i:u:d:l:a:x:o:v" option; do
    case $option in
        i)
            #echo " Interface/Device: \"$OPTARG\""
            DEV=$OPTARG
            ;;
        u)
            #echo " (Upstream) Line speed: \"$OPTARG\" kbit"
            LINE_CEIL=$OPTARG
            ;;
        a)
            #echo " Rate reserved to ACKs: \"$OPTARG\" kbit"
            ACK_rate=$OPTARG
            ;;
```

```
        d)
            #echo " Downstream speed (ONLY used to calc ACK_rate): \"$OPTARG\" kbit"
            LINE_DOWNSTREAM=$OPTARG
            ;;
        x)
            EXTRA=$OPTARG
            ;;
        o)
            overhead=$OPTARG
            ;;
        v)
            VERBOSE=yes
            ;;
        l)
            LIST=yes
            ;;
        ?|*)
            echo ""
            echo "[ERROR] Unknown parameter \"$OPTARG\""
            usage
            exit 2
    esac
done
shift $[ OPTIND - 1 ]

if [ -z "$DEV" ]; then
    echo "ERROR: no device specified"
    exit 1
fi

if [ -z "$LINE_CEIL" ]; then
    echo "ERROR: no speed specified"
    exit 1
fi

if [ -n "$ACK_rate" -a -n "$LINE_DOWNSTREAM" ]; then
    echo "ERROR: Cannot specify ACK_rate and Downstream at the same time!"
    echo "       Downstream is ONLY used to calculate the ACK_rate"
    exit 1
fi

if [ -z "$EXTRA" ]; then
    EXTRA=0
fi

#if [ -z "$ACK_rate" ]; then
#    ACK_rate=0
#fi
```

B.6.2 HTB script: Functional solution

Listing B.27: Real-world HTB script: htb_overhead_kernel_03.sh

```
#!/bin/sh

# ADSL-optimizer HTB script
# ------------------------------------
# Author: Jesper Dangaard Brouer <hawk@diku.dk>
#
# HTB class prio script with overhead setup for ADSL
# This script requires my kernel and iproute2 patch.
#
# Note:
# HTB needs to support the overhead parameter/option.

configfile =/usr/local/etc/ADSL-optimizer.conf
if [ ! -f $configfile ]; then
    echo "ERROR missing configuration file: \"${configfile}\""
    exit 1
fi
source ${ configfile }

# Use specific "tc" util
if [ -z "$tc" ]; then
   bin=/usr/local/ADSL-optimizer/bin
   tc=${bin}/tc.atm_overhead_kernel
```

```
fi


source ${QUEUE_INC}/parameters.inc
source ${QUEUE_INC}/functions.inc

CEIL=$LINE_CEIL

# ======================================================
# Packet overhead per TCP/IP packet due to ATM/AAL5
# ======================================================
#
# Overhead per AAL5 packet
# AAL5 tail    : 8 bytes per packet (incl. 4 bytes checksum)
#
aal5_tail=8

# AAL5 SSCS headers (SSCS = Service Specific Common Sublayer)
# ------------------
#
# VC (Virtual Circuit) vs. LLC (Logical Link Control)
# There is a basic choice between LLC and VC when configuring
# the encapsulation mode on ADSL. LLC gives more overhead.

# Routed modes:
overhead_PPPoA_VC=2
overhead_PPPoA_LLC=6
#
overhead_rfc2684R_VC=0
overhead_rfc2684R_LLC=8

# Bridged modes:
# There is a special feature bridged mode which some equipment
# support where the MAC-checksum (FCS) can be dropped, which
# reduces the overhead by 4 bytes.
#
overhead_rfc2684B_VC=20
overhead_rfc2684B_VC_noFCS=16
overhead_rfc2684B_LCC=28
overhead_rfc2684B_LCC_noFCS=24
#
overhead_PPPoE_VC=28
overhead_PPPoE_VC_noFCS=24
overhead_PPPoE_LLC=36
overhead_PPPoE_LLC_noFCS=32

# ************************************************
# *** Choose the SSCS overhead encapsulation mode ***
# ************************************************
overhead_SSCS=${overhead_PPPoA_VC}

# Encapsulation overhead:
# ------------------------
if [ -z "$overhead" ]; then
    overhead=$[ ${overhead_SSCS} + ${aal5_tail} ]
fi

# ATM cell padding/aligning
# --------------------------
# The ATM padding/aligning cost is accounted for by
# modifying tc, thus applying my tc (tc_core.c) patch ;-)


# Calculate the ACK_rate from the $LINE_DOWNSTREAM
# ------------------------
if [ -z "$ACK_rate" -a -n "$LINE_DOWNSTREAM" ]; then
    # Will set the variable $ACK_rate
    ACK_rate_calc $LINE_DOWNSTREAM
    #echo "ACK rate calculated to: $ACK_rate "
    #echo "(for a $LINE_DOWNSTREAM Kbit downstream link)"
fi

# Set ACK_rate to zero if not set
if [ -z "$ACK_rate" ]; then
    ACK_rate=0
fi

#ACK_ceil=$CEIL
ACK_ceil=$[90*${CEIL}/100]
#ACK_ceil=$[2*${ACK_rate}]
```

```
# Rate bandwidth left for Classes
#
RATE=$[${CEIL}-${ACK_rate}]

# Print setup information:
#
print_info2

# Update the event file ${EVENTDIR}/${QUEUE_EVENT}
event_startup


# Deletes previous classification (if any)
# ------------------------------------------
# Suppress output, because no classes/qdiscs result
# in an error message, which is expected ...
fun_tc qdisc del dev $DEV root > /dev/null 2>&1


# Create the root of tree
# ----------------------
# (default class: 1:50)
#
fun_tc qdisc add dev ${DEV} root       handle 1: htb default 50 r2q 1

fun_tc class add dev ${DEV} parent 1: classid 1:1 htb \
    rate ${CEIL}kbit ceil ${CEIL}kbit \
    overhead $overhead


##############################################
# Class: 1:10
# Mark : 10
# Description: Interactive traffic
##############################################
procent=20
ceil_procent=20
fun_tc class add dev ${DEV} parent 1:1 classid 1:10 htb \
    rate $[${procent}*${RATE}/100]kbit \
    ceil $[${ceil_procent}*${CEIL}/100]kbit \
    prio 0 \
    overhead $overhead

fun_tc qdisc add dev ${DEV} parent 1:10 handle 4210: \
    sfq perturb 10 limit 50

fun_tc filter add dev ${DEV} parent 1:0 protocol ip \
    prio 10 handle 0x10 fw classid 1:10


##############################################
# Class: 1:20
# Mark : 20
# Description: ACK "class"
##############################################
fun_tc class add dev ${DEV} parent 1:1 classid 1:20 htb \
    rate ${ACK_rate}kbit \
    ceil ${ACK_ceil}kbit \
    prio 1 \
    overhead $overhead

fun_tc qdisc add dev ${DEV} parent 1:20 handle 4220: \
    sfq perturb 10 limit 50

fun_tc filter add dev ${DEV} parent 1:0 protocol ip \
    prio 20 handle 0x20 fw classid 1:20


##############################################
# Class: 1:30
# Mark : 30
# Description: Good traffic
##############################################
procent=28
ceil_procent=80
fun_tc class add dev ${DEV} parent 1:1 classid 1:30 htb \
    rate $[${procent}*${RATE}/100]kbit \
    ceil $[${ceil_procent}*${CEIL}/100]kbit \
    prio 4 \
```

```
    overhead $overhead
#    burst 10000 cburst 5000 \

fun_tc qdisc add dev ${DEV} parent 1:30 handle 4230: \
    sfq perturb 10 limit 64

fun_tc  filter  add dev ${DEV} parent 1:0 protocol ip \
    prio 30 handle 0x30 fw classid 1:30

##############################################
# Class:  1:40
# Mark : 40
# Description: Bulk
##############################################
procent=22
ceil_procent=80
fun_tc class add dev ${DEV} parent 1:1 classid 1:40 htb \
    rate $[${procent}*${RATE}/100]kbit \
    ceil $[${ceil_procent}*${CEIL}/100]kbit \
    prio 4 \
    overhead $overhead

fun_tc qdisc add dev ${DEV} parent 1:40 handle 4240: \
    sfq perturb 10 limit 128

fun_tc  filter  add dev ${DEV} parent 1:0 protocol ip \
    prio 40 handle 0x40 fw classid 1:40



##############################################
# Class:  1:50
# Mark : 50
# Description: Default fallthrough traffic
##############################################
procent=20
ceil_procent=95
fun_tc class add dev ${DEV} parent 1:1 classid 1:50 htb \
    rate $[${procent}*${RATE}/100]kbit \
    ceil $[${ceil_procent}*${CEIL}/100]kbit \
    prio 4 \
    overhead $overhead

fun_tc qdisc add dev ${DEV} parent 1:50 handle 4250: \
    sfq perturb 10 limit 64

fun_tc  filter  add dev ${DEV} parent 1:0 protocol ip \
    prio 50 handle 0x50 fw classid 1:50



##############################################
# Class: 1:666
# Mark : 666
# Description: Bad traffic
##############################################
procent=10
ceil_procent=100
fun_tc class add dev ${DEV} parent 1:1 classid 1:666 htb \
    rate $[${procent}*${RATE}/100]kbit \
    ceil $[${ceil_procent}*${CEIL}/100]kbit \
    prio 7 \
    overhead $overhead

fun_tc qdisc add dev ${DEV} parent 1:666 handle 666: \
    sfq perturb 10 limit 128
    #sfq perturb 10 limit 64

fun_tc  filter  add dev ${DEV} parent 1:0 protocol ip \
    prio 666 handle 0x666 fw classid 1:666


###
# List
list
```

B.7 ADSL-optimizer: Filter

B.7.1 Filter: Rules configuration files

Listing B.28: scp.rules

```
#
# SCP traffic
#
# (maybe: -m tos --tos Maximize-Throughput)
#
# SCP traffic to external machines (upstream)
Append -p tcp -s LOCALNET --dport 22
#
# ... from external machines and back (downstream)
Append -p tcp -d LOCALNET --sport 22
#
#
# Local machines running SSHD (return/upstream traffic)
Append -p tcp -s LOCALNET --sport 22
#
# Local machines running SSHD (downstream)
Append -p tcp -d LOCALNET --dport 22
```

Listing B.29: ssh.rules

```
#
# SSH traffic
#
# Make more specifik... fx. match TOS bits
#
# SSH traffic to external machines (upstream)
Append -p tcp -s LOCALNET --dport 22 -m tos --tos Minimize-Delay
#
# ... from external machines and back (downstream)
Append -p tcp -d LOCALNET --sport 22 -m tos --tos Minimize-Delay
#
#
# Local machines running SSHD (return/upstream traffic)
Append -p tcp -s LOCALNET --sport 22 -m tos --tos Minimize-Delay
#
# Local machines running SSHD (downstream)
Append -p tcp -d LOCALNET --dport 22 -m tos --tos Minimize-Delay
```

Listing B.30: dns-server.rules

```
#
# Match DNS lookups
#
# Misuse can be limited, by only allowing the DNS server
# to use this prio service, all other should ask the DNS server
# when doing DNS lookups
#
# It's possible to further protect the DNS port by matching
# the length of the UDP packet, because RFC1035 specifies that
# a DNS UDP packet limited to 512 bytes (but in this case we
# should trust the servers we specify ... but just in case ...)
#
MyTarget -p udp -m length --length 513: -j RETURN
#
# DNS server doing lookup (upstream)
Append -p udp -s DNS_LOCAL_01 --dport 53
#
# DNS server getting lookup respons (downstream)
Append -p udp -d DNS_LOCAL_01 --sport 53
#
#
# Lookups going to the DNS server (downstream)
Append -p udp -d DNS_LOCAL_01 --dport 53
#
# Responses going back from the DNS server (upstream)
Append -p udp -s DNS_LOCAL_01 --sport 53
#
# We have configured the clients via DHCP to also use
# an external DNS server to lookups:
```

```
#                 193.162.159.194 == ns.tele.dk
# (Also include: 194.239.134.83  == ns3.tele.dk
#             and 194.239.134.82  == ns2.tele.dk)
# Also prio TCP/DNS connection for zone transfer
# (upstream)
Append -p udp -d DNS_EXTERN_01 --dport 53
Append -p tcp -d DNS_EXTERN_01 --dport 53
Append -p udp -d DNS_EXTERN_02 --dport 53
Append -p tcp -d DNS_EXTERN_02 --dport 53
Append -p udp -d DNS_EXTERN_03 --dport 53
Append -p tcp -d DNS_EXTERN_03 --dport 53
# (downstream)
Append -p udp -s DNS_EXTERN_01 --sport 53
Append -p tcp -s DNS_EXTERN_01 --sport 53
Append -p udp -s DNS_EXTERN_02 --sport 53
Append -p tcp -s DNS_EXTERN_02 --sport 53
Append -p udp -s DNS_EXTERN_03 --sport 53
Append -p tcp -s DNS_EXTERN_03 --sport 53
#
# Hmm... somebody is using "ns1.earthlink.net/207.217.126.41" directly.
Append -p udp -d DNS_EXTERN_04 --dport 53
Append -p udp -s DNS_EXTERN_04 --sport 53
#
#
# NOTE:
# Mystisk der er også nogle der spørger 194.239.210.254
# om kollegiegaarden.dk addresser ... Er det også en DNS server IP?
```

Listing B.31: http.rules

```
#
# Filter rules file which matches http(s) local browsers
#
# HTTP
# ----
# Http GET's to external www-servers from koll (upstream)
# (this unfortunately is misused quite often)
Append -p tcp -s LOCALNET --dport 80
#
# Http responses to koll machines (downstream)
## Append     -p tcp -d LOCALNET --sport 80
#
# HTTPS
Append -p tcp -s LOCALNET --dport 443
#
#Append -p tcp --sport 443
#Append -p tcp --dport 443
```

Listing B.32: webserver.rules

```
#
# Filter rules file which matches
#  local web servers ...
#
# Http(s) GET's to the webservers (downstream)
# ----------------------------------------
Append -p tcp -d WEBSERVER_01 --dport 80
Append -p tcp -d WEBSERVER_01 --dport 443

Append -p tcp -d WEBSERVER_02 --dport 80
Append -p tcp -d WEBSERVER_02 --dport 443

Append -p tcp -d WEBSERVER_03 --dport 80
Append -p tcp -d WEBSERVER_03 --dport 443

Append -p tcp -d WEBSERVER_04 --dport 80
Append -p tcp -d WEBSERVER_04 --dport 443
Append -p tcp -d WEBSERVER_04 --dport 8080
#
# Http(s) responses from webservers (upstream)
# ----------------------------------------
Append -p tcp -s WEBSERVER_01 --sport 80
Append -p tcp -s WEBSERVER_01 --sport 443

Append -p tcp -s WEBSERVER_02 --sport 80
Append -p tcp -s WEBSERVER_02 --sport 443
```

```
Append -p tcp -s WEBSERVER_03 --sport 80
Append -p tcp -s WEBSERVER_03 --sport 443

Append -p tcp -s WEBSERVER_04 --sport 80
Append -p tcp -s WEBSERVER_04 --sport 443
Append -p tcp -s WEBSERVER_04 --sport 8080
#
# HTTPS
#Append -p tcp --sport 443
#Append -p tcp --dport 443
```

Listing B.33: tcp_handshake.rules

```
#
# Match TCP connection establishment
#
# SYN packets
Append  -p tcp -m tcp --tcp-flags SYN,ACK,RST,FIN SYN
#
# SYN+ACK packets
Append  -p tcp -m tcp --tcp-flags SYN,ACK,RST,FIN SYN,ACK
```

Listing B.34: tcp_syn_limit.rules

```
#
# Match TCP connection establishment
#
# SYN packets
Append -p tcp -m tcp --tcp-flags SYN,ACK,RST,FIN SYN -m limit --limit 20/sec
#
# SYN packet for explicit port numbers
Append -p tcp --dport 80 -m tcp --tcp-flags SYN,ACK,RST,FIN SYN -m limit --limit 10/sec
Append -p tcp --dport 22 -m tcp --tcp-flags SYN,ACK,RST,FIN SYN -m limit --limit 10/sec
#
# SYN+ACK packets
Append -p tcp -m tcp --tcp-flags SYN,ACK,RST,FIN SYN,ACK -m limit --limit 20/sec
```

Listing B.35: tcp_syn_dstlimit.rules

```
#
# Match TCP connection establishment
#
# SYN packets
Append  -p tcp -m tcp --tcp-flags SYN,ACK,RST,FIN SYN -m dstlimit --dstlimit 4/sec --dstlimit-name syn --dstlimit-mode srcip
#
# SYN+ACK packets
Append  -p tcp -m tcp --tcp-flags SYN,ACK,RST,FIN SYN,ACK -m dstlimit --dstlimit 4/sec --dstlimit-name synack --dstlimit-m
```

Listing B.36: mail_client.rules

```
#
# Mail client  protocols
#
# POP3
Append  -p tcp -m tcp --dport 110
# IMAP
Append  -p tcp -m tcp --dport 143
# IMAPS
Append  -p tcp -m tcp --dport 993
# POP3S
Append -p tcp -m tcp --dport 995
```

Listing B.37: mail_server.rules

```
#
# Mail server
#
# Only the mailserver gets  prioritized  for sending mails,
# because of to many email "viruses" doing direct SMTP connections
#
# External SMTP servers (upstream)
Append -p tcp -m tcp -s MAILSERVER --dport 25
#
```

```
# External SMTP servers (downstream)
Append -p tcp -m tcp -d MAILSERVER --sport 25
#
#
# SMTP connection to the mailserver (downstream)
Append -p tcp -m tcp -d MAILSERVER --dport 25
#
# SMTP responses from the mailserver (upstream)
Append -p tcp -m tcp -s MAILSERVER --sport 25
```

Listing B.38: chat.rules

```
#
# Chat protocols...  (secure/correct  enough ?)
#
# IRC
Append  -p tcp -m tcp --sport 6667
Append  -p tcp -m tcp --dport 6667
# ICQ ??? Port:5190 ???
Append  -p tcp -m tcp --sport 5190
Append  -p tcp -m tcp --dport 5190
# MSN Port:1863???
#  207.46.104.20 :  messenger.hotmail.com
#  207.46.108.11 :  baym-sb11.msgr.hotmail.com
#  207.46.106.110:  baym-cs110.msgr.hotmail.com
#  207.46.106.76 :  baym-cs76.msgr.hotmail.com
#  207.46.xxx.xxx:  Pattern???
Append  -p tcp -m tcp --sport 1863
Append  -p tcp -m tcp --dport 1863
```

Listing B.39: ftp_upload.rules

```
#
# FTP upload
#
#ftp-data        20/tcp
#ftp             21/tcp
#
# FTP control channel (upstream)
Append  -p tcp -m tcp -s LOCALNET --dport 21
#
# FTP data channel (upstream)
Append  -p tcp -m tcp -s LOCALNET --dport 20

#
# Usage of the connection track  helper
# (requires  the  modul "ip_conntrack_ftp"  is  loaded)
#
Append -p tcp -s LOCALNET -m helper --helper ftp
```

Listing B.40: bad-simple.rules

```
#
# Simple matching of bad P2P traffic  using  our upstream  traffic
#
# eDonkey uses port 4662
#  local  machines is sending  (upstream) to  external  machines
# from port 4662.
Append -p tcp -s LOCALNET --sport 4662
Append -p tcp -s LOCALNET --dport 4662
# (also uses  UDP for something...)
Append -p udp -s LOCALNET --dport 4661:4663
Append -p udp -s LOCALNET --dport 4672

#
# A lot of users change this  port  ...  :-(((
Append -p tcp -s LOCALNET --sport 50
Append -p tcp -s LOCALNET --sport 6346
Append -p tcp -s LOCALNET --sport 6348
Append -p tcp -s LOCALNET --sport 1445
Append -p tcp -s LOCALNET --sport 4661
Append -p tcp -s LOCALNET --sport 4663
Append -p tcp -s LOCALNET --sport 3414
Append -p tcp -s LOCALNET --sport 4508
Append -p tcp -s LOCALNET --sport 10920
```

```
# kg108: 00:0B:6A:29:EB:82 (d.5/1-2005)
Append -p tcp -s LOCALNET --sport 8288

# kg220: 00:11:2F:4A:2F:21 (d.5/1-2005)
Append -p tcp -s LOCALNET --sport 12449

# kg177: 00:00:39:39:88:E7 (d.5/1-2005)
Append -p tcp -s LOCALNET --sport 3077

Append -p udp -s LOCALNET --sport 10920
#Append -p tcp -s LOCALNET --sport 46620
#
# 194.239.210.63 / 00:30:4F:12:0F:40
Append -p tcp -s LOCALNET --sport 6699
#
# Morpheus and Kazaa
Append -p tcp -s LOCALNET --sport 1214
Append -p tcp -s LOCALNET --dport 1214
#
# Alot of machines are sending data to dest port 6881
# properly BitTorrent ... 6881-6889
#Append -p tcp -s LOCALNET -m multiport --dports 6881,6882,6883,6884,6885,6886,6887,6888,6889
Append -p tcp -s LOCALNET --dport 6881:6889
Append -p tcp -s LOCALNET --sport 6881:6889
```

Listing B.41: layer7-p2p.rules

```
#
# Layer 7 rules
#
# Requires the kernel and iptables have been patched with
# the Layer 7 Classifier .
#
# http://l7- filter . sourceforge .net/
#
# See available patternes in /etc/l7-protocols/
#
# This is the rules file for the P2P services
# ----------------------------------------------
#

# eDonkey2000 - P2P filesharing (http://edonkey2000.com)
# (From:weakpatterns)
Append -m layer7 --l7proto edonkey

# FastTrack - Peer to Peer filesharing (Kazaa, Morpheus, iMesh, Grokster, etc)
# Pattern quality: marginal
Append -m layer7 --l7proto fasttrack

# Bittorrent - http:// sourceforge .net/ projects / bittorrent /
# Pattern quality: good
Append -m layer7 --l7proto bittorrent

# Gnutella - Peer-to-peer file sharing
Append -m layer7 --l7proto gnutella

# WinMX - a Gnutella client (use gnutella pattern)
# Pattern quality: poor
# (From:weakpatterns)
# Append       -m layer7 --l7proto winmx

# Bearshare - a Gnutella client (use gnutella pattern)
# (From:weakpatterns)
# Append       -m layer7 --l7proto bearshare

# Direct Connect - Peer to Peer filesharing http://www.neo-modus.com/
# Pattern quality: good
# Direct Connect "hubs" listen on port 411
Append -m layer7 --l7proto directconnect

# MUTE - Peer to Peer filesharing - http://mute-net.sourceforge.net/
# Append       -m layer7 --l7proto mute

# Audiogalaxy - Peer to Peer filesharing
# Pattern quality: ok
Append -m layer7 --l7proto audiogalaxy

# Apple Juice - P2P filesharing - http://www.applejuicenet.de/
# Append       -m layer7 --l7proto applejuice
```

```
# GoBoogy - A Japanese (?) P2P protocol
# Append        -m layer7 --l7proto goboogy

# Hotline - An old P2P protocol
# Append        -m layer7 --l7proto hotline

# OpenFT : P2P network (implemented in giFT library)
# Append        -m layer7 --l7proto openft

# Tesla Advanced Communication - P2P file sharing (?)
# Append        -m layer7 --l7proto tesla
```

Listing B.42: simple_vpn.rules

```
#
# Jon runs a VPN connection
#  with uses UDP packets on port 5000 (both hosts)
#
Append -p udp --dport 5000 --sport 5000
```

Listing B.43: ack.rules

```
#
# Try to match pure ACK's
#
Append -p tcp --tcp-flags SYN,ACK,FIN ACK -m length --length :64
#
# We have seen (S)ACK packet being 72 bytes,
# due to the timestamp option and SACK options
Append -p tcp --tcp-flags SYN,ACK,FIN ACK -m length --length :72
```

Listing B.44: ack_extra.rules

```
#
# Try to match pure ACK's
#
Append -p tcp --tcp-flags SYN,ACK,FIN ACK -m length --length :64
Append -p tcp --tcp-flags SYN,ACK,FIN ACK -m length --length :65
#
# We have seen ACK packet being 72 bytes,
# due to the timestamp option and SACK options
# (to get some  statistics  we do some extra matching)
Append -p tcp --tcp-flags SYN,ACK,FIN ACK -m length --length 64:80
Append -p tcp --tcp-flags SYN,ACK,FIN ACK -m length --length 65:80
Append -p tcp --tcp-flags SYN,ACK,FIN ACK -m length --length 64:64
Append -p tcp --tcp-flags SYN,ACK,FIN ACK -m length --length 65:65
Append -p tcp --tcp-flags SYN,ACK,FIN ACK -m length --length 66:66
Append -p tcp --tcp-flags SYN,ACK,FIN ACK -m length --length 67:67
Append -p tcp --tcp-flags SYN,ACK,FIN ACK -m length --length 68:68
Append -p tcp --tcp-flags SYN,ACK,FIN ACK -m length --length 69:69
Append -p tcp --tcp-flags SYN,ACK,FIN ACK -m length --length 70:70
Append -p tcp --tcp-flags SYN,ACK,FIN ACK -m length --length 71:71
Append -p tcp --tcp-flags SYN,ACK,FIN ACK -m length --length 72:72
Append -p tcp --tcp-flags SYN,ACK,FIN ACK -m length --length 73:73
Append -p tcp --tcp-flags SYN,ACK,FIN ACK -m length --length 74:74
Append -p tcp --tcp-flags SYN,ACK,FIN ACK -m length --length 75:75
Append -p tcp --tcp-flags SYN,ACK,FIN ACK -m length --length 76:76
Append -p tcp --tcp-flags SYN,ACK,FIN ACK -m length --length 77:77
Append -p tcp --tcp-flags SYN,ACK,FIN ACK -m length --length 78:78
Append -p tcp --tcp-flags SYN,ACK,FIN ACK -m length --length 79:79
Append -p tcp --tcp-flags SYN,ACK,FIN ACK -m length --length 80:80
Append -p tcp --tcp-flags SYN,ACK,FIN ACK -m length --length 81:81
Append -p tcp --tcp-flags SYN,ACK,FIN ACK -m length --length 82:82
Append -p tcp --tcp-flags SYN,ACK,FIN ACK -m length --length 83:83
Append -p tcp --tcp-flags SYN,ACK,FIN ACK -m length --length 84:84
```

Listing B.45: ping-ask.rules

```
#
# Test rule,  to  measure the (ping)  latency
#
Append  -p icmp -d ask.diku.dk
Append  -p icmp -s ask.diku.dk
```

Listing B.46: ping-munin.rules

```
#
# Test rule, to measure the (ping) latency
#
Append -p icmp -d munin.diku.dk
Append -p icmp -s munin.diku.dk
```

Listing B.47: ping-hugin.rules

```
#
# Test rule, to measure the (ping) latency
#
Append -p icmp -d hugin.diku.dk
Append -p icmp -s hugin.diku.dk
```

Listing B.48: ping-brok.rules

```
#
# Test rule, to measure the (ping) latency
#
Append -p icmp -d brok.diku.dk
Append -p icmp -s brok.diku.dk
```

Listing B.49: ping-www.diku.dk.rules

```
#
# Test rule, to measure the (ping) latency
#
Append -p icmp -d www.diku.dk
Append -p icmp -s www.diku.dk
```