Helsinki University of Technology
Department of Computer Science and Engineering

Piia Pulkkinen

Design and Implementation of a Policy Control Agent for a Differentiated Services Router

Master's thesis submitted in partial fulfilment of the requirements for the degree of Master of Science in Technology

Espoo, 24th May, 2004

Supervisor: Professor Raimo Kantola

Instructor: Marko Luoma, Lic.Sc.(Tech.)

Contents

Acronyms

AF	Assured Forwarding
ALTQ	Alternate Queuing
API	Application Programming Interface
BA	Behaviour Aggregate
BGP	Border Gateway Protocol
BSD	Berkeley Software Distribution
CBQ	Class Based Queuing
CBS	Committed Burst Size
CIR	Committed Information Rate
CPU	Central Processing Unit
CTR	Committed Target Rate
DARPA	Defense Advanced Research Projects Agency
DiffServ	Differentiated Services
DRD	Derivative Random Drop
DRR	Deficit Round Robin
DSCP	Differentiated Services Code Point
ECN	Explicit Congestion Notification
EF	Expedited Forwarding
EWMA	Exponentially Weighted Moving Average
FCFS	First Come First Served
FEA	Forwarding Engine Abstraction
FEC	Forwarding Equivalency Class
FIFO	First In First Out
GUI	Graphical User Interface
HFSC	Hierarchical Fair Share Curve
HTB	Hierarchical Token Bucket
HTTP	Hypertext Transfer Protocol
IETF	Internet Engineering Task Force
IGMP	IP Group Membership Protocol

IntServ	Integrated Services
I/O	Input/Output
IPC	Inter-Process Communication
IPSec	Internet Protocol Security
IPv4	Internet Protocol version 4
IPv6	Internet Protocol version 6
ITU-T	International Telecommunications Union, Telecommunications Sector
JoBS	Joint Buffer Management and Scheduling
LDP	Label Distribution Protocol
LSR	Label-Switched Router
MAC	Media Access Control
MF	Multi-Field
MPLS	Multiprotocol Label Switching
NAT	Network Address Translation
NIST	National Institute of Standards and Technology (U.S.)
OSPF	Open Shortest Path First
PBS	Peak Burst Size
PEP	Policy Enforcement Point
PHB	Per Hop Behaviour
PIM-SM	Protocol Independent Multicast - Sparse Mode
PIR	Peak Information Rate
PRIQ	Priority Queuing
PTR	Peak Target Rate
QCMD	Queue Command
QOP	Queue Operation
QoS	Quality of Service
RED	Random Early Detection
RIB	Routing Information Base
RIO	Random Early Detection In/Out
RSVP	Resource Reservation Protocol
RSVP-TE	Resource Reservation Protocol Traffic Engineering
SNMP	Simple Network Management Protocol
TCB	Traffic Conditioning Block
TCif	Traffic Control interface
TCP	Transmission Control Protocol
TOS	Type of Service
trTCM	Two Rate Three Colour Marker
TSWTCM	Time Sliding Window Three Colour Marker

UDP	User Datagram Protocol
URL	Uniform Resource Locator
VPN	Virtual Private Network
WRED	Weighted Random Early Detection
WRR	Weighted Round Robin
WWW	World Wide Web
XORP	Extensible Open Router Platform
XRL	XORP Resource Locator

Chapter 1

Introduction

1.1 Background

The story of the Internet began in the 1960s, when the U.S. Defense Advanced Research Projects Agency, also called DARPA, launched a network project in co-operation with a group of universities in the United States. The ARPANET network stayed in the use of researchers for a long time until it was made publicly accessible. The introduction of the World Wide Web (WWW) in the 1990s started a new era in the history of the network communication. The public network began to expand rapidly from country to country and from continent to continent, and soon there was the Internet, a multinational, multilingual and in no way controllable network. The baby had grown to a dinosaur.

Since the early days the traffic in the Internet has been based on datagrams and best-effort packet forwarding. In practice, all packets are delivered to their destinations independently of the other packets. The best-effort service means that all efforts are made to ensure fast and correct packet delivery, but nothing is guaranteed. Depending on various network conditions packets may end up to wrong destinations, be delayed, duplicated, corrupted or even get dropped. However, the best-effort service worked tolerably well until the Internet exploded, because the traffic volumes were low enough and the network was able to carry most of the traffic through. During the last ten or fifteen years, the amount of network traffic and the variety of applications have increased enormously which has revealed the limitations of the original network design. The best-effort architecture is facing an impossible task when trying to handle all packets in a mixture of traffic sufficiently.

All applications suffer from the lousy network conditions, but the actual victims are the intolerant real-time applications, like IP phone calls and video conferencing. If several packets get lost on the way and they need to be retransmitted over and over again, or the connection has for some other reasons huge delays, file transfer will work, but will be extremely slow. On the contrary, the real-time applications can become totally unusable due to packet losses or intolerable delays. The architecture, like best-effort, offering only one service cannot meet the needs of various types of applications whose requirements from the network differ substantially from each other.

The urgent demand for Quality of Service (QoS) architectures on the Internet was realised also by the IETF. Its first attempt to bring quality differentiation to the Internet was the introduction of the Integrated Services (IntServ) [BCS94] architecture. However, due to the complex resource reservation feature IntServ was soon found unscalable and thus not an optimal solution. Some years later the IETF published another QoS architecture proposal, namely Differentiated Services (DiffServ) [BBC+98]. Unlike IntServ, DiffServ does not use resource reservations and does not itself create services. Instead, DiffServ uses provisioning and provides only mechanisms for building services which makes it more scalable and practicable architecture than IntServ.

1.2 Aim of the Thesis

Although DiffServ is currently deployed in the Internet only in a small scale, it is already facing new challenges set by the changing world. People are more and more moving around with pocket PCs and mobile phones with Internet access and laptops, which can be used on business trips or taken over to a friend's place, are replacing desktop computers. Thus, the support for mobility of the users is becoming an essential feature of the QoS networks. In a DiffServ network, the support for mobility requires a centralised management of customer profiles and authentication. Hereby, the user is allowed to log into the network through any access point and also to change the access point at any time. User's own customer profile is always installed into the edge router he is at the moment connected to, no matter of the location. When the user changes the access point or logs out from the network, his customer profile is removed from the edge router to make room for other profiles.

The DiffServ architecture brings configurability to the network, but the increase of control is not achieved for free. The traffic management mechanisms in the routers can be configured for only one traffic situation, although the amount and types of traffic vary all the time. An inappropriate configuration has an unwanted effect on the traffic and can decrease the network utilisation even below the best-effort service level. Therefore, it is crucial to have a mechanism in the network that modifies the configurations on-the-fly according to the changing network conditions. Without any adaptability, the DiffServ architecture can provide QoS only on paper, not in a real, heterogeneous network.

We have built a prototype of an adaptive policy-based network which bases the resource provisioning on policies and traffic volumes and allows the users to move from one access point to another. The routers in the network are DiffServ routers whose DiffServ functionalities were realised with a traffic management software called Alternate Queuing (ALTQ) [Cho01]. The configuration of ALTQ can be changed dynamically, and the Policy Control Agent software is responsible for performing the modifications in the edge routers. The task of the Agent is to receive the new configuration parameters from the databases in the Policy Server and configure ALTQ according to them. This thesis describes the design of the Policy Control Agent, takes a look at the most important implementation issues and finally presents and analyses the results of the performance measurements.

1.3 Structure of the Thesis

Chapter 2 gives an overview of the Quality of Service concepts. It explains briefly the most common QoS mechanisms and introduces Integrated Services [BCS94], Differentiated Services [BBC+98] and Multiprotocol Label Switching [RVC01] architectures. Chapter 3 takes a look at the research and development work that has been done all over the world concerning traffic management. The chapter presents five different software solutions which all have their own viewpoints to the subject. Chapter 4 introduces the ALTQ traffic management software [Cho01] emphasising on the functionalities and design issues relevant to the Policy Control Agent implementation. Chapter 5 describes the design and implementation of the Policy Control Agent. Chapter 6 defines the performance test procedure for the ALTQ/Policy Control Agent router and presents the results with brief analyses. In addition, it gives some suggestions for further development. Finally, Chapter 7 summarises the work.

Chapter 2

Overview of Quality of Service

2.1 Introduction

In the traditional best-effort network, all packets receive the same treatment and therefore have equal possibilities to get delayed, corrupted or lost on the way. In the networks providing QoS, traffic is divided into several groups which receive different treatment. Traffic differentiation can be based on the traffic type or customer, for example, and the treatment is defined in the contract between the customer and the service provider.

This chapter gives some definitions for QoS and introduces the most popular methods and mechanisms to realise QoS in the IP networks. The chapter takes a look at two QoS architectures, namely Integrated Services and Differentiated Services, and at Multiprotocol Label Switching architecture, which provides one way to implement the Qos architectures. Due to the field of this thesis, Differentiated Services is studied closer than Integrated Services.

2.2 Defining Quality of Service

According to ITU-T, Quality of Service is 'the collective effect of service performances which determine the degree of satisfaction of a user of the service' [IT94]. Therefore, from the users' point of view QoS is easily a psychological issue. Service providers see QoS as a set of adjustable parameters, but in practice measuring QoS and tuning the parameters is not an easy task. In the service contract between the customer and the service provider, the offered forwarding service is defined with appropriate QoS properties. It is important that the properties are measurable for both parties, so that the fulfilment of the contract can be controlled.

2.2.1 QoS Properties

The performance of the network is commonly characterised with properties including latency, jitter, throughput and error rate. As different kinds of traffic have different needs, QoS is actually optimisation of one or more of the parameters.

The total delay that the packet encounters when travelling from the source to the destination is often referred to latency. Latency consists of three parts: propagation delay, transmission delay and queuing delay. Propagation delay is the time that the packet spends on the wire and it exists always, because a packet cannot travel faster than the speed of light. Transmission delay indicates the time it takes to transmit all bits of the packet, and it depends on the size of the packet and the capacity of the link. Queuing delay is the time the packet has to spend in the queue waiting for transmission. Packet sizes and link capacities are usually hard to modify, but queuing delay can be affected a bit easier. Latency is an especially important characteristic for intolerant real-time applications which require strict bounds for latency in order to work properly. Another important attribute for the applications is jitter which stands for latency variation in a short time-scale. Jitter prevents the applications from using the packets right away when they arrive, because the timing between them is arbitrary. Therefore, the applications store all the incoming packets to a playback buffer and smooth the jitter by buffering the less delayed packets some extra time. [PD00]

Throughput defines the maximum number of bits that the network is able to transfer from end-to-end in a certain amount of time. Throughput is affected by the physical restrictions of the transmission media and the traffic volumes inside the network, hence it can be substantially lower than the available bandwidth. [PD00]

The reliability of the network can be characterised as a number of errors per a certain amount of data or time. Errors include packet loss, when the transmitted packet never reaches the destination, packet duplication, when the packet gets copied en route and is thereby received in the destination multiple times, and packet corruption, when some bits get altered and make the packet unusable. TCP (Transmission Control Protocol), which is a reliable protocol and uses acknowledgements to ensure the correct transmission of the packet, can tolerate an unreliable network quite far. UDP (User Datagram Protocol), on the contrary, is an unreliable protocol and it suffers greatly from the continuous packet losses, because it does not implement a packet retransmission mechanism. [PD00]

2.2.2 Implementing QoS in the IP Networks

In order to put QoS in practice, working QoS architectures are needed. The architectures consist of different elements which are implemented as separate mechanisms. The QoS mechanisms have for some time been a hot research topic, hence currently several mechanisms exist and new ones are studied. Some mechansims have been deployed wider than the others, but there is no such thing as a common QoS architecture or even a common mechanism for any functionality. All architectures and mechanisms are just proposals.

The lack of a prevalent QoS solution may be one reason for the slow deployment of QoS in the Internet. Some service providers have made their decision and chosen one architecture to support, but most service providers have done nothing. In this situation, the end-to-end QoS between different operators' networks is hard to achieve. The more networks there are between the communication partners the more difficult it is to receive end-to-end QoS. There are ways to fit the services offered in separate networks together, but it is expensive and requires detailed and complex agreements between the service providers. Because there is no point in offering a half-way QoS, the services are generally defined only inside the service provider's own network.

2.3 Integrated Services and RSVP

When the real-time applications became common in the Internet during the 1990s, it was soon realised that the existing Internet architecture could not meet the requirements of the real-time traffic. Jitter and packet loss were too severe for that type of traffic. Hence, IETF developed a service model called Integrated Services (IntServ) [BCS94]. Its purpose was to make sure that the real-time traffic was provided the network resources it needed and at the same time, the existence of the best-effort traffic was not forgotten. [Wan01]

2.3.1 Services

The IntServ model offers two kinds of services targeted to the real-time traffic. Guaranteed Service [SPG97] is tailored to meet the needs of the intolerant real-time applications that need the packets to arrive at a certain time. The service sets strict bounds to the maximum queuing delay and, to be able to control the delay, reserves some amount of network resources for the use of the traffic flow. Controlled Load Service [Wro97] is intended for tolerant real-time applications which can accept packet losses and adapt to jitter to some extent. Controlled Load service offers better than best-effort service by providing the network conditions similar to lightly loaded best-effort network. The delay bounds or bandwidth cannot be fixed, thus the service does not provide strict guarantees. [Wan01]

2.3.2 Architecture

The IntServ router reference model includes four components that implement the QoS functionalities: classifier, packet scheduler, admission control and reservation setup protocol. The classifier separates the incoming packets to the different classes according to the flow specification by examining the packet header. It also monitors that the hosts stay inside the agreed traffic profile and do not send too much traffic. The packet scheduler controls the packet forwarding by using different queues. Depending on the class the packet belongs to, it receives a certain kind of treatment from the scheduler. When a new packet stream wishes to enter the network, the admission control in every router checks if there are resources available. If the requested service is Guaranteed Service, the reservation setup protocol called Resource Reservation Protocol (RSVP) [ZBHJ97] is used to set up the route and make the reservations. In Controlled Load Service, RSVP messages are used to find out whether there are enough resources for the new packet stream to enter the network, but no reservations are made. In addition to the IntServ components, the router includes the normal routing and network management elements. The components are illustrated in Figure 2.1 which presents a simplified reference model of an IntServ router. [BCS94]

2.3.3 RSVP

RSVP is a signalling protocol that is used to carry reservation messages between the traffic sender and the receiver. The protocol is not tied to IntServ or any other

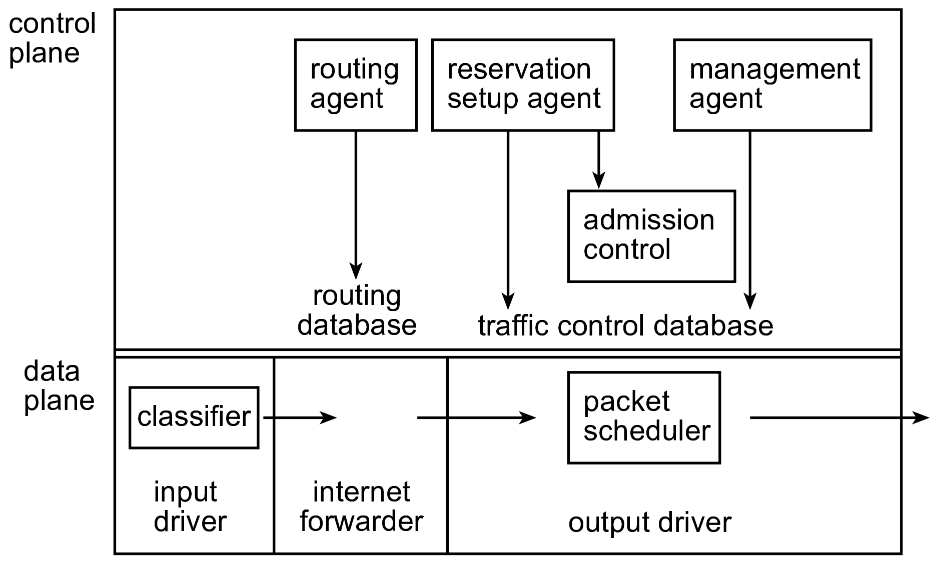


Figure 2.1. *[BCS94, Luo00] IntServ router reference model*

architecture, nor has it anything to do with routing. The reservations are soft-state which means that they expire after a certain time if they are not renewed before that. In some cases, the reservations can be merged together or shared among several packet streams. [Wan01]

Setting up a reservation begins with a PATH message that the source host sends to the destination host by using RSVP. The message includes a specification of the traffic that the source is willing to send. To find the way to the destination, RSVP uses the guidance of some routing protocol. At the destination end, the host generates a RESV message based on the information in the received traffic specification. The message contains a description of the requested resources and a definition of the packets which are eligible to use the resources. The RESV message is sent back to the source host using the same route as the PATH message. Routers on the way make resource reservations according to the RESV message. Because the reservation process in the routers begins from the receiver end, the reservation mechanism of RSVP is called receiver oriented. [Wan01]

2.4 Differentiated Services

Some years after the deployment of IntServ it was clear that the architecture lacked scalability and was not able to solve all QoS problems. There was a need for an advanced QoS architecture that would be scalable and simple at the same time. In 1998, the IETF published a definition for Differentiated Services (DiffServ) [BBC+98], an architecture designed to offer better than best-effort service in different service levels. One major difference to the IntServ was that in DiffServ there was no resource reservation with special protocols. Instead, DiffServ achieved resource assurance with a set of different methods consisting policing, provisioning and traffic priorisation. [Wan01, Kil99]

2.4.1 Concepts

The basic idea of DiffServ is to divide the traffic on the edge of the network into a set of groups which receive different forwarding treatment inside the network. The treatment is defined by Per-Hop Behaviours (PHBs) which are marked to the packet IP headers with a 6-bit Differentiated Services Code Point (DSCP). The place for the DSCP in the IPv4 header is the former Type of Service (TOS) field, nowadays called a DSCP field. One DSCP value refers to exactly one PHB, but one PHB can map to several DSCP values. [Kil99, Wan01] Figure 2.2 shows the transformation of the TOS field into the DSCP field. The DSCP field reserves the first six bits of the TOS field bits, while the last two bits are currently unused. Experimentally the bits have been used by ECN (Explicit Congestion Notification).

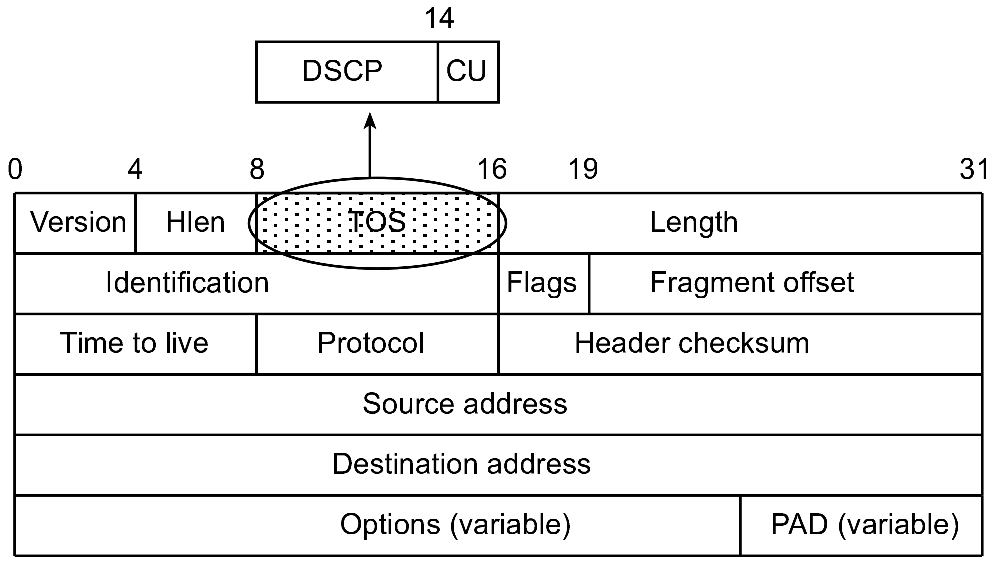


Figure 2.2. *IPv4 header*

A PHB defines the packet treatment in the network, and all the packets that have the same PHB form a behaviour aggregate and receive the same treatment. PHB neither defines a service nor makes any reference to the actual implementation. Instead, it is a service building block situated between services and mechanisms. The description of a PHB covers all the main quality aspects: latency, packet loss ratio and bit rate. The whole DiffServ architecture can be seen as a set of logical best-effort networks in parallel, each having its own network resources and characteristics and mapping to one PHB. [Kil99, Luo00]

PHBs are divided into PHB groups containing PHB classes that are sets of PHBs. There are currently two standardised PHB groups: Assured Forwarding (AF) [HBWW99] and Expedited Forwarding (EF) [JNP99]. They can be referred to service classes which have a certain kind of equality with the IntServ services. AF equates to Controlled Load and EF to Guaranteed Services. AF PHB consists of four PHB classes, each with three drop precedence levels. One flow can be destined to only one PHB class, but the packets inside the flow may have different drop precedencies. Each AF class is assigned a certain amount of forwarding resources, and if the class is congested, the packets with the highest drop precedence are discarded first. The class can be selected by the customer, and the packet drop precedence is usually selected by the edge router. EF offers end-to-end assured service, where the QoS properties can have firm limits. Unlike IntServ Guaranteed Service, EF does not reserve network resources. Instead, the adequacy of the resources is achieved with provisioning. In addition, EF has only one PHB and all packets within the behaviour aggregate share the same resources. Thus, the resource allocation is done per traffic aggregate, not per-flow, as in IntServ. [HBWW99, Wan01, Kil99]

A network or a part of a network which implements the DiffServ architecture under the same PHB and service specifications is called a DiffServ domain. The domain consists of two types of routers: the edge routers and the core routers. Edge routers are located in the boundary of the domain, hence they are the access points of the DiffServ network and have some network interfaces outside of the domain. The edge routers that connect to the customer networks are called DiffServ access routers, because they are the customer access points to the domain. Routers connecting DiffServ domains together are called DiffServ border routers. Core routers are completely inside the domain, thus all links are connected to the DiffServ capable routers. An example of a network consisting of two DiffServ domains is presented in Figure 2.3. [Luo00, Wan01]

Both types of routers have different roles. The edge routers are the brains of the network. They examine the headers of all incoming packets and mark the correct DSCP value to the DSCP field according to the predefined rules. The routers also

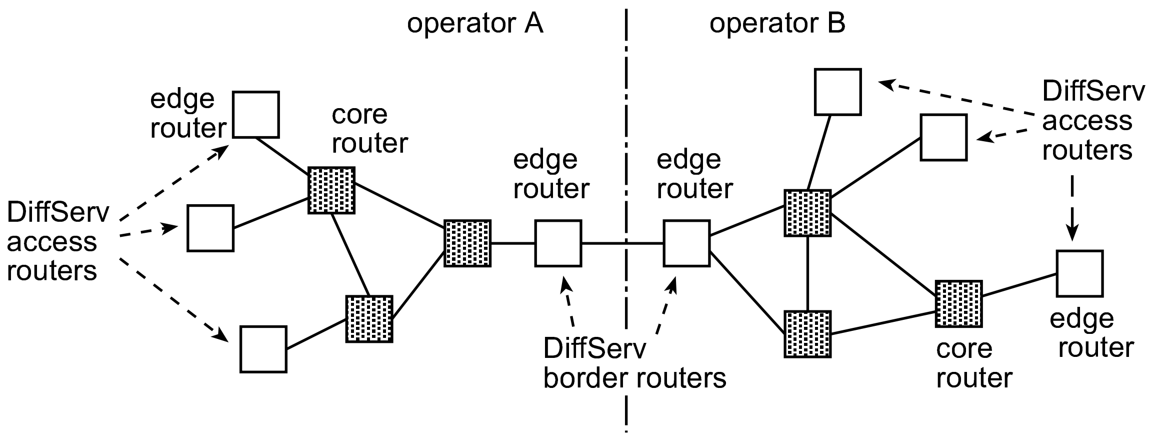


Figure 2.3. *Two DiffServ domains connected together*

supervise that the customers do not violate the service contract by sending traffic which is out of the agreed profile. The core routers examine the DSCP field of every packet and determine the requested forwarding treatment according to the DSCP value. The scalability and efficiency of the DiffServ architecture are based on the distribution of the router functionalities and intelligence. The time-consuming work is done on the edge, where the traffic volumes and data rates are low, and the busy core routers are allowed to concentrate on other operations. [Wan01]

2.4.2 Comparison of the Best-Effort and DiffServ Router Architectures

The ordinary network router which handles the best-effort traffic takes the incoming packet, checks its destination address and forwards the packet to the correct output link according to the packet destination address. The components needed for this operation are a forwarder, a scheduler and a routing agent. The routing agent calculates the routes by using the routing algorithm and maintains the routing table. From the routing table the forwarder gets the information which link the packet has to be directed to. The packets are queued to wait for their turn to get forward, and the scheduler decides which packet is transmitted next. The scheduler works on the First Come First Served (FCFS) basis which means that the packets are handled in the arriving order. The queues are managed in tail drop basis. If there are too many incoming packets and the queues which are always of finite size become full, all packets that do not fit into the queue are discarded. The packets that are already in the queue will not get dropped. [Luo00, PD00, Wan01]

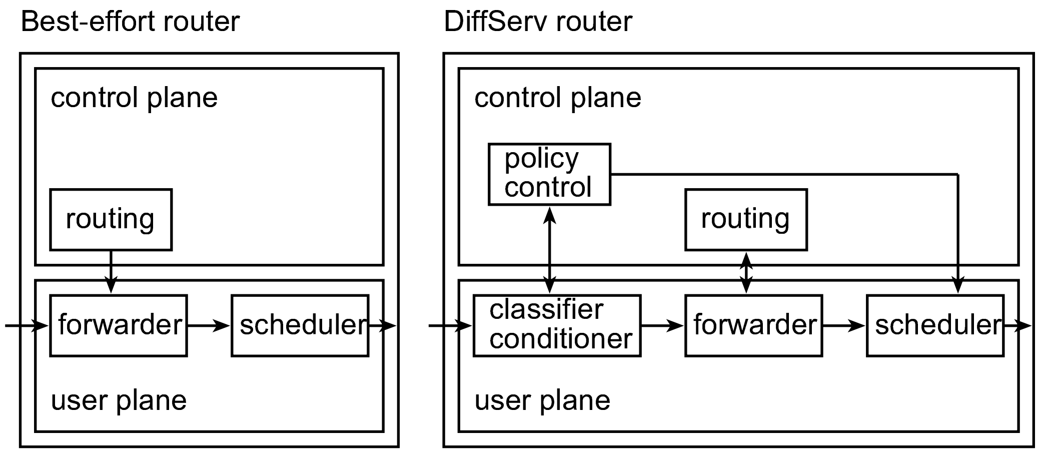


Figure 2.4. *[Luo00] Components of a best-effort router and a DiffServ router*

The DiffServ router architecture adds some elements to the best-effort router, as Figure 2.4 shows. The first component in the forwarding path is the classifier which divides the incoming packets into different groups based on the header information. The conditioner measures the traffic properties, sets a DSCP value to the header and alters the traffic shape to fit inside the committed profile, if needed. Policy controller is a management unit which takes care of the configuration of the classifier, conditioner and scheduler. [Wan01, Luo00]

2.4.3 Classifier

The classifier on the edge router is usually a multi-field (MF) classifier, because the edge routers often need to examine several fields of the packet header. The packets are divided into different groups according to the predefined rules called filters and forwarded to the conditioner. When the packets arrive at a core router, they already have the DSCP value set. So the classifier in the core router is a behaviour aggregate (BA) classifier that separates packets according only to the DSCP value. [Wan01]

2.4.4 Conditioner

The conditioner is located in the DiffServ router forwarding path after the classifier. Together the conditioner and the classifier form a bigger element called a traffic conditioning block (TCB). The conditioner is used especially in the edge routers but in some cases also in the core routers. The functional components inside the conditioner are meter, marker, shaper and dropper. The meter is used to measure the temporary properties of the traffic which are compared to the committed traffic profile to find out whether the traffic is inside or outside the profile. The common properties are peak rate, average rate and maximum burst size of the traffic. The marker sets the DSCP value to the packet header either straight away or according to the meter information. The shaper can force the traffic back into its profile limits by keeping the packets queued for awhile and thus delaying them. If shaping is not enough to bring the traffic into compliance with the profile, the dropper may discard certain packets. Shaping traffic by dropping is known as policing. [BBC$^+$98, Luo00]

Token bucket is a common meter which uses tokens to determine if the traffic is inside or outside the profile. Tokens are generated at a certain rate to the bucket, and when a packet arrives, the tokens are calculated. If there are not enough tokens for the packet, it is considered to be out-of-profile. Otherwise the packet is inside the profile. [Wan01]

Dividing the traffic into two, like the token bucket does, is not enough for DiffServ AF due to its three drop precedencies. Hence, AF uses a dual token bucket scheme called a Two Rate Three Colour Marker (trTCM) [HG99] which is able to separate the packets into three groups. The traffic exceeding the Peak Information Rate (PIR) and the associated Peak Burst Size (PBS) is marked as red, whereas the traffic below the PIR and PBS but over the Committed Information Rate (CIR) and Committed Burst Size (CBS) is marked as yellow. The traffic that fits nicely inside the CIR and CBS limits is marked as green. The trTCM is implemented as two consecutive token buckets, of which the first one leaves out the red traffic and the second one separates the green traffic from the yellow traffic. The other possible marker for AF is TSWTCM (Time Sliding Window Three Colour Marker) [FSN00] which is not based on token buckets but on a long-term traffic rate estimating. TSWTCM marks the packets as red, yellow and green, using Committed Target Rate (CTR) and Peak Target Rate (PTR) as thresholds. [Wan01]

2.4.5 Scheduler

The scheduler decides which packet is transmitted next to the output link. The packets waiting to be scheduled are placed into one or more queues which can be managed with various methods. The simplest and most common queue management algorithm is tail drop. The queue is filled in order and when the queue gets full, all packets trying to get in are dropped. New packets are accepted as soon as there is room in the queue. Random Early Detection (RED) [FJ93] and RED In/Out (RIO) [CF98] are more sophisticated, active queue management algorithms. RED was designed to provide a new mechanism for congestion notification. As shown in Figure 2.5, the algorithm accepts all packets to the queue until a certain threshold (MinTH) on the average queue length calculated with an exponentially weighted moving average (EWMA) algorithm is reached. After that, the incoming packets are dropped with a linearly increasing probability (Pdrop) until the second threshold (MaxTH) is reached. When the queue fills up, all packets are discarded. RIO scheme is the basic idea behind DiffServ AF. RIO consists of two parallel RED algorithms, one for high precedence traffic and the other for low precedence traffic. To be able to separate the low and high precedence traffic, RIO expects the packets to be marked beforehand in the conditioner element. A generalisation of RIO is called Weighted RED (WRED), and it enables using several RED algorithms in parallel. [Wan01, PD00, Luo00]

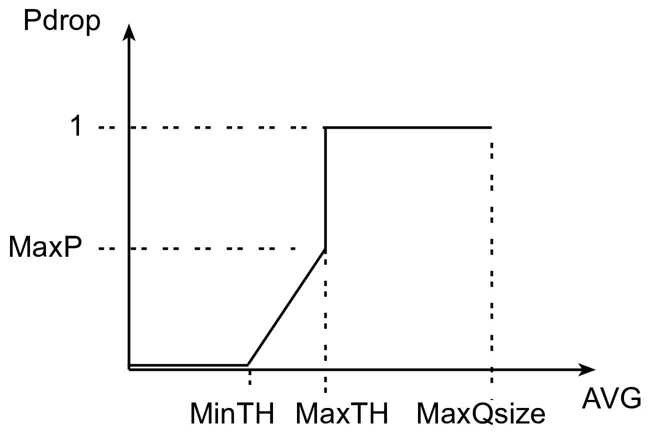


Figure 2.5. *[Luo00] Drop probability function of RED*

The scheduling is taken care of by a scheduling algorithm. The algorithm can be run for a single queue, several queues or even over several other scheduling algorithms. The simplest and the most common scheduler is First Come First Served (FCFS), also called First In First Out (FIFO). FCFS schedules the packets in the order they arrive to the router, and does not provide any differentiation. Priority Queuing (PRIQ) is a slightly enhanced FCFS which observes the priority values marked in the packets. One way to implement PRIQ is to assign own FCFS queue for each priority and schedule always the higher priority packets before the lower priority packets. Another way is to use only one FCFS queue where the higher priority packets are placed in front of the lower priority packets. [PD00]

Round Robin schedulers assign a time slice of certain size to each connections and rotate around the queues scheduling packets according to the time slices. In Weighted Round Robin (WRR), the time slices are not of the same size, thus during one rotation, the scheduler selects more packets from some connections than from others. WRR is inefficient if the packet sizes vary greatly, because the operation of WRR is based on packets. Deficit Round Robin (DRR) counts bits, not packets, and divides the frames into quantums according to the weights of the connection. The size of the quantum governs the number of packets that can be sent in one rotation. If the quantum is too small even for one packet, the connection has to wait several rotations to be served. [Luo00, PD00]

Class Based Queuing (CBQ) [FJ95] is the most popular scheduling mechanism in DiffServ. CBQ is a hierarchical scheduler which supports link capacity sharing among different classes. The traffic is divided into classes using predefined rules called filters. Sharing is based on the tree-like class structure whose leaves represent the actual scheduled packet queues. An example of a CBQ class tree is presented in Figure 2.6. When the leaf classes are not congested, CBQ uses a general scheduler which is some variant of Round Robin schedulers. If there are one or more congested leaf classes, the general scheduler is replaced by a link share scheduler which operates according to the rules set by the user. [PD00, Luo00]

2.5 MPLS

The definitions of the DiffServ and IntServ architectures do not say anything about the implementation issues. There are several ways to implement the QoS architectures in the IP networks, and one option is to use the Multiprotocol Label Switching (MPLS) [RVC01] architecture. In general, MPLS is suitable for realising traffic engineering and guaranteed QoS and building Virtual Private Networks (VPNs).

The forwarding process in a router is based on examining every incoming

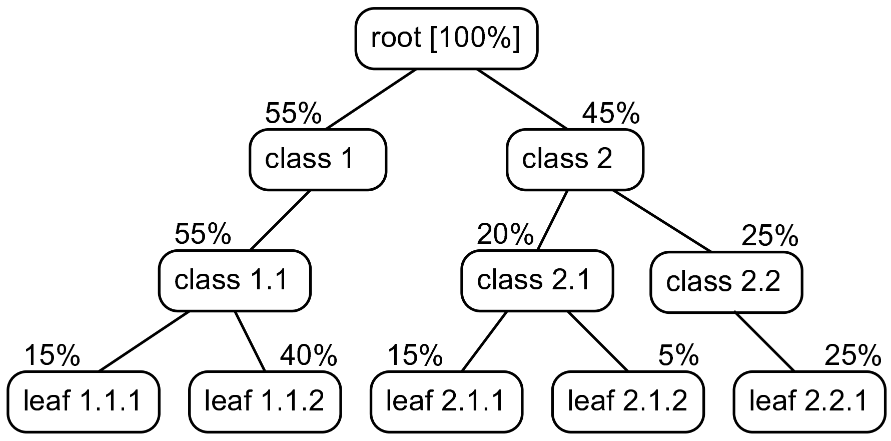


Figure 2.6. *An example of a CBQ class tree with link sharing percentages*

packet header and making the decision of the next hop independently. The operation is reliable but time-consuming. MPLS was developed to lighten the work of the routers and to speed up the routing decisions. The basic idea behind MPLS is that packets are routed over the predefined paths, and a short, fixed identifier called a label is stamped on each packet header to determine the next hop. The name 'multiprotocol' comes from the ability to have multiple protocols implemented over the same forwarding path. [Wan01, RVC01]

When a packet arrives to the edge of an MPLS network, its header is examined and it is classified into one of Forwarding Equivalency Classes (FEC). FEC is a group of packets which are forwarded identically inside the MPLS network. The packets take the same route and receive same kind of treatment. The MPLS header is added to the packet header, and a label in it specifies the FEC of the packet. Inside the MPLS network there are predefined paths called label-switched paths (LSPs) for different FECs. The label-switched routers (LSRs), as the MPLS capable routers are called, forward the packet along the appropriate LSP according to the label in the packet header. The labels are defined locally and one-way between two LSRs, hence every router has a table which maps the labels meaning the same FEC together. As a FEC has a different label in the different links, the router needs to change the labels for all incoming packets before forwarding them. When the packet arrives to the egress router of the MPLS network, the MPLS header is removed. An example of an MPLS network is illustrated in Figure 2.7. The figure also shows the basic operation of the network by presenting two LSPs between the hosts and the labels along the path. [Wan01]

A label distribution protocol is responsible for setting up the LSPs by informing the

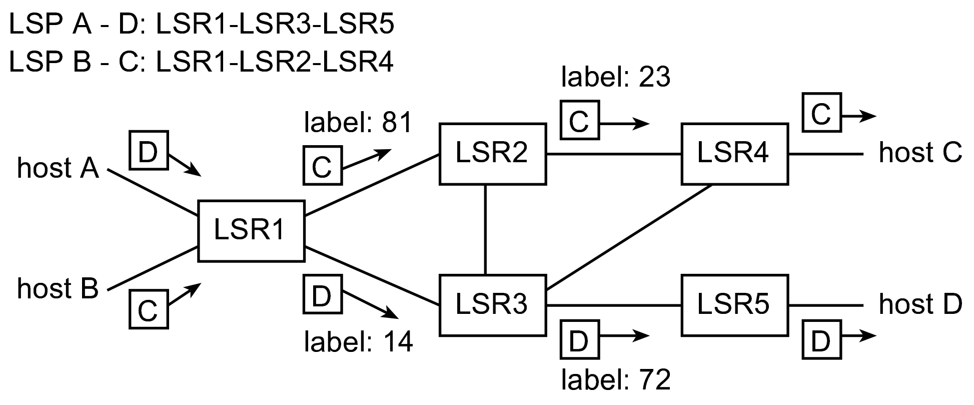


Figure 2.7. *Basic operation of an MPLS network*

neighbouring routers about the label mappings. The protocols that can be used for that purpose include LDP (Label Distribution Protocol) [ADF+01] and an RSVP extension RSVP-TE (RSVP Traffic Engineering) [ABG+01].

Chapter 3

Related Research and Development Work

3.1 Introduction

Traffic management in the IP networks is currently a hot research topic worldwide. Numerous mechanisms and tools have already been developed in various research projects. The emphasised aspects, operation and architecture of the mechanisms depend on the primary objectives of the project. Thus, the points of view to the traffic management and the design decisions in the implementation issues vary greatly. This chapter introduces five freely distributable traffic management tools: Dummynet, NIST Net, Click, XORP and Linux Traffic Control. Alternate Queuing (ALTQ) traffic management software which we used as a basis in the router development is presented in Chapter 4.

3.2 Dummynet

Dummynet [Riz97, Riz98] is a simple software tool initially designed for performance testing of the network protocols and applications. In addition, it has been introduced as a traffic manager. Dummynet operates inside the protocol stack, where it captures all network traffic and creates effects of delays, limited bandwidth and bounded queues, for example. Thus, it is able to simulate network conditions which otherwise would need a number of machines and a large network to produce. Dummynet was developed by Luigi Rizzo in the University of Pisa in Italy. The first version of the software was released in 1996 as a part of a research project. Dummynet was designed for the FreeBSD operating system, and since FreeBSD version 3.1 it has been integrated into the distribution. [Riz02, Riz97]

3.2.1 Design Goals

Researchers have for a long time struggled with the problem how to build a realistic, reasonable-sized and affordable testing network environment. Simulation is an option in rare cases, since the real network conditions are seldom known exactly. However, accuracy is required to configure the simulator to correspond the real network. The other option, putting up an actual-sized network, is not usually a sensible idea. The design goal of Dummynet was to offer a third option. With Dummynet, one computer can be configured to represent the whole network and emulate its conditions. The experiments can be made like in a real-sized network with the difference that the network between the end nodes is a single dummynet machine. If there is a need to get different conditions to different links in a real test network, Dummynet machines can be sprinkled around the network. [Riz98]

Network conditions that Dummynet is capable of producing are bounded-sized queues and queuing delay, limited bandwidth, propagation delay, packet losses and the effects caused by multipath packet forwarding. As Dummynet was not initially designed for traffic management, further development was needed to equip the software with all necessary components for traffic management. The original implementation was lacking two important features: packet filtering mechanism and flexible configuration. They were realised with the IP firewall (ipfw) code which was already included in FreeBSD. Due to this design decision Dummynet is configured with extended ipfw commands. [Riz98]

Dummynet is distributed with the FreeBSD operating system and as a stand-alone version. The software integrated into the operating system works as any other program, and it is targeted for FreeBSD-based routers and bridges. The stand-alone Dummynet is packetised together with a minimal version of FreeBSD called picoBSD. The whole packet fits into one bootable floppy disk, and it is intended for workstations which do not primarily implement routing functionalities. [Riz02]

3.2.2 Architecture

The requirements that Dummynet sets to the underlying system are fairly minimal. The stand-alone Dummynet does not even demand a hard disk to be present in the machine. Everything the software needs is in the floppy disk, and after loading the program operates in the main memory. However, the system timer granularity has to be small enough for the experiments. The faster the test network is, the higher the timer resolution should be. The buffer space to store the packets is another noteworthy matter. Dummynet needs the buffering resources from the underlying system, because it does not store the packets in process internally. [Riz97]

The location of Dummynet inside the protocol stack is between the IP and TCP layers. The implementation takes only a few hundred lines of code and is fairly simple. It consists of two queue abstractions and functions needed to move packets between the queues. As Dummynet does not take copies of the packet data, it introduces practically no overhead to the system. [Riz97, Riz98]

3.2.2.1 R-queues and P-queues

The operation of Dummynet is based on a queue abstraction consisting of two queues: R-queue and P-queue. The queues are placed in the middle of the protocol stack, as shown in Figure 3.1, thus all network traffic flowing up and down the stack passes through the queues. The names of the queues come from their functionality. The R-queue is a router queue simulating the impacts of routers in the network. These are the effects of the finite queue size and queuing policy. The P-queue represents the links between the machines and their effects on the packet transfer which are the bandwidth limitations and propagation delay. The letter P in the name of the queue stands for 'a pipe' which is the term used for a communication link in Dummynet. [Riz98]

The effects are generated by keeping the packets in the R- and P-queues for a certain amount of time. The R-queue implements the bounded-sized queue by limiting the accepted amount of packets to some predefined value. The queuing policy of the R-queue is specified by the user. The choices include, for example, FIFO and a WFQ (Weighted Fair Queuing) variant called WF2Q+ implemented especially for Dummynet. The bandwidth limitation is created by moving the packets from the R-queue to the P-queue at a certain packet rate at the maximum. The P-queue simulates the propagation delay simply by keeping the packets queued long enough. When the P-queue releases the packet, it is transmitted to the next protocol stack

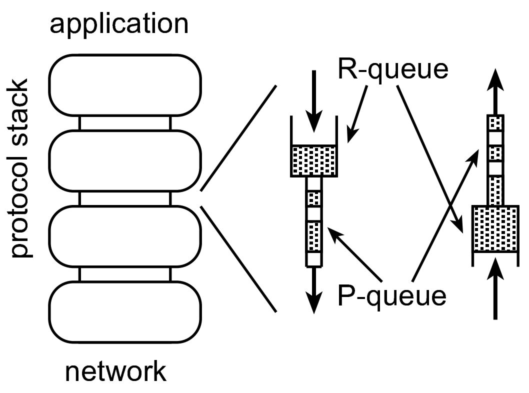


Figure 3.1. *[Riz98] R- and P-queues of Dummynet*

layer. [Riz98, Riz02]

3.2.2.2 Traffic Management with ipfw

Ipfw separates the packets according to their headers. An IP packet header carries many kinds of information and several fields can be used in dividing the packets to different groups. The user can freely define which pieces of information are used and how strict the matching rules are. The possible options for the examined fields include source and destination addresses, source and destination ports and a protocol. [ipf00]

The integration of the ipfw code into Dummynet enabled the configuration of multiple independent pipes. The traffic can be divided with ipfw rules to different groups which have separate Dummynet queues. Thus, not all traffic passing through Dummynet experiences the same kinds of network conditions. In addition, ipfw allows the packets to be examined more than once. Therefore, it is possible to configure a hierarchical structure of pipes where the packets travel through several pipes. [Riz98]

As a default, Dummynet is configured as a router but it can be used also as a bridge. In both cases, the packets are handled by ipfw, which is located above the bridge functionalities in the TCP/IP protocol stack. The bridge handles the packets by using the MAC-addresses which are lost if the packets are passed upwards in the protocol stack. Therefore, the Dummynet bridge caches the MAC-addresses before passing the packets to the next layer upwards and maps the addresses back to correct packets when getting them back from ipfw. The feature has been implemented by modifying drivers for the network interface cards. Because the number of modified drivers is small, the Dummynet bridge supports only a limited set of network interface cards. With a Dummynet router the same problem does not exist, but unlike a bridge a router always stands out in the network topology which is not always desirable.

3.2.3 Configuration

The configuration of Dummynet consists of pipes and queues. The pipes represent communication links that have a certain bandwidth. The queues are packet queues that are connected to one pipe and have a weight attached to them. The weight specifies how big a share of the pipe bandwidth is assigned to the queue. [Riz02]

The user interface of Dummynet is extremely simple. A bare command line is the only way to configure the software. Dummynet does not support a configuration file, which means that all configurations have to be made by typing the commands one after another to the command line.

The configuration commands are either ipfw commands or sysctl variable settings. The ipfw commands are used to configure pipes and queues inside Dummynet. The commands are detailed, and one command contributes to only one pipe or queue. Sysctl variables are the internal variables of the kernel which are numerous and used for various purposes, not only for ipfw or Dummynet controlling. The variables define the environment where Dummynet works, so they set the global configuration which affects on every pipe and queue in Dummynet. [Riz02]

3.3 NIST Net

NIST Net [CS03] is a network emulation tool for Linux targeted for research purposes. It is able to emulate different kinds of network conditions and also record the existing conditions and playback them later. The software is developed in the United States at the National Institute of Standards and Technology (NIST) as a part of a research program. The first initial beta release in the NIST Net project was in 1998, and the current version of NIST Net was released in June 2002 under the version number 2.0.12. [CS03, oSN03]

3.3.1 Features

Several network emulators were developed already before NIST Net. However, they lacked support for high data rates and their capabilities were restricted. The design goal of NIST Net was to create a software tool capable of emulating various network conditions in a sophisticated manner. Packet loss, duplication and reordering, delay with adjustable distribution and limited bandwidth were the most important implemented features. In addition, NIST Net was designed to allow the user to define his own packet handlers which can be used for various purposes including data capturing and performance monitoring. The implementation of NIST Net is loosely based on Hitbox network emulator [ADLY95] and MOST radio network emulator [DBCF95]. [oSN03, CS03]

3.3.2 Operating Requirements

NIST Net runs in a Linux machine configured as a router. The software observes the traffic passing through the router and applies the user-defined actions to the selected flows. NIST Net is implemented as an extension to the Linux kernel, hence it is not a stand-alone system like Dummynet which is distributed jointly with an operating system. Most Linux distributions with new enough kernel work fine with NIST Net. Only Red Hat has been reported to cause some troubles in installation, but they can be avoided by following special installation instructions. [CS03, oSN03]

The hardware requirements of NIST Net are quite moderate. The critical characteristics of the underlying system are the amount of kernel memory and processing power. If the system properties are sufficient, the software is able to run without causing a distracting processing overhead. [oSN03]

3.3.3 Usage

Before NIST Net is ready to be used, it has to be loaded into the kernel memory. The loading is done with a special command or using a general *insmod* command. The operation can also be automated by adding the software to the list of modules that are loaded in the system boot. [oSN03]

NIST Net provides three types of user interfaces: two command line interfaces, one graphical user interfaces and an API for programmers. The command line interfaces are called Cnistnet and Hitbox. Hitbox is the original and thus older interface, whose set of commands is restricted. In addition, it does not automatically provide the user any information about the system state. Cnistnet is the new command line interface, and it implements a wider range of commands than Hitbox. With Cnistnet the user is able to use all features of NIST Net and define the software behaviour accurately. The command line interfaces suit well for complex, non-interactive testing which is performed through scripting. [oSN03]

The second way to use the software is via APIs. NIST Net contains a set of APIs for scripting and programming. For non-interactive testing the APIs can be used instead of the command line interfaces. [oSN03]

Probably the most popular user interface is the graphical user interface called Xnistnet. The GUI looks like a spreadsheet which contains the same parameters that can be set with the command line commands. The user fills in the appropriate cells and launches the software. Xnistnet is an interactive user interface, thus it shows some bandwidth and queue statistics in real-time. The GUI suits mainly for simple testing. More complex testing can be executed by running scripts either in a command line interface or via APIs. [oSN03]

The configuration of NIST Net consists of a set of rules, which are used to divide packets into different action groups. Rules include the identification part and the action part. The identification part lists some of the packet header values: source and destination address, source and destination ports and used protocol. Not all of them needs to be defined, as they can also be marked as default, meaning that they match to every packet. The action part defines the treatment of the matching packet. The possible actions are dropping or duplicating packets, increasing delay, limiting bandwidth and applying a RED-variant DRD (Derivative Random Drop) queuing policy [Gay96]. All of the actions have adjustable parameters. [oSN03]

3.3.4 Implementation

The NIST Net implementation includes the actual program, three user interfaces and an API for the programmers. The program itself is a kernel module extension which works on Linux. NIST Net maintains a table of emulator entries based on the configuration. The entries consist of the packet matching rules, actions to be taken and statistics of the matching packets. [oSN03]

NIST Net is constructed of two parts: a kernel module and a set of user interfaces. The structure of the NIST Net architecture is illustrated in Figure 3.2. The NIST Net kernel module captures the packets entering the IP layer in the Linux kernel and compares the packet header values to the packet rules in the emulator entries. If the packet matches some entry, the actions defined in the entry are performed on the packet. In case of dropping, the packet is simply discarded. If the packet is duplicated, two copies of the packet are passed back to the Linux kernel IP layer. A delayed packet is returned to the IP layer after a certain amount of time. The fast timer uses the kernel real time clock as the time source when scheduling the delayed packets back to the IP layer. NIST Net is controlled from the user interface which operates the kernel module through the control APIs. [CS03]

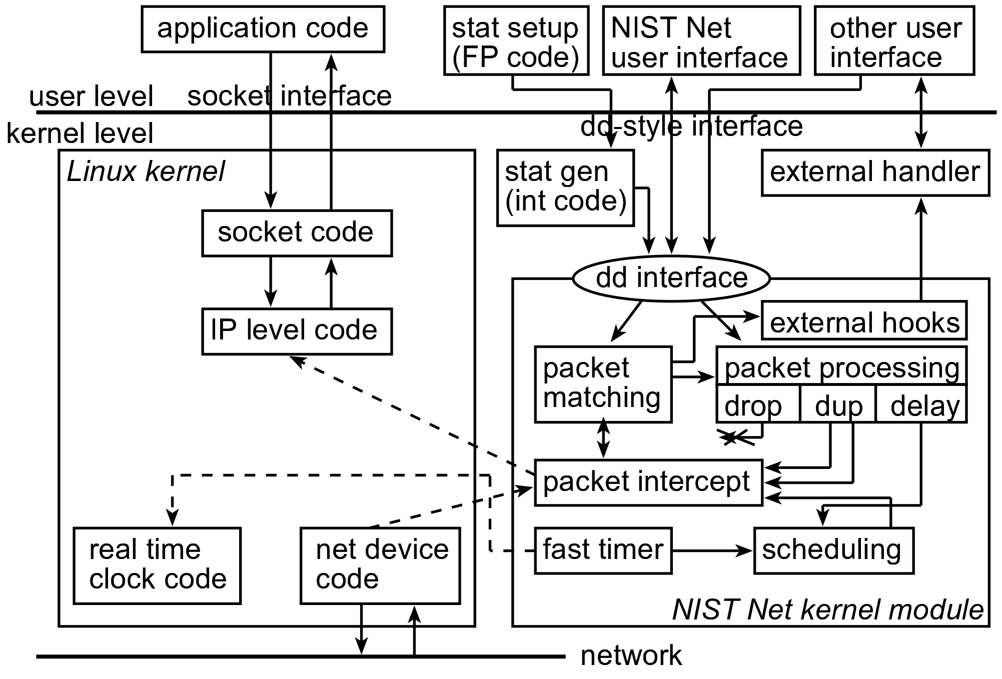


Figure 3.2. *[CS03] Architecture of NIST Net*

The external hooks in the NIST Net kernel module enable the existence of the external handlers for the NIST Net program code. Handlers can be used in applying external statistics when creating the emulator entries. An external handler can cooperate with NIST Net or take care of the packet processing entirely, bypassing the NIST Net code. The handlers have been implemented for flow monitoring and Voice over IP testing, for example. [CS03]

3.4 Click

One fundamental problem with most routers is the inflexible architecture. Routers allow some functionalities to be turned on and off but more detailed subtractions or extensions are hard to make because of the closed design. To study if a router could be made flexible with a fully modular design and easy configuration, a software called Click [KMC$^+$00] was invented in MIT in 1999. Click increases the flexibility of router implementation by allowing the router forwarding path to be built from individual modules. The modules can be connected together in several ways like the toy building blocks, and the final result depends greatly on the builder. If the existing modules do not implement some required feature, it is possible to write a new module. In addition to the basic modules, Click has many extension modules which include DiffServ support, for example. The current version of Click is 1.3pre1 and it was released on March 2003. [KMC$^+$00]

3.4.1 Architecture

Each Click module, called an element, represents one small part of the router functionality. Click handles the configuration as a directed graph, where the elements are vertices and connections between the elements are edges. Packets move along the edges, so every connection represents a possible transfer path for a packet. [KMC$^+$00]

3.4.1.1 Elements

Elements are the building blocks of the Click router. All router functionalities have been divided into separate pieces which can be connected together to form a router forwarding path. The input and output interfaces of the elements are strictly defined which allows a free combination of the elements, as long as the connected interfaces are of the same type. On the other hand, strict definitions prevent illegal combinations that would not work in practice. [KMC$^+$00]

Functionally the elements may be very small. For example, a queue is one separate element - no other element implements it. Division of the router functionality into small enough pieces which are still of a reasonable size increases the flexibility of the architecture. All actions are taken inside the elements, nothing happens outside of them. [KMC$^+$00]

There are four important properties on every element: class, ports, configuration string and method interfaces. The class specifies the code to be used in packet processing and some other element characteristics. Every element belongs to exactly one element class. The input and output interfaces of the element consist of ports which are defined in ports property. The optional configuration string consists of extended attributes for the element. Method interfaces are the interfaces that the elements use to communicate with each other. All elements support at least the packet transfer interface, which is the simplest interface, but also other interfaces can be defined. Figure 3.3 illustrates a example of an element and presents some of the properties. [KMC+00]

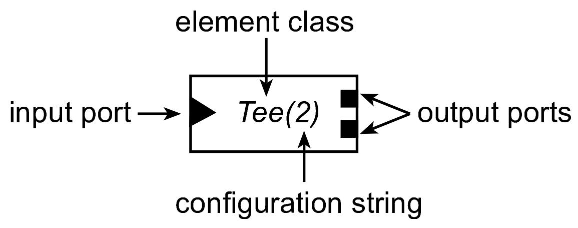


Figure 3.3. *[KMC+ 00] An example of a Click element*

The element is not only a functional component but also a basic unit for scheduling. When the CPU time is assigned to different tasks, the time slots are divided to elements. Click implements a task queue which contains the elements that wait for CPU processing. One by one the elements get their share on FCFS basis. There are still only few elements that end up to the queue, because most elements get the processing time through their push and pull functions. [KMC+00]

3.4.1.2 Connections

Connections are created between the elements and they are the possible transfer paths for the packets. There are two types of connections: push and pull. In a push connection, the source element passes the packet to the destination element which has no idea that a packet is coming until it arrives to the input interface. In the pull connection, the destination element requests the packet from the source element which then passes the packet. Push connections are used when the element on the other end is allowed to send a packet whenever it suits to it, and the destination element is prepared to store the packet if it is not able to handle it right away. Pull connections are practical, when the destination element needs to control the timing of the packet arrivals. [KMC+00]

All ports of the element are either push or pull ports. This creates the rule of connecting two elements: only two push ports or two pull ports can be connected together. A push port cannot be connected to a pull port, or vice versa. There are also so called agnostic ports that can be either push or pull ports, depending on to which type of port they are connected to. Every port can be connected to exactly one port, so there is no danger of one port having both push and pull connections. [KMC+00]

3.4.2 Inside the Kernel

Click implementation is an extension to the Linux kernel. The scheduling of the elements is taken care of by the router driver which is controlled by the Click kernel thread. There can be several router driver threads running at the same time but they can be active only one by one. Interrupts are allowed to preempt the thread, but the driver also releases the CPU voluntarily every now and then to keep the system running. [KMC+00]

There are four main object categories in a runnig Click system: elements, packets, timers and a router. Click has element objects for all elements that appear in the system - either in the current configuration or among the elements that could be used. The objects for the latter type of elements are called prototype objects. The packet objects are for the packets that are stored in the memory, and in the kernel the packet objects are similar to the packet abstraction used by Linux. The timers in Click use Linux timer queues and their abstractions are timer objects. The router object takes care of the router configuration in many levels. It collects the information for the current configuration, configures elements, checks the connections and puts the router on line. It also has a responsibility of handling the task queue which is used in scheduling the elements. [KMC+00]

3.4.3 Configuration

The configuration of Click consists of simple commands written in Click's own programming language. The language has two important concepts: declaration and connection. The declaration creates the elements, and the connection describes how the elements should be connected together. In addition, there is an abstraction called a compound element which consists of the user-defined, consecutive elements which are handled together like they were a single element. The language is used only in the descriptive configuration. The information on how the packet processing is performed in practice is stored in another way. [KMC$^+$00]

3.4.3.1 Installation and Changing Configuration

The kernel driver takes care of the installing of the configuration. It reads the configuration file written in Click language, parses it, checks if it has any errors, does the initialisation of the elements and puts the router on line. Normally the system can store only one configuration at a time which causes losing of information when the configuration is changed on-the-fly. For example, all packets that are in queues are dropped, because the new configuration starts from a clean system state. [KMC$^+$00]

There are two ways to change from one configuration to another without losing information. If the modifications are made locally inside one element, the element specific handlers can be used to change the configuration. If the modifications are too complex for the handlers, the whole configuration file can be changed using hot swapping. The new configuration file is accepted only if it does not contain any errors, otherwise the old configuration remains effective. When the new configuration has been accepted, it adopts the system state which the old configuration left behind. Therefore, no packets or any other information are lost. [KMC$^+$00]

3.4.3.2 Extensions

Click implements several extensions, including different scheduling and dropping policies, queue management, IP header compression and decompression and security features like firewalling and NAT (Network Address Translation). The DiffServ support in Click is implemented by taking all DiffServ components defined in [BBC$^+$98] and transforming them into Click elements. The system administrators can combine the elements together to form a tailored DiffServ implementation. [KMC$^+$00]

3.5 XORP

Extensible Open Router Platform (XORP) [HHK03] is another router software developed to meet the needs of an open router architecture. The main design goals of XORP were extensibility, performance and robustness. XORP bases heavily on Click implementation, as its forwarding path was constructed of Click elements. The software was targeted for research purposes in both laboratory and production networks which naturally set high requirements for the program stability and development. The home of XORP is at the International Computer Science Institute in Berkeley, California. The current version of XORP is 0.3 alpha, released in June 2003. [HHK03]

3.5.1 Design Goals

The primary goal of XORP is to work as a bridge between the protocol and mechanism testing in the laboratory environment and in the real production networks. New ideas developed in the laboratories easily remain inside the laboratory walls because introducing the new protocols and new mechanisms in the real world is extremely difficult. The commercial routers have not been designed for research purposes, so their architecture is closed and static. The open source routers, on the other hand, tend to be too experimental for production networks. XORP aims to be flexible, open source router software suitable for research and despite of its experimental nature, stable enough to be connected to a production network. Thus, with XORP the researchers are able to do the main development work and testing in the laboratories and the final testing in a real network on the same platform. [HHK03]

Originally XORP was targeted for network edge routers, whose capacity does not normally differ much from a conventional PC. In addition, on the edges of the network, the routing tables are relatively small and the amount of interfaces is reasonable. Since the early days, XORP has been deployed also in the network core. The current objective of XORP is to become a software that could be used as a basis of all kinds of routers. [HHK03]

3.5.1.1 Router Features

In order to meet the requirements which are set for a general core of various types of routers, XORP needs to implement a great amount of features. The software is not ready yet, since the implementation is only half-way, but there is an extensive list of existing and forthcoming features. The list includes different routing protocols, network management tools and forwarding path support. The current version supports routing protocols like BGP4, OSPF, PIM-SM, IGMPv1 and IGMPv2, and in the future more unicast and multicast protocols will be implemented. All protocols are at the moment tested only with IPv4, but XORP has been designed to work also with IPv6. Network management is currently handled via a command line interface, but there are plans for SNMP and WWW interfaces. As a forwarding path, XORP can use a common Unix forwarding path or a Click router forwarding path. Adding a support for alternative forwarding paths is in progress. [XOR03]

3.5.1.2 Extensibility

For a router that is used in research, extensibility is a key issue. Every structural level of a router from packet forwarding to protocols needs to be extensible. Sufficient extensibility is achieved by taking care that there are well-defined interfaces in all points where the extensions could be attached to and by implementing easy-to-use APIs to all interfaces. The router should not limit the amount of coexistent extensions as long as there are no interference between them. [HHK03]

3.5.1.3 Performance

Forwarding performance is an essential aspect for any router, no matter whereabouts in the network it is located. For the forwarding path which has to handle every packet that enters the system the performance issue is critical, while the upper layers of the router prioritise other matters. In XORP, the sufficient performance level is achieved with the Click router software. XORP can use a forwarding path constructed of Click elements which provide good performance. In addition, Click can increase the flexibility, configuration abilities and hardware support of a XORP router. [HHK03]

3.5.1.4 Robustness

XORP is designed so that if a routing process crashes, other processes will not be affected. The routing and coordinating processes run in the user space of the Unix system and they are all protected from each other. If a crash happens, the system cleans up after the dead process, so that there are no harmful traces left to distract other processes. The robustness is an important matter also in the forwarding path, but there it is not possible to use the same memory-based protection methods as in the user space. However, if the forwarding path is constructed of the Click elements, the necessary robustness is achieved. [HHK03]

3.5.2 Architecture

Internally the XORP architecture is modular. The system is divided into two subsystems, upper and lower one, of which the upper subsystem is split further into several modules consisting of various functional blocks. The upper level subsystem is also called a user level subsystem, as it runs in the user space of Unix. All routing protocols and management functionalities are located in the user level subsystem. The architecture of the user level XORP is a multi-process architecture, where every routing protocol has one process. [HHK03]

3.5.2.1 XRLs

Messages between the XORP processes are carried by XORP Resource Locators (XRLs). The format of XRLs is similar to URLs: in the beginning there is the protocol which is the type of IPC (Inter-Process Communication) transport, and after that there are the name of the communication party, the name of the used method and a list of arguments. Furthermore, it is possible to define the desired type of the response. [HHK03]

The original idea of XRLs was to hide the protocols inside the IPC framework. The aim was to enable the addition of new protocols to the XORP system simply by writing the necessary elements to the XRL client library. The library is an essential component for the operation of XRLs. When the process sends an XRL message, it is caught by the XRL library that is linked to all XORP processes. The library examines the XRL parameters and invokes the required IPC transport mechanism. The library also takes care of the response or a possible error condition. [HHK03]

3.5.2.2 Process Model

The subsystems consist of various processes as can be seen in Figure 3.4. The lowest level is the kernel subsystem which consists of the forwarding engine. The forwarding path is built on the Click elements. [HHK03]

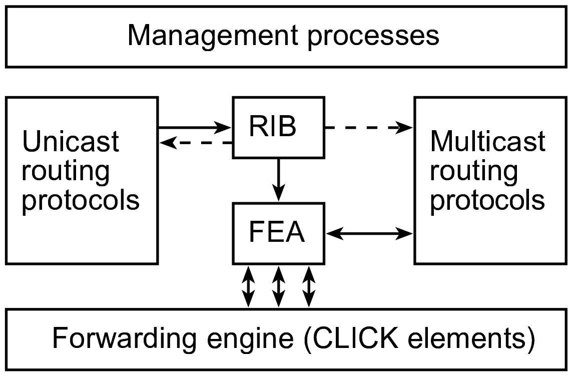

Management processes
Unicast routing protocols
RIB
Multicast routing protocols
FEA
Forwarding engine (CLICK elements)

Figure 3.4. *XORP process model (simplified from [HHK03])*

The Forwarding Engine Abstraction (FEA) manages the forwarding path and hides all its details from the processes above. The FEA provides a common interface for the routing processes to the forwarding path and keeps the router forwarding table up to date. FEA also takes care of the networking interfaces by managing them and passing information on their state and actions to the routing processes. Usually all information crossing the user and kernel subsystem boundary goes through the FEA, but in some special cases the processes are allowed to bypass the FEA. [HHK03]

The current version of XORP implements both unicast and multicast routing protocols. Unicast routing processes use the Routing Information Base (RIB) to propagate the calculated routes. The RIB decides whether the route should be passed to the forwarding path or to the other routing processes. The RIB keeps track on who has sent which route, and with this information it maintains a copy of the forwarding table. Multicast routing protocols need the RIB only occasionally, because the protocols manage the routing tables themselves. The RIB is provided only the information which routes are capable of multicasting. In Figure 3.4 the dashed arrows represent the communication which does not take place with all protocols in the routing protocol block. [HHK03]

The management processes block contains all management functionality of the router. XORP router manager process, called rtrmgr, is responsible for the router on the whole. It monitors the processes, starts new and restarts the failed ones, all according to the configuration. It keeps the router components running and configures them. The command line interface provides a user access to the router for configuration and information collecting purposes. The IPC finder is needed by the XRLs in communication between the processes. All XORP processes need to inform the finder where they are located, so that the finder is always aware of which processes exist and where. If the finder notices that the information of a process is not up-to-date, it can advice the process to send the valid contact information. When a process wants to communicate with another process, it does it via an XRL. The XRL library captures the message and consults the finder in order to get the name of the actual communication protocol. When the library has received the information, it is able to take care of the rest of the process communication. [HHK03]

3.6 Linux Traffic Control

Linux operating systems include a built-in traffic control environment [Alm99] which provides a great variety of mechanisms for traffic management. The environment consists of the kernel parts which contain the actual traffic control functions, and the user-space applications which are used in commanding and configuring the kernel elements. Linux traffic control implements the most common queuing and policing mechanisms as well as the necessary components to support IntServ and DiffServ architectures.

Traffic control has been included in the Linux distribution since the kernel version 2.2. The improved implementation of traffic control called Linux traffic control next generation (tcng) [Alm02] has advanced to the version tcng-9f which was released in June 2003. It has not yet been integrated to the Linux distribution but it is available via the tcng home page in the Internet [tcn03].

3.6.1 Architecture

The architecture of Linux traffic control has a layered structure as shown in Figure 3.5. The traffic control subsystem resides in the kernel space, while the rest of the components are located in the user space. Traffic control utility (tc utility) is situated in the user space between the kernel traffic control block and the user applications. [Alm99]

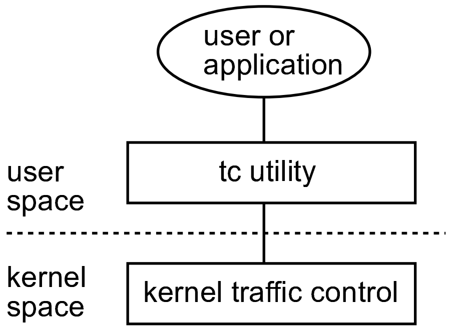


Figure 3.5. *[Alm02] Architecture of Linux traffic control*

In Linux traffic control implementation, the classification is taken care by several distinct components, while the queue management and scheduling are performed by a single component called a queuing discipline (qdisc). The qdiscs together with the class, filter and policing components are located in the kernel traffic control block. The relationship between the qdiscs, classes and filters is a hierarchical structure, where one qdisc is a root and classes are its children. Filters can be attached either to the classes or to the qdiscs. More complex configurations, where the classes have their own inner qdiscs and other classes as children, are also allowed. The fourth component, policing, is a functional component, not an element as the three others can be seen. Policing can be performed in different ways and by several elements. [Alm99]

3.6.2 Queuing Disciplines

The queuing discipline can be either classful or classless. A classful qdisc can contain classes which are allowed to contain further classes and qdiscs. It is also possible to attach filters to the classes and qdiscs. A classless qdiscs operates always without classes and filters. Linux traffic control implements several classless and classful qdiscs, including tail-drop FIFO, RED FIFO, Priority Queuing and an improved CBQ called Hierarchical Token Bucket (HTB) [Dev03]. [tc01]

Qdiscs implement functions for packet enqueuing, dequeuing and dropping, and for qdisc configuration initialisation, changing and destroying. Collecting statistics is obligatory for the qdiscs, and the required information includes the cumulative number of bytes enqueued, packets enqueued and packets dropped, as well as the present queue length. Each instance of a queuing discipline has a unique name, a 32 bits long two-piece number, on every interface. The number is of format *major:minor*, where each part is 16 bits long and the minor number has always the value zero. The major number is called a handle and it is needed if the qdisc has child classes. [Alm99]

3.6.2.1 Classes

A classful qdisc uses classes to create different traffic profiles. The classes do not actually handle the packets, as they only provide the conditions for traffic differentiation. The class functions include operations like changing and deleting of the classes, changing of the qdisc and information gathering about the current configuration. [Alm99, tc01]

Classes have two names: a user-defined class ID and an internal ID which is given by the parent qdisc. Both names are unique inside the queuing discipline. The class ID has the same structure as a qdisc ID, *major:minor*. The major number is the number of the parent qdisc and the minor number identifies the class inside the qdisc. The internal ID is of a different type than the class ID, and it can be a pointer or an index, for example. The internal ID is the name that the kernel normally uses when referring to the class. In special cases, different classes that obviously have different class IDs, may have the same internal ID. [Alm99]

3.6.2.2 Filters

Filters collaborate with classes and define which packet belongs to which class. Although filters are tied to classes, depending on the implementation, they can be attached either to classes or to qdiscs. The possible types of the filters are generic and specific. A generic filter can divide the packets into different classes alone, no matter how many classes there are. Under a single qdisc there is a need for only one generic filter. A specific filter can capture only one type of packets, thus if specific filters are used, every class needs at least one filter to collect the packets. The functions to control filters include filter initialisation, changing and deleting, as well as printing statistics and parameters of the filter. [Alm99]

In the system configuration, the filters are stored in the class specific or qdisc specific list. The order of the filters is determined by the filter priorities which are arranged in ascending order. Therefore, if there are two or more filters for the same protocol, they need to have different priorities. Filters do not have filter IDs in the same way as classes do. Instead, they contain a list of arguments that specifies the filter. The arguments are the name of the class or qdisc which the filter is attached to, the protocol and the priority in the filter list. Internally filters have elements which are controlled through their identification names called handles. If the element needs to be identified outside the filter, it can be done with the filter argument list combined to the handle. [Alm99]

3.6.2.3 Policing

Linux traffic control is able to perform policing in five different ways along the forwarding path. The policing decision can be made by filters which pass the information to the queuing discipline. The qdisc has then the final responsibility to decide whether to take the action or not. Policing can take place also in the ingress interface, where the nonconforming packets are dropped before they have used any more resources. The filters are used to select the packets to be dropped. Discarding a packet in the enqueuing operation is another alternative, and it takes place if the qdisc is not for some reason able to enqueue the packet. Packet can also be dropped inside a nested qdisc if there is a need to make room for a more important packet. For the same reason a packet which has already been successfully enqueued may be discarded from the queue. [Alm99]

3.6.3 Traffic Control Usage

Linux traffic control is used via command line commands. The user application is called tc, and it is invoked with the command 'tc'. The application commands can concern about a queuing discipline, a class or a filter. The options inside the tc commands are add, remove, change, replace and link. All of them are common to qdiscs, classes and filters, except 'link' which is used only for qdiscs. [tc01]

3.6.4 Linux Traffic Control Next Generation - tcng

Although the Linux traffic control architecture is not very old, it has certain weaknesses especially with usability, extensibility and performance. In 2001, a project called Traffic Control Next Generation was launched in Swiss Federal Institute of

Technology and currently the project continues in the Internet as a voluntary work. The aim of the project is to tackle the problems discovered in Linux traffic control and implement an improved traffic control which is called tcng after the project name. [Alm02]

3.6.4.1 New Architecture

The old traffic control has a three layered structure, as shown in Figure 3.5. The new traffic control, whose architecture is illustrated in Figure 3.6, adds one layer between the tc utility and the user. The layer is a traffic control compiler called tcc, whose aim is to make the configuration of tcng user-friendly. It takes scripts written in the tcng language as an input and translates them into the tc language commands which are brought to the underlying tc layer. The tcng language resembles Perl and C which makes the language more understandable and more flexible than the old tc command language. [Alm02]

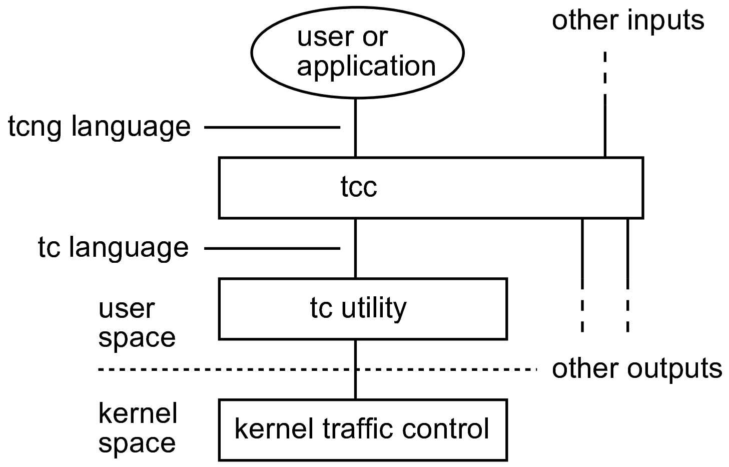


Figure 3.6. *[Alm02] Architecture of Linux tcng*

To avoid faulty or illegal commands to reach the tc level, the scripts are automatically checked in the tcng level before the translation. The checking is done by simulating the effects of the configuration in the user space with a simulator called tcsim. The simulator copies the tc layer and kernel traffic control subsystem codes, adds some simulator code around them and thereby creates an authentic testing environment. As the output, the simulator creates a trace of all events. [Alm02]

Tcng enables the replacing of the kernel traffic control block with a third-party software. The usage of hardware accelerators is also possible. Tcng translates the output from the user space system suitable for the alternative traffic control module in the kernel. Hence, if there is a need to make modifications to the input configuration and scripts because of the new kernel module, the modifications are likely to be small. [Alm02]

Chapter 4

Alternate Queuing Framework

4.1 Introduction

Alternate Queuing (ALTQ) [Cho01] is a traffic management software for BSD Unix operating systems. It has a similar operation philosophy to Linux Traffic Control, although ALTQ is more versatile than Linux TC. ALTQ implements several mechanisms for controlling traffic, including various queuing disciplines and RSVP and DiffServ support, as well as a number of tools for gathering statistics and monitoring the traffic. It offers a modular and extendable platform for researchers studying the QoS mechanisms and related issues. This chapter introduces ALTQ, its architecture, design goals and capabilities. The aim is to give the basic knowledge of the software which has been used as a basis for the Policy Control Agent implementation presented in the next chapter.

4.2 ALTQ History and Current Status

The history of ALTQ dates back to year 1997 when the designer of the software, Kenjiro Cho from Sony Computer Science Laboratory in Japan, made the first version of ALTQ publicly available. A year later a group of Japanese companies launched the KAME project, which studies IPv6, IPSec and state-of-the-art networking issues in BSD environment, and the development of ALTQ was transferred under the project. In addition to the KAME integrated component, ALTQ was released as a standalone software. The latest version is 3.1 and it was published in 2002. The software is freely distributable and available in the Internet. [Cho98, kam04, Cho03]

The current version of ALTQ was built on FreeBSD version 4.5. In the newer FreeBSD versions, ALTQ requires some kernel patches in order to work properly. NetBSD and OpenBSD already have ALTQ integrated in the operating system. Since its first release ALTQ has been actively deployed into several research projects and at present it is used in various projects worldwide. [Cho03]

4.3 Design goals

The name of ALTQ, Alternate Queuing, indicates the main purpose of the software. The BSD operating system does not implement any other queuing disciplines than a FIFO queue, but this bias is hard to go around, because there is no way to add new queuing disciplines to the system. ALTQ does not only provide a number of ready-made alternative queuing disciplines but also a method to introduce new ones. The feature is valuable for the researchers, who need the research platform to be as flexible and extensible as possible. Another important element is the versatility of the software. ALTQ has been designed to support a variety of queuing disciplines and other QoS mechanisms, and the list of supported functions has been improved for every release. [Cho01]

Attempts have been made to keep the implementation of ALTQ simple enough. Simplicity, stability and robustness are key factors for the systems connected to production networks, and a simple design helps to achieve the required stability and robustness. Production networks have been taken into account in the design of ALTQ, because the software is intended for research use both in the laboratory environment and in the operational networks. [Cho98]

4.4 Features

ALTQ provides a framework for different QoS functionalities. The emphasis is on the queuing disciplines, but ALTQ supports also some other traffic engineering mechanisms. If the set of queuing discipline algorithms does not include the required one, ALTQ allows the addition of the new queuing disciplines. The programmers are encouraged to use the FIFOQ template as a basis for their own queuing discipline implementations. Changes can be made practically to any part of the system, so the researchers are able to adapt the ALTQ system suitable for their experiments. [Cho01]

The current version of ALTQ implements six queuing disciplines including FIFOQ, PRIQ, CBQ, WFQ, HFSC and JoBS, and three queue management algorithms including RED, RIO and Blue [Cho03]. ECN (Explicit Congestion Notification) mechanism is supported experimentally.

FIFOQ (First In First Out queuing) is the simplest queuing method in ALTQ consisting of a FIFO queue and tail-drop queue management. It is aimed mainly for template use when implementing new queuing disciplines. [Cho01] PRIQ (Priority queuing) adds 16 priority levels to FIFO queuing. The idea is that the higher priority class gets always the service first. [alt99a] CBQ (Class Based Queuing) [FJ95] provides partitioning and link capacity sharing among traffic aggregates based on a hierarchical class tree. Each class is assigned an own queue, and child classes are allowed to borrow the excessive capacity of their parents. [Cho02] WFQ (Weighted Fair Queuing) [DKS89] is an approximation of a theoretical model of an ideal scheduling mechanism, where the packets are prioritised according to the earliest transmission finishing time. Several variations of WFQ exist, and in ALTQ, WFQ implements a Weighted Round-Robin scheduler and a set of queues, each with a certain weight indicating its share of the link capacity. [Cho01] HFSC (Hierarchical Fair Service Curve) [SHN00] is designed to support both link-sharing and real-time services. It bases its operation on two so called service curves for the real-time and link-sharing criteria, and implements independent scheduling mechanisms for both types of service. [Cho02] JoBS (Joint Buffer Management and Scheduling) [LC00] can provide hop-by-hop service guarantees for traffic aggregates. The guarantees are realised with bandwidth allocation, and their type can be either absolute or proportional. [Cho02]

RED (Random Early Detection) [FJ93] is an active queue management mechanism which implements congestion notification. RED observes the average queue length and starts dropping packets with an increasing probability after a certain threshold. In ALTQ, RED is used in the experimental implementation of another congestion notification method, ECN (Explicit Congestion Notification). An enhanced version of RED is RIO (RED In/Out) [CF98] which implements several parallel RED queues. In ALTQ, RIO is implemented according to the Assured Forwarding definition [HBWW99] with three precedence levels. [alt99a] Blue [FKSK02] is an active queue management algorithm using packet loss and link utilisation as the basis of congestion handling.

The traffic conditioning elements provide the functionalities required by Diff-Serv in the input interface of the router. The elements consist of simple elements that are built for only one action and more complex elements that are able to choose from several actions according to certain conditions. Classifiers and meters belong to the latter group. ALTQ implements three complex conditioning element: a token bucket meter [BBGS02], two-rate three-colour marker (trTCM) [HG99] and time-sliding window three-colour marker (TSWTCM) [FSN00] which were all introduced in Chapter 2. The simple elements perform the same action to every incoming packet which can be passing the packet, marking the packet with a certain DSCP value or dropping the packet. Shapers which also belong to the DiffServ conditioning elements are not currently supported in ALTQ. If shapers are needed, they can be simulated by using the queuing disciplines. [Cho01]

The BSD operating systems do not implement a traffic control module, which is a necessity for the RSVP daemon program (rsvpd). ALTQ has included a traffic control module since the first version and thus opened up a possibility to use RSVP in the BSD platform. The lack of the traffic control module in BSD was considered so critical that integrating such a module into ALTQ was one of the original design goals. [Cho01]

4.5 ALTQ Tools and Applications

The QoS mechanism implementations reside in the kernel space of the operating system. Above them in the user space there are many kinds of tools for configuring and monitoring the underlying mechanisms. In addition to the stand-alone programs, ALTQ provides an application programming interface (API) as a part of the software architecture.

4.5.1 ALTQ Daemon - altqd

The ALTQ management tool called altqd is used for configuring the QoS mechanisms. It is a stand-alone program which enables the ALTQ functions in the kernel and sets the parameters according to the given commands. Altqd reads the primal setup from a configuration file and initialises the system. Afterwards, the configuration can be modified through a command line interface, if ALTQ is in the interactive mode, or via ALTQ API commands from an external program. [alt99b]

Altqd is responsible for taking care of the QoS components in the kernel space by enabling, configuring, reconfiguring and disabling them. When altqd is launched, it wakes up the QoS routines and turns the router into an ALTQ router, and when it is shutdown, it stops the routines and the router returns to the normal state. As default, altqd detaches after initialisation and runs as a daemon with the same initial configuration. If altqd is invoked to the debug mode, it does not detach but opens a command line which can be used for giving management commands and thereby changing the configuration on-the-fly. In addition, in both modes altqd serves the monitoring tools by providing them information about the state and functions of the QoS components. [alt99b, Cho01]

4.5.2 Statistics and Monitoring

Providing information on the ALTQ system and component states and functioning is essential for debugging and program development. There is a common monitoring tool called altqstat, which displays real-time statistics of queuing discipline and traffic conditioning elements. Discipline specific monitoring tools are also available, and they can be used as stand-alone programs as well, although their main purpose is to gather information for altqstat. In addition, altqstat is able to display the current component structure in the ALTQ configuration, according to the information provided by altqd. [Cho01]

4.5.3 Application Programming Interface

The libaltq library which plays the main role in the ALTQ management architecture is built from varying components located in three layers, as shown in Figure 4.1. The upmost layer includes the parser and the command interpreter module which reads and interprets the configuration file and the commands given through the command line. The middle layer consists of the Queue Command (QCMD) API and the lowest layer of the Queue Operation (QOP) API. QCMD and QOP layers are divided vertically into a common module and a discipline specific module. Below the layers, there are the system call interface and functions which handle the communication with the kernel. [Cho01]

The QCMD API is principally meant to be used by the parser, but also programs that do not need detailed control over the QoS components are capable of using

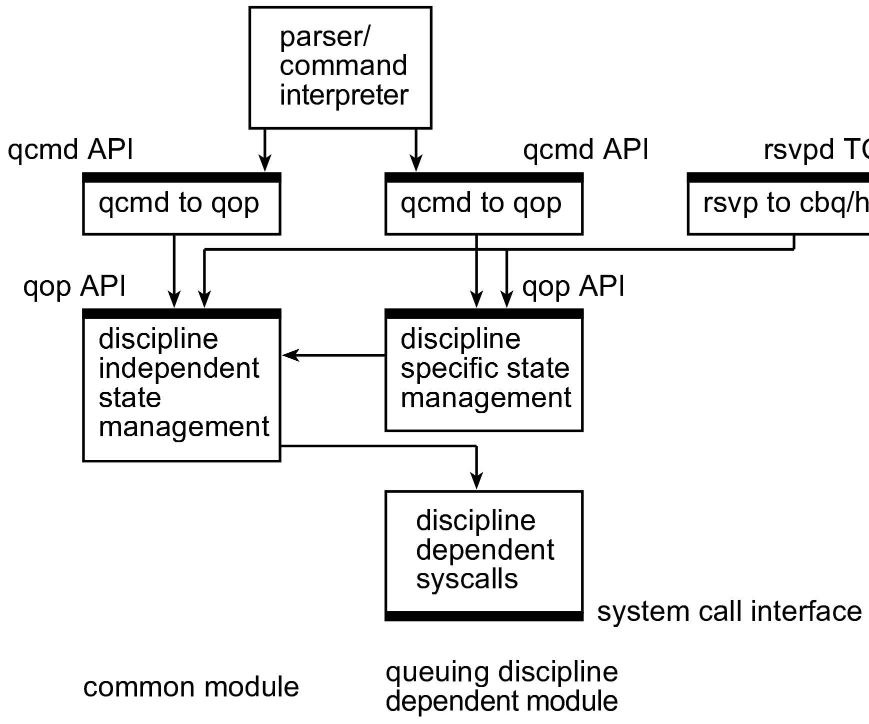

parser/ command interpreter
qcmd API
qcmd to qop
qcmd API
qcmd to qop
rsvpd TCif
rsvp to cbq/hfsc
qop API
qop API
discipline independent state management
discipline specific state management
discipline dependent syscalls
system call interface
common module
queuing discipline dependent module

Figure 4.1. *[Cho01] The libaltq library*

it. [Cho01] The actual API is nevertheless the QOP API which contains more complete functions to control the behaviour of the software. The QCMD API is actually a simplified QOP API, since the QCMD functions only provide a different interface to the QOP functions. The simplicity of the QCMD functions is based on the required parameters, which are mainly strings, whereas the QOP functions include strings, pointers and various structures as parameters.

The QOP API provides for both the ALTQ package programs and the user-written programs a common interface to the functions controlling the QoS components. The API also makes it possible to use a single configuration file for all ALTQ configurations. The QOP functions, and therefore also the QCMD functions, belong either to the common module or to some discipline specific module. The common module consists of functions that are common to all disciplines and can be used with any of them. Enabling and disabling ALTQ are examples of such functions. The discipline specific functions require parameters that are unique to a particular discipline. Thus, the functions can be used only with the discipline assigned to them. [Cho01]

In the QCMD layer, next to the QCMD common and discipline specific modules, there is a place for additional APIs. Currently there are two of them, rsvpd Traffic Control interface (TCif) and DiffServ Policy Enforcement Point (PEP). Both work as interpreters by translating the incoming parameters into the discipline specific parameters. [Cho01]

4.6 ALTQ Architecture

The architecture of ALTQ can be presented with two nested models. The topmost of them is the abstract model of ALTQ functionalities and their location in the traffic forwarding path inside the router. Below the abstraction there is the actual implementation which consists of several components attached to the operating system. Both the traffic control model and the implementation model explain the ALTQ architecture from their own point of view, completing each other.

4.6.1 ALTQ Traffic Control Model

The main parts of the ALTQ architecture are the framework, forwarding mechanisms and management mechanisms. The framework draws the outlines of the system and creates the interfaces towards the operating system. The framework does not know anything about the QoS operations, it just implements abstractions of the available QoS mechanisms and offers the operating system a common interface to all of them. [Cho01]

The forwarding mechanisms are the heart of the ALTQ system, for they implement the actual QoS functionalities. The mechanisms are separated into two groups, one consisting of queuing elements and the other consisting of traffic conditioning elements. Queuing elements include queuing disciplines, whereas traffic conditioning elements include classifiers and markers, for example. [Cho01]

The forwarding mechanisms are small and simple and they reside in the kernel, as they need to be efficient. The management mechanisms, on the contrary, are not so small and definitely not simple, and they operate in the user space. Their task is to control and manage the forwarding mechanisms. The management mechanisms consist of different types of tools and programs for configuring and monitoring the traffic management, like the ALTQ daemon altqd and the altqstat program. [Cho01]

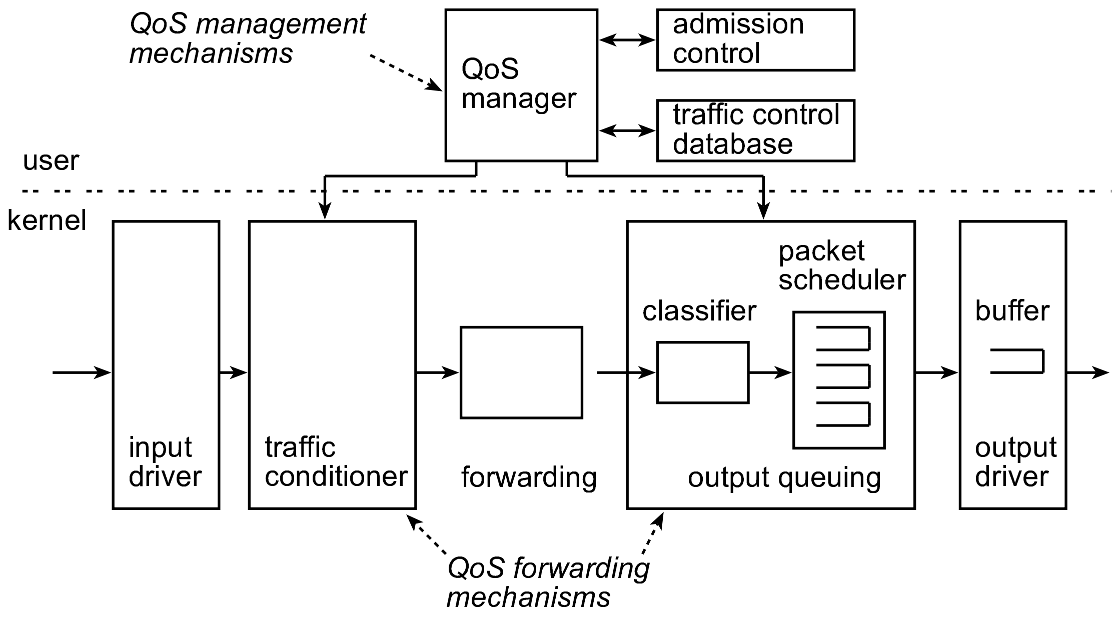


Figure 4.2. *[Cho01] ALTQ traffic control model*

The ALTQ traffic control model is shown in Figure 4.2. The QoS manager containing the management mechanisms resides in the user space of the operating system and communicates with the admission control mechanisms and with the traffic control database. The QoS manager has a control over the kernel-located forwarding mechanisms which in the figure are divided in two blocks. The traffic conditioning block handles the incoming packets. It classifies and meters them, and either marks or drops them according to the customer contract. The output queuing block includes the scheduling mechanisms and the packet classification operations. [Cho01]

4.6.2 ALTQ Implementation Model

One of the main goals in ALTQ implementation was to change the existing operating system code as little as possible. However, especially the output queuing required a lot of modifications to the BSD networking code and the network drivers. The output queuing was implemented as a switch from the common kernel code to the ALTQ queuing discipline code, as illustrated in Figure 4.3. The switch was attached to the BSD network interface abstraction called *if_output*. The ALTQ queuing discipline code exists in the kernel parallel to the original queuing code, thus when ALTQ is not enabled, the switch is ignored and the kernel uses the original queuing structure. The enqueue operation queues the packets and is responsible for the related traffic shaping actions, whereas the dequeue operation handles packet scheduling. [Cho01]

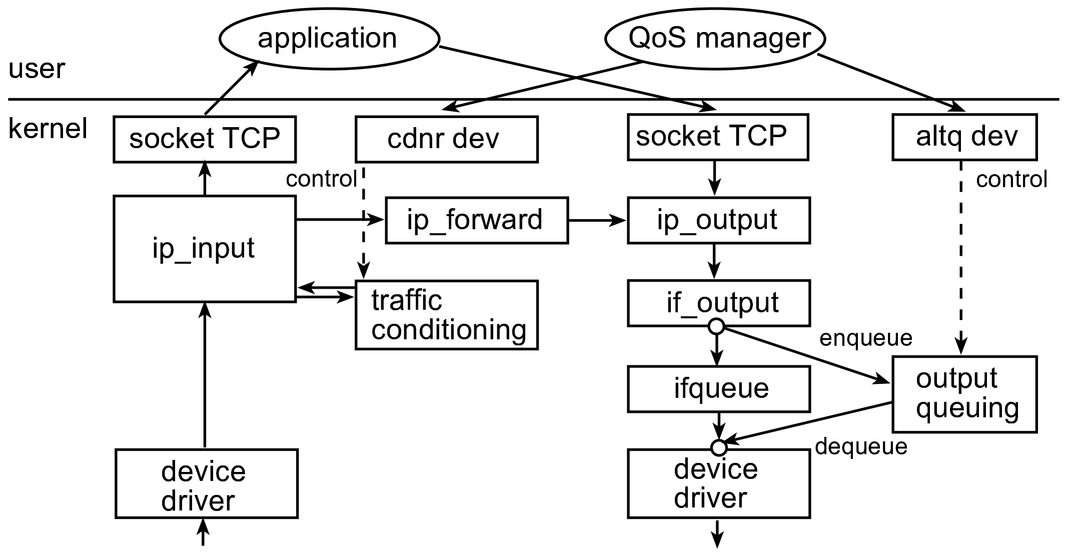


Figure 4.3. *[Cho01] ALTQ system implementation model*

The same type of solution as in the queuing implementation was made with the traffic conditioning. A hook for traffic conditioning code was inserted into the kernel code, and when ALTQ is enabled, all traffic flows through the conditioning component. The forwarder abstraction *ip_forward* connects the conditioning to the queuing and scheduling and thus forms the ALTQ router forwarding path. [Cho01]

The QoS operations are controlled by the QoS manager, which uses the corresponding kernel devices to command the traffic conditioning and output queuing blocks. Traffic conditioning is reached through *cdnr dev* and output queuing through *altq dev.* In reality, all queuing disciplines have special devices in kernel, hence CBQ uses *cbq dev*, for example. [Cho01]

Chapter 5

Policy Control Agent

5.1 Introduction

The adaptive policy-based network prototype consists of several parts which are combined together. The network includes routers, database servers and mechanisms for traffic classification and measurement. In the edge routers, the QoS functions are realised with ALTQ software. The ALTQ configuration parameters are updated based on the traffic measurements and stored into the database. The currently valid ALTQ configuration parameters are kept safe in another database.

The database server containing the ALTQ parameters is called a Policy Server. Because ALTQ and the server are not capable of communicating with each other, there is a need for a block that takes care of the parameter and other information passing between them. The block is a software called a Policy Control Agent, which is located in the edge routers. This chapter describes the architecture and functionality of the Policy Control Agent and also gives some viewpoints to the implementation issues.

5.2 Overview of the Policy Control Agent

The main task of the Policy Control Agent is to create a communication channel between the Policy Server and ALTQ. Therefore, in the architectural model the location of the Agent is between the Policy Server and ALTQ, but in practice it resides inside the edge router. All edge routers have their own Policy Control Agents, thus each router communicates independently with the Policy Server. The information flow model in Figure 5.1 shows the communication chain from the Policy Server to ALTQ.

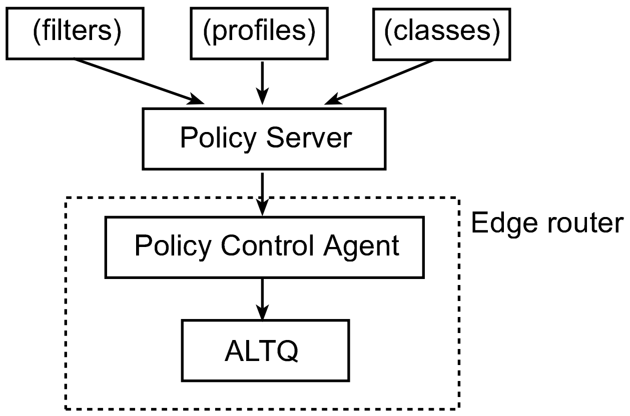


Figure 5.1. *Model of the information flow from the Policy Server to ALTQ*

The Policy Server contains a set of MySQL databases which store information about the current configuration of ALTQ as well as the latest updates to the configuration parameters. At the moment there are three separate databases: the class database, the profile database and the profile filter database. The profiles are based on the service contracts made with the customers and they define the behaviour of the edge routers in the input interface. The class database contains the CBQ configuration parameters needed in the edge router output interfaces and in the core routers. The databases are accessed via a special interface written in Perl. The actual communication between the router and the databases is transparent to the Agent, since it uses the functions provided by an external database access library. The library functions take care of the connection establishment to the Policy Server and making queries to the databases. The updated ALTQ configuration parameters are returned as lists to the Policy Control Agent which stores them temporarily into its local databases.

The communication between the Policy Control Agent and ALTQ is handled via the ALTQ API. The API, which is described in more detail in Chapter 4, includes commands to control ALTQ and change its configuration. The Agent reads the parameters which it has got from the Policy Server and uses them in the ALTQ API function calls. ALTQ functions reply to the configuration modification with an error code which indicates whether the command was executed successfully. At the moment, the error codes are mostly used for debugging, but in the future the Agent might include an error detecting system based on the ALTQ error codes.

The Policy Control Agent can be configured only when it is not running. Once the Agent has been invoked, the interaction between the outside world and the Agent is minimal. The Agent does not take any commands while running, but for debugging reasons it prints some information and statistics of the current actions to the standard output, usually on the screen.

5.3 Implementation Issues

The main goal in the development of the Policy Control Agent was to implement a working prototype, which included all essential functionalities. Therefore, the emphasis was in the operational matters, not in the elegancy of the implementation. Simplicity, flexibility and expandability were the main criterias in the design decisions. The performance of the Agent depends on the performance of the Policy Server and on the capacity and reliability of the communication link between them. The most time-consuming operation in the Policy Agent itself was assumed to be the maintenance of the local databases.

The Policy Control Agent was designed to work in FreeBSD 4.5 and with Perl 5.005_03. The FreeBSD platform was a natural choice, since ALTQ is written for FreeBSD and therefore the edge routers of the network also run FreeBSD. The version of FreeBSD was frozen to 4.5 firstly, because of ALTQ and secondly, to avoid the compatibility problems caused by operating system and software updates.

5.3.1 Language

The languages used in the implementation of the Agent were C and Perl. The part of the Agent that communicates with the Policy Server was written in Perl. There were two reasons why Perl was chosen: firstly, communication with the databases requires many operations on strings, which are easy to implement in Perl, and secondly, Perl was suitable also in the Policy Server side. Thus, Perl was agreed to be the language of the interface between the Policy Server and the Agent. Majority of the Agent code was written in C, because it is the language used also in the ALTQ API.

5.3.2 Communication with ALTQ

ALTQ provides a two-layered API for the external programs. The structure and functionality of the ALTQ API is described in Chapter 4. The function calls that the Agent uses belong to the QCMD API which despite of its simplicity provides all necessary functionalities. Sticking to the QCMD commands spares the Agent and the Policy Server from handling various data structures and parameters, which are needed with the QOP API.

To create the communication channel ALTQ is invoked inside the Policy Control Agent. This does not affect the functionality or performance of ALTQ, but it gives the wrapper program, the Policy Control Agent, a possibility to have full control over ALTQ. As ALTQ runs inside the Policy Control Agent process, it terminates when the Agent terminates. The other way round, the Agent can disable ALTQ and keep running itself.

There are three important concepts in the ALTQ configuration. First there are classes which in our case are CBQ classes and located in the core routers and in the output interfaces of the edge routers. Secondly there are profiles, called also conditioners, which reside in the input interfaces of the edge routers and define the actions that have to be taken for the certain kind of incoming traffic. Finally there are filters which are traffic classifying rules attached either to classes or profiles. An overview of CBQ and conditioning is given in Chapter 2.

5.3.3 Internal Databases

The Policy Agent maintains two internal databases called a modification database and a maintenance database. Together the databases are referred to local databases in distinction from the the Policy Server databases. The local databases are located in the edge routers and every Policy Control Agent is responsible for the maintenance of its own databases. Both databases consist of a directory tree, which has files as its leaves. The files are common text files and each of them contains one parameter value. The parameter name is not included into the file contents but the file is named after the parameter. The core structure of the databases is static, but the number of stored classes, profiles and filters can be arbitrary. The directory path from the database root to the parameter tells where the parameter belongs to in the ALTQ configuration scheme. Thus, the directory structure can be seen as a navigation map of the database.

The modification database works as a temporary storage for the parameters received from the Policy Server. Classes, profiles and profile filters all have the corresponding directories in the database, as can be seen in Figure 5.2. In the profile directory, the login subdirectory contains the profiles of the customers who have recently connected to the network and need their profiles to be added to the edge router ALTQ configuration. The logout directory includes the customer profiles that can be removed from the ALTQ configuration, since the customers have disconnected from the network. There are no profile parameters in the logout directory, only the names of the profiles, since it is the only information needed to delete profiles. The profile filters are divided into the add and remove directories, which are located under the corresponding profile directory. Both directories exist only if there are filters to be added or removed. The class branch is simpler than the profile related branches, because the classes are not added or removed, only their parameters are modified. The class filters are static and cannot be changed at all, so there is no place reserved for them in the database. In Figure 5.2, the static directories are bolded, and the files containing the actual parameters are italicised.

The effective ALTQ configuration parameters are stored into the maintenance database, whose structural model is presented in Figure 5.3. The purpose of the database is to keep the static parameters safe and outside of the ever changing modification database, and also to provide backup parameters in a case of an error. The database structure is quite alike to the modification database, although the structure is a bit simpler. Classes, class filters, profiles and profile filters have their own directory branches of which the class filter branch is completely static. The other branches are updated every time when the ALTQ configuration has been changed successfully. Like in Figure 5.2, also in Figure 5.3 the parameter files are italicised and the static directories and files are bolded.

5.4 Architecture

In the operational level, the Policy Control Agent architecture consists of four functional modules which all perform different tasks. As Figure 5.4 shows, three modules have a close relationship with each other and the local databases, while one module communicates only with one local database inside the Agent. The triplet Main Module, ALTQ Communication Module and Local Database Management Module takes care of configuring ALTQ, which is the main task of the Policy Control Agent. The Policy Server Communication Module is actually only an auxiliary module but ex-

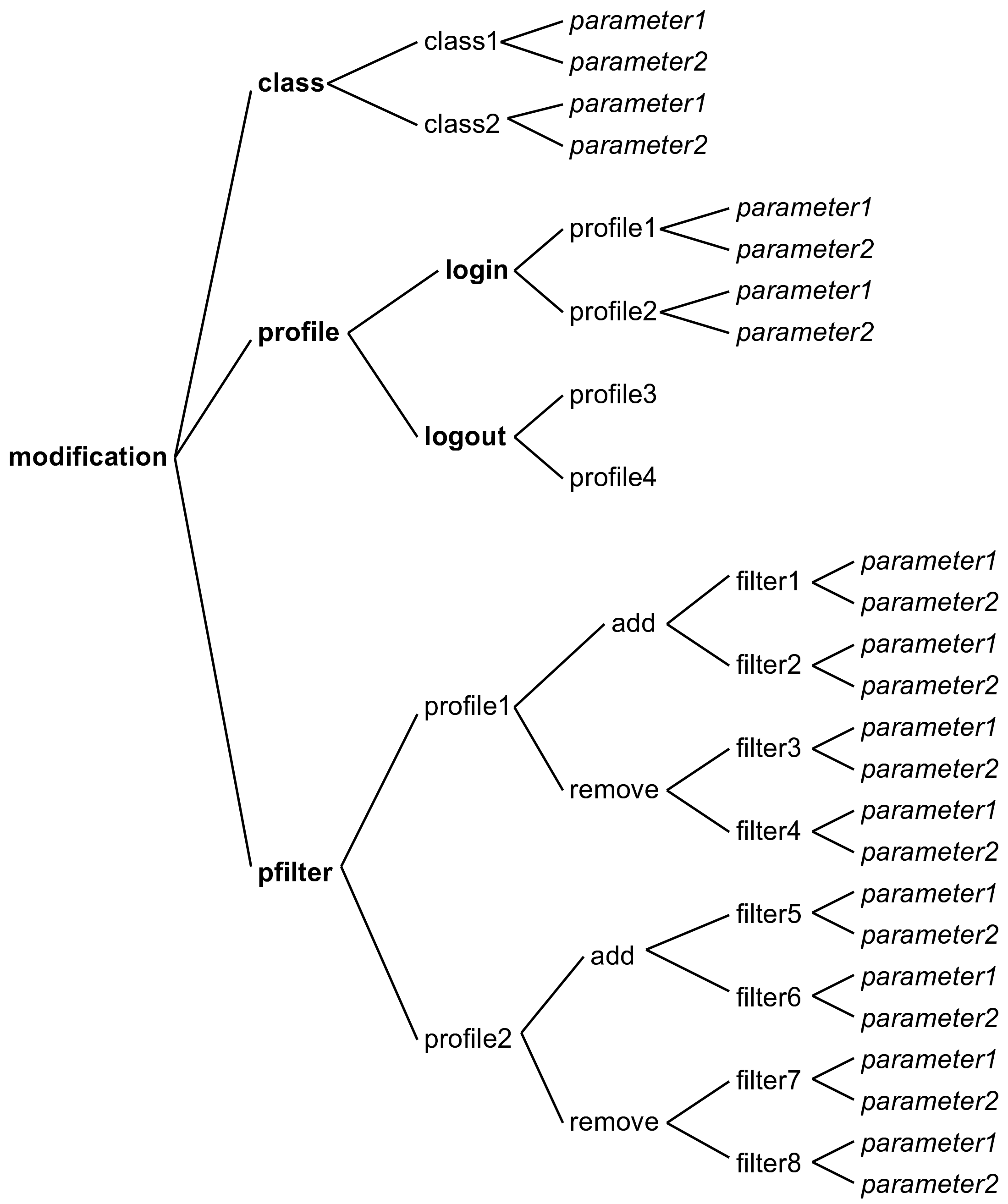


Figure 5.2. *Reference model of the modification database*

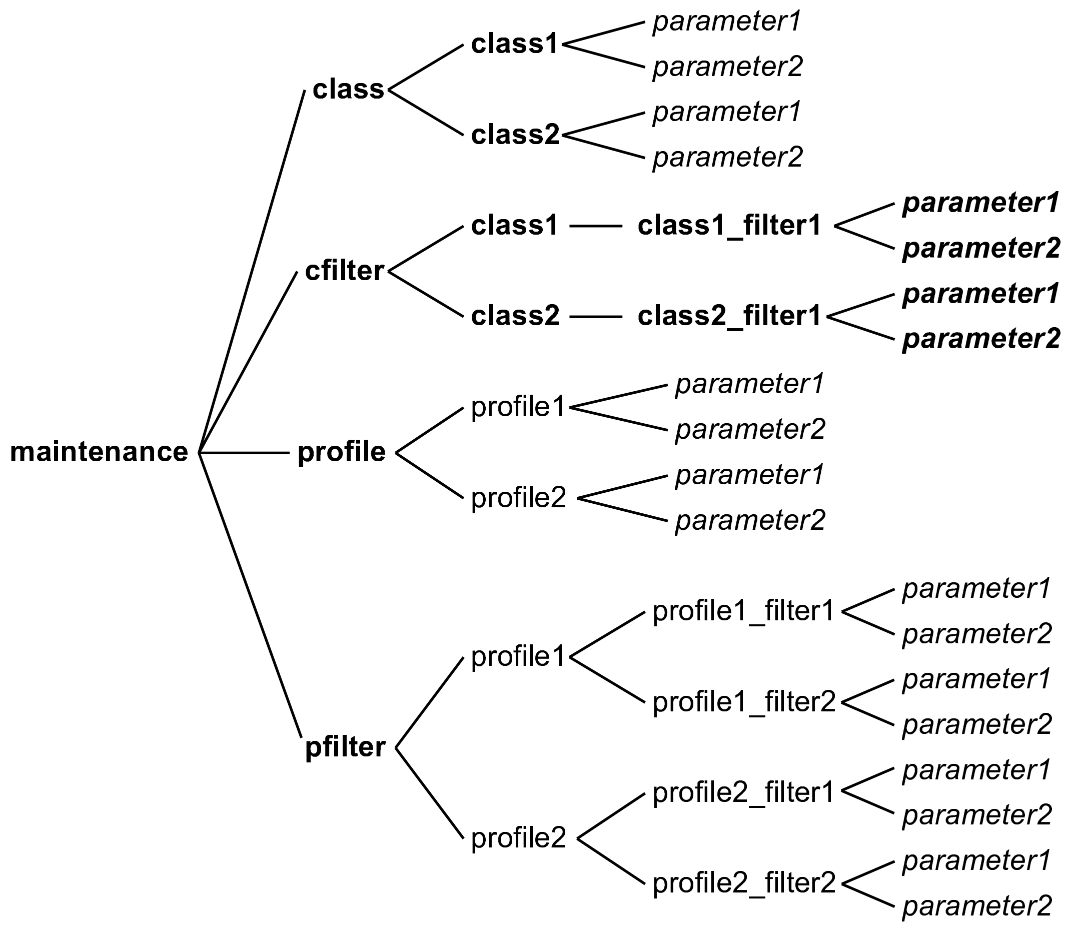


Figure 5.3. *Reference model of the maintenance database*

tremely necessary, because it handles the parameter passing from the Policy Server to the local database.

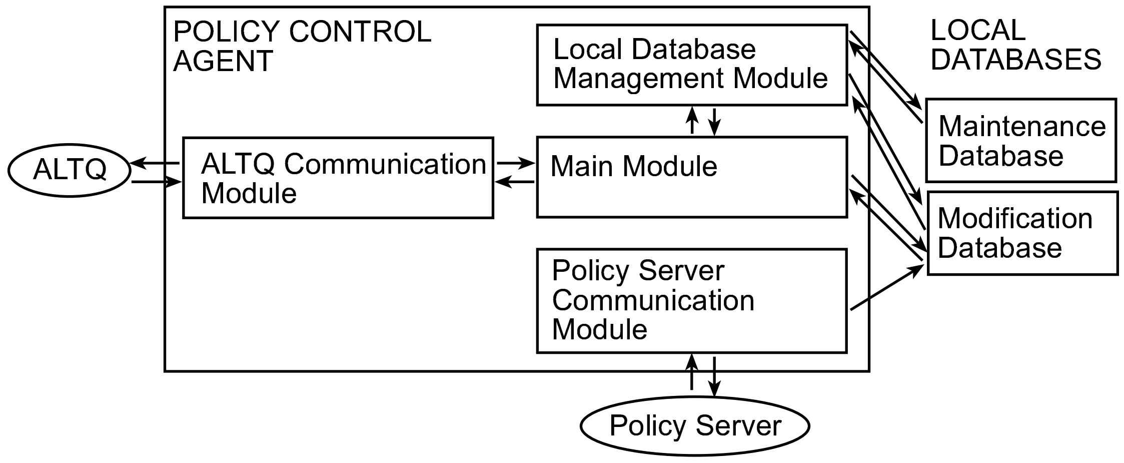


Figure 5.4. *Architectural model of the Policy Control Agent*

The Policy Control Agent is a single runnable C program apart from the Policy Server Communication Module which is an individual Perl program. The communication between the C modules operates through function calls, but currently straight communication between the Policy Server Communication Module and the rest of the Agent is not needed. Hence, the Main Module which controls the two other modules in the triplet is not able to have any influence on the operation of the Policy Server Communication Module.

5.4.1 Main module

The Main Module is the topmost module in the module hierarchy. It ties the functionalities provided by the other modules together and in that way builds up the system called Policy Control Agent. Therefore, the Main Module has a control over the Agent, excluding the Policy Server Communication Module, and besides it works as a communication link between the local databases and the ALTQ Communication Module. In the code level, the Main Module includes the definitions for the major data structures, interfaces and other parameters used throughout the Policy Control Agent.

The first task of the Main Module is to take care of the initialisation of ALTQ when the Policy Control Agent is invoked. Forwarding the parameters that the Policy Server Communication Module has received from the Policy Server is the other main task of the Main Module. The parameters are located in the modification database, from where the Main Module reads them and stores them into the data structures. The filled structures are passed as function call arguments to the ALTQ Communication Module, which uses the parameters in the ALTQ API function calls. The third task of the Main Module is to maintain the information of the class hierarchy, so that the class tree can be rebuilt when needed.

5.4.2 ALTQ Communication Module

The module that interacts with ALTQ is called the ALTQ Communication Module. The module has two tasks: to initialise ALTQ and to modify the ALTQ configuration based on the commands and parameters received from the Main Module. The ALTQ Communication Module does not handle the parameters itself. It is the responsibility of the Main Module to read the parameters from the local databases and place them into the correct data structures. The structures are forwarded to the ALTQ Communication Module as function call arguments, thus they can be used in the ALTQ API function calls without modifications.

The ALTQ Communication Module includes all functions necessary to control ALTQ. Besides initialisation of the ALTQ software there are functions for adding and removing profiles and profile filters, modifying classes and adding class filters. ALTQ supports configuration changing on-the-fly, and the new configurations are in effect as soon as they are installed. No rebooting of ALTQ, not to mention the whole router, is required.

5.4.3 Local Database Management Module

The function of the Local Database Management Module is to keep the local databases in order. It removes the parameters that are not needed anymore from the modification database and updates the maintenance database according to the changes in the ALTQ configuration. The modification database is cleaned up after every ALTQ modification round, because the database should contain only the static components when the Policy Server Communication Module starts writing the new parameters into it. The maintenance database contains the backup copies of the current ALTQ configuration parameters, so the database has to be updated after every modification in the configuration.

5.4.4 Policy Server Communication Module

The only module that is not controlled by the Main Module is the Policy Server Communication Module. It is an individual program written in Perl, and it is not in any way dependent on the other Policy Control Agent modules. The module works as a link between the Policy Agent local databases and the Policy Server databases. Its task is to establish a connection to the Policy Server with certain intervals, fetch any new configuration parameters it may have and store them into the modification database. The module does not implement the Policy Server interface itself. Instead, it uses an external library which includes functions for getting the different parameters from the Policy Server.

5.5 Operation of the Policy Control Agent

Both programs inside the Policy Control Agent work in continuous loops. The loops start when the Agent is launched and they keep running until the Agent is shutdown. The idea behind the Agent operation is that the programs take turns in executing. First the Policy Server Communication Module writes the new ALTQ parameters to the modification database and only after it is done, the Agent main loop starts reading the parameters and modifying the ALTQ configuration. In the meantime, the Policy Server Communication Module waits for its turn. Since there is no communication between the programs, the mutual exclusion was implemented with timers. However, there is still a danger of a race condition, if the normal operation of the modules is disturbed by network congestion, for example. The files are protected with locks during the operations, but at the moment there is no possibility to prevent the reading and writing modules to access the same database at the same time. One way to avoid such situations would be to use a lock file in the root of the database tree. The file includes the ID of the process that is currently accessing the database. The other process willing to access the database has to wait until the first process has exited the database and removed its ID from the lock file.

5.5.1 Overview

The initialisation of ALTQ starts on the operation of the Policy Control Agent. ALTQ is initialised with a single API command, which launches the ALTQ program and makes it read the configuration file *altq.conf*. Because the aim is to modify the

ALTQ configuration dynamically, the configuration file can be extremely simple, but it is still required in ALTQ initialisation process. When ALTQ is up and running, the Agent enters the main loop, as represented in Figure 5.5.

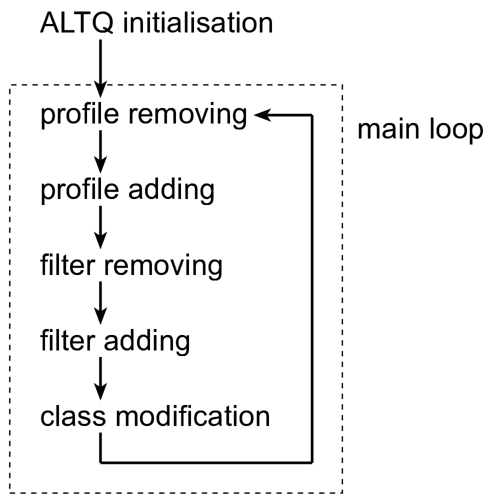


Figure 5.5. *The basic operation of the Policy Control Agent*

In the main loop, the profile modifications are examined first. To avoid possible overlaps, the removal of the profiles is carried out before the addition. The profile filters are operated next, in the same order as the profiles. The final step is to modify the classes after which the execution of the program pauses for a while and then starts again from the beginning of the main loop.

At the moment, the error cases with the router and the Agent are noted but they are not handled properly. The maintenance database is meant to provide backup copies of parameters in case of errors, but in the current Agent implementation the feature has not yet been taken into use. If the malfunction that requires reconfiguration of ALTQ happens during the database updating process, extra care has to be taken of that the information read from the database is consistent. If the router restarts when the Agent is running, the Agent is launched anew and its operation starts from the beginning. However, a mechanism that restores the previous configuration and harmonises the configuration and the local databases has not yet been implemented.

5.5.2 Profile Operations

The main operations for the profiles are adding and removing. The Agent searches the modification database and if there are parameters in the profile login directory, there are profiles to be added. Likewise, if the profile logout directory is not empty, there are profiles to be removed. The outlines of the addition and removal processes are presented in Figure 5.6.

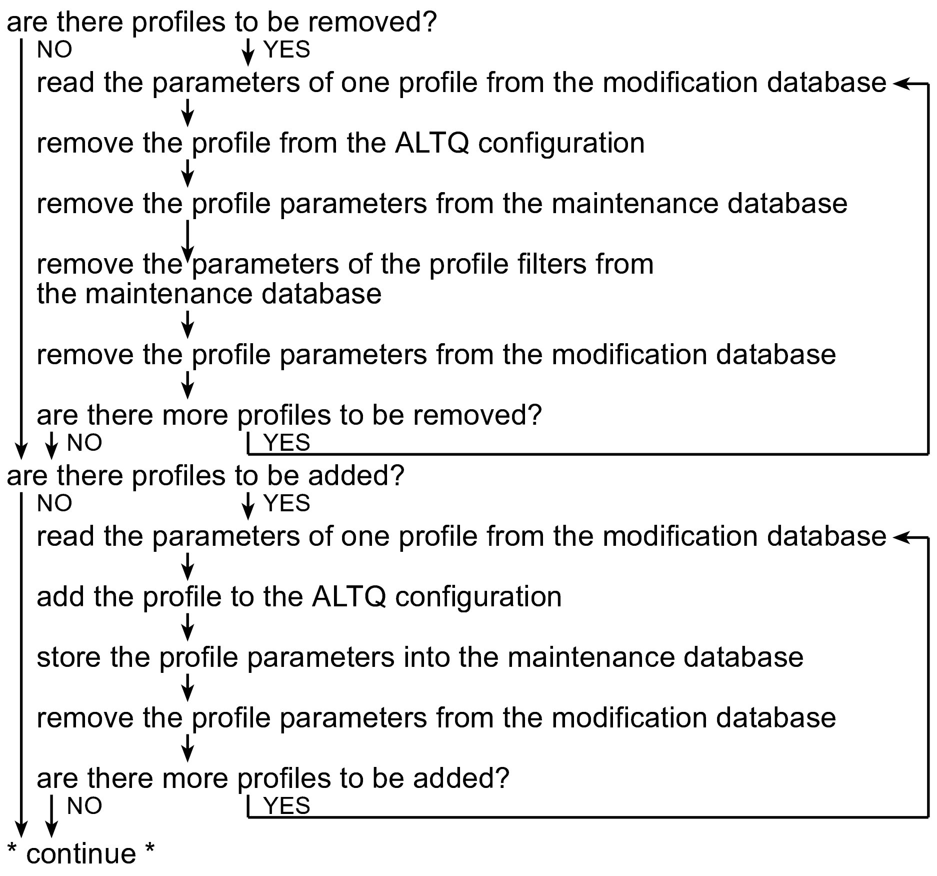

are there profiles to be removed?
NO
YES
read the parameters of one profile from the modification database
remove the profile from the ALTQ configuration
remove the profile parameters from the maintenance database
remove the parameters of the profile filters from the maintenance database
remove the profile parameters from the modification database
are there more profiles to be removed?
NO
YES
are there profiles to be added?
NO
YES
read the parameters of one profile from the modification database
add the profile to the ALTQ configuration
store the profile parameters into the maintenance database
remove the profile parameters from the modification database
are there more profiles to be added?
NO
YES
* continue *

Figure 5.6. *Adding and removing profiles in the Policy Control Agent*

When the profile is removed, its parameters are first read from the modification database. Parameters needed for removing a profile are the name and interface of the profile. The parameters are stored into a data structure which is forwarded to the ALTQ Communication Module. The module removes the profile from the ALTQ configuration. The profile parameters do not include information about whether the profile is a new one to be added or an old one to be removed. Therefore, the Local Database Management Module is informed by the Main Module, if the profile belongs to the login or logout profiles. Because the profile is a logout-profile, its parameters are removed from both local databases. If the profile has any filters, they are automatically removed from the ALTQ configuration at the same time when the profile is removed. Thus, the filter parameters are removed from the maintenance database, too. If there are more profiles stored under the logout directory in the modification database, the profile removal loop starts from the beginning. Otherwise the program moves to the profile additions.

The parameters for the profiles to be added are read from the modification database and placed into a special data structure. Unlike in the profile removal process, all provided parameters are now needed. The data structure is forwarded to the ALTQ Communication Module which takes care of adding the profile to the ALTQ configuration. The Main Module informs the Local Database Management Module that the profile is a login-profile, hence the profile parameters are stored into the maintenance database and removed from the modification database. If there are more profiles to be added, the addition loop starts from the beginning. In other case, the Agent main loop moves forward.

5.5.3 Filter Operations

The filter operations consist of adding and removing profile filters. Because in the ALTQ configuration the filters and the profiles are tied together, the profiles cannot be ignored in the filter operations. The filter changes are examined for one profile at a time, mostly because of the structure of the modification database. The filter addition and removal process is outlined in Figure 5.7.

If the profile has filters that need to be removed, first the filter parameters are read from the modification database. Currently all the filter parameters are stored into the data structure, although only the name of the filter and the profile it is attached to are actually needed. The name of the interface is also required, but as the filter parameters do not provide the information, it has to be read from the profile parameters in the maintenance database. The parameters are forwarded to the ALTQ Communication Module which removes the filter from the ALTQ configuration. Like in the profile operations, the Main Module informs the Local Database Management Module about the type of the filter. The filter parameters are then removed both from the maintenance and modification database. If the profile has more filters that need to be removed, the removal process is started again. If not, the program continues with the filter additions.

The parameters of the new filter are read from the modification database

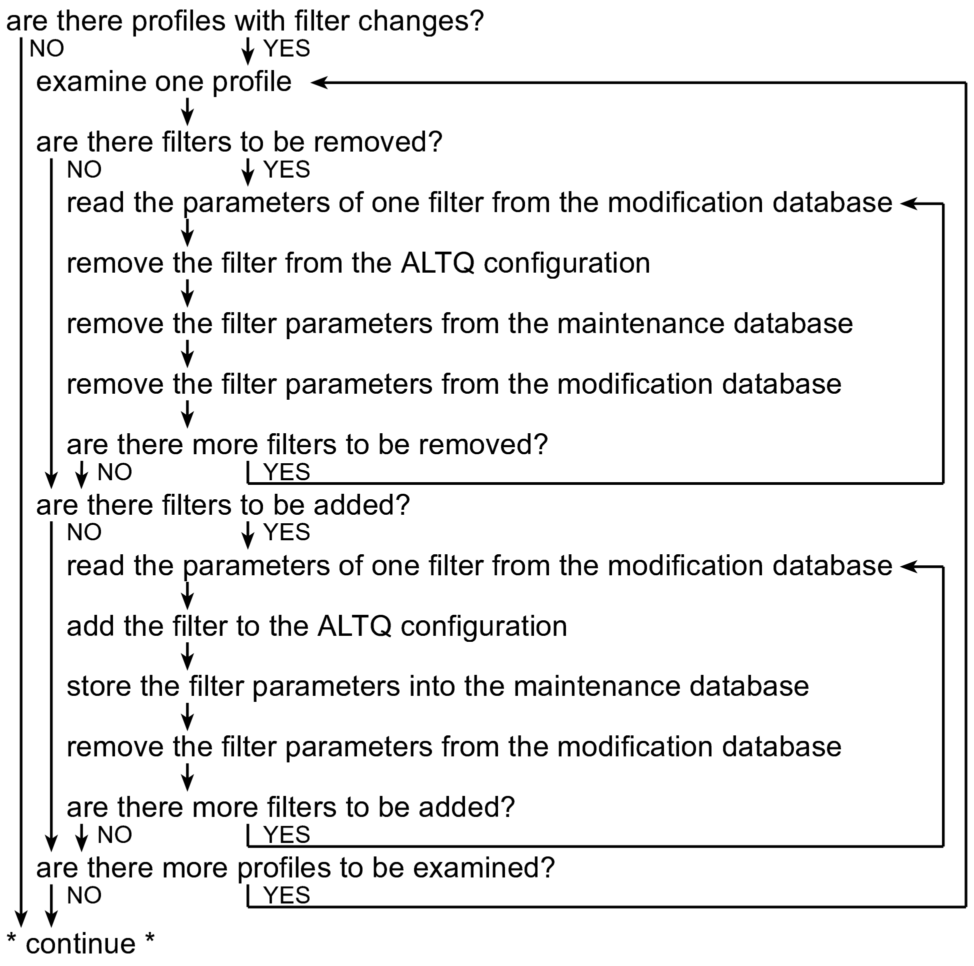

are there profiles with filter changes?
NO
YES
examine one profile
are there filters to be removed?
NO
YES
read the parameters of one filter from the modification database
remove the filter from the ALTQ configuration
remove the filter parameters from the maintenance database
remove the filter parameters from the modification database
are there more filters to be removed?
NO
YES
are there filters to be added?
NO
YES
read the parameters of one filter from the modification database
add the filter to the ALTQ configuration
store the filter parameters into the maintenance database
remove the filter parameters from the modification database
are there more filters to be added?
NO
YES
are there more profiles to be examined?
NO
YES
* continue *

Figure 5.7. *Adding and removing filters in the Policy Control Agent*

and stored into the data structure. The ALTQ Communication Module takes again care of the addition of the filter to the ALTQ configuration according to the given parameters. The Main Module provides the type of the filter to the Local Database Management Module which stores the parameters of the newly added filter into the maintenance database and removes them from the modification database. If there are more filters waiting for the addition, the process is started from the beginning. Otherwise, the search moves one level higher to find out if there are more profiles that have filter modifications. When the last profile and its filters are handled, the Agent main loop continues forward.

5.5.4 Class Operations

Class operations include only the modification of the existing class. Classes are special cases in the ALTQ configuration and also in the Policy Control Agent implementation, since the class structure is static. In theory, all class parameters can be modified, but in practice, the only parameter that is modified is the bandwidth. As Figure 5.8 shows, the class modification consists of only one functional block, unlike the filter and profile operations.

Modifying a class is a more complex operation than any of the filter or profile operations. The class is modified by replacing the old class with a new class containing the new parameters. Therefore, the operation requires removing and adding the modified class. However, a class cannot be removed if it has any child classes. So all child classes need to be removed first, before the actual class can be replaced with a new one. After the replacement, the underlying class hierarchy has to be built anew and also the possible class filters have to be added to the ALTQ configuration. In the modification operation, the old class needs to be removed before adding a new class, because in the ALTQ configuration, there cannot be two classes with identical filters present at the same time.

The class modification process starts with the reading of the parameters from the modification database. As the ALTQ configuration is equal in all CBQ interfaces, one class is modified throughout the interfaces before moving on to the next class. The Main Module stores the structure of the class hierarchy, and if the class has any child classes, the ALTQ Communication Module is ordered to remove them temporarily. After that, the class itself is removed. If the class or any of its child classes has filters, they are automatically removed at the same time as their host class is deleted. The modified class is added to the ALTQ configuration, and if

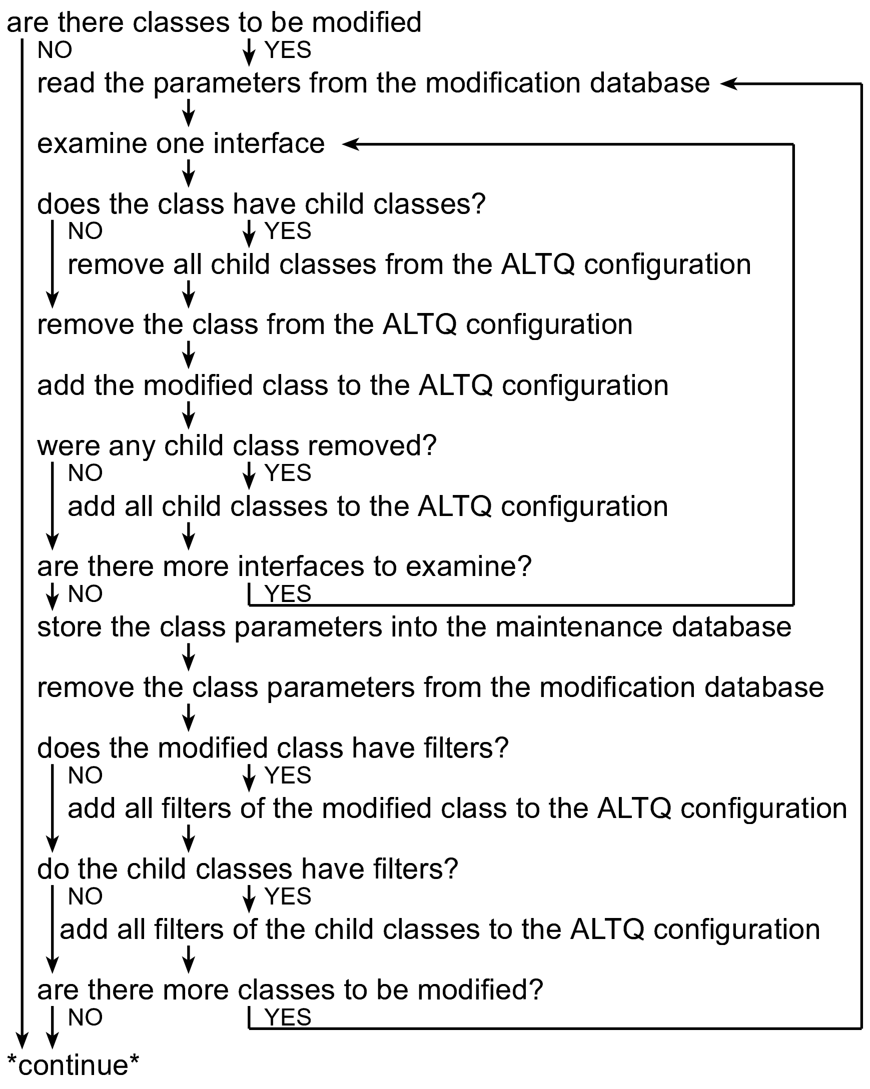

are there classes to be modified
NO
YES
read the parameters from the modification database
examine one interface
does the class have child classes?
NO
YES
remove all child classes from the ALTQ configuration
remove the class from the ALTQ configuration
add the modified class to the ALTQ configuration
were any child class removed?
NO
YES
add all child classes to the ALTQ configuration
are there more interfaces to examine?
NO
YES
store the class parameters into the maintenance database
remove the class parameters from the modification database
does the modified class have filters?
NO
YES
add all filters of the modified class to the ALTQ configuration
do the child classes have filters?
NO
YES
add all filters of the child classes to the ALTQ configuration
are there more classes to be modified?
NO
YES
continue

Figure 5.8. *Class modifications in the Policy Control Agent*

any child classes were removed earlier, they are now added back. If there is more than one class interface, the previous operations are performed in all of them before moving forwards. When all interfaces have been handled, the modified class parameters are stored into the maintenance database and removed from the modification database. If the modified class has filters, they are added to all interfaces. In addition, the possible filters of temporarily removed child classes are added back to the configuration. If there are more classes waiting to be modified, the class modification process is run again. After the class modifications, the Agent is ready to start a new loop, as shown in Figure 5.5.

5.5.5 Receiving New Parameters

The updated ALTQ configuration parameters are transferred from the Policy Server to the edge router via the Policy Server Communication Module. The functions for making database queries are implemented in an external library. There are separate functions for accessing the different databases in the Policy Server, thus the Policy Server Communication Module makes three queries to the Policy Server databases. In the outline of the process, in Figure 5.9, the external functions are written with italic fonts.

The class parameters are handled first. If the parameter list received from the Policy Server is empty, there are no class modifications. Otherwise, the parameters are stored into the modification database class by class. The profile parameter list which is received next actually consists of two lists. The first list includes the login-profiles and the second list includes the logout-profiles. If there are parameters in the login-list, they are stored into the modification database under the login directory. Likewise, the parameters in the logout-list are stored into the modification database under the logout directory. If either of the lists is empty, it means that there are no such profile modifications. The last list to handle is the filter list. Like the profile list, also the filter list consists of two lists: one for filters to be added and one for filters to be removed. If the addition list is not empty, the filter parameters are stored into the modification database under the add directory. The parameters in the removal list are stored under the remove directory. Also in this case, either or even both of the lists can be empty, if there are no such filter modifications.

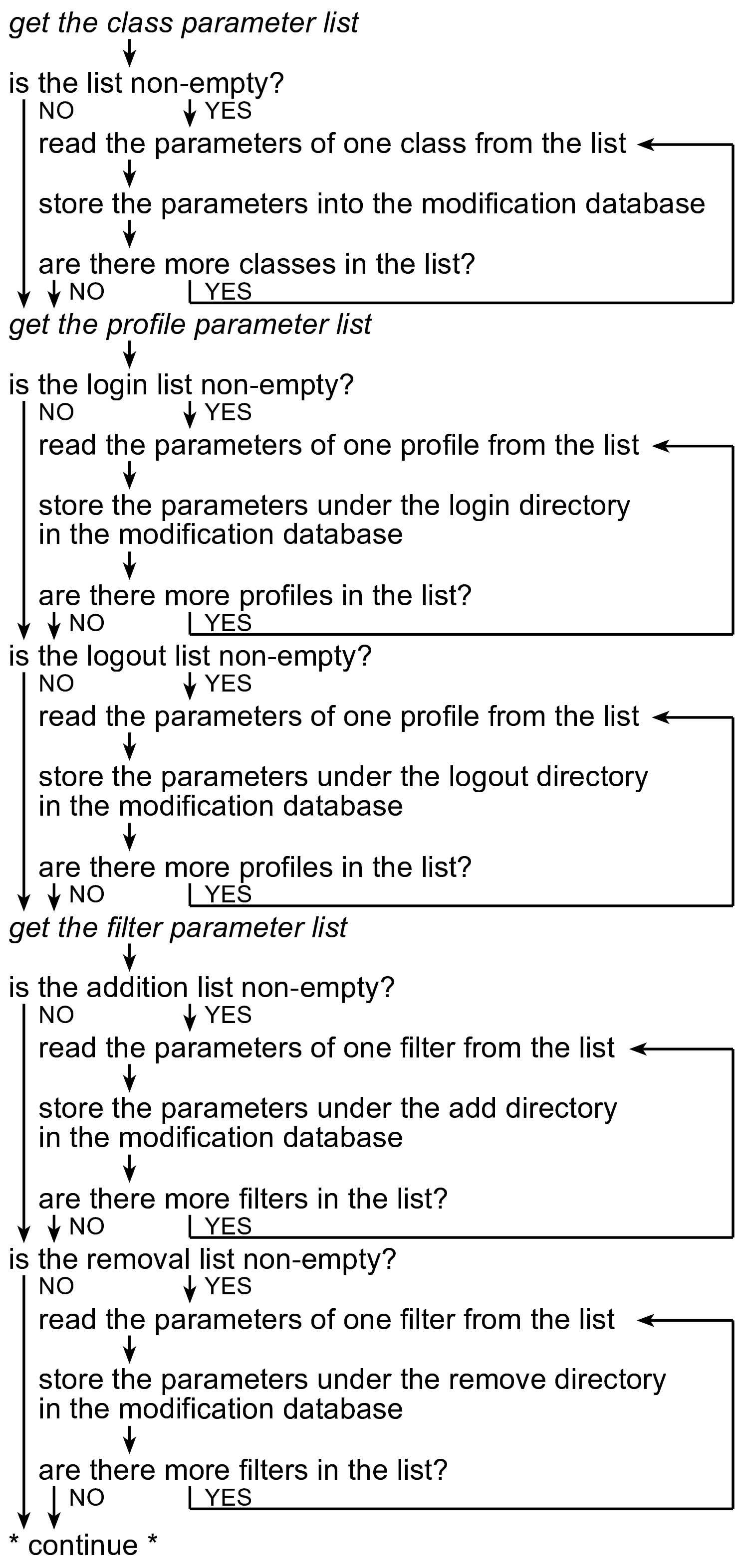

get the class parameter list
is the list non-empty?
NO
YES
read the parameters of one class from the list
store the parameters into the modification database
are there more classes in the list?
NO
YES
get the profile parameter list
is the login list non-empty?
NO
YES
read the parameters of one profile from the list
store the parameters under the login directory in the modification database
are there more profiles in the list?
NO
YES
is the logout list non-empty?
NO
YES
read the parameters of one profile from the list
store the parameters under the logout directory in the modification database
are there more profiles in the list?
NO
YES
get the filter parameter list
is the addition list non-empty?
NO
YES
read the parameters of one filter from the list
store the parameters under the add directory in the modification database
are there more filters in the list?
NO
YES
is the removal list non-empty?
NO
YES
read the parameters of one filter from the list
store the parameters under the remove directory in the modification database
are there more filters in the list?
NO
YES
* continue *

Figure 5.9. *Operations of the Policy Server Communication Module*

Chapter 6

Policy Control Agent Measurements

6.1 Introduction

The FreeBSD-based ALTQ/Policy Control Agent router was put under various tests in order to define the system performance. The aim was to conclude the current state of the router implementation and to point out the issues which needed further development. In the measurements, only one router was tested, although the future target was to install the Policy Control Agent to all ALTQ edge routers in the prototype network.

In this chapter, the test cases and the testing environment are introduced. The results from the measurements are presented and analysed case by case. Finally, the overall performance and functionality of the Policy Control Agent is discussed.

6.2 Test Setup

All experiments were made with one router which was connected to the prototype network but did not serve as a router in it. The router functionalities of the system were strictly limited for testing. Figure 6.1 shows the basic test setup. Two special testing devices with dedicated hardware and software were used in the measurements. They were exploited as full-featured performance test systems in some measurements, but in most cases they served only as traffic generators. The devices were connected to the router-under-test one at a time, and the decision of which one to use in each test was made based on the device suitability and ease-of-use in the measurement in question. Due to the present FreeBSD hardware, only two of the four available network interfaces were enabled for testing purposes.

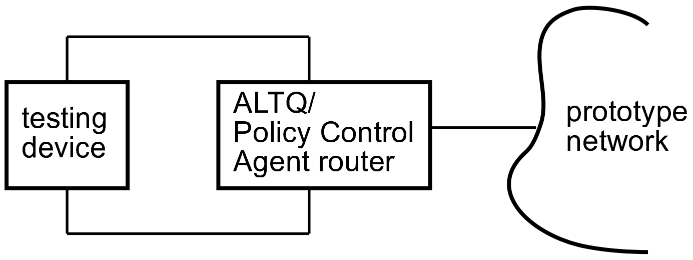


Figure 6.1. *Test setup in the network*

ALTQ was configured in two network interfaces which were dedicated for testing. The actual configuration varied a little between some test cases, but the basic configuration stayed the same throughout all measurements. In the input interface, the traffic conditioning was taken care of by token buckets and two-rate three colour markers (trTCM), and the queuing method used in the output interface was Class Based Queuing (CBQ) with tail-drop, RED and RIO queue management mechanisms.

6.2.1 Router Hardware and Software

The hardware of the router was a standard PC hardware with AMD 1300 MHz processor, 256 MB memory and 30 GB hard disk. The network interfaces consisted of one D-Link 4-port network interface card which could operate in both 10 Mbps and 100 Mbps Ethernet networks.

The operating system of the router was freely distributable Unix clone called FreeBSD, version 4.5. Both the operating system and its version were chosen because of the requirements of the ALTQ software. ALTQ 3.1 which was the latest version of the software was designed for FreeBSD 4.5 and was not known to cause any problems in that operating system version. The newer versions of FreeBSD were reported not to be fully compatible with the software without patching.

6.2.2 Measurement Tools

General measurements were carried out with the Spirent Communications SmartBits 600 network performance analysis system. SmartBits is a device consisting of special hardware for measurement purposes, and it can be connected with cross-over cables to one or several devices to be tested. The testing software was Spirent Communications SmartFlow which was designed for QoS testing with SmartBits. It is able to measure for example throughput, frame loss and different latency characteristics from the traffic that SmartBits generates to the test network. The traffic generator creates UDP-based packets, whose attributes - header type and content, for example - are defined by the user. [Spi01]

The other device used in general measurements was the Spirent Communications Adtech AX/4000 test system [Com]. AX/4000 has been developed for performance and QoS testing of broadband technologies, and although its basic functionality resembles SmartBits, it is much more versatile. Like SmartBits, AX/4000 was connected with cables to the tested device or system. AX/4000 was needed in the measurements, because it was able to gather more detailed statistics than SmartBits. It was used in the latency test which studied the latency variations in changing network conditions and therefore required measuring the latency at short intervals. The AX/4000 control software was also able to draw a real-time graph of the latency which made the progress of the ongoing measurement easy to follow.

With both testing devices, the bit rate was chosen to be 10 Mbps, the same as in the test network. The other option could have been to choose 100 Mbps connection, because that was supported both by the tested router and by the testing devices. Although the main reason for the decision was to make the results valid also in the prototype network, the other weighty reason was that the network card in the router was a 4-port card which worked more reliably with a smaller bit rate.

In the modification and Policy Server Communication Module measurements, SmartBits was used as a background traffic generator. The purpose of the traffic was to keep the router processor busy, not necessarily to pass through the router. The actual measurements were performed inside the Agent software independently of the traffic. The modification measurements used a standard C library function 'gettimeofday' which expressed the current time of the system clock with the accuracy of microseconds. The time that the Agent spent in some operation could be calculated as a difference between two timestamps. The function was also available for Perl, so it was used in the measurements with the Policy Server Communication Module. The timestamps were not written into a file but printed to the screen in order to avoid the disk operation overhead in the results. The other timestamping function in use was 'getrusage' which was also a standard C library function. It measured the time that the predefined function spent in the user and system operations. The fine-grained time-usage information was needed in the I/O operation measurements.

6.3 Testing Procedure

The target of the measurements was to find out the key figures of the Policy Control Agent performance. The main emphasis was on the trends, not on the absolute time values. The reference points were defined with the baseline tests which consisted of general router testing without ALTQ. In the ALTQ router tests, all the Policy Control Agent functionalities were tested with and without background traffic. The hypothesis was that the overall performance drops if the router needs to handle packets at the same time when making changes to the ALTQ configuration or handling the local databases. Because the functionality of the Agent included plenty of disk writing and reading operations, the other assumption was that the I/O operations form one of the major bottlenecks of the Agent performance. Therefore, the affect of the disk operations were studied more closely. In all test cases the measurements were repeated at least five times in order to improve the reliability of the results. The final results are averages of the individual measurement results.

Due to the restrictions of the testing devices, background traffic was always UDP-based traffic with modified packet headers. Although TCP packets were used in the background traffic, neither of the testing devices was able to establish real TCP connections. Thus, the traffic was connectionless in every case.

6.3.1 General Performance

The general performance tests were carried out with SmartBits testing device and SmartFlow software. In the tests, the router was first configured as a normal FreeBSD router without ALTQ. The aim of the measurements was to find out the performance of a 'clean' system and therefore get the baseline values for later testing. The same baseline measurements were executed with the ALTQ/Policy Control

Agent router to define the basic performance of an ALTQ system. ALTQ was configured so that it did not disturb the measurements by dropping packets because of the capacity restrictions.

6.3.1.1 Throughput

In SmartFlow, the maximum throughput was defined to be the maximum transmission rate at which the router can forward the traffic without any frame loss. SmartFlow calculated the frame loss by comparing the number of transmitted and received frames, and if there were lost frames, the test failed. The throughput test started at the user defined traffic load which was increased in every iteration round until there were dropped packets. After the first failed round the test continued as binary (scaled) search as long as the final rate between the last successful and failed test rounds was found. [Spi01]

Throughput was measured for three router configurations: general FreeBSD router, ALTQ/Policy Control Agent router with a simple and light ALTQ configuration and ALTQ/Policy Control Agent router with a more complex and heavy ALTQ configuration. All router configurations were tested with two traffic variations. First, the traffic consisted of 1024-byte UDP packets, and then of 128-byte UDP packets. Thus, totally there were six test cases.

6.3.1.2 Latency

The latency measurements determined the minimum, maximum and average latency of the transmitted packets at different traffic loads. SmartFlow defined the latency to be the difference between the transmit timestamp and the receive timestamp in the frame. The latency was calculated from one port to another unidirectionally and only for the received frames. The testing procedure started from the load percentage defined by the user which specified the traffic load relative to the wire capacity. The duration of the test rounds was changeable and specified in seconds. After each round the load percentage was increased until the maximum defined load was reached. [Spi01]

Latency was measured at 1 Mbps intervals starting from 1 Mbps and ending at 10 Mbps. For the traffic with 1024-byte packets the granularity was enough, but for the traffic with 128-byte packets an additional test had to be performed. In the new test, the latency was measured between 200 kbps and 1600 kbps at 200 kbps intervals. Thereby, the traffic with 1024-byte packets was measured at ten points and the traffic with 128-byte packets at 17 points.

6.3.1.3 Packet Order

The aim of the packet order test was to define how well the ordinary FreeBSD router and ALTQ router were able to maintain the correct packet order. In SmartFlow, the packet order was observed in the frame loss test. SmartBits marked each transmitted frame with a sequence number which was examined when the frames were received from the tested router. If the sequence number was equal to one more than the sequence number of the previous frame, SmartFlow considered the received frame to be in sequence. If the sequence number was inequal to the reference number, the frame was out of sequence. The out-of-sequence frames could be caused by the packet reordering in the router as well as frame loss. Therefore, if there were lost frames, the number of out-of-sequence frames was not valid for the packet order measurement purposes. [Spi01]

The measurements were performed with an ordinary FreeBSD router and with an ALTQ router which contained a simple ALTQ configuration. The traffic used in the measurements consisted of 128-byte UDP packets. The traffic load was 1 Mbps in the beginning and it was increased by 1 Mbps until it reached 10 Mbps.

6.3.2 Profile Modifications

In a real network, a profile is added every time a user logs in and removed when he logs out. The support for mobility of the users is therefore implemented with the profile operations. The profile modifications are assumed to happen comparatively often, and usually they are related to filter operations. This is natural, since if a profile is deleted, also all its filters are deleted, and adding a profile without any filters is useless. However, the profile measurements without filters were needed in order to find out the time spent in deleting or adding profiles alone.

The three observed time consumers were ALTQ, the local databases and the complete Agent loop. In case of ALTQ, the time that ALTQ used in adding or removing one profile was measured. In the local databases, the point of interest was the time that was spent in the database operations when one profile without filters was added and when one profile with zero, one or two filters was removed. In the loop measurements the observed cases were adding and removing 1, 2, 3, 4, 20, 50, 100 and finally 200 profiles without filters.

All cases were studied both with and without background traffic. When traffic was required, SmartBits was used as a traffic generator. In the ALTQ and database measurements, the traffic load was 10 Mbps and the packets were 1024-byte UDP and TCP packets. In the Agent loop measurements, the packet size was reduced to 128 bytes to load the processor even more.

6.3.3 Filter Modifications

Adding and removing filters are the most common Policy Control Agent operations. Therefore, it is important that they are as fast as possible. The aim of the filter measurements was to conclude the current performance of the filter operations, so that the need for optimisation could be resolved.

In the filter measurements, the time usage of ALTQ, the local database management and the Agent loop was measured, thus the testing procedure resembled the profile measurements. In the ALTQ measurements, the time that was spent by ALTQ in adding and removing one filter was observed, so one profile was required in the ALTQ configuration. In the local database measurements, one filter was added to the local database and removed from there. The Agent loop was measured for the time spent in adding and removing 1, 2, 3, 4, 20, 50, 100 and 200 filters. Filters were added under one profile and after each addition loop all filters were deleted. Therefore, all addition loops started from the clean configuration.

The background traffic was generated with SmartBits and in the ALTQ and local databases test cases, it was 10 Mbps of 1024-byte TCP and UDP packets. In the loop measurements the packet size was reduced to 128 bytes.

6.3.4 Class Modifications

The class modification testing procedure differs from the profile and filter measurements due to the complexity of the class modification operation. The total time used in one class modification can be measured, but it will be valid only for the prevailing test case and configuration. The class modification time depends on the place of the class in the CBQ hierarchy and the number of the class filters, thus the results cannot be generalised.

The ALTQ and local database measurements could be performed, because the results were independent of the CBQ configuration. In the ALTQ measurements, one class without child classes and class filters was added to the ALTQ configuration and removed from there. In the local database measurements, the parameters of one class were modified. Because the CBQ hierarchy inside the the router was static, there was no need to measure the time consumption of the class addition and removal operations separately.

The test cases were studied both with and without background traffic. The 10 Mbps background traffic was generated with SmartBits and it consisted of 1024-byte TCP and UDP packets.

6.3.5 Policy Server Communication Module

The Policy Server Communication Module was written in Perl and it forms an independent program inside the Policy Control Agent. The module is not controlled by the Main Module like the other modules. The Policy Server Communication Module has two tasks: to communicate with the Policy Server and to write the parameters into the modification database. The data handling abilities of the module were measured, but the performance of the communication between the module and the Policy Server was out of the scope of this study. Although the module operates with one internal database, it does not perform as many disk operations as the rest of the Agent. Yet, because writing to disk was considered time-consuming, it was necessary to examine the time spent in the database operations.

In the measurements, a profile, filter and class were added to the modification database one at a time. Before every addition, the database was cleaned from the parameters so that it consisted only of the static components. Thus, in all test cases the situation was like in the worst case, because the necessary directory hierarchy needed to be created from the beginning. Since the traffic handling at the same time with the operations was supposed to have an effect on the performance, the measurements were done both with and without background traffic. The traffic generated by SmartBits consisted of TCP and UDP packets at the rate of 10 Mbps and with the packet size of 1024 bytes.

6.3.6 Latency Behaviour during Modifications

The filter modifications were exploited in an additional latency test performed with Adtech AX/4000 testing device. The aim of the test was to find out, what happened to the packets during the heavy ALTQ configuration modifications. The assumption was that during the inconsistent state in the configuration, ALTQ would try to keep the packets queued rather than drop them. Thus, the variation of the latency before, after and during the configuration modifications was measured. The measurements consisted of adding and removing 1, 50, 250 and 750 filters. There were two profiles in the ALTQ configuration and both of them were assigned their own set of filters. In the measurements, first a certain number of filters were added under one profile, then the same number of filters were added under the other profile, and finally all filters were removed from the configuration at the same time. The filters were added and removed in turns, so that the filters were added to the clean configuration. Only the last part of the measurement made an exception: additional 750 filters were added on top of already configured 750 filters.

The traffic created by the testing device consisted of four streams of 128-byte TCP or UDP packets at the bit rate of 500 kbps. Therefore, the total traffic load was 2 Mbps. The load was defined low, because it was necessary to get all packets through the router. With a low traffic load, it could be observed, if the router dropped packets solely because of the modification, not because of a temporary congestion. In addition, it was expected that if the latency increased, it would show also with lower traffic loads. The traffic streams and the ALTQ configuration were chosen so that one stream was always outside and one always inside the defined profiles. The other two streams were either inside or outside the profiles, depending on the added filters. The purpose was to take different situations into account, as it was not known beforehand if there were any differences in the latency behaviour between them.

The results were stored on a per stream basis. The short-time average latency, the number of lost packets and the number of reordered packets were logged at 0.1-second interval. The AX/4000 control software saved the results into a special result file which could be converted into a text file.

6.3.7 Effect of the I/O Operations

It was assumed that the management of the local databases would cause the main overhead of the Policy Control Agent. In order to define the actual effect on the Agent performance, some of the earlier measurements were repeated, this time with the function 'getrusage' which provided detailed timestamping. The timestamps showed the time the process had spent for the user and system mode operations by far. The system mode operations consisted of the disk operations required by the Agent, whereas the actual Agent functions belonged to the user mode operations. The proportion of the user time to the system time was supposed to show the time consumption of the I/O operations in relation to the rest of the Agent process.

The measurements were realised as Agent loops starting from the baseline measurement with no modifications. The loop without any changes provided the reference values for later measurements. In the other measurements, profiles and filters were added and removed in the series of 1, 20, 50, 100 and 200. Background traffic was not present in the test cases, because the aim was to find out the real performance values without any additional stressing of the router.

6.3.8 Summary of the Testing Procedure

The testing procedure is summarised in Table 6.1. The test cases are divided into groups according to the topic. The first column presents the topics and brief names of the tests. The measurements were performed either with or without the Agent, so the content of the column Agent can be *yes*, *no* or *yes/no* if the measurements covered both cases. The traffic was either required or not. The traffic load and the packet size of the traffic are characterised in the Traffic type column. The number of modified profiles, filter or classes is shown in the Number of items column.

6.4 Measurement Results

The purpose of the measurements was to define the basic performance of the ALTQ/Policy Control Agent router. The aim was to use the results for giving insight into the capabilites and limitations of the router and also for finding the functionalities of the Agent which needed further development. Thus, the relations between the results were considered more important than the comparisons with

Test case	**Agent**	**Traffic**	**Traffic type**	**Number of items**
General performance				
Throughput	yes/no	yes	var., 128/1024 B	
Latency	yes/no	yes	var., 128/1024 B	
Packet order	yes/no	yes	var., 128/1024 B	
Profile modifications				
ALTQ	yes	yes	10 Mbps, 1024 B	1
Local databases	yes	yes	10 Mbps, 1024 B	1 with 0-2 filters
Agent loop	yes	yes	10 Mbps, 128 B	1-4/20/50/100/200
Filter modifications				
ALTQ	yes	yes	10 Mbps, 1024 B	1
Local databases	yes	yes	10 Mbps, 1024 B	1
Agent loop	yes	yes	10 Mbps, 128 B	1-4/20/50/100/200
Class modifications				
ALTQ	yes	yes	10 Mbps, 1024 B	1
Local databases	yes	yes	10 Mbps, 1024 B	1
Policy Server Comm. Module				
Adding a profile	yes	no		1
Adding a filter	yes	no		1
Adding a class	yes	no		1
Latency caused by modifications				
Agent loop with filters	yes	yes	2 Mbps, 128 B	1/50/250/750
I/O operations				
Adding a profile	yes	no		1/20/50/100/200
Adding a filter	yes	no		1/20/50/100/200
Removing a profile	yes	no		1/20/50/100/200
Removing a filter	yes	no		1/20/50/100/200

Table 6.1. *Summary of the testing procedure*

external performance values. The majority of the main results are illustrated as figures to show the trend of the results. Only the minor results and the results that did not produce a clear enough graph are listed in tables.

The operating system included several processes which required processor time every now and then. In addition, there were some daemon processes which needed their share of the processor time. Thus, all measurement results that were calculated as a difference between the operation starting and finishing time contained always some overhead caused by the operating system and daemon processes. The processes which had a chance to affect to the results are listed in Table 6.2. Most of the processes seldom needed the processor time, but some of the processes were more active. Especially sshd consumed the processor time, because the router was controlled through an ssh connection.

Process name	**Purpose**
sshd	SSH daemon
ntpd	Network Time Protocol (NTP) daemon
cron	executes scheduled commands
syslogd	logs system messages
csh	shell
getty (8 processes)	virtual terminals
inetd	socket listener (internet 'super-server')
adjkerntz	manages CMOS and kernel clocks
syncer	(kernel) filesystem synchroniser
bufdaemon	(kernel) virtual memory buffer daemon
vnlru	(kernel) flushes and frees vnodes
pagedaemon	(kernel) pagination daemon
init	(kernel) process control initialisation
swapper	(kernel) system scheduler
vmdaemon	(kernel) virtual memory daemon

Table 6.2. *Operating system and daemon processes running in the tested router*

6.4.1 General Performance

The general performance tests included throughput, latency and packet order measurements. The first two measurements consisted of two traffic cases both with three router configuration variations which are listed in Table 6.3. The packet order measurements were performed for test cases 1a, 1b, 2a and 2b.

Test Case	Packet Size	Configuration
Case 1a	128 bytes	no ALTQ
Case 1b	128 bytes	ALTQ with simple configuration
Case 1c	128 bytes	ALTQ with complex configuration
Case 2a	1024 bytes	no ALTQ
Case 2b	1024 bytes	ALTQ with simple configuration
Case 2c	1024 bytes	ALTQ with complex configuration

Table 6.3. *Traffic in the different test cases*

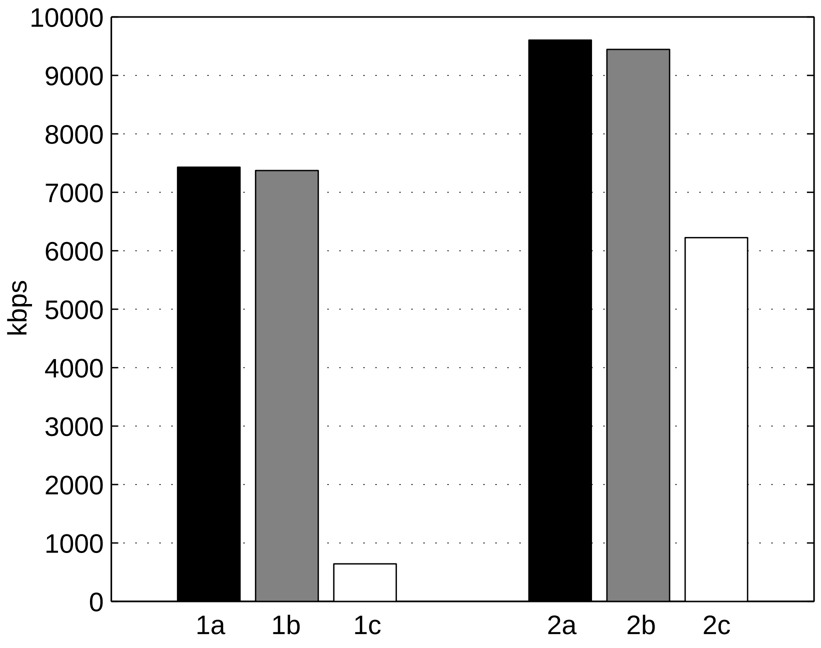
In the test cases 1a and 2a ALTQ was not enabled, so the tested router was an ordinary FreeBSD router. In the other test cases, ALTQ was enabled with a certain configuration. The test cases 1b and 2b used a simple configuration which in practice meant one profile with one filter in the input interface. On the contrary, in the test cases 1c and 2c the ALTQ configuration was complex and heavy - more than one hundred profiles and more than five thousand filters altogether. The CBQ configuration in the output interface was designed to be as simple as possible, and it was kept the same in all four ALTQ related test cases.

6.4.1.1 Throughput

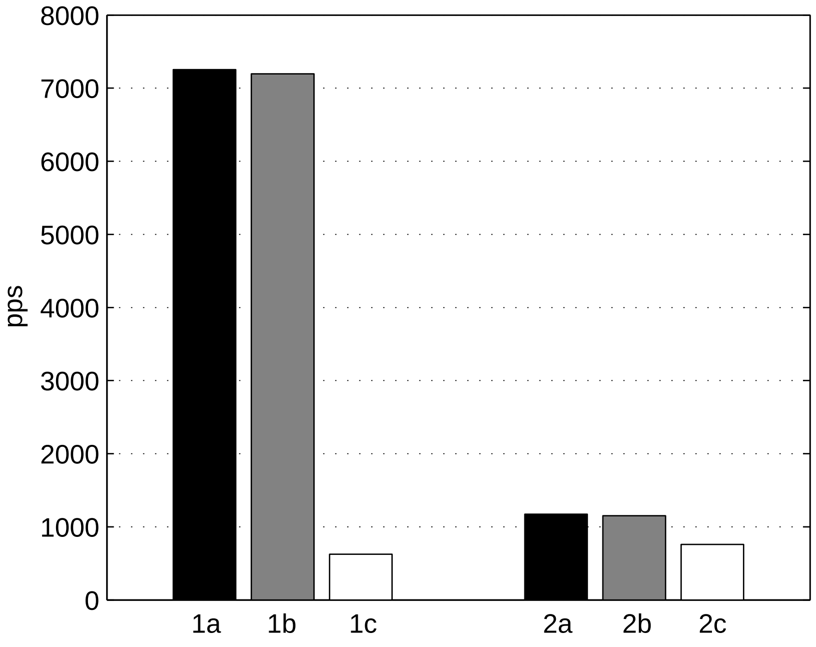
The throughput measurements were performed with SmartBits hardware and SmartFlow software. The results of each test round were taken from SmartFlow, and Figure 6.2(a) presents the calculated averages of the individual throughput values. Figure 6.2(b) shows throughput as packets per second.

As can be seen from Figure 6.2(a), the test cases 1a and 1b produced nearly the same throughput - slightly over 7 Mbps. Similar situation can be seen in the test cases 2a and 2b, although the actual throughput was higher than in the previous case, over 9 Mbps. Results in the cases 1c and 2c differed greatly from the earlier ones. In the case 2c, the throughput was only 6 Mbps, which is about 30% less than in the cases 2a and 2b. The case 1c was even worse, since the throughput was no more than 700 kbps. The result was only 10% of the earlier cases 1a and 1b.

The throughput presented as bit rate does not give a clear picture of the ef-



(a) Throughput presented as kbps



(b) Throughput presented as packets per second

Figure 6.2. *Throughput in different test cases*

fect of ALTQ on the results. Thus, Figure 6.2(b) illustrates the throughput calculated as packets per second and makes it possible to compare the performance of ALTQ in diverse test cases. In the test cases 1a, 1b, 2a and 2b, the results varied greatly between the different packet sizes, but in the test cases 1c and 2c the results were nearly the same.

The original assumptions were that the complex ALTQ configuration and small packet size of the traffic would decrease the throughput. As Figure 6.2(a) shows, both factors were alone enough to decrease the throughput notably, and together their influence was drastic. ALTQ itself did not seem to have any mentionable affect on the throughput results, but as can be seen from Figure 6.2(b), a heavy configuration changed the situation. The size of the effect depended on the configuration because of the ALTQ architecture. ALTQ examines every incoming packet to see if it matches to any of the filters. The filters are stored into a simple list structure which ALTQ scans from top to bottom until it finds the match. If there is no match, ALTQ goes through the list for nothing. ALTQ does not implement a sophisticated searching algorithm for filters which could prevent it from wasting time with huge filter lists. In the test, the ALTQ configuration and the traffic were chosen so that none of the packets matched to any of the filters. Thus, ALTQ searched the list in vain and consumed the maximum amount of processing time for every packet.

The results presented in Figure 6.2(b) indicate that the bottleneck in the packet processing is ALTQ, not the I/O hardware. The number of processed packets per second were nearly the same with both packet sizes when the ALTQ configuration was complex, thus the configuration plays more important role than the packet size. Searching the filter list causes the majority of the packet processing time and adds a processing delay independent of the packet size to every packet. In the other test cases, the number of processed packets per second varied greatly between the packet sizes. If Figure 6.2(a) and Figure 6.2(b) are compared to each other, it can be seen that in the cases 1a and 1b, ALTQ was doing its best but was not able to process the packets faster than at the rate of 7.5 Mbps. In the cases 2a and 2b, the highest possible throughput in a 10 Mbps link was reached with a small number of processed packets per second, because of the bigger packet size.

6.4.1.2 Latency

The latency measurements for the test cases 2a, 2b and 2c consisted of one set of measurement rounds. At first, the traffic load was set to 1 Mbps, and in each round it was increased by 1 Mbps. In the end, the traffic load was 10 Mbps which was the maximum wire capacity. Figure 6.3 illustrates the results.

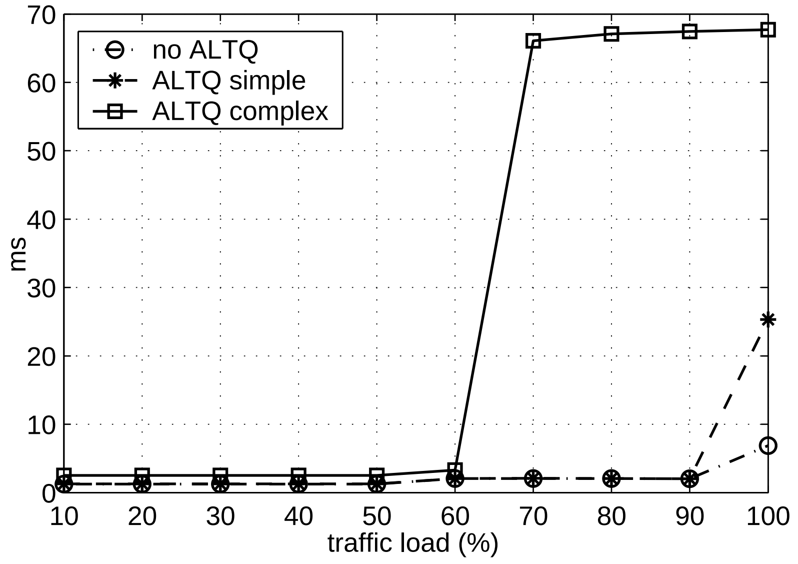


Figure 6.3. *Latency for traffic consisting of 1024-byte packets*

For the test case 1c, the test set of the previous latency test was not sufficient alone. The additional set was needed between 200 kbps and 1600 kbps, because it was assumed that the delay of the test case would increase more rapidly in that area. Since the results were supposed to be comparable inside the traffic case, the same additional measurements were performed also for the test cases 1a and 1b. At first, the traffic load was set to 200 kbps and it was increased with 200 kbps intervals round by round to 1600 kbps.

The results in Figure 6.4 show clearly, that the assumption of the quick increase of the latency in the case 1c was correct. At the traffic load of 1 Mbps (10%) the latency jumped from less than 2 ms to nearly 70 ms, after which it grew slowly. The next jump occured at 7 Mbps (70%), when the latency of the test case 2c increased from 3 ms to nearly 70 ms. After that, the growth of the latency was only moderate. The other four test cases were quite alike in the latency behaviour. The differences showed only at 10 Mbps traffic load, where the test case 1a reached 33 ms, 1b 25 ms, 2a 7 ms and 2b 25 ms. With the lower traffic loads, the latency values varied from 0.5 ms to 3 ms.

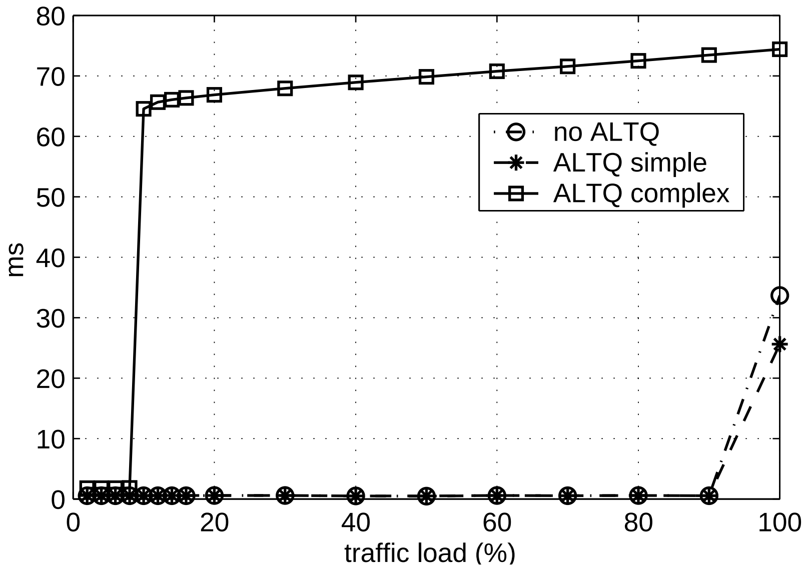


Figure 6.4. *Latency for traffic consisting of 128-byte packets*

The results in Figure 6.3 and Figure 6.4 show how closely related latency was to throughput. If the latency measurement results are compared to the throughput results, it can be noticed that the latency jumped upwards when the traffic load exceeded the maximum throughput of the test case. The reason for the behaviour is the same as with throughput: a complex ALTQ configuration and traffic consisting only of mismatching packets caused a huge processing time per packet. The packet queues filled up when the traffic load reached the maximum throughput and after that the router was forced to start dropping packets. Packet discarding increased the latency considerably which shows as a jump in the result graphs.

6.4.1.3 Packet Order

The purpose of the packet order measurements was to find out, how well the router could maintain the correct packet order. The test was run with an ordinary FreeBSD router and with an ALTQ/Policy Control Agent router. The assumption was that both routers could forward the packets in correct order. As expected, the results indicated no packet reordering for 1 Mbps – 10 Mbps traffic load with 128-byte and 1024-byte packets, unless there was also frame loss.

Packet reordering in the router without any frame loss may happen for various reasons which are not always errors [BPS99]. In a router built on a general PC hardware, reordering may still indicate a problem in packet forwarding more often than in a router with special hardware. However, both the ordinary FreeBSD router and the ALTQ/Policy Control Agent router were able to forward the packets in the correct order which gave a good starting point for further measurements.

6.4.2 Profile Modifications

The profile modification measurements consisted of three individual parts. In the first two, the performance of ALTQ and the local database management were measured, and in the third part, the target was to define the overall performance of the Policy Control Agent profile modification process.

In the ALTQ test cases, one profile was added to the ALTQ configuration and removed from there. The measurements were done first without traffic and then with 10 Mbps background traffic using 1024-byte packets. Similar traffic was used also in the local database measurements. The test case included adding one profile without filters and removing one profile first with zero, then with one and finally with two filters attached to it.

Test Case	**Without traffic (ms)**	**With traffic (ms)**
ALTQ		
Adding a profile	0.290	0.475
Removing a profile	0.297	0.513
Local databases		
Adding a profile	1.988	3.379
Removing a profile (no filters)	1.662	2.609
Removing a profile (1 filter)	2.131	3.490
Removing a profile (2 filters)	2.997	4.710

Table 6.4. *Performance times of ALTQ and the local database management in the profile modifications*

Both profile operations seem to take the same time inside ALTQ, as shown in Table 6.4. However, the local database management spent more time in adding one profile than removing one. It was clear that removing filters together with a profile would require more processing compared to the removal of a profile without filters. The measurement results confirmed the expectation. Background traffic was presumed to increase the operation time, and according to the results, it added an overhead of approximately 60%.

The overall performance of the profile modifications was examined with a series of profile additions and removals. The test set consisted of 1, 2, 3, 4, 20, 50, 100 and 200 profiles, and the measurements were done both without traffic and with 10 Mbps background traffic consisting of 128-byte packets. Initially, the ALTQ configuration did not contain any profiles on the traffic conditioning interface, and the measurements were performed in such order that the profiles were always added to the clean configuration. So in the results which are illustrated in Figure 6.5, there is no overhead caused by the underlying ALTQ configuration.

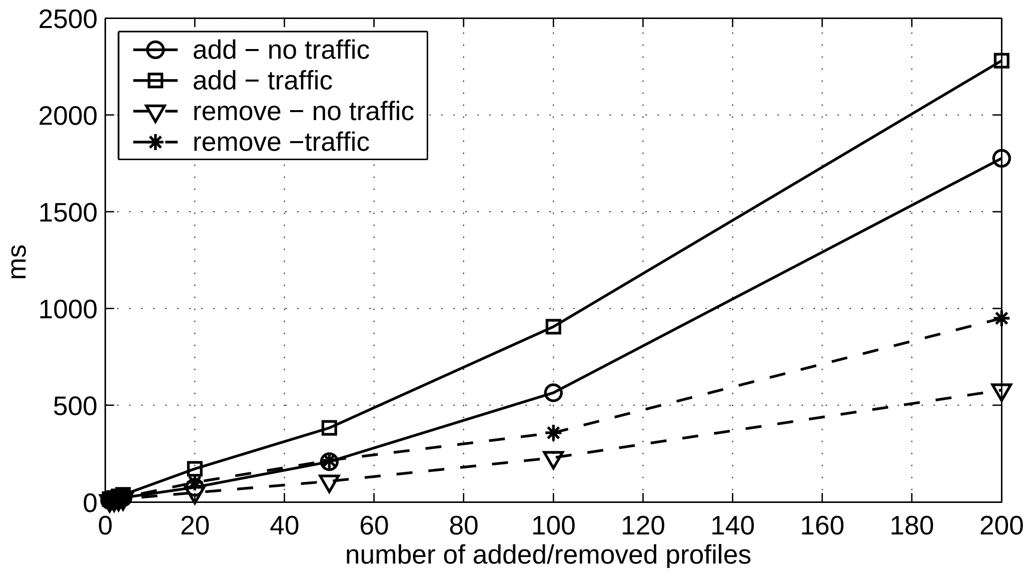


Figure 6.5. *Performance times of the profile modification loop*

The solid lines in the graph represent the profile adding operations and the dashed lines the profile removing operations. Although the time consumption of all operations increases nearly linearly, it took more time to add profiles than to remove them. For example, removing 200 profiles without traffic took less than 600 ms, while adding the same number of profiles took almost 1800 ms. Background traffic introduced varying overheads in different measurement points. When the number of added profiles was at most four, the overhead was roughly 70%. For removed profiles the overhead for the same measurement points was around 50%. At 20 added and removed profiles, there was a spike in the overhead - in both cases the overhead was over 100%. After that, the overhead decreased near or even below 50%.

The results show that in both cases the processing time grew nearly linearly.

In addition, with small number of profiles doubling the number of handled filters did not double the processing time. When the number of profiles grew, also the processing time increased somewhat rapidly. ALTQ used the same amount of time for adding and removing one profile, but adding a profile to the local databases takes more time than removing a profile. The reason could be that writing data to the disk is more time-consuming than removing data from the disk. The difference between the operation processing times cumulated when the number of profiles grew, so that finally profile adding was clearly more time-consuming operation than profile removing.

6.4.3 Filter Modifications

The filter modification measurements consisted of the same three parts as the profile modification measurements. They included ALTQ and the local database management performance measurements, as well as defining the overall filter modification capabilities.

In the ALTQ test cases, one filter was added to the ALTQ configuration and removed afterwards. As the filters always needed a profile or a class to be attached to, the initial ALTQ configuration included one profile in the traffic conditioning interface. Otherwise the configuration of the output interface was clean. The local database management was tested by letting it handle one filter which was to be added and one filter which was to be removed from the ALTQ configuration. In both measurements, the background traffic consisted of 1024-byte packets at the bit rate of 10 Mbps.

Test Case	**Without traffic (ms)**	**With traffic (ms)**
ALTQ		
Adding a filter	0.280	0.526
Removing a filter	0.272	0.520
Local databases		
Adding a filter	2.249	4.522
Removing a filter	1.699	3.645

Table 6.5. *Performance times of ALTQ and the local database management in the filter modifications*

The results in Table 6.5 show that the time spent by ALTQ in adding and removing one filter is nearly the same. Instead, the local database management needs more time for adding a filter than for removing one. The overhead caused by the background traffic is quite high in all cases. For the ALTQ measurement results it is about 90% and for the local database results as much as over 100%.

The performance of the Agent in the filter modifications was studied with the same series of adding and removing operations as in the profile modification measurements. The test set consisted of 1, 2, 3, 4, 20, 50, 100 and 200 profiles to add or to remove. The measurements were carried out so that the adding of the profiles was always done to the clean ALTQ configuration. All measurements were performed first without background traffic and later with 10 Mbps background traffic consisting of 128-byte packets. The behaviour of the operations is visualised in Figure 6.6, where the solid lines represent the filter adding and the dashed lines the filter removing operations.

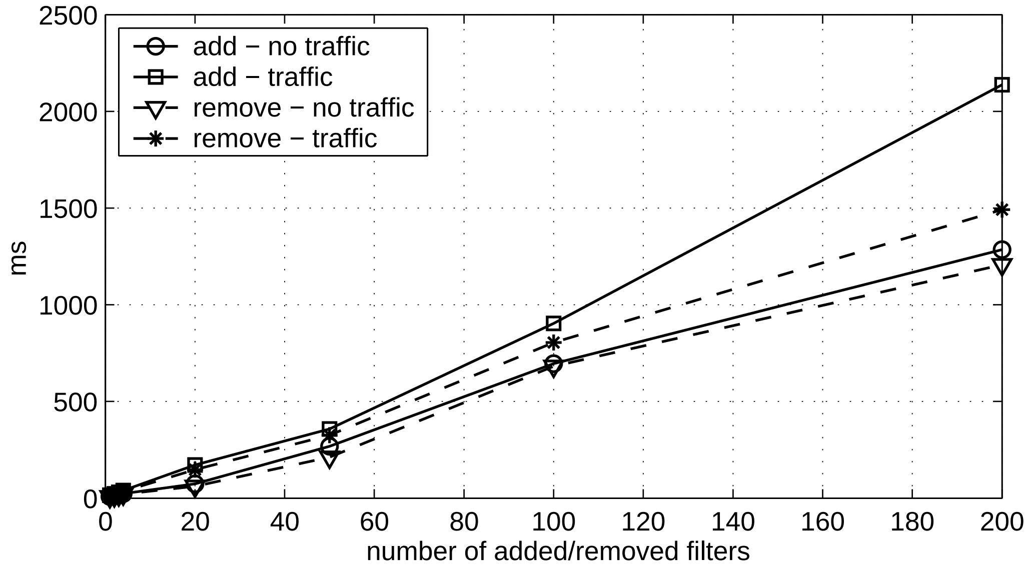


Figure 6.6. *Performance times of the filter modification loop*

As the figure shows, the filter adding and removing operations without the background traffic consumed nearly the same amount of time. It took 9 ms to add one filter and 8 ms to remove it. Further, adding 200 filters took 1280 ms and removing them 1200 ms. The time-usage of the operations increased almost linearly, even when there was background traffic. The traffic caused approximately 70% overhead to both operations when the number of filters was at most four. When 20 filters were added or removed, the overhead jumped to 130% but dropped way below 70% already in the next measurement point.

The processing times of the filter adding and removing operations were fairly similar throughout the test case. However, the overhead caused by the background traffic in the last measurement points was bigger for the adding operations than for the removing operations. The time-usage in the other cases increased less than linearly which was desired, because filter modifications are the most common operations of the Agent.

6.4.4 Class Modifications

The class modification measurements consisted only of the ALTQ and local database management tests. The overall performance test was omitted because the results would have been valid only for one certain configuration. The testing procedure had also other differences to the previous test cases. The performance of ALTQ was measured the same way as with the profiles and filters, but the local database measurements consisted of only one test. In the Agent implementation, the classes are static, and therefore only their parameters can be modified - the classes themselves cannot be added or removed. Thus, the database management performance was measured by changing the parameters of one class. All measurements were performed first without background traffic and later with 10 Mbps traffic load consisting of 1024-byte packets.

Test Case	**Without traffic (ms)**	**With traffic (ms)**
ALTQ		
Adding a class	0.281	0.417
Removing a class	0.248	0.428
Local databases		
Modifying a class	1.345	2.066

Table 6.6. *Performance times of ALTQ and the local database management in the class modifications*

The measurement results which are listed in Table 6.6 show that the class addition and removal operations in ALTQ took nearly the same amount of time. The background traffic created approximately 50% overhead to all results.

The Agent loop measurements were not suitable for the class modifications which impeded defining the performance of the operation. The time spent for a class modification operation is difficult to estimate, but the more child classes the class has and the more filters are present in the subtree, the longer the modification takes. Thus, if the frequency of the class modifications is expected to be higher than moderate, the class structure should be simple enough.

6.4.5 Policy Server Communication Module

The Policy Server Communication Module fetches the new ALTQ configuration parameters from the Policy Server and stores them into the modification database. As the performance of the communication between the Policy Server and the Communication Module was outside the scope of this study, only the database handling was examined.

In the measurements, one profile, one filter and one class were added into the modification database. The parameters were given to the Communication Module as static program parameters. The database was emptied between the measurements, so that it consisted only of the static components. Thus, the situation was the worst possible, because in each case, the Policy Server Communication Module had to create the whole directory hierarchy for the parameters. The results presented in Table 6.7 include the measurements done without background traffic and with 10 Mbps traffic load of 1024-byte packets.

Test Case	Without traffic (ms)	With traffic (ms)
Adding a profile	2.004	3.704
Adding a filter	2.002	3.182
Adding a class	1.753	3.054

Table 6.7. *Performance times for updating the modification database by the Policy Server Communication Module*

As can be seen from the results, the addition of a filter and a profile to the database took the same amount of time, but they were slower operations than the addition of a class. The difference can be explained with the varying number of parameters and complexity of the directory hierarchy. A class has only five parameters and its directory tree in the modification database is low. A profile has eight and a filter six parameters, and also their directory trees in the modification database are deep. The background traffic causes 60-80% increase to the operation execution time and evens up the time differences.

Storing one profile and one filter into the modification database took the same amount of time, while storing one class took a little less time than the other operations. The reason was assumed to be, as already mentioned, the difference in the number of parameters and in the database hierarchy. However, the test cases described the worst situation where the database contained only the static directories. Storing the next filter for the same profile as before would have taken less time, because most of the directory hierarchy would have already been created. The same applies for the profiles. As the reference model of the modification database in Figure 5.2 in Chapter 5 shows, each class is stored right under the static 'class' directory. Hence, the time that is spent storing one class into the database is approximately the time that is used for storing one profile or filter into the database when the necessary directory hierarchy already exists.

6.4.6 Latency Behaviour during Modifications

Filter modifications were used also in a supplementary latency test whose purpose was to define how the latency was affected by the modification operations. The test was performed with Adtech AX/4000 testing system. It was configured to create four 500 kbps traffic streams, each consisting of 128-byte TCP or UDP packets. One of the streams was always inside a profile defined in the ALTQ configuration and another stream was one always outside of all profiles. Two remaining streams were either outside or inside some profile, depending on the configured profile filters. The AX/4000 control software was configured to log the average latencies, packet loss and packet reordering per stream every 0.1 second and draw a real-time graph of them to the screen.

In the measurements, 1, 50, 250 and 750 filters were added and removed in turns under two profiles. The test case was divided into four separate parts based on the number of handled filters. The first three parts consisted of two filter addition and one filter removal cases. The fourth part consisted of three filter addition cases. The assumption was that the effect of the modifications would increase case by case, because it was expected to be dependent on the number of filters. Figure 6.7 and Figure 6.8 illustrate the latency of the stream which stayed always inside one profile in the ALTQ configuration. Figure 6.7 presents the case with 250 filters and Figure 6.8 the case with 750 filters.

There are six different events that can be pointed out from the latency graph in Figure 6.7. The events are marked with numbers to the picture. In cases 1) and 3), 250 filters were stored into the modification database by the Policy Server Communication Module. In cases 2) and 4), the filters were added to the ALTQ

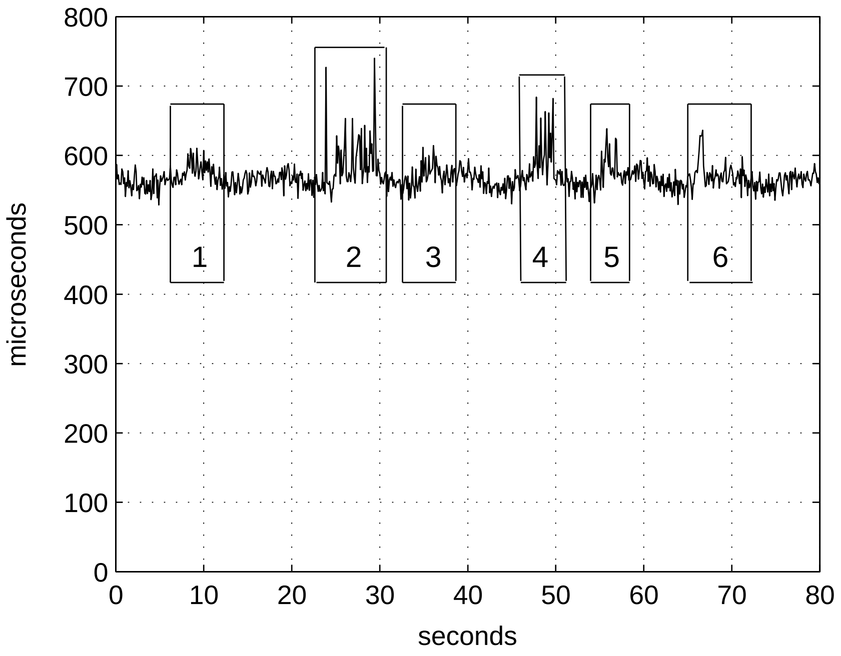


Figure 6.7. *Latency behaviour in the case of 250 modified filters*

configuration. In case 5), the filters that were to be removed from the configuration were stored into the modification database, and in case 6), the filters were removed from the ALTQ configuration. In the last two cases the number of filters was 500.

All six cases presented in Figure 6.7 required processor time, thus they slowed down the normal operation of ALTQ. The packets needed to be queued extra time which increased the packet latency. According to the results, the ALTQ operations disturbed the packet forwarding most. The adding operations increased the packet latency even more than the removing operations, although the number of filters in the latter case was twice the number of filters in the former case.

Figure 6.8 shows the behaviour of the latency when the number of handled filters was tripled from the previous measurements. The numbers in the figure refer to different operations. In case 1), 3) and 5) in Figure 6.8 the Policy Server Communication Module stored 750 filters into the modification database. In cases 2) and 4), the filters were added to the ALTQ configuration under two different profiles. In case 6), the filters were added to the ALTQ configuration under one profile and on top of the previously configured 750 filters.

Compared to the previous results, the modification database operations stood out clearly, and the influence of all operations stretched longer, as can be seen

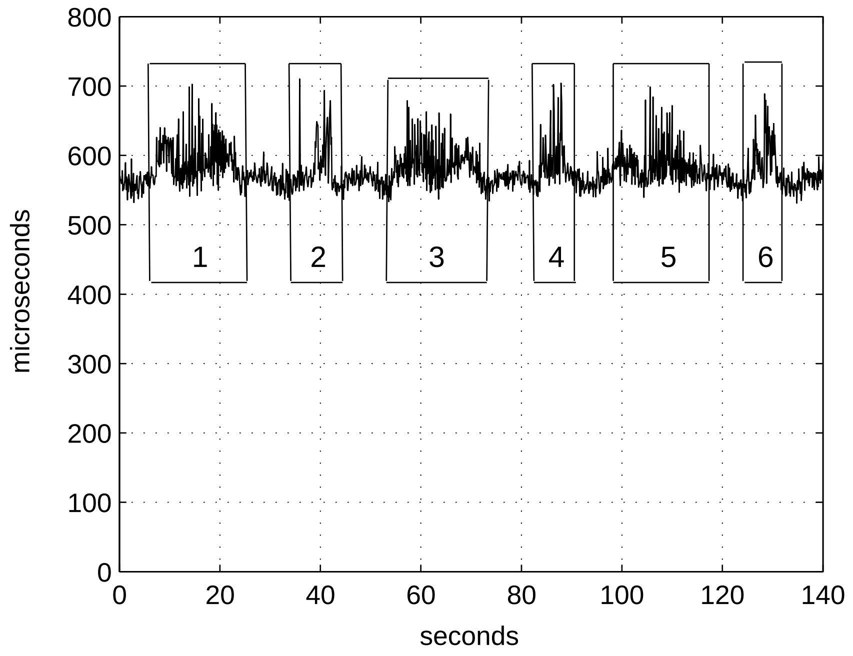


Figure 6.8. *Latency behaviour in the case of 750 modified filters*

in Figure 6.8. However, the maximum latency did not increase during the ALTQ operations despite the grown number of the filters. The maximum latency stayed during all operations at approximately 0,7 ms which was the maximum latency value also in the previous measurements. The same behaviour could be seen also in the latencies of the other three traffic streams. When compared the results of the stream presented above with the results of three other streams, no clear difference can be pointed out.

Packet loss and the number of reordered packets were measured throughout the test. Because the traffic load was low, it was assumed that the packet loss as well as the number of reordered packets would be small or even nonexistent. The results showed that the assumption was correct. No packet loss or reordering occured during the measurements.

The results indicate that the operations had an effect to the latency behaviour, as expected. The more there were operations, the more their influence showed in the results. In the test cases with one and 50 filters, the operations did not cause any clear effect on the latency behaviour. Still, the modification operations were allowed to use only a certain amount of the processor time. The increase of the maximum latency stopped at 0,7 ms and the total processing time of the operations started to grow, instead.

ALTQ was able to pass the measurements without dropping packets, although there were traffic flows whose classification changed on-the-fly because of the modifications. Thus, at least with moderate traffic loads, ALTQ manages to handle all incoming packets during the modification process.

6.4.7 Effect of the I/O Operations

The first target of the I/O operation measurements was to define the time-usage of the disk operations. It was expected that the disk operations would be the worst time-consumers in the modification operations. The second target was to define the difference between the real processing time of the operations and the 'wallclock' time used in the previous modification measurements.

The I/O operation measurements were performed for the profile and filter modification tests. The tests consisted of adding and removing 1, 20, 50, 100 and 200 profiles and filters, thus the number of measurement points was smaller than in the earlier test cases. The detailed timestamping function 'getrusage' was used for providing the time consuming of the Agent process. No background traffic was used in the measurements, because the aim was to define the basic performance values.

The results for the profile removing are illustrated in Figure 6.9. The user time shows the time spent in processing in the user mode, and the system time declares the time consumption of the disk operations. The total time is a calculated sum of the user and system times. The wallclock time which was measured already in the earlier test tells the complete execution time of the operations.

As can be seen from the results, the disk operations (system time) used always more time than the Agent software operations in the user mode. Both times increased linearly, and also the wallclock time grew nearly linearly. The difference between the total processing time and the wallclock time indicated clearly that the results from the earlier modification results had quite much air in them. The time between the lines was used by the operating system and daemon processes listed in Table 6.2 earlier in this chapter. Especially the ssh daemon consumed processing time, because the router was configured through an ssh connection. In practice, there were always one or two ssh sessions open. In addition, context switches between the user and kernel modes required processing time which was not included into the total processing time. As ALTQ operates in the kernel mode and the Agent in the

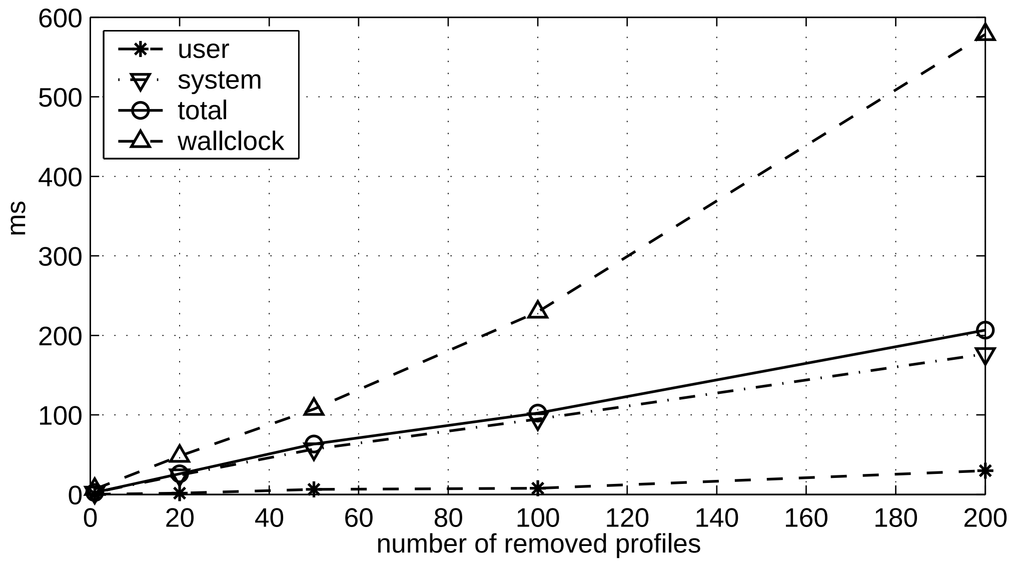


Figure 6.9. *Performance of the profile removing operation measured with detailed timestamping*

user mode, context switching happened often.

Table 6.8 summarises the total (actual) processing times and the wallclock times for the different Agent operations. The number of handled profiles or filters is presented on top of the column.

Test Case	**1**	**20**	**50**	**100**	**200**
Profile adding					
total	3.066	43.869	103.880	210.555	476.006
wallclock	8.538	76.897	208.013	564.970	1776.079
Profile removing					
total	2.349	25.874	63.269	102.323	206.442
wallclock	6.436	48.369	107.304	229.567	578.501
Filter adding					
total	4.088	43.045	98.137	220.732	468.226
wallclock	8.884	74.464	267.952	695.696	1284.938
Filter removing					
total	3.367	35.464	79.196	150.788	273.996
wallclock	8.024	62.030	212.276	681.670	1206.856

Table 6.8. *The wallclock and actual processing times (ms) of the operations*

The summarised results show the clear difference between the actual processing times and the wallclock times. The adding operation processing times and, on the other hand, the removing operation processing times are close to each other. This indicates the already mentioned difference between the time-consumption of writing to the disk and deleting from the disk. If the wallclock times of the different cases are compared to each other, it can be noticed that after all they do not differ greatly from each other. The only exception is the profile removing case which takes much less time than the other operations. The simplest assumption is that for some reason the profile removing does not cause as many context switches as the other operations. The profile removal and addition test were made in parallel, so it is not very likely that during the profile removal tests there were less processes than usual requiring processor time. However, because the sources of the extra time-usage were not studied closer, no actual conclusions can be made from the results.

The size of the unveiled differences between the actual processing times and the wallclock times seem to impugn the earlier measurement results. The actual processing times give a totally different impression of the performance of the Agent than the modification measurement results. However, either the new results do not tell the whole truth. The gap between the times include also operations like context switches which are part of the Agent process, so the difference is not plain overhead caused by other processes. Thus, when estimating the performance of the Agent, the total processing times can be used as basis, but they are not enough alone.

6.4.8 Conclusions from the Measurements

Due to the overhead caused by the external processes the modification measurement results cannot easily be generalised. The results are directly valid only for the system which contains the same hardware and software combination as the tested router. The results are still usable in the prototype network, because the edge routers are similar to the tested router. If the results are applied outside the prototype network, their limitations need to be taken into account. Although the results include some overhead caused by other processes, they do not show randomness. Naturally, the results of individual measurements vary from each other, but the difference stays small enough to maintain the trend. Also the throughput and delay measurement results suffer from the overhead, but the rest of the results are less problematic, either because of the nature of the measurement or the nature of the results.

One approach to the Agent performance is to contemplate the operation processing times in relation to the frequencies of the operations. The more common the operation is, the faster it should be completed. The class modifications are estimated to happen quite seldom, so in that sense the operation does not have to be extremely fast. On the other hand, when the class tree is in an inconsistent state, as it is during the modification operation, it affects all outgoing traffic on the interface. Hence, the performance of the class operation should be at least moderate. The profile operations are needed when a customer enters or leaves the network, so the frequency of the operations depends on the number of users accessing the network through the edge router. Good customer service presumes that when the customer enters the network, the network connection is available as soon as possible. This sets performance requirements also for the profile operations. The filter operations are expected to be the most common operations of the Agent. They are executed in certain time intervals, so one filter modification round may include even hundreds of filters. Thus, the operations need to be fast.

ALTQ seems to manage all operations nearly in an equal time (see Table 6.4, Table 6.5 and Table 6.6). Therefore, the differences between the performance times are created by the local database management and the rest of the Agent process. The current performance of the different operations is enough for testing purposes in the prototype network, but if and when the performance values need to be improved, there are several issues that can be reconsidered. The implementation of the local databases is one of them. If the parameters were stored into the virtual disk in the system memory, there would be no need for time-consuming physical disk operations. In addition, the number of context switches could be reduced to a minimum. The filter handling routines in ALTQ could also be improved. With a sophisticated searching algorithm the time that ALTQ uses in searching the matching filter could be considerably reduced. Another option could be arranging the filters to the list according to the popularity. Sorting the filters like this would already be possible, because in the prototype network, the filter generation process knows the traffic volumes for each filter. The sorting could also be performed by adding packet counters to the filters and sort the filters according to the gathered information. In general, improving the filter handling in ALTQ would affect not only the modification operations but also the throughput. Yet another point of view is to take a look at the router as a whole; how it could provide more processing resources for the ALTQ and Agent processes. The simplest action is to shut down all processes that are not necessarily needed. In our case, special hardware is not considered, because the edge routers of the prototype network are supposed to use general PC hardware.

Chapter 7

Conclusions

Differentiated Services (DiffServ) seems to be a promising QoS architecture, and it has already been deployed in the Internet to some extent. However, the original DiffServ architecture is not able to adapt to the changes in traffic load and types, and it does not support the users moving around and changing their network access point.

Our adaptive policy-based network prototype includes the support for mobility as well as the adaptability to the changes in the network conditions. The DiffServ functionalities in the edge routers were realised with an ALTQ traffic management software, introduced in Chapter 4, and our aim was to design a mechanism that would dynamically reconfigure ALTQ according to the traffic and customer changes. The control software implemented for the edge routers to communicate with ALTQ is called a Policy Control Agent, presented in detail in Chapter 5. The new ALTQ configuration parameters are calculated elsewhere in the network based on the traffic measurements and stored into the centralised database server called a Policy Server. The task of the Policy Control Agent is to retrieve the parameters from the Policy Server and reconfigure ALTQ.

As described in Chapter 5, the Policy Control Agent consists of four functional modules and two local databases containing the current ALTQ configuration parameters and the new, not yet installed parameters. The new parameters are fetched from the Policy Server by the Policy Server Communication Module which stores the parameters into the database. The Main Module reads the parameters from the database and passes them to the ALTQ Communication Module which uses the parameters to reconfigure ALTQ. The Local Database Management Module keeps the local databases up-to-date by removing the old entries and modifying the definition of the current configuration according to the changes. The triplet Main Module, ALTQ Communication Module and Local Database Management Module forms a combination which is controlled by the Main Module. The Policy Server Communication Module is an independent part of the Agent, as it is not commanded by any other module.

The current implementation of the Policy Control Agent is only the first working prototype, but it already includes all the necessary functionalities for operating in the network. However, before installing the Agent into the edge routers in the prototype network, a single ALTQ/Policy Control Agent router was put under several performance tests, as described in Chapter 6. The purpose of the tests was to find out if the Agent was suitable for further testing of the prototype network.

The majority of the test cases related to modification of the ALTQ configuration, but also general performance tests consisting of throughput, latency and packet order measurements were run. In the modification tests, all features of the Policy Control Agent were examined one at a time by adding and removing from one up to even several hundred profiles, profile filters and CBQ classes in turns. The results indicated that the overall performance of the Agent was good and the time-usage of the Agent process increased at most linearly in all test cases.

The general performance tests showed what happened to the packet forwarding performance with different ALTQ configurations and also during the configuration modifications. The results of the throughput test revealed that the greatest performance bottleneck was the packet processing in ALTQ. A complex and heavy ALTQ configuration together with a small packet size seemed to create a deadly mixture for the packet forwarding performance. The increase of the packet size increased the throughput measured in bits per second but not in packets per second. This observation proved that the bottleneck was really ALTQ, not the limitations of the I/O buses, for example. The latency measurement results illustrated clearly that the packet latency increased substantially during the ALTQ configuration modifications. The more there were changes, the more distinct were the effects.

In general, the results showed that there are some issues in the Policy Control Agent that need refinement, but no major performance problems caused by the Agent turned up. Thus, the Agent already suffices well for the network-wide testing at the moment, without any changes. The Agent implementation will be improved and optimised in the course of time according to the results from the previously described and forthcoming tests. The possible actions to increase the Agent and

ALTQ performance were discussed more thoroughly in Section 6.4.8.

The work with the Policy Control Agent is only beginning. The next step is to install the Agent to all edge routers in the prototype network and test the software in a real network environment using real traffic. The bits and pieces that were implemented independently are soon to be combined together to form the mechanism that controls and configures the prototype network. The final target with the Policy Control Agent is to integrate it as a seamless part of this mechanism.

Bibliography

[ABG$^+$01] D. Awduche, L. Berger, D. Gan, T. Li, V. Srinivasan, and G. Swallow. RSVP-TE: Extension to RSVP for LSP Tunnels. Technical report, IETF, Dec 2001. RFC 3209.

[ADF$^+$01] L. Andersson, P. Doolan, N. Feldman, A. Fredette, and B. Thomas. LDP Specification. Technical report, IETF, Jan 2001. RFC 3036.

[ADLY95] J.S. Ahn, P.B. Danzig, Z. Liu, and L. Yan. Evaluation of tcp vegas: emulation and experiment. *SIGCOMM Compututer Commununication Review*, 25(4):185–195, 1995.

[Alm99] W. Almesberger. Linux Network Traffic Control - Implementation Overview. In *5th Annual Linux Expo*, pages 153–164, May 1999.

[Alm02] W. Almesberger. Linux Traffic Control - Next Generation. In *9th International Linux System Technology Conference*, pages 95–103, Sep 2002.

[alt99a] altq.conf - altq configuration file. FreeBSD manual page, Sep 1999.

[alt99b] altqd - altq daemon. FreeBSD manual page, Sep 1999.

[BBC$^+$98] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An Architecture for Differentiated Services. Technical report, IETF, Dec 1998. RFC 2475.

[BBGS02] Y. Bernet, S. Blake, D. Grossman, and A. Smith. An informal management model for diffserv routers. Technical report, IETF, May 2002. RFC 3290.

[BCS94] R. Braden, D. Clark, and S. Shenker. Integrated Services in the Internet Architecture: an Overview. Technical report, IETF, Jun 1994. RFC 1633.

[BPS99] J.C.R. Bennett, C. Partridge, and N. Shectman. Packet reordering is not pathological network behavior. *IEEE/ACM Transactions on Networking*, 7(6):789–798, 1999.

[CF98] D. Clark and W. Fang. Explicit Allocation of Best-Effort Packet Delivery Service. *IEEE/ACM Transactions on Networking*, 6(4):362–373, Aug 1998.

[Cho98] K. Cho. A Framework for Alternate Queuing: Towards Traffic Management by PC-UNIX Based Routers. In *USENIX 1998 Annual Technical Conference*, pages 247–258, Jun 1998.

[Cho01] K. Cho. *The Design and Implementation of the ALTQ Traffic Management System*. PhD thesis, Keio University, 2001. 155 pages.

[Cho02] K. Cho. Altq tips, Feb 2002.

[Cho03] K. Cho. ALTQ: Alternate Queuing for BSD UNIX. Web page, May 2003. URL:<http://www.csl.sony.co.jp/person/kjc/programs.html>.

[Com] Spirent Communications. Ax/4000 broadband test system. Web page. URL:<http://www.spirentcom.com>.

[CS03] M. Carson and D. Santay. Nist net - a linux-based network emulation tool. *SIGCOMM Computer Communications Review*, 33(3):111–126, Jul 2003.

[DBCF95] N. Davies, G.S. Blair, K. Cheverst, and A. Friday. A network emulator to support the development of adaptive applications. Technical report, Department of Computing, Lancaster University, 1995.

[Dev03] M. Devera. Htb home. Web page, Dec 2003. URL:<http://luxik.cdi.cz/ devik/qos/htb/>.

[DKS89] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queuing algorithm. In *Symposium proceedings on Communications architectures & protocols*, pages 1–12, Sep 1989.

[FJ93] S. Floyd and V. Jacobson. Random Early Detection gateways for Congestion Avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, Aug 1993.

[FJ95] S. Floyd and V. Jacobson. Link Sharing and Resource Management Models for Packet Networks. *IEEE/ACM Transactions on Networking*, 3(4):365–386, Aug 1995.

[FKSK02] W. Feng, D. Kandlur, D. Saha, and Shin K. The BLUE Active Queue Management Algorithms. *IEEE/ACM Transactions on Networking*, 10(4):513–528, Aug 2002.

[FSN00] W. Fang, N. Seddigh, and B. Nandy. A Time Sliding Window Three Colour Marker (TSWTCM). Technical report, IETF, Jun 2000. RFC 2859.

[Gay96] M. Gaynor. Proactive Packet Dropping Methods for TCP Gateways. Draft, Nov 1996.

[HBWW99] J. Heinanen, F. Baker, W. Weiss, and J. Wroclawski. Assured Forwarding PHB Group. Technical report, IETF, Jun 1999. RFC 2597.

[HG99] J. Heinanen and R. Guerin. A Two Rate Three Color Marker. Technical report, IETF, Sep 1999. RFC 2698.

[HHK03] M. Handley, O. Hodson, and E Kohler. XORP: An Open Platform for Network Research. *SIGCOMM Computer Communications Review*, 33(1):53–57, Jan 2003.

[ipf00] ipfw - controlling utility for ip firewall and traffic shaper. BSD manual page, Feb 2000.

[IT94] ITU-T. Terms and definitions related to quality of service and network performance including dependability. Technical report, ITU-T, 1994. ITU Recommendation E.800.

[JNP99] V. Jacobson, K. Nicholas, and K. Poduri. An Expedited Forwarding PHB. Technical report, IETF, Jun 1999. RFC 2598.

[kam04] Kame project home page. Web page, Jan 2004. URL:<http://www.kame.net/>.

[Kil99] K. Kilkki. *Differentiated Services for the Internet.* Macmillan Technical Publishing, 1999. 356 pages. ISBN 1-57870-132-5.

[KMC$^+$00] E. Kohler, R. Morris, B. Chen, J. Jannotti, and F.R. Kaashoek. The Click Modular Router. *ACM Transactions on Computer Systems*, 18(4):263–297, Aug 2000.

[LC00] J. Liebeherr and N. Christin. Buffer management and scheduling for enhanced differentiated services. Technical Report CS-2000-24, University of Virginia, Aug 2000.

[Luo00] M. Luoma. *Simulation Studies of Differentiated Services Networks.* Helsinki University of Technology, 2000. Licentiate Thesis. 109 pages.

[oSN03] National Institute of Standards and Technology (NIST). NIST Net home page. Web page, Jan 2003. URL:<http://snad.ncsl.neist.gov/itg/nistnet/>.

[PD00] A.L. Peterson and B.S. Davie. *Computer Networks - A Systems Approach.* Morgan Kaufmann Publishers, 2nd edition, 2000. 747 pages. ISBN 1-55860-577-0.

[Riz97] L. Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *SIGCOMM Computer Communications Review*, 27(1):31–41, Jan 1997.

[Riz98] L. Rizzo. Dummynet and Forward Error Correction. In Proceedings of Freenix 98, 1998, Jun 1998.

[Riz02] L. Rizzo. Dummynet. Web page, Jun 2002. URL:<http://info.iet.unipi.it/ luigi/ip_dummynet/>.

[RVC01] E. Rosen, A. Viswanathan, and R. Callon. Multiprotocol Label Switching Architecture. Technical report, IETF, Jan 2001. RFC 3031.

[SHN00] I. Stoica, Zhang H., and T.S.E. Ng. A hierarchical fair service curve algorithm for link-sharing, real-time, and priority services. *IEEE/ACM Transactions on Networking*, 8(2):185–199, 2000.

[SPG97] S. Shenker, C. Partridge, and R. Guerin. Specification of Guaranteed Quality of Service. Technical report, IETF, Sep 1997. RFC 2212.

[Spi01] Spirent Communications. *SmartFlow User Guide*, 2001.

[tc01] Tc - show/manipulate traffic control settings. Linux manual pages, Dec 2001.

[tcn03] Project home page, Jun 2003. URL:<http://tcng.sourceforge.net/>.

[Wan01] Z. Wang. *Internet QoS: Architectures and Mechanisms for Quality of Service.* Morgan Kaufmann Publishers, 2001. 239 pages. ISBN 1-55860-608-4.

[Wro97] J. Wroclawski. Specification of the Controlled-Load Network Element Service. Technical report, IETF, Sep 1997. RFC 2211.

[XOR03] XORP Project. XORP Design Overview. Technical report, International Computer Science Institute, Jun 2003.

[ZBHJ97] L. Zhang, S. Berson, S. Herzog, and S. Jamin. Resource ReSerVation Protocol (RSVP) - Version 1 Functional Specification. Technical report, IETF, Sep 1997. RFC 2205.