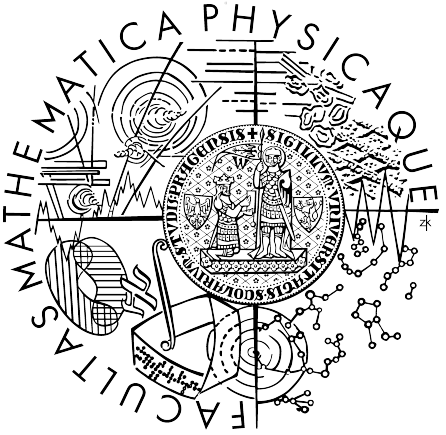
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Jan Poduška

UNIVERSAL FRAMEWORK FOR MANAGEMENT AND MONITORING OF LINUX TRAFFIC CONTROL FACILITIES

Department of Software Engineering

Supervisor: Ing. Lubomír Bulej

Study program: Computer Science, Software Systems

I would like to thank my advisor Lubomír Bulej for his valuable suggestions and observations, which contributed to the creation of this work. I would also like to thank my family and friends for their support in both my life and studies.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

I hereby declare that I am the sole author of this master thesis and have properly listed all the references used herein. I give permission for the public use of this thesis.

Prague, 15th December 2006

Jan Poduška

Contents

List of Figures

Název práce: Universal Framework for Management and Monitoring of Linux Traffic Control Facilities
Autor: Jan Poduška
Katedra: Katedra softwarového inženýrství
Vedoucí diplomové práce: Ing. Lubomír Bulej
e-mail vedoucího: bulej@nenya.ms.mff.cuni.cz

Abstrakt:

Traffic Control a Quality of Service (QoS) jsou termíny z oblasti počítačových sítí založených na přepojování paketů. Vztahují se k řídícím mechanismům, které mohou přiřadit různou prioritu různým datovým tokům nebo garantovat určité vlastnosti podle požadavků aplikace (šířka pásma, dostupnost, zpoždění). Garance Quality of Service je důležitá, pokud kapacita linky je omezená, obzvláště pro aplikace potřebující přenášet data v reálném čase (IP telefonie), protože takové aplikace většinou potřebují pevný datový tok a jsou citlivé na zpoždění.

Tato práce analyzuje současnou situaci v oblasti nastavování Traffic Control v Linuxu. Linuxový kernel poskytuje velké množství funkcí pro klasifikaci a plánování síťového provozu. Náležité konfigurování traffic control je ale složité, kvůli základnímu použitému konceptu, který je obtížně srozumitelný a velmi náročnému konfiguračnímu jazyku - "tc". Na základě analýzy současného stavu je navržen a implementován univerzální a rozšiřitelný framework jpQOS, který umožňuje snadné a přehledné nastavování a monitorování QoS pomocí jednotného grafického rozhraní.

Klíčová slova: Traffic Control, Quality of Service (QoS), Linux, správa, monitorování

Title: Universal Framework for Management and Monitoring of Linux Traffic Control Facilities
Author: Jan Poduška
Department: Department of Software Engineering
Supervisor: Ing. Lubomír Bulej
Supervisor's e-mail address bulej@nenya.ms.mff.cuni.cz

Abstract:

The terms "Traffic Control" and "Quality of Service" are used in the terminology of packet-switching based computer networks. They refer to control mechanisms, which can assign different priorities to different data flows or guarantee certain properties according to the requirements of an application (e.g. bandwidth, latency, accessibility). The properties guaranteed by the Quality of Service are especially important if the line capacity is limited, especially for applications that need to transmit data in real-time (IP telephony), since these types of applications usually require stable dataflow and are sensitive to delays.

This thesis analyzes the current situation in Traffic Control configuration in Linux. The Linux kernel offers a variety of functions for the classification and scheduling of network traffic. However the configuration of traffic control is proving challenging, since the basic concept used is difficult to understand and the configuration language – "tc"- is very difficult to use. Based on the analysis of the current situation, a universal and extensible framework - the jpQOS - was designed and implemented. This allows easy to use and well arranged configuration and monitoring of the QoS using a unified graphical user interface.

Keywords: traffic control, Quality of Service (QoS), Linux, management, monitoring

1 Introduction

The topic of this thesis is the universal framework for the management and monitoring of Linux traffic control facilities, which is a highly specific and complex sub-topic of computer networking.

1.1 Context and Motivation

The topic of Traffic Control mainly applies to computer networks based on packet switching (including the most widespread protocol – the IP). Data transmitted on the network between two computers are divided into parts called packets, which are transferred separately. The transmission route does not have to be a direct connection between the sender and the recipient, it can contain (and it usually does) many other network elements (switching nodes) which take care of routing (choice of the next route), packets are enqueued in buffer waiting for the line to become available and when it does they are sent. The amount of time a packet can spend in the switching node is not limited, it only depends on the intensity of the traffic. All the packets are serviced equally, the best effort principle is applied. The result of this is that it is never guaranteed, how much time it will take for the packet to arrive to its final recipient. This is not a problem for "classical" application run on computer networks, but for the upcoming multimedia services incorporating live transmission of sound and picture this represents a major problem (latency, irregularities in continuous deliveries).

A solution to this problem is the implementation of support for the Quality of Services (QoS). Under this term we understand the ability to guarantee certain data transfer properties – bandwidth, latency and jitter. The Quality of Service is ensured by different ways of treating different packets. For example in some queues on the switching nodes some packets receive priority above others. This also means that a mechanism is necessary that can divide the packets into categories – classify them. Other mechanisms are required as well, which are able to treat packets according to specific requirements (ordering, sending at a certain speed). To guarantee Quality of Service, it has to be installed on all of the switching nodes between the sender and the recipient of the data.

Linux does contain the infrastructure and the tools for configuring Traffic Control, which allows guaranteeing the required Quality of Service of data transfers on the network. The Linux kernel offers a large set of functions for classifying and scheduling network traffic. But properly configuring traffic control in Linux is very complicated, because of the obscure underlying theoretical concepts and a very difficult configuration language. Users must understand the variety of principles involved and the tools have a large number of adjustable parameters, the level of abstraction is very low. The documentation is completely insufficient and there are not many tools available for the configuration. Everyone who wants to approach such task has to spend a great deal of time studying various materials, configuring, etc.

As the Internet and computer networks are developing, more and more people are starting to get involved in the setting issues. Many local computer networks with shared Internet connection are developing, etc. As the requirements of the users for network service quality are increasing and the capacity of the line is restricted, there is a need to use the available bandwidths as effectively as possible. This can be achieved by setting up the QoS.

The problems with Traffic Control configuration on Linux and a growing demand for a well designed tool to facilitate the configuration were a large motivation for creating this thesis.

1.2 Goals of the Thesis

The aim of this study is to analyze the infrastructure provided by the Linux kernel for implementing integrated and differentiated services in IP networks and to evaluate the current approaches to management and configuration of traffic control primitives found in the Linux kernel. Furthermore, based on this analysis, design a universal and extensible framework for the configuration and management of Linux traffic control primitives and implement a tool for interactive design and run-time monitoring of user defined quality of service policies on a set of remote gateways. The program created should provide network administrators with and easy-to-use and efficient tool for setting up the QoS via a unified user interface.

1.3 Results

The outcome of this work is an application, which allows users to set up and monitor the network's QoS. The program is designed and implemented with respect to the above mentioned aims. It allows set up of the QoS not only locally, but also on remote gateways via the network. The set up is done via an unified interface for all QoS disciplines using a well arranged and intuitive GUI. The program can be easily extended to deal with new types of disciplines using configuration scripts and shared libraries.

1.4 Structure of the Text

The text is divided into 9 chapters. The first chapter briefly describes what the aim is and which ideas lead to the creation of such a thesis. The second chapter contains a thorough analysis of the current state, all the relevant associations are explained and the basic principles of traffic control in Linux are described. Its main problems and imperfections are also stated in this chapter. The third chapter deals with the suggested solutions to these problems and describes the main goals of this thesis, which, if accomplished successfully, would help to eliminate all the problems. These can be summed up into one main goal – the creation of the universal jpQOS framework. In the fourth chapter the architecture of the suggested solution and the structure of the proposed framework are described.

The fifth chapter contains a description of each part of the implemented framework and its main basic principles. In each part all of the possible ways of implementation were analyzed and the choice made is explained.

The sixth chapter describes the practical use of the jpQOS framework on a router, thus checking and proving the correctness of the suggested solution. In the seventh chapter we evaluate the achieved goals and suggest the future development of the framework and possible future improvements. In the eights chapter we compare the jpQOS to other similar programs. The last, ninth chapter contains a brief conclusion.

2 Analysis

2.1 Best effort

This work focuses on computer networks based on the most widespread protocol - IP. The IP transmission protocol is based on the "best effort" / no guaranteed result principle. Reliability, right ordering etc. were managed by higher level protocols – TCP. Originally the design did not contain any settings for the Quality of Service (QoS). This made the technology inexpensive and caused it to spread quickly. With the coming of multimedia applications the need for a certain set of guaranteed properties arose, like minimum data flow, latency and jitter. The first obvious solution was quantitative – an increase in bandwidth. However the resulting improvement was only statistical (it is unlikely for the requirements to decrease any further), because the best effort method cannot, due to its fundamental principles, ensure the required properties. The qualitative solution – the introduction of QoS support - does on the other meet the required properties. It replaces the "best effort" principle, but its implementation is complex and expensive. [PET06]

This work deals with the set up of the QoS, therefore it will focus on this qualitative solution in detail.

2.2 Quality of Service (QoS)

The term Quality of Service (QoS) on the network, can be defined as a control mechanism adjusting the following properties of the network:

- bandwidth
- availability
- latency
- jitter (latency variation)
- losses

There are two basic principles involved in defining the QoS:

- prioritization
- reservation

The principle behind **prioritization** is that different types of transmissions are assigned different priority levels and are treated differently: transmissions with higher priority receive better "quality of service" (via allocation of the sources) than lower priority transmissions. This cannot guarantee properties but does ensure different treatment of various transmissions. The main representative of such principle is An Architecture for Differentiated Services. [BLA98].

The principle behind **reservation** is, that it enables the allocation and reservation of the required sources exclusively for one transmission, which then uses them as needed. Reservation deals with bandwidth and switching capacity. This solution finally guarantees the required properties, however it also wastes the resources and is difficult to install. The main representative of this principle is An Architecture for Integrated Services. [BRA94].

2.3 Hierarchical link-sharing mechanism

The function of a link-sharing mechanism is to enable network elements (gateways) to control the distribution of bandwidth in response to user needs. One requirement for link-sharing could be to share bandwidth on a link between multiple organizations, where each organization wants to receive a guaranteed share of the link bandwidth during congestion, but where bandwidth that is not being used by one organization should be available to other organizations sharing the link. Another requirement for link-sharing could be to share bandwidth on a link between different protocol families (e.g., IP and SNA), where controlled link-sharing is desired because the different protocol families have different responses to congestion. A third example for link-sharing is to share bandwidth on a link between different traffic types, such as telnet, ftp, or real-time audio and video.

The various requirements for link-sharing naturally lead to requirement for **hierarchical link-sharing**. For example, the bandwidth on a link might be shared between multiple agencies, and each agency might want to share its allocated bandwidth between several traffic types.

The link-sharing structure specifies the desired policy in terms of the division of bandwidth for a particular link. To illustrate on an example, see the hierarchical link-sharing structure in Figure 1. The telnet and ftp classes are examples of leaf classes in the link-sharing structure, and the aggregated Agency class is an interior class.

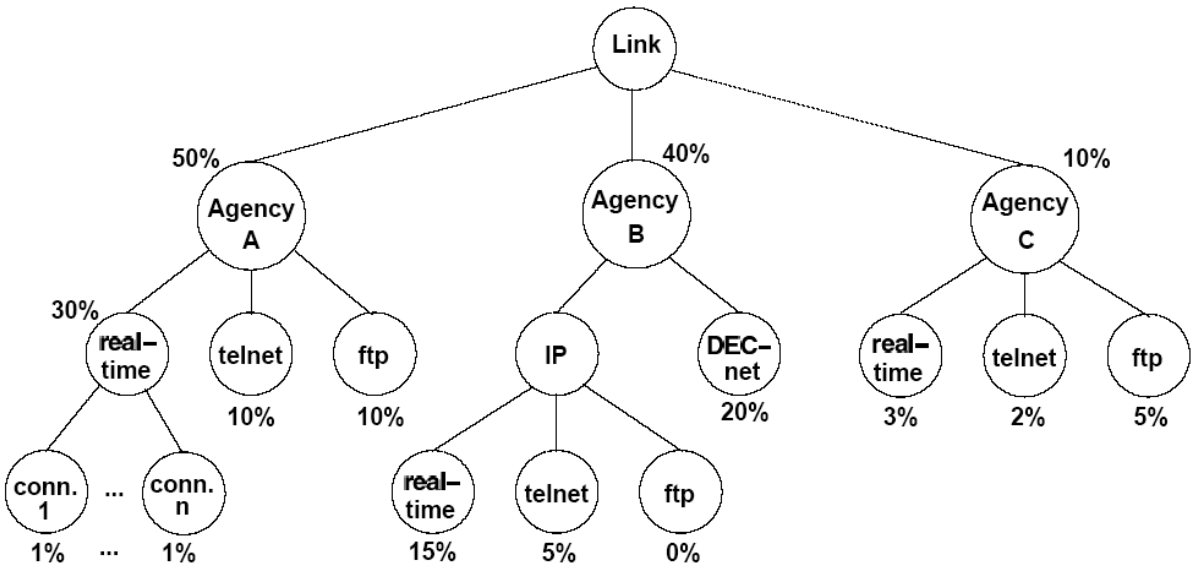


Figure 1: Hierarchical link-sharing structure

The first link-sharing goal is that each class with sufficient demand should be able to receive roughly its allocated bandwidth. As a consequence of this link-sharing goal, in times of congestion some classes might be restricted to their link-sharing bandwidth.

A secondary link-sharing goal is that when some class is not using its allocated bandwidth, the distribution of the "excess" bandwidth among the others classes should not be arbitrary, but should follow some appropriate set of guidelines.

All arriving packets at the gateway are assigned to one of the leaf classes; the interior classes are used to designate guidelines about how “excess” bandwidth should be allocated. [FLO95]

2.4 Analysis of the infrastructure provided by the Linux kernel

The topic of this thesis is the QoS configuration on Linux systems, hence the support for traffic control that is already provided by the Linux kernel needs to be analyzed first. It is necessary to determine how it works and what exactly does it provide for the user.

The term “traffic control” can be defined as the management of data flow on the network, including both incoming and, more importantly, outgoing messages. The data flow management is done for each network device separately.

Detailed information about processing of network data:

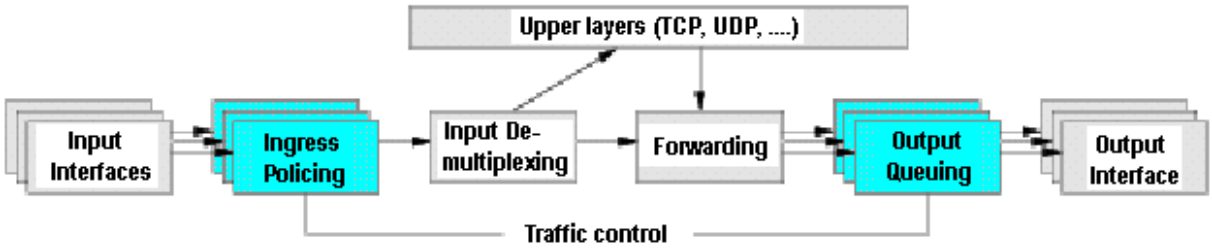


Figure 2: Processing of network data

Figure 2 shows in detail how the kernel processes data received from the network, and how it generates new data to be sent on the network. Packets arrive via an input interface, where they can be policed. Policing discards undesirable packets, for example if traffic is arriving too fast. After policing, packets are either directly forwarded to the network (e.g. onto a different interface, if the machine is acting as a router or a bridge), or they are passed on to higher layers in the protocol stack (e.g. to a transport protocol, like UDP or TCP) for further processing. Those higher layers may also generate data on their own and hand it to the lower layers for tasks like encapsulation, routing, and eventually transmission. "Forwarding" includes the selection of the output interface, the selection of the next hop and encapsulation among others. Once all this is done, packets are queued on the respective output interface. This is the second point where traffic control intervenes. Traffic control can, among other things, decide if packets are queued or if they are dropped (e.g. if the queue has reached some length limit, or if the traffic exceeds some rate limit), it can decide in which order packets are sent (e.g. to give priority to certain flows), it can delay the sending of packets (e.g. to limit the rate of outbound traffic) etc. Once traffic control has released a packet for sending, the device driver picks it up and dispatches it onto the network.

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

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

Each network device has a queuing discipline associated with it, which controls how packets enqueued on that device are treated. The default queuing discipline is very simple - it consists only of a single queue, which works on the FIFO principle. All packets are stored in the order in which they have been enqueued. They are emptied as fast as the respective device can send them. See Figure 3 for such a queuing discipline.

Figure 3: A simple FIFO queuing discipline

More complex queuing disciplines can be used. They may use filters to classify packets, i.e. to assign them to different classes. Each class is processed in a different way. One class can have higher priority than others, higher transfer rate, etc. An example of such a queuing discipline is on figure 4. Note that multiple filters may classify packets to the same class.

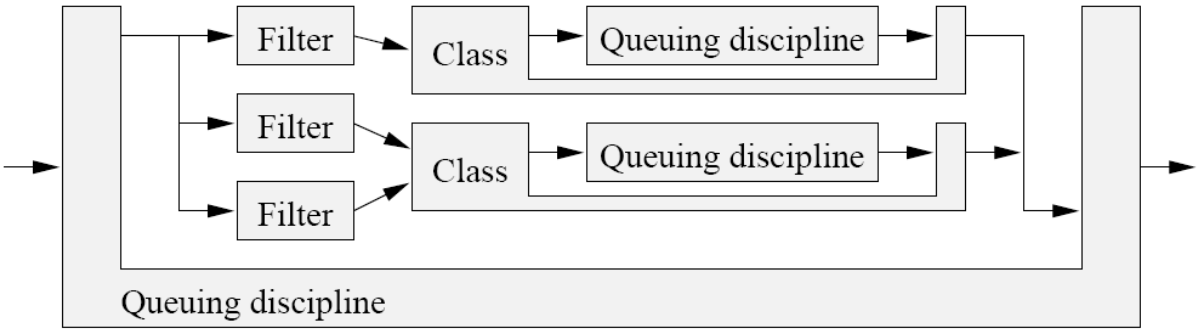


Figure 4: A simple queuing discipline with multiple classes

Queuing disciplines and classes are tied together. The presence of classes and their semantics are fundamental properties of the queuing discipline. In contrast to that, filters can be combined arbitrarily with queuing disciplines and classes. The system is even more flexible. Classes can choose how to store their packets. It means that in a class can be another queuing discipline, which takes care of storing the classes packets. That queuing discipline can be arbitrarily chosen from the set of available queuing disciplines, and it may also have classes, which in turn use queuing disciplines, etc.

Figure 5 shows an example of this flexibility application. There is a queuing discipline with two classes of different priorities. Packets which are selected by the filter go to the high-priority class, while all other packets go to the low-priority class. If there are packets in the high-priority queue, they are sent before packets in the low-priority queue. In order to prevent high-priority traffic from starving low-priority traffic, a rate limit of traffic with high priority is used. To specify, a token bucket filter (TBF) queuing discipline is used for rate limit of high-priority class to 1 Mbps and FIFO queuing discipline is used for low-priority packets. Note that there are better ways to accomplish this example, for example by using class-based queuing (CBQ).

Packets are enqueued in the following way: the enqueue function of a queuing discipline is called. It runs one filter after the other until one of them indicates a match. It then queues the packet for the corresponding class, which usually means invoking the enqueue

function of the queuing discipline in that class. Packets which do not match any of the filters are typically stored in some default class.

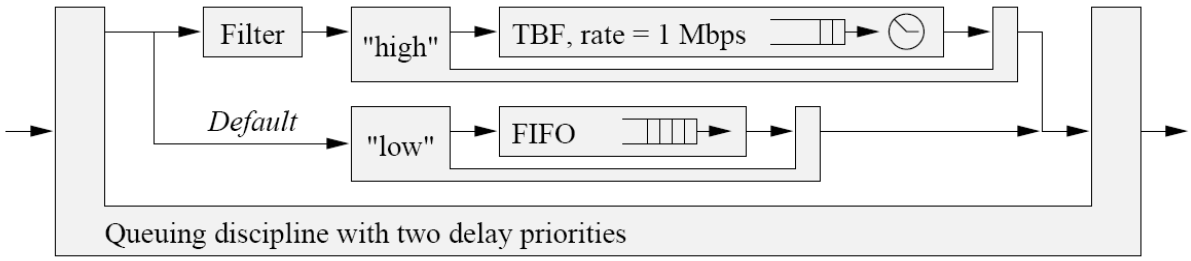


Figure 5: Combination of queuing disciplines

Typically, each class has one queue, but it is possible for several classes to share the same queue or that a queuing discipline only uses a single queue for all its classes. Note that packets do not carry any explicit indication of class to which they were classified. Queuing disciplines that change per-class information when dequeuing packets will therefore not work properly if the classes queues are shared, unless they are able either to repeat the classification or to pass the classification result from enqueue to dequeue.

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

For more detailed information about Linux traffic control implementation see [ALM98].

2.5 Classless queuing disciplines

Linux provides a support for many types of queuing disciplines. The common ones are briefly described in the following part to illustrate, how QoS work on Linux. The category of classless queuing disciplines means that they contain only one qdisc[1] and are not divided into classes.

2.5.1 FIFO queuing discipline

First-in, first-out (FIFO) queuing is the most basic queue scheduling discipline. In FIFO queuing, all packets are treated equally by placing them into a single queue, and then servicing them in the same order that they were placed into the queue. FIFO queuing is also referred to as First-come, first-served (FCFS) queuing. FIFO is the default queuing discipline used by Linux. In case no qdisc is specified, Linux assembles its interfaces with this type of queue.

[1]queuing discipline

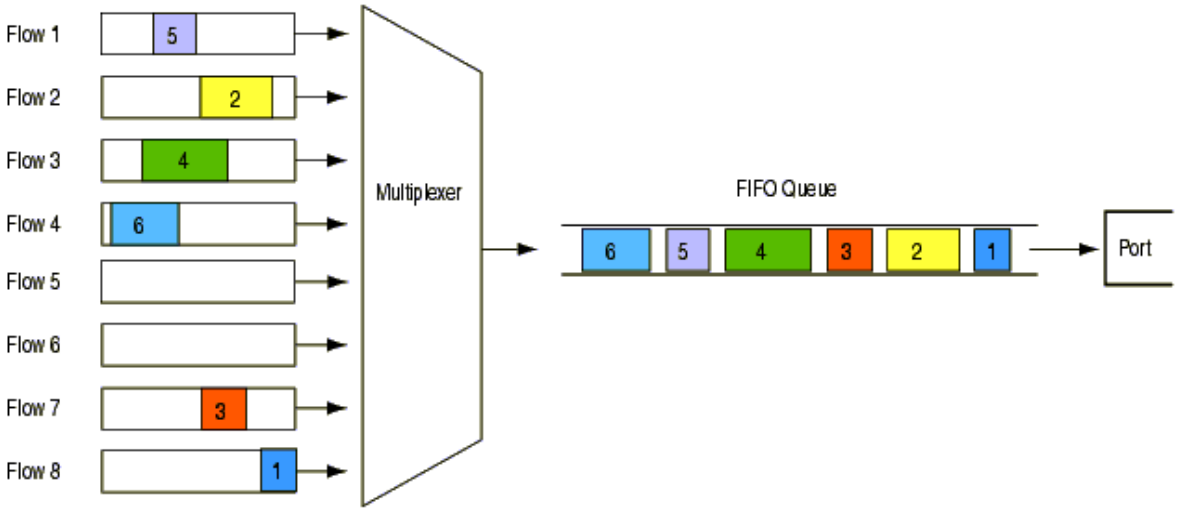


Figure 6: FIFO queuing discipline

2.5.2 TBF queuing discipline

The Token Bucket Filter (TBF) is a simple queue that only passes packets arriving at a rate which is not exceeding some administratively set rate, with the possibility to allow short bursts in excess of this rate.

TBF is very precise, network and processor friendly. It should be the first choice if the user simply wants an interface to slow down. The TBF implementation consists of a buffer (bucket), constantly filled with some virtual pieces of information called tokens, at a specific rate (token rate). The most important parameter of the bucket is its size that is the number of tokens it can store.

Each arriving token collects one incoming data packet from the data queue and is then deleted from the bucket. Associating this algorithm with the two flows -- token and data, gives three possible scenarios:

- The data arrives in TBF at a rate that is equal to the rate of incoming tokens. In this case each incoming packet has its matching token and passes the queue without delay.
- The data arrives in TBF at a rate that is smaller than the token rate. Only a part of the tokens are deleted at output of each data packet that is sent out the queue, so the tokens accumulate, up to the bucket size. The unused tokens can then be used to send data at a speed that is exceeding the standard token rate, in case short data bursts occur.
- The data arrives in TBF at a rate bigger than the token rate. This means that the bucket will soon be devoid of tokens, which causes the TBF to throttle itself for a while. This is called an "overlimit situation". If packets keep coming in, packets will start to get dropped.

The last scenario is very important, because it allows to administratively shape the bandwidth available to data that is passing the filter. The accumulation of tokens allows a short burst of overlimit data to be still passed without loss, but any lasting overload will cause packets to be constantly delayed, and then dropped.

2.5.3 SFQ queuing discipline

Stochastic Fairness Queuing (SFQ) ensure that each flow has fair access to network resources and to prevent a bursty flow from consuming more than its fair share of output port bandwidth. In SFQ, packets are first classified into flows by the system and then assigned to a queue that is specifically dedicated to that flow. Queues are then serviced one packet at a time in round-robin order.

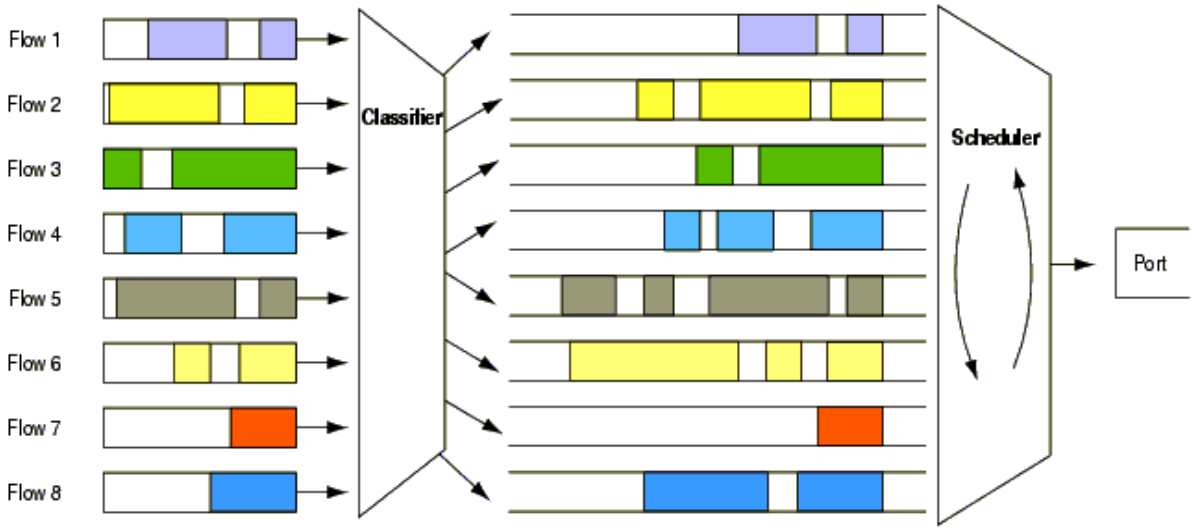


Figure 7: SFQ queuing discipline

2.6 Classful queuing disciplines

To support the Differentiated Services described in [BLA98] and to support the hierarchical link-sharing described in Link-sharing and Resource Management Models for Packet Networks [FLO95], Linux contains classful queuing disciplines[2] as well.

The link-sharing goals can be summarized as follows:

1. Each interior or leaf class should receive roughly its allocated link-sharing bandwidth over appropriate time intervals, given sufficient demand.
2. If all leaf and interior classes with sufficient demand have received at least their allocated link-sharing bandwidth, the distribution of any `excess' bandwidth should not be arbitrary, but should follow some set of reasonable guidelines.

The fundamentals of the main classful queuing disciplines in Linux:

Packet Flow within classful qdiscs & classes

When traffic enters a classful qdisc, it needs to be sent to any of the classes within - it needs to be 'classified'. To determine what to do with a packet, the so called 'filters' are consulted. The filters attached to that qdisc then return with a decision, and the qdisc uses this to enqueue the packet into one of the classes. Each subclass may try other filters to see if further instructions apply. If not, the class enqueues the packet to the qdisc it contains. Besides containing other qdiscs, most classful qdiscs also perform shaping. This is useful to perform both packet scheduling (with SFQ, for example) and rate control.

[2]queuing discipline containing structure of classes

The qdisc family: roots, handles, siblings and parents

Each interface has one egress 'root qdisc'. By default, it is the earlier mentioned classless fifo queueing discipline. Each qdisc and class is assigned a handle, which can be used by later configuration statements to refer to that qdisc. Besides an egress qdisc, an interface may also have an ingress qdisc , which polices traffic coming in. The handles of these qdiscs consist of two parts, a major number and a minor number: <major>:<minor>. It is customary to name the root qdisc '1:', which is equal to '1:0'. The minor number of a qdisc is always 0. Classes need to have the same major number as their parent. This major number must be unique within a egress or ingress setup. The minor number must be unique within a qdisc and his classes.

How filters are used to classify traffic

Packets get enqueued and dequeued at the root qdisc, which is the only part the kernel interacts width. To determine which class shall process a packet, the so-called 'classifier chain' is called each time a choice needs to be made. This chain consists of all filters attached to the classful qdisc that needs to decide. When enqueueing a packet, at each branch the filter chain is consulted for a relevant instruction. A typical setup may have a filter in 1: that directs a packet to 1:12 and a filter on 12: that sends the packet to 12:2. Packet can't be filtered upwards. Packets are only enqueued downwards. When they are dequeued, they go up towards the interface again. They do not fall from the end of the tree to the network adaptor directly.

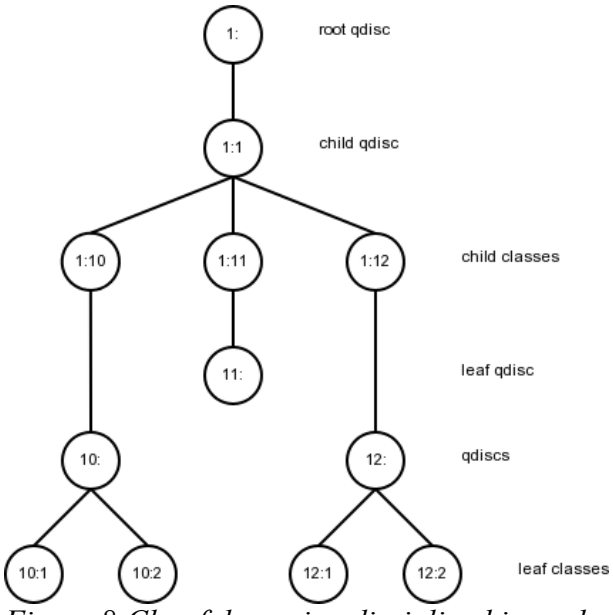


Figure 8:Classful queuing discipline hierarchy

How packets are dequeued to the hardware:

When the kernel decides that it needs to extract packets and send them to the interface, the root qdisc 1: gets a dequeue request, which is passed to 1:1, which is in turn passed to 10:, 11: and 12:, each of which queries its siblings, and tries to dequeue from them. Thus, nested classes only talk to their parent qdiscs, never to an interface. Only the root qdisc gets dequeued by the kernel. To conclude, classes never get dequeued faster than their parents allow. And this is exactly what is desired: this way the SFQ can be in an inner class, which doesn't do any shaping, only scheduling, and other outer qdisc can be used to do the shaping.

2.6.1 PRIO queuing discipline

Priority queuing (PQ) is the basis for a class of queue scheduling algorithms that are designed to provide a relatively simple method of supporting differentiated service classes. In classic PQ, packets are first classified by the system and then placed into different priority queues. Packets are scheduled from the head of a given queue only if all queues of higher priority are empty. Within each of the priority queues, packets are scheduled in FIFO order.

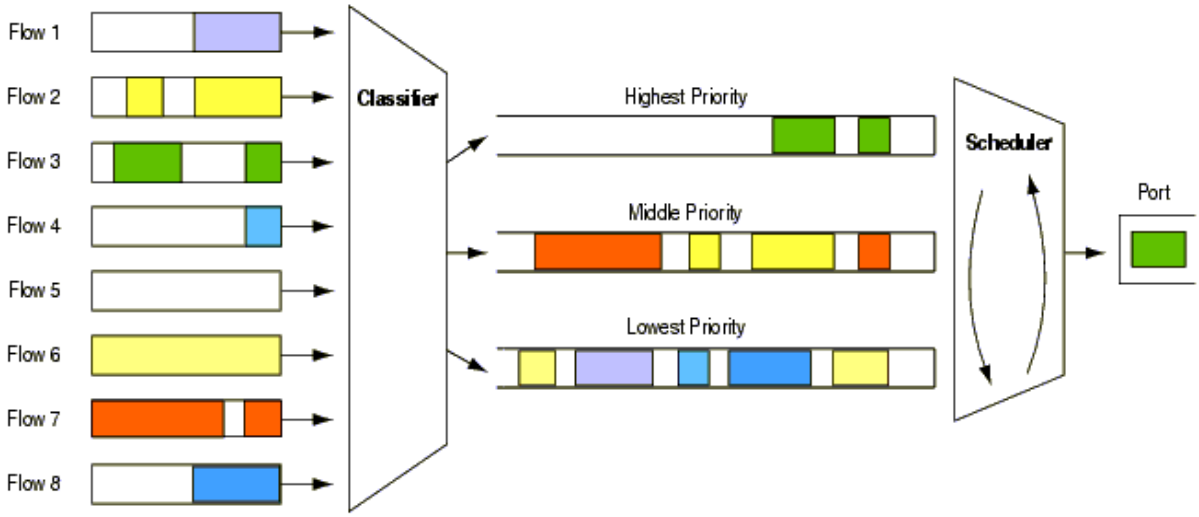


Figure 9: PRIO queuing discipline

2.6.2 CBQ queuing discipline

Class Based Queuing (CBQ) is a well known classful queuing discipline. Classful qdiscs are very useful if there are different kinds of traffic which should be treated differently. CBQ is the most complex qdisc available, the most hyped, the least understood, and probably the most complicated one to correctly set up and use effectively. CBQ is also a shaper, but this function of the CBQ queuing discipline doesn't work well, thus it is better to use HTB for this purpose.

2.6.3 HTB queuing discipline

The Hierarchical Token Bucket (HTB), like CBQ, is a classful queuing discipline. It uses the Token Bucket filter for shaping. It fulfills the concept of hierachical link-sharing. For more detailed information refer to the HTB Linux queuing discipline manual [DEV02b] .

2.7 Other queuing disciplines

Linux supports many other interesting queuing disciplines, for example RED, GRED, DSMARK or HFSC. More detailed overview of the above mentioned disciplines can be found in the Linux Advanced Routing and Traffic Control [LAR04], Differentiated Service on Linux HOWTO [BAL03] or other documents mentioned in the references part of this thesis.

2.8 Filters

The Linux kernel includes support for many types of filters. It is possible to classify packets by almost any property. The most commonly used are the following:

- source / destination address
- source / destination port
- ip protocol (tcp, udp, icmp, gre, ipsec,...)
- fw mark (marks from routing, firewall)
- TOS filed (type of service)

The most frequently used filters are:

- u32 – the strongest and most complex filter, it is able to classify by the comparison of any bit of the packet (including higher level comparison using addresses, ports, protocols, etc.)
- fw – classifies using firewall flags
- route – classifies using routing decisions
- rsvp – classifier for the Integrated Services
- tcindex – classifier for the Differentiated Service (DSMARK)

2.9 How to set up QoS in Linux (analysis of the current state)

This chapter will analyze in detail how to work with traffic control from the users (administrators) point of view, how to adjust its behaviour and settings.

2.9.1 Netlink socket

Each of the components (queuing disciplines, classes, filters) is implemented as a kernel module. To allow the user to set their parameters according to his requirements, there has to be a user-space – kernel interface. This interface takes on the form of the netlink socket in Linux. It is a socket similar to the standard socket used for network communication. The netlink socket allows two way communication. It can be used to transfer messages that configure the individual traffic control components or to read messages describing the status of the component. The messages have a predefined structure, i.e. they have a given set of options and their parameters that can be set. In addition, each component has its individual parameters, which get transmitted via the message and the are dealt with by the individual compoment, depending on their meaning (which is only known by the target component). More detailed information can be found in Linux - Advanced Networking Overview [RAD99] or Netlink Sockets – Overview [DHA99]. Figure 10 shows example of the netlink socket interface.

To adjust traffic control properties, a modified netlink socket is used – the rtnetlink (route table netlink). Figure 11 shows the structure of rtnetlink message. The predefined general structure of all the messages flowing through the rtnetlink interface can be seen on the picture. Every component sends its specific parameters in the “Attributes“ section of the message. These attributes are marked by a known identifier (each discipline has its own) followed by the actual data of the attribute, usually stored in C structure. Each component defines its own C structure, which allows any type of parameter to be transferred. However this makes it very hard to create a single unified interface for all the components.

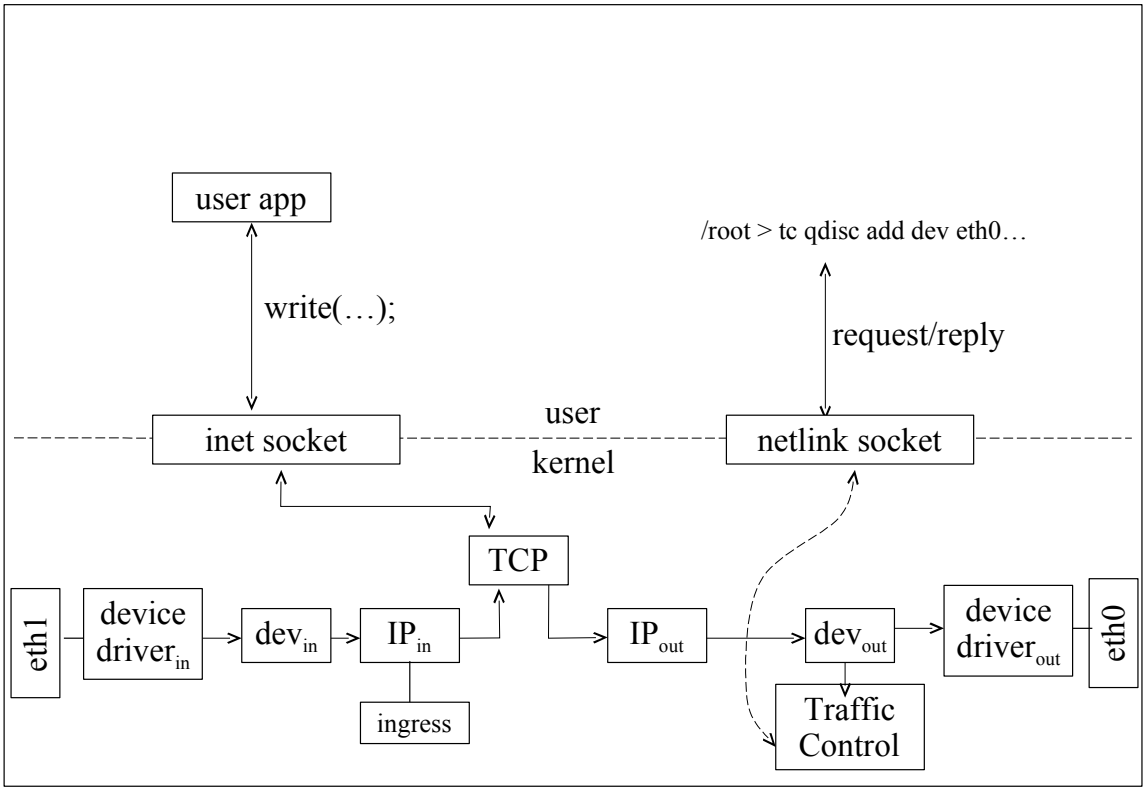


Figure 10: Netlink socket interface

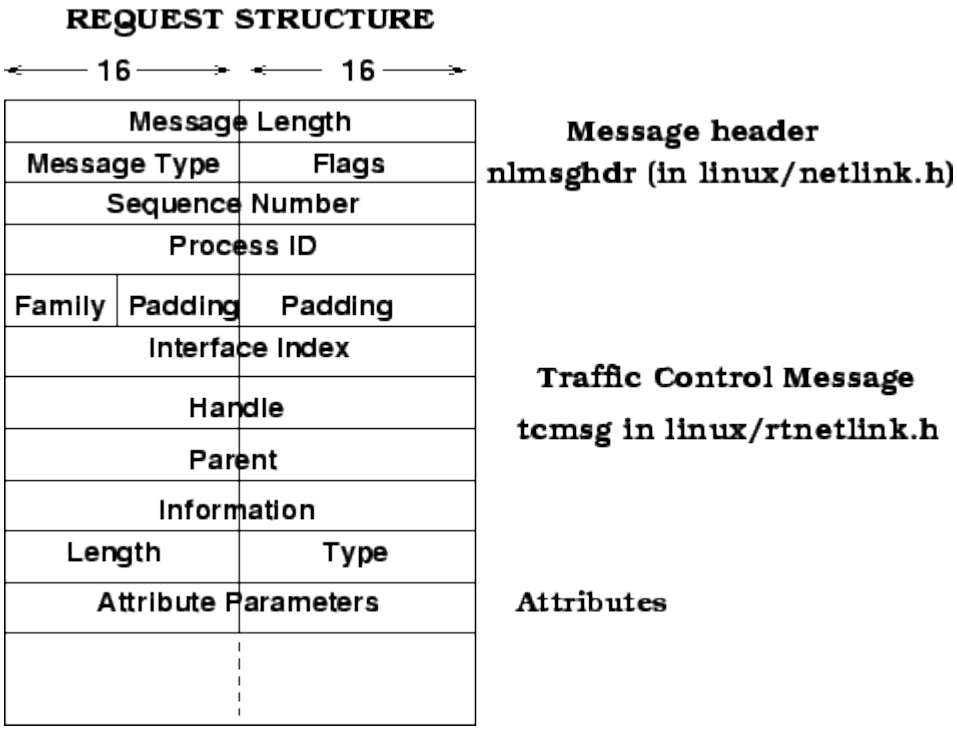


Figure 11: Rtnetlink message structure

An example of TBF discipline message [OLS02]:

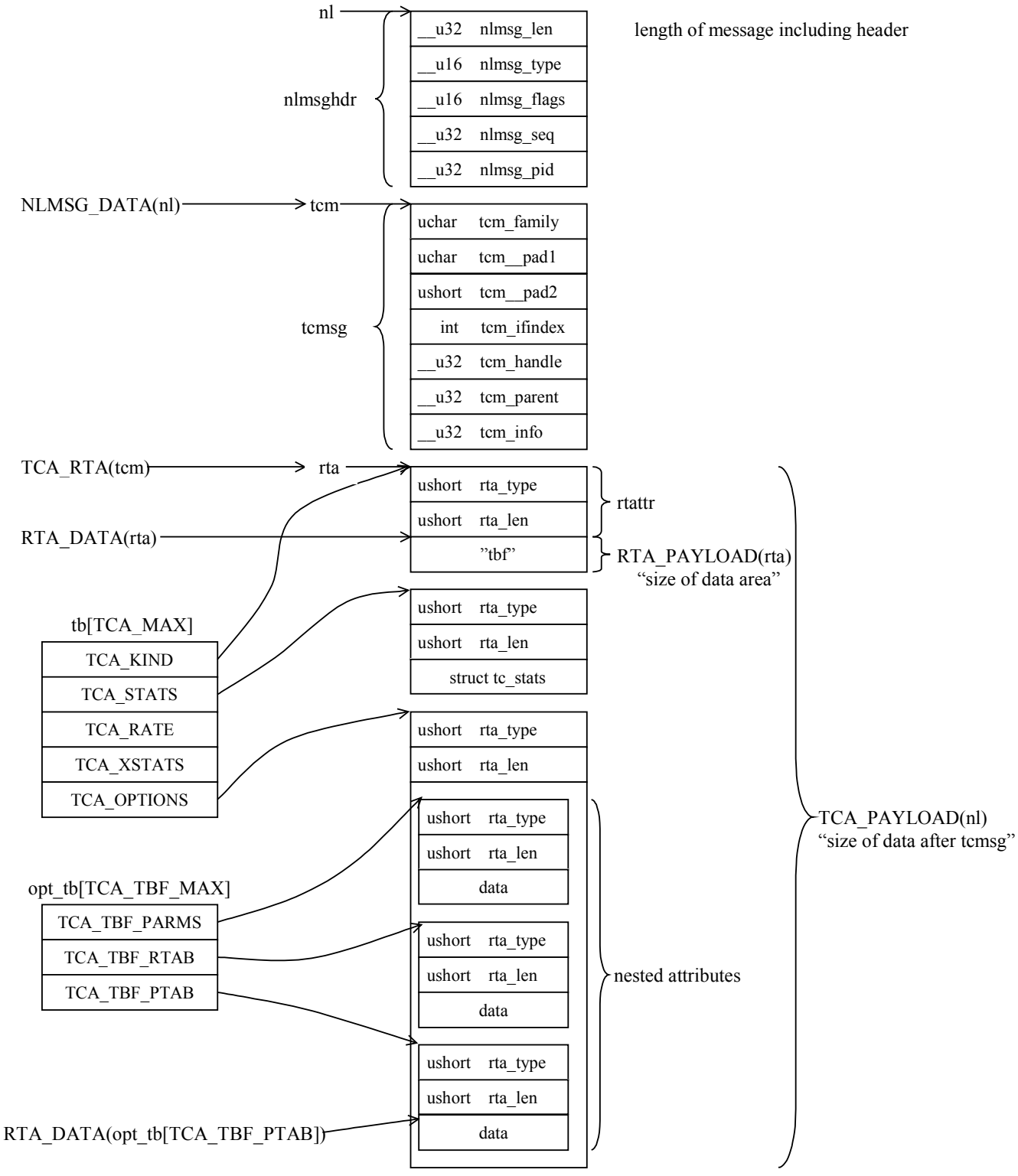


Figure 12: Format of messages passed to/from kernel through a netlink socket.

This design of the rtnetlink interface also has its drawbacks. One of them is the way the attributes are passed – it is not universal enough. Each type of discipline and filter passes attributes using their specific structures marked with their specific identifiers. This makes it impossible to create an unified and universal interface that would allow users to input attributes and would be easily extensible with new types of disciplines. Another problem is the feedback of the traffic control set up via the rtnetlink interface, since this contains almost no error messages at all. And since the input parameters are not checked properly, incorrect input parameters can be processed, that can even result in an infinite cycle in the kernel. Also, the structure of the available error messages is to simple, therefore we do not receive specific information about the problem, just that it occured.

2.9.2 TC

The system resources for setting up the components have already been described. A common tool which allows the users to set up the traffic control is the TC (traffic controller) command line utility. This tool has large capabilities but it is unfortunately extremely difficult to use. It is simply parsing the command line and creating and sending a message via the netlink socket in accordance with the user defined parameters. This makes the utility fairly simple and easily extensible to other traffic control components by adding a module, which is able to parse the parameters of the new component and saves the obtained values into a netlink socket message.

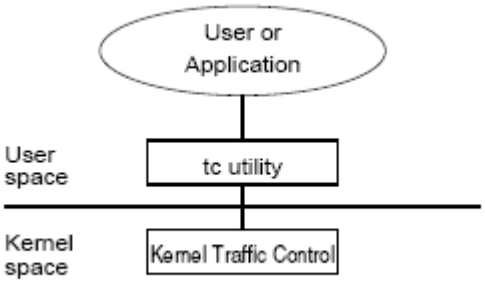


Figure 13: TC architecture

However the ease of use for the creators of new disciplines comes at the cost of the difficulty of configuration for the system administrator. The reasons for this are manifold:

First, since all the individual components of traffic control configuration can be manipulated individually, the location and role of each component needs to be specified in every single configuration command. This redundancy makes configuration scripts very hard to read or change. Even finding a small typing error can be very time consuming.

Second, some elements just happen to be complex by design, for example the u32 classifier. While having full control over all details allows the construction of highly efficient classifiers, it burdens users with a fair amount of information they need to understand before expecting to accomplish anything, and the sheer number of individual configuration steps is likely to cause result in frequent mistakes.

To get a picture of the final configuration in its tree layout just from reading the scripts is very difficult. And it is even more difficult to find out the current QoS configuration when the source configuration scripts are unavailable.

The TC script language problems used to be worsened by the near-total absence of documentation, and incomplete and outdated online help. Fortunately, the Linux Advanced Routing and Traffic Control project [LAR04] has meanwhile provided the much needed relief.

To summarize, the main difficulties of Linux traffic control are that configuration – much like programming in assembler – focuses almost exclusively on low-level details, and that the low level of abstraction of the interface complicates the integration of performance optimizations. [ALM02]

2.9.3 TCNG

Mostly due to the sheer complexity and difficult parameter input when using the TC tool, other programs were created for traffic control configuration. One example is the TCNG, which defines its own language (similar to C) for the configuration of components. This language, thanks to abstraction, allows a simpler and easier-to-read notation of the traffic control configuration. From this language, TCNG then translates the code into scripts for TC. In addition TCNG contains a simulator for configuration tuning. The main disadvantage of TCNG is its bad extensibility – individual components are “hardcoded” into the entire system which therefore cannot cooperate with new components. This was one of the main reasons why this project stopped developing further and never spread between users. [ALM02]

3 Solution

On the basis of a thorough analysis of the current situation in the configuration of Traffic control in Linux, the following problem solutions were proposed:

Problems of system resources for setting up the QoS:

The problem of insufficient versatility of the rtnetlink interface can be solved by the creation of a more universal, higher-level interface. The problem of inaccurate error message reports and insufficient component settings checking in its kernel module can be solved by an elaborate error checking before the data are sent to the rtnetlink. Another option would be to modify the rtnetlink itself, though that would mean rewriting all existing components. However this solution seemed like too radical and should be first discussed by experts.

Problems with the available user tools for setting up the QoS:

These problems of course depend on which particular tool is used. The previously described TC tool is the most common tool in Linux, others are marginal, and therefore a solution related to this tool is examined in detail.

Based on the usage of the TC tool and the problems that need to be solved with when setting up the QoS using TC, the creation of a new tool for system administrators was proposed, to allow easy and more effective configuration of the QoS. Thus, the goal is to design a universal and extensible framework for the configuration and management of the Linux traffic control primitives and implement tool for interactive design and run-time monitoring of user defined quality of service policies on a set of remote gateways. The main objectives are described in the following part of the chapter.

3.1 Setting the QoS

The main contribution of this work should be the improvement in the way traffic control (QoS) is configured in the Linux. We should evolve from a text script based setup, where each component is separate with a lot of individual parameters, which repeat and almost nobody understands their purpose, to a model of the **uniform graphical interface for all types of components**. The interface (GUI) should allow **simple** and **well-arranged** QoS configuration. It should be intuitive and user friendly. Due to the quantity of different types of components with their special approaches, it has to be **universal and flexible** enough to cover all cases. The QoS configuration system has to be easily **extensible** to support new types of components (disciplines and filters), in order to keep up to date. In case it is not possible to use the graphic interface, the text script setup option is included.

3.2 The QoS Monitoring

While setting up the QoS it is very helpful to have a feedback on how the system reacts to the particular settings. Using this feedback, the administrator can find out if he managed to setup the QoS as he intended and that everything works properly. Therefore, one of the further aims is the real-time QoS monitoring which would allow the continuous data flow checking for the individual components.

3.3 Remote settings

With the spreading of Linux into all kinds of devices like routers or gateway, which were up to now dominated by proprietary systems, the possibility to set up QoS on these device in the same manner as they would be set up on a computer with Linux was gained. Since these devices are usually locked up somewhere in a rack and do not have a monitor or keyboard connected, most of the set up is done remotely. Therefore it is desirable to include the possibility of remote settings for our program. In addition, the remote settings support has to be implemented with minimum requirements, so that for example a server running on a router has minimum requirements (to allow extensibility to hw/sw minimalist architectures). The GUI and all the operations with high requirements are only included in the client running on a regular computer.

3.4 Robust architecture

The framework architecture should be designed in such a manner that any changes in the surrounding environment can be easily integrated into the framework and so that it is not necessary to rewrite the whole framework or to make many changes each time a function is added. Under the changes in the surrounding environment we understand change of the TC interface, changes in individual component parameters, etc.

3.5 Platform independence

The remote settings requirement implies a client/server framework architecture. In order for the framework to become widely used, platform independence is required. This would enable communication between the client and the server, even if each runs on different hardware architecture (big-endian / little-endian, 32bit / 64bit, ...), as well as the option of running the client on operating systems other than Linux (Windows).

3.6 Security

Since we require the option of remote settings and are configuring very sensitive parameters, to which only a limited group of users (system administrators) is allowed to access, we have to ensure a good level of security. Hence, we need to ensure proper authentication (identity check), authorization (assign rights), as well as a secure client/server connection and protection against attacks and unauthorized access.

4 Architecture

This chapter describes the framework distribution into modules, in order to achieve the properties defined earlier.

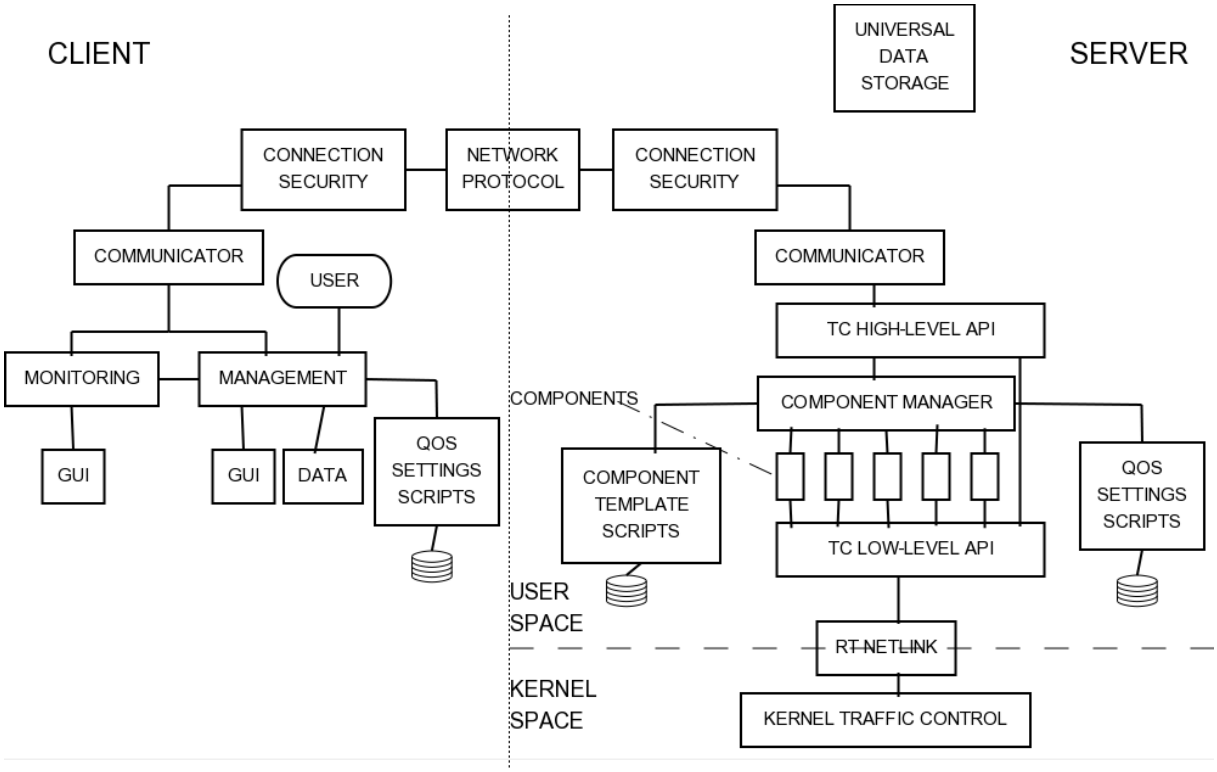


Figure 14: Architecture

4.1 Client / Server

The framework is primarily divided into a client and a server part. This structure was chosen to allow remote settings and to allow different HW and SW requirements on the client and server side. The server thus only provides low-level functions (QoS setup and monitoring) and enables the client to access its functions, while the client contains the high-level graphical user interface.

4.2 Configuring Traffic Control

Traffic Control is configured on the server via the rtnetlink interface. Access to this interface is provided through the TC library, which takes care of the creation and reading of messages with rtnetlink settings. To achieve **extensibility** of the application with new components (disciplines and filters), the framework can be extended with a shared library module that will enable support for that new component of the traffic control. This module solves all specific issues of that particular component, using configuration template scripts which describe the individual parameters of the components. The module is managed by the component manager, which provides a unified interface for managing components and ensures **universal** access. The component manager is then topped with a High-level traffic

control API, which provides all the settings and monitoring of traffic control in this framework. The high-level traffic control API uses the component manager (for issues related to the specific component type) as well as a direct access to the TC library. Besides that, the High-level traffic control API cooperates with the module for administrating QoS configuration scripts. These scripts are used to permanently save the QoS settings, and to allow direct editing by users.

4.3 Transfer protocol

The high-level traffic control API on the server is made accessible for the user on the client side of the framework by using the communication module and transfer protocol. The transfer protocol between the client and the server must be designed in order to make the client and the server independent on the architecture used (SW and HW), i.e. **platform independence**. Furthermore, the security of the connection and control over the client access to the server must be secured.

4.4 Universal data storage

To allow working with any type of component, the need for a module that is able to process any number of parameters of various types arose. The module also has to be able to structure the data according to their semantics and it has to support nested structures. This will then be used for working with templates for the creation of actual QoS configurations, as well as for working with actual QoS setting. This module is also used to store data on the client and the server. **Platform independence** has to be ensured. The transfer protocol has to be able to transfer data in this module via network. The module has to enable conversion of data from/to the form, in which it can be saved to a file to support persistent data saving.

4.5 Client

Traffic Control management and configuration is done via a client using high-level traffic control API accessed by a communication module. From data measured on the server it creates graphical representations of the individual traffic control components data flow rate, i.e. enables real time **monitoring**. It also enables the user to configure the QoS on a high level via a sophisticated GUI.

5 Implementation

The implementation of individual framework modules is described in detail in the following chapter, including reasons why a particular progress of work was chosen.

5.1 Program structure

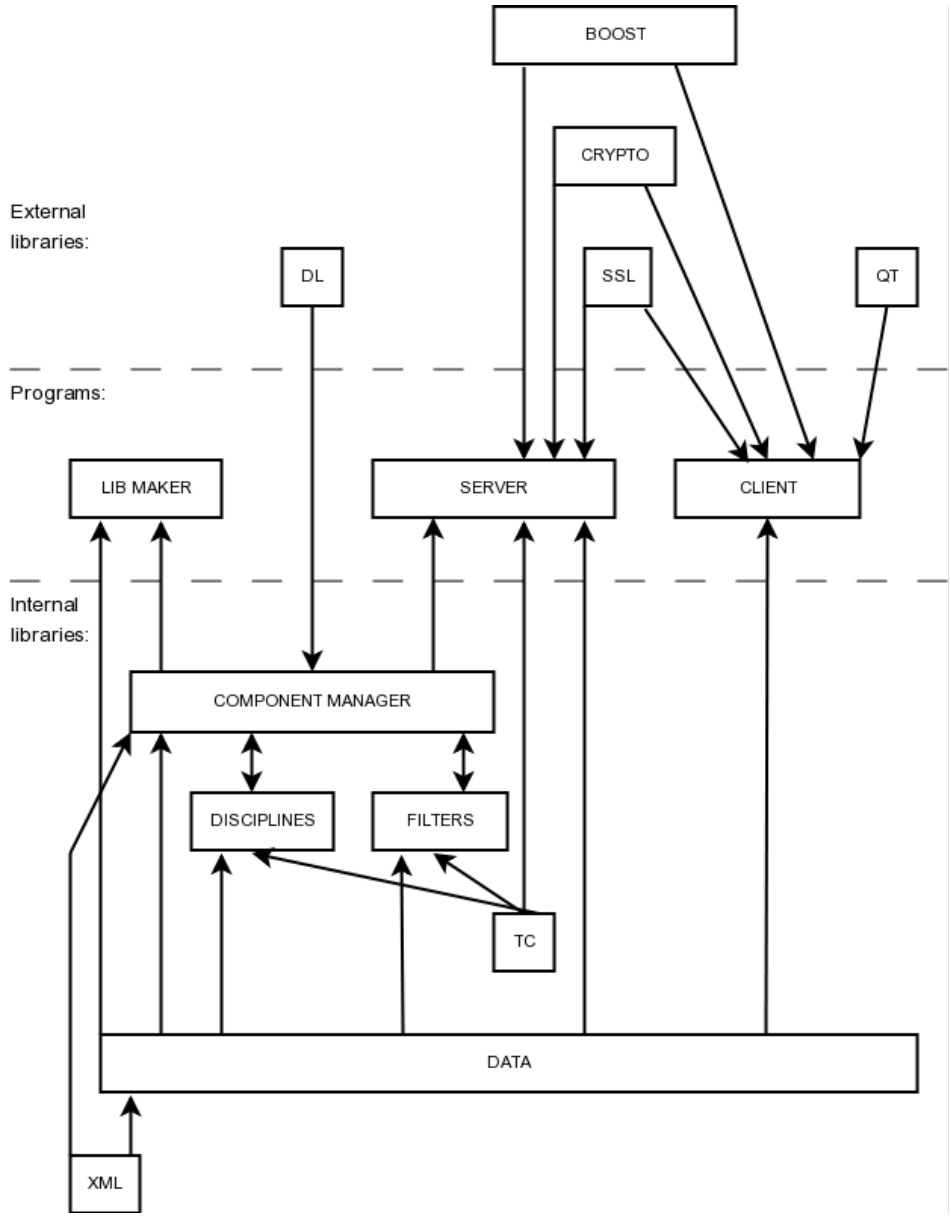


Figure 15: Program structure

Modules:

- jpData
- jpComponent
- tc
- tinyXml
- basicQOSdisciplines
- basicQOSfilters
- jpQOSd
- jpManager
- jpLibMaker

The jpQOS framework is divided into several modules each taking care of some of the programs required properties, which can be summed up in the following - universality, extensibility and support for remote settings.

Further modules emerged because of the use of external libraries and the division of the framework into a Linux dependent part and an independent part, which provides support for clients in different Operating Systems.

jpData:

The library that manages work with program data, i.e. their loading, saving and forwarding. It contains the basic function required to work with the data (conversion of units). It treats the data structure of one discipline (datastorage), as well as data of the whole QoS configuration tree (object_tree). The library also contains a uniform system for reporting the log messages about the program status and its possible errors (this is used both by client and by server). And finally it also contains the protocol for transferring data via the network between the client and the server (netprotocol). It is the only library shared by the client and the server.
System independent.

jpComponent:

The library providing support for treating various disciplines and QoS filters. It defines the interface for the framework extension on new disciplines and filters on the basis of dynamically loaded libraries. It implements loading of template configuration scripts for generating templates of disciplines and filters, which are applied by the user to create a specific configuration.
System dependent on Linux. (dynamic loading of libraries)

tc:

The Alexey Kuznetsovs' TC library from the iprout2 package. It is adjusted so that it can be used as a shared library in C++ environment. It takes care of low-level communication with the kernel via the netlink socket interface – rtnetlink.
System dependent on Linux.

tinyXml:

A library created by the authors Lee Thomason and Yves Berquin. The TinyXml is a simple and small C++ XML parser.
System independent.

basicQOSdisciplines:

A library providing support for the basic set of QoS disciplines - CBQ, HTB, SFQ, TBF, FIFO.
System dependent on Linux.

basicQOSfilter:

A library providing support for the basic set of QoS filters - U32, Route, fw.
System dependent on Linux.

jpQOSd:

This is the server part of the jpQOS framework. It communicates with the kernel via the TC and provides high-level API for user control. It listens on the network, receiving clients commands and sending results.
System dependent on Linux.

jpManager:

This is the client of the jpQOS framework. It communicates with the server via a network connection and the netprotocol. It provides the user with a GUI, which enables him to monitor and configure the QoS on the server.
System independent.

jpLibMaker:

This is a support tool for designers of new QoS disciplines and filters. It pre-generates the source code of a new discipline / filter support library based on the corresponding template configuration script. The designer only fine-tunes details and deflections.
System dependent on Linux.

5.2 Server

5.2.1 The configuration of Traffic Control Primitives – rtnetlink

As described in the analysis, traffic control components in Linux are implemented as kernel modules. To adjust them via the user's application, some user-space/kernel-space interface has to be used. The Linux provides the rtnetlink interface with all its advantages and problems (see the netlink socket analysis). Some projects (LKM) chose a different approach and tried to implement the configuration via syscalls. This method is too invasive. It assumes rewriting all the traffic control components of the kernel and patching the kernel. There are any reasons to substantiate this. Therefore in jpQOS framework was decided to use a standard rtnetlink socket interface for the traffic control component configuration. Finally, decision, whether to access it directly or to use ready high-level tools, had to be made. It also had to be considered the requirement of the framework extensibility for new components and the requirement of universality. When this module was being implemented, the TC command-line tool from Alexey Kuznetsov [KUZ03] and the TCAPI library from David Olshefski [OLS02], were available. TCAPI library seemed to be exactly the one it was needed. It created API for the communication with the rtnetlink socket. However, when inspected in detail, it was discovered that its architecture did not enable universal access to all components and that extensibility for new components was difficult. Each component type was tightly "hardcoded" in the library and there were special functions and special data structures prepared for it. Therefore, it was refused to use TCAPI. The other option was to use the TC command- line tool. The advantage of this would be minimal work difficulty - the whole support had already been made. Moreover, the new component author always creates the service module for TC, so extensibility is assured. The significant disadvantage was the way of transmitting parameters via the command-line, which was suitable for human use, not for the program. Some problems might occur while changing the format or the name of the parameters (in case of TC version modification), that would result in a fragile solution which does not correspond with the requirement of robustness. Another possible solution might be to create the library from the whole TC for direct use in program (avoiding the command-line and direct parameter entering). However, that was not possible because command line parsing was grown through the whole TC (in all component service modules), so it would mean rewriting the entire TC. It has been decided to communicate directly with the rtnetlink socket while using the part of TC as a library with low-level functions – marschaling/unmarschaling of messages for rtnetlink socket, the creation of the socket, sending requests and primary processing of responses. The advantage is that the setting can be modified exactly according to the framework requirements. There is no extra work transferring parameters directly via

structures (conversion, etc.), so it is a faster solution. However, this decision has disadvantages as well. It is necessary to write each component its own module, existing modules from TC cannot be used. When a new component is created, its author has to write its module not only to the TC but to jpQOS framework as well.

5.2.2 Components library

As it was mentioned in the previous chapter, each type of component needs to write a special module to framework - a shared library. In this chapter, there are explained reasons, why it is so. Framework is a universal tool which has the possibility to work with any type of component. Each component has its specific parameters that are transferred during the configuration via rtnetlink in specific component C structures (see rtnetlink analysis). Messages for rtnetlink cannot be created automatically by means of some own descriptive language, because the declarations of used C structures would need to be known to create the correct message (The declarations of future types of component cannot be known now). This problem could be solved by putting data directly to the memory, without knowledge of structure declaration. This solution is inappropriate because it is a dirty operation. Its correct function can depend on the compiler that is used and its parameters (data item alignment and its size can differentiate in structures). It is necessary to realize that the data is transferred between two programs (kernel and framework) when each of them could be compiled in different conditions. Besides that, some components use demanding preprocessing of the parameters entered by user (miscellaneous table calculations, etc.) until they forward them via rtnetlink. It would result in high demands on the components descriptive language, so the language wouldn't be easily and quickly understood. There is no point in the new component author learning a new language when it is possible to do it all in the C language that he already knows (kernel modules of components are written in the C language).

Because of these reasons, it was decided to solve each component specifics via shared libraries written in C/C++ language. Every component is implemented in one of the shared libraries. When this component is needed in the framework, the library is dynamically loaded and a particular method is called. That ensures the framework extensibility without the necessity of its recompilation. In order to make it all work properly, it is necessary to define precisely the interface 'framework – components library', which determines what exactly the library has to implement. Because of the differences of filter components and queuing disciplines, there are two interfaces defined, one for disciplines (*discipline.h*) and one for filters (*filter.h*).

The discipline class is the interface for disciplines that has to be implemented by the new discipline author. It contains *modifyQDisc* and *modifyClass* functions that have to create and send the message for rtnetlink according to the entered parameter values of the discipline (qdisc/class). This will lead to setting the particular component in kernel. The discipline class contains *dumpQdisc* and *dumpClass* functions as well. These functions create data representation of actual component configuration, which is usable in framework, from values acquired from kernel via rtnetlink. Finally, the discipline class also contains *testConfigValidity* and *calcRateLimits* supportive functions for verifying the values for setting and calculation of data flow speed limit for the display in graphs. All specifics of the particular discipline are implemented in the library (including all entered parameter modifications, calculations above them, etc.); and make externally an uniform interface. The interface for filters is similar; the main difference is that filters are not divided into qdisc and class. The mentioned low-level part of the TC library can be used while implementing the component library, as well as it is used while implementing the module to TC tool. That helps

the new discipline author because he works in the well-known environment. The only difference is that he does not acquire the parameter values from the command-line but from the special framework data module '*DataStorage*' that will be described later. The creation of shared libraries is simplified further by the supportive *jpLibMaker* tool that will be described later in detail.

5.2.3 Component manager

Individual shared libraries with the implementation of particular component specification are covered by the component manager (separately for disciplines and filters, so by the discipline manager and the filter manager). Component manager assures uniform and universal access to components for the rest of the framework. It assures a dynamic loading of shared library in case it is needed. It also forwards the implemented interface according to components name, when there is a request for the use of the components functions. All framework functions use the component manager when necessary to implement an action dependent on the components type.

It was necessary to use a trick because the system of dynamic linking of shared libraries in Linux does not support objects but only functions. Every library besides the Discipline class implements, in addition the function for the creation of Discipline object – *getDiscipline_t()*. The Discipline manager finds a handle for this function via which it gets the created object with a particular implementation. It can call Discipline class virtual methods with the knowledge of its virtual ancestor.

The essential disciplines are implemented in the "libbasicQOSdisciplines.so" library. These are FIFO, SFQ, TBF, CBQ, Ingress and HTB disciplines. If the Discipline manager does not find the required discipline in its basic library (it is searched by the name), it tries to open the new library with the name according to the prior pattern that comes from the disciplines name (for example for HFSC it is the libhfscQOSdiscipline.so). Libraries are searched in the "/usr/lib/jpqos" directory or eventually in the directory set up in the environment variable „jpQOS_LibPath". If the library is not found, it is a fatal error and the discipline cannot be used.

The Discipline manager creates cache, which contains used disciplines. When the discipline is requested, it is searched firstly in cache (by the name), if it is not found there, the Discipline manager tries to open the appropriate library. One object implementing a particular discipline manages to cover all requests directing to this discipline (for example in the case of five HTB classes one discipline object is enough.)

Filters and the Filter manager work at the same principle.

5.2.4 Component template scripts

As described in the analysis, each component has a different number of its specific parameters of different types. Parameters of each component have to be described (each has to have its name, type and unit specified) in order to enable their setting via framework. That is provided by the component template script (descriptive language) and *templates_xml* module, which can load the script. To each component is created one script. This script is loaded in the Component manager, when its components library is loading.

Component template script is written in the XML. XML have been chosen because it is well-known and wide-spread. The author of the script does not have to learn a new language - he manages if he knows the component template script structure definition. It is defined by the *component.dtd* file, which contains Document Type Definition. XML have been chosen also for its good element structure and the possibility of adding some other eventually needed attributes easily.

Description of component template script structure, its attributes and their purpose:

The component template script describes one kind of component so it has to contain its name and type, eventually the comment used as a help for user, who is editing the particular component (a general instruction about the whole component philosophy). The type determines the particular component, whether it is the filter or discipline, classless / classbased. Also supportive "component" – *supplement,* can be one of the type. It is used for the specialized configuration of the data structures used in more components (for example estimator, policing filter; more about it later). The type determines the basic form of the framework access to the component. Then there is a list of all component data structures (*data_storage*). Typically it is one structure for qdisc and one structure for class in case of the discipline and one "root" structure in case of the filter. Each data structure is identified by its name and eventually supplemented with the comment used as a help for the user. Then there is a list of data structure parameters. They are divided into three main categories – basic, advanced and special. The purpose of these divisions is to make the setting of parameters, edited by user, well-arranged. The user is no longer stressed with the setting of parameters which he doesn't understand the meaning of. In the basic category, there are basic component parameters by which the user sets the basic component properties (for example rate at TBF). These properties should always be set, otherwise there is no point in using the component. In the advanced category, there are also important component parameters that are set only when the user wants to change the basic component behavior (for example "crate" at HTB). In the special category, there are parameters of a very special character that should be edited after careful consideration and with clear understanding of whole component principles. It is used for detailed tuning of components behavior or for specification of technical parameters of used hardware. Finally, parameter itself is defined by its name, type of variable in which its value is stored, by unit (kbit, mbit, etc.), by the default value and by the comment with help for the user. The type can be also, for example, the reference to another data structure, by which means it is possible to create recursively whole data structure trees.

The parameter default value is a value that the component author determines as default. To simplify editing, especially of some complicated components, framework contains the recommended values principle. These are recommended values resulting from miscellaneous practical examples and they work well by common usage. Recommended values may depend on the network interface type (for example the frame size, the interface speed, etc.). Therefore it is possible to define different recommended values for each interface - these are used only whilst configuring the traffic control on a particular interface. For parameters that don´t depend on the used interface, there are default recommended values, that can be used for configuration on any interface. Recommended values are stored in XML scripts, their structure is very simple and is defined via the Document Type Definition in the *recommend.dtd* file.

Component template scripts and Recommended values scripts are loaded to framework via *templates_xml* module that uses tinyXml library. This library is very small and has a simple C++ interface for access to the loaded XML document. These properties are useful for jpQOS framework. Therefore it was preferred tinyXml library to XML Xerxes library, which has more functions, but it is too big and difficult to configure.

Loaded component template scripts are represented in the framework in the form of recursive structures – *DataStorages*. Each component has one root *DataStorage*, which contains a list of *DataStorages* described in its component template script. All the loaded components and their *DataStorages* are then compiled into one *DataStorage*. This "global" *Data*Storage is transferred from the server to the client, which uses the information contained within to create configuration options for the user (it contains a list of all available components – disciplines and filter, a description of all their parameters, hints).

Recommended values are represented in the framework the same way as the component template scripts, i.e. in the form of recursive structures – DataStorages. Their basic principle is the same, they just contain different information. In addition, the Recommended Values differ for each network interface, thus we get several "global" DataStorages – there are as many as there are network interfaces plus one default.

DataStorage can also contain an entry containing a link to another DataStorage – the principle of creating nested structures. The entry contains the name of the DataStorage that can be created and a tag determining, whether only one or more instances can be created. The template for the DataStorage being created is first being searched for the DataStorage component template scrip of that component, which contains the entry with the link. If the DataStorage with the required name is not present, it is sought for in the component template script supportive "component" – *supplement.* All the DataStorages shared between two or more components are described here. A DataStorage usable only in one component is only described in the component template script of its respective component.

5.2.5 Supportive tool jpLibMaker

The framework contains jpLibMaker supportive tool to simplify the creation of shared library for each component kind (and to simplify the extensibility as well). This tool generates the source code of the library implementation according to the knowledge of Discipline/Filter interface and Component template script of the particular component. The jpLibMaker tool cannot know the exact implementation specifics of the particular component - therefore, the generated code has to be tuned before use. However, in most of the components, this concerns just small corrections - for example the change of the used component structure name, addition of the parameter value control, etc. The access to all component parameters via the data framework interface is ready, as well as the creation/processing of the message for rtnetlink socket.

5.2.6 Data management

For a possibility to work with any component universally, a universal system for the components data management is needed. Each component has an miscellaneous number of parameters of various types. These parameters are arranged into numerous structures which can be interconnected, one included in another, etc. Furthermore, for a good work of the components with the given data manager, it is required a reasonable interface for data conversion from the specific component representation into the universal data system and vice versa. The universal data system has to contain as well information according to user can set up parameters at high level. (help, parameter classification, recommended values,...) Additionally, in this system, it must be possible to transfer the data via the network. All these requirements are fulfilled by the designed Datastorage system, which will be described now in detail.

The basic element is the DataItem class, which represents the data of one component's parameter. It contains: name of the parameter, type (int, string, double, ...), unit (kbit, mbit, ...) and several other supportive variables. The value of the parameter (which can be of various types) is stored via the boost::variant data structure. The Variant data structure of the Boost library have been chosen because: 1) its capability of saving different types into a single variable; 2) Compile-time type-safe value visitation; 3) Efficient implementation and possibility of serialization, which is used while transferring objects via a network. Moreover, the DataItem includes: the interface for controlled access to the value of a variable according to its type; the conversion functions string->value, value->string with the conversion of units.

The basic DataItem elements are associated to the DataStorages, which comport with the data structures according to the component structure. DataStorages enable direct access to DataItem elements according to names; adding and deleting the elements. In addition, this structure is recursive; it can contain the list of the DataStorage's structures. DataStorage prepares its elements (component parameters) according to the component template script. Elements prepared in this way are then forwarded to the user, who can edit them in the GUI.

So each component manages its data via the DataStorage. Moreover, the framework needs the data structures to management the entire QoS configuration in its tree structure. *NIC*, *TCObject* and *TCObjectTree* structures serve this purpose. *NIC* represents all settings of one network interface card. Traffic control in Linux is based on the network device, so that one *NIC* contains the complete traffic control configuration (independent from the others) for the given interface. First of all, this is the QoS configuration tree for egress & ingress queue, interface index & name, hw & ip address. The QoS configuration tree consists of *TCObjectTree* classes. *TCOobjectTree* class enables the creation of the QoS tree structure out of a particular component. It contains: 1) link to a component (*TCObject*) which is represented by the given tree node; 2) the list of the child components (*TCObjectTree*); 3) the list of filtres (*TCObjectTree*). *TCObject* represents the configuration of one component in the QoS tree, which could be the qdisc, the class or the filter. *TCObject* contains mainly the name of the represented component, its type, handle, parent handle and the setting of parameters (in the DataStorage form).

5.2.7 Obtain QoS tree from Kernel

It is not easy to obtain the configuration of the traffic control primitives and to create the QoS tree. Obtaining information via the rtnetlink socket is done by sending a request (specifying exactly what is wanted) and subsequently calling up the set function on the received data. The procedure is as follows: the network interface is chosen, for which the QoS configuration tree is created. First the list of qdiscs is obtained at the given interface and then the component manager is used to process the specific parameters of each component and save them to the DataStorage. Then, for each qdisc, the list of its classes (which can have the tree structure) is obtained, including its parameters and the list is saved to the appropriate qdisc. Then, out of all these lists, the QoS tree is created, using the knowledge of: 1) which class belongs to which qdisc 2) parent class handle 3) child qdisc handle. Then, for each node of the tree, the list of filters is obtained and save it in order of use

5.2.8 Monitoring

Whilst obtaining the data about each qdisc/class, also information about the transmitted data is acquired, which is used for the real-time monitoring by measuring it at

regular intervals. The monitoring is realized via the timer, function that measures the time precisely and calculation the difference between the transmitted data at each component at a given time period. Measuring of the transmitted data is done in a similar way used to obtain the whole QoS tree configuration. The difference is that there is no need to create a new tree - the original one is used and the information (about the transmitted data and the measured time) is only updated in it.

5.2.9 Set QoS tree to Kernel

Setting the Traffic Control Primitives according to the configuration created by user is done in this way: The whole configuration is represented by *NIC*, which contains the QoS configuration tree. First, it is necessary to check the validity of each tree node by using the validation function of the component, which is called via the component manager. If an error is discovered, it is reported (the name of the incorrect parameter, handle of its component and error message) and the setting of the new QoS configuration is canceled. If the validation check is successful, the configuration can begin. It is started on the root of QoS configuration tree, which must be the qdisc. Each tree node has a tag, which determines whether the node configuration had been changed by user or not. When changing the qdisc configuration it is necessary to delete all the subtree under it because the change of qdisc parameters is not possible - it is only possible to create a new qdisc with different parameters. It is easier while changing the class configuration because it is possible to set changes locally – it is not necessary to change the rest of the tree. While changing the filters configuration the situation is problematic because it is not possible to change the parameters (analogous to the qdisc configuration), it is only possible to create new filters. Moreover, it is not possible to delete filters individually - every time the whole set of filters with the same priority is deleted. This problem can be avoided by deleting and then recreating the node, in which the changed filter was placed.

While setting up the QoS tree configuration, the same tree structure is used – until changes of configuration make this possible. The tree is set level by level, starting from the root. If an unchangeable node occurs (because of the above-mentioned rules), it has to be deleted and recreated, which means deleting and recreating its entire sub-tree while proceeding with the rest of QoS tree configuration. Node configuration is set up via the component manager by calling the appropriate component setting function, its parameters is in the DataStorage form.

5.2.10 QoS Settings File

The QoS configuration tree is represented by the *NIC* class in the framework. The QoS Setting File (QSF) scripts were designed to enable a permanent form of the QoS configuration, which could be saved to a disc. These scripts are written in XML, which easily implements the tree structure with its nodes. The QSF scripts structure is described by The Document Type Definition in the config.dtd file. Conversion of *NIC*>QSF and QSF>*NIC* is implemented by the framework module *settings_xml*. Loading of the QSF script to the framework is so robust that the user can easily edit the QSF script or create a new one without worrying about the order of given tree nodes, etc. Loading of the QSF scripts has three stages, much like obtaining the QoS settings from the kernel. All qdiscs are first loaded sequentially, then followed by classes (inserted into their own qdisc) and finally follow by the filters (inserted directly into their own node). After all nodes have been loaded, their lists are used to create the QoS configuration tree.

5.3 Communication Client / Server

5.3.1 Monolithic application

The client/server architecture was necessary because of the Remote Settings requirement of the tool. This requirement however, could also be met by different means – to build a monolithic application providing all the functions based on X Window. Remote access to this application via the network would be provided through the X Window server, which has these options included. As far as implementation is concerned, this would be the easiest solution. The remote access is provided implicitly, by the type of the application. This solution however has a very significant disadvantage, which is the necessity of running the X window server on any device, where the QoS setup should take place. By choosing this option, we would rule out the use of our application on most networking devices with an insufficient HW configuration to run X window server. In addition running the X window server on machines that fulfill its requirements just because of the jpQOS framework is not reasonable (X window server is not typically used for networking devices with minimalist HW architecture like routers, gateways etc.). This possibility was thus discarded and only solutions based on a client/server architecture were considered further.

5.3.2 Communication protocol

The client/server architecture requires the formation of some sort of a communication channel between the client and the server. There are several possibilities, how to achieve this. We have to consider, what is the best level to create the communication interface on, with respect to the actual properties of the framework and with respect to the requirements.

As a consequence of the required properties, the server has to minimalist. The server only carries out those necessary operation, which are impossible or meaningless on the client (low-level adjustments and monitoring of the QoS via the rtnetlink). Everything else is carried out by the client (high-level management of the QoS, GUI etc.) which has better conditions to do them – higher performance hw and software tools. This implies the division of the application into components and the creation of links between them also incorporating network communication.

CORBA:

One of the possibilities of how to implement the client/server communication interface is to use the CORBA system [OMG02]. To design a system of objects, which are implemented by the server, but their methods can also be called by the client. In the case of the jpQOS this interface could be the Communicator object and a system of objects forming a universal data storage – DataStorage (see chapter 5.2.6 – Data management). This solution has several advantages:

- First, it is the large versatility of the CORBA system, allowing for example the implementation the client and server in different programming languages.
- Furthermore it is the solution of all the platform dependent issues.
- In case the program needs to be changed and updated, we can also easily update the object communication interface in CORBA using the IDL programming language and the IDL compiler.
- The communication interface can be very detailed and the client only calls the methods of objects without the need to differentiate between remote and local objects.

On the other hand a solution based on CORBA also has its drawbacks:

- The use of CORBA on routers is not very common and the installation and maintenance of CORBA is quite challenging. This could discourage some of the less skilled users of the jpQOS framework.
- To allow the connection of client to the server, name service daemon needs to be installed and configured, there is no complete control over the establishing of connection.
- The CORBA system itself is a fairly large and robust application and its requirement on the side of the server does not fulfill the aim which is to create an application with minimalist requirements on the server of the jpQOS framework.
- Since there is no complete control over the establishing of connections, it is harder to implement authentication. It has to be done not only during the establishing of the connection, but also before every access to an object.
- Another aspect is the network load. In designing the communication interface on a detailed level, access to a large number of small objects would be required (the QoS configuration tree), thus the network load to amount of data transferred ratio would be very large, due to the flooding of the network with a large number of small packets.
- In addition, there is the question of caching the data that were transferred from the server to the client, which are unchanged and accessed repeatedly. The Corba system does not deal with this by default, thus it would have to be implemented, which would again be very complicated.
- Another problem is the connection of CORBA and the Qt client. As the principles of event-driven mechanisms imply, in the case of Qt the client cannot call upon an object function implemented on the server via CORBA and wait for a response (which can take a long time) without processing the Qt event queue. The application would seem to be "frozen" during that period of time. The QtCorba framework offers a solution to this problem, since it is integrating the Qt and CORBA event loops so that both Qt and CORBA will receive their own events. The problem is that this framework is not offered publicly free of charge, it requires a paid license. Another possibility is to use more threads, but then we would run into problems with synchronization.

Custom protocol:

Another solution of how to implement the client/server communication interface is to design a custom protocol. This protocol would be based mainly on the transfer of the data containing the configuration of the QoS tree from the client to the server and vice versa. The client would create the QoS configuration tree locally and then send it to the server via the protocol. The server receives the data a uses them to configure the QoS. Monitoring is based on the same principle, just in the opposite direction. The server acquires data describing the measured transmission speeds for the whole QoS tree, sends it to the client which then processes the data and shows it to the user. Data are represented in the form of a universal data storage system – DataStorage- both on client and server. The only remaining task is to define, how this system of objects will be transmitted on the network. The use of the **serialization of objects** would be ideal in this case, however C++ does not support this option. It does however offer the **Boost** library, which allows the serialization of objects under C++. Additionally, the Boost library deals with object serialization on the whole, including all the related tasks – Code portability, code economy, deep pointer save and restore, proper restoration of pointers to shared data and data portability (Streams of bytes created on one platform should be readable on any other). The Boost library also has other advantageous properties – it is not too large and it is easy to use. The drawback of a solution based on a custom made protocol is the necessity to "manually" update it in case of some changes in the

framework, which can be extremely complex and laborious. This however, can be minimized by a well designed protocol with enough versatility.

Due to the requirements imposed on the jpQOS framework and the characteristics of the individual solutions, it has been decided to use a custom designed protocol utilizing the Boost library. This solution does not limit the possibilities of the framework, it is easily expandable even on simple architectures (routers), has low HW requirements, low network requirements and is easily implemented.

NetProtocol

The actual jpQOS framework network protocol was named NetProtocol and is based on these principles:

- The structure of the data of the network packets is precisely defined by *netprotocol.h*.
- Packets are transmitted through a standard TCP/IP protocol.
- Marshalling/Unmarshalling, sending/receiving of data, establishing/closing of connection is provided by the Communicator module.

The format of the packet is designed to be platform independent (little/big endian, structure alignment, 32/64 bit,...). Each packet has 3 main parts: header, data and information the size of a header. The size of the header is written as the first item, in the format of a 32 bit unsigned integer in network byte order. Then follows the packet header, which is a structure describing the type of the packet. It mainly contains information about the data type, their size, count and a command saying what should be done with them (change configuration of the QoS, check, save to disk…). The structure of the header is serialized using the Boost library. The types and the number of structures and the sum of their sizes are known from the header. Some packets do not contain actual data, all the relevant information are contained within the header (a command what to do). The reason for this format of the packet is the versatility and platform independence. This way it is possible to transfer any desired data structure, all that is needed is to register the type into the header and then serialize the data using the Boost library. Platform independence is secured by the Boost library and a fixed format for saving the header size.

The Communicator module deals with the network packets, both on the side of the server and the client. Both sides do however have their specific properties given by the difference in running and controlling the application and are therefore described separately.

Server:

The communication module of the server is implemented using low level network API. The communication module first creates a network socket and sets it to listen for new TCP connections on a given port. Then it waits in standby mode for the connection request from the client. After the request arrives, a TCP connection is established with the client on a new socket, authentication of the client takes place using the X.509 certificate (for more details see chapter 5.3.3 – Security). If the authentication is unsuccessful, the connection is terminated. Until the client communication socket is closed, the listening socket ignores any new arriving connections, which ensures that only one administrator at a time can configure the QoS (to allow multiple administrator configuration at the same time would only cause confusion). The server can also be run through a superserver (inetd), in which case the TCP connection with the client is established by the superserver and the jpQOS server only receives a descriptor on the socket in stdin/stdout. In this case the server only runs while communicating with the client, then it shuts down automatically, saving resources.

The socket used for communication with the client is in blocking mode, thus calling, reading and writing into/from the socket are synchronous. Thus, since the reading and writing

is synchronized, data consistency is easily guaranteed. For example two outgoing packets cannot mix into each other etc. During reading and writing however the server is not blocking, because these calls only take place if the data are ready.

The communication with the client is executed in a loop, where the server waits for the clients command, the server executes the command and sends the response, etc. An exception is the rate monitoring of transferred data, which when requested by the client, results in the sending of transmission statistics from the server to the client in regular intervals. This is accomplished using a timer, which sends a signal. There still is, however, the problem of how to wait for the signal from the timer and for the event on the socket at the same time. A solution to this is offered by the "pselect" library function, which can work with events and signals at the same time. It is not, however, implemented in all C libraries. To achieve the highest possible expandability of the tool being made, the standard "select" function is used instead and the signal problem is removed using the following trick. The signal from the timer is converted into a writing to a pre-created pipe using a callback function, which can already be examined using the ordinary "select" function.

The loading of the packet itself proceeds in parts, as they arrive on the network. The data are saved into the prepared buffer and they are processed when the whole part is loaded. First the size of the header is loaded (the first 4 bytes of the packet). Then the header itself is loaded (according to its length in bytes, as previously detected). From the information in the header it is determined whether any more data will arrive and their size. If yes, they are loaded. After loading the whole packet it is processed, i.e. the required command is executed and a response is sent.

The server is designed to allow long term operation without the need of administration, thus its functioning must not be interrupted by an error on the clients side for example, etc. Thus if any critical error occurs, the server terminates the connection with the client and is ready to establish a new connection. If there is an error on the listening socket, the whole communication module is reinitialized.

Client:

The communication module of the client is implemented using a low level network API. It is based on a similar concept as the server. The difference is that it follows the principles of QT applications (event driven GUI program) and follows the users' commands.

The QT contains a QSocket class, which serves as a high level API for the TCP connection. Unfortunately, this class cannot be used, because it is not possible to adjust it to use SSL encryption (see chapter 5.3.3 – Security). Hence basic low level network API is used, which does bring new problems. The QT applications are event driven. A function called by the program cannot be blocked and for example to wait for further data, otherwise the application would "freeze". That's why it is necessary to use a more complicated asynchronous access to the socket, which is in a non blocking mode. However we cannot wait for new events by blocking the application in the "select" function, it is necessary to use an analogue of the QT and that is its QsockeNotifier. The QsocketNotifier sends a signal corresponding to the status change on the socket, i.e. read/write/error.

Furthermore, SSL can force renegotiating of the connection even during read and write operations on the socket, which means the possibility of reading data during writing and vice versa, writing data during reading into the socket, needs to be preserved. Thus the QsocketNotifier has to be modified to always call the appropriate functions as needed.

All of the clients' network communication is implemented by the Cmconneciton class, which provides the rest of the application with higher level API for the communication with

the server. The term high level API means the calling of a specific function to send the command to the server and the reservation of signal for the calling of a specific service function for the receiving of the response from the server (Qt signal and slot model).

Asynchronous access requires the repeated calling of some functions. After each event arriving to the socket the QsocketNotifiers service function is called, which ensures the processing of the currently prepared data and return of the control to the program. After processing the complete request (connect, read, and write) its service is called – (successful connection to the server, loading of the packet from the server, sending the packet to the server).

5.3.3 Security

Authentication and authorization

The jpQOS framework is base on the client/server model. Thus there is the advantage of the user (client) being able to control the system (server) remotely on the network. In order to prevent abuse of this remote access, the framework has to have network security elements protecting against unauthorized use.

The jpQOSd server performs operations, which are only allowed for a limited number of users (QoS system administrators). Hence there has to be a way to define a user group, which has access to server administration and also a way to verify the identity of the user trying to log on.

There are several authentication and authorization tools. Not all are however suitable for our purpose. After studying the documentation of the PAM [MOR06], SASL [MYE97] and SSL [COX06] tools it was decided to use the X.509 certificates and the OpenSSL project. Our requirements were mainly: security, user friendliness, ease of configuration for the administrator and a reasonable level of difficulty in the implementation. The OpenSSL project and the X.509 certificates do comply with all of these requirements. Using the X.509 certificates authentication is taken care of and SSL ensures an encrypted connection between the client and the server, thus the transmitted data cannot be manipulated.

So how does the whole thing work? The user generates his X.509 certificate. This is sent to the server administrator, and signed using his certification authority. The signing of the client certificate by the certification authority of the server renders the certificate usable for logging on to the given server. The signed certificate is saved by the user a reused when logging the client on to the server.

The server administrator specifies a group of users with the right to access the server by only signing their certificates (**authorization).** Any other users without the signed certificate cannot successfully connect to the server.

The user can have his X.509 certificate signed by several certification authorities, thus the possibility to log on to several servers using one certificate.

By using the users server-signed certificate the user is **authenticated** on the server. It is still necessary to check, if the client is really connecting to the server to which he desires, or if it is an attacker that has placed a fake server and is trying to extract sensitive data from the

user, etc. In other words server authentication has to be done. This is done using a certificate of the server certification authority. Only servers with a certificate signed by the given certification authority is accepted, otherwise the connection is terminated.

The process of client-server login:

The client establishes a TCP connection with the server. The client exchanges certificates with the server. The server authenticates the client using his certification authority. The client verifies the server using the certificate of certification authority. If the whole verification process is successful, an encrypted connection is established, which is used until the client log off the server.

Encryption is done on the basis of two encryption keys, one public and one private. Both the client and the server have their own set of two keys. The private key always remains with its owner, no one must acquire it, so that there is no possibility of a data leak. The public key, which is generated on the basis of the private key, is sent to the other side. The transmitted data are encrypted using the public key. The recipient then uses his private key to decode it.

TCP Wrappers

The jpQOS framework can be used in combination with the inetd and TCP Wrapper tools. Inetd is a superserver used to run actual servers. Instead of the jpQOSd daemon (server) running continuously and waiting for a connection from a client, it is just necessary to register him at the superserver inetd. The superserver checks all registered ports and in case of a new connection request on any of them it runs the corresponding registered server, which will process the clients request.

This principle has two main advantages. First, it minimizes the waste of system resource. The actual servers are only run when needed, they do not wait idly. Second, a connection check can be made. The connection check is done directly using xinetd or another TCP Wrapper tool. For example IP addresses can be checked, domains, user names, etc. If the check is positive, the connection is established, otherwise it is immediately terminated and the server is not called at all. The advantage of this approach is that the check itself is completely transparent for the program itself and it does not have to deal with it. Everything is administered by the system administrator via the superservers and TCP Wrappers' configuration.

5.4 Client

The clients' task, according to the design of the jpQOS framework, is to allow the user to comfortably set up the QoS. The client part of the framework does not deal with low-level communication with the kernel via rtnetlink, etc., all of this is done by the server, with whom the client communicates via the Communicator module. The Communicator module makes the servers functions accessible to the client on a high level (commands like configure the QoS, check current configuration, turn on/off monitoring are transmitted through it).

The communication of the client with the server is taken care of; the main focus of the clients work is on allowing sophisticated configuration of the QoS by the user, i.e. on the creation of the data containing the configuration which are then sent to the server for

processing. And vice versa, based on the data received from the server, the client should create a well-arranged and schematic diagram describing the QoS configuration comprehensively and intelligibly to the user.

To provide a well arranged and user friendly interface, it was decided to use a GUI (graphical user interface) instead of a script editing based configuration. Qt C++ graphic toolkit [TRO06] was chosen for client implementation. The Qt toolkit allows the creation of multiplatform GUI application based on object programming in C++. In addition it offers good documentation. The Linux dependant part of the server and the multiplatform GUI application allows for example setting up the QoS configuration on a Linux router under MS Window, etc.

The client is based on the Qt toolkit, thus it is an event driven application. The client application consists of the main application window *ConfigMWindow*, which determines the visual design of the application and the layout of the main controls (main menu, toolbars, tab bar…). It also takes care of the connection of these controls with the implementation of their respective functions using the Qt – signal and slot principle.

After the client is started it first establishes connection with the server via the Communicator communication module. The client allows only one simultaneous connection to one server. The configuration of more servers at the same time has no practical application, if needed several clients can be run. Without connection to the server the client does not allow any operations, everything depends on the network interface used.

After connection is established, all network interfaces on the server are detected. To each network interface a *DeviceManager* is assigned, which thence-forth manages all the data and controls of that particular interface.

After choosing a network interface we can start configuring its QoS. The QoS in Linux is network interface oriented, that is why the client configuration is also designed in this manner (it is not possible to create one configuration for more interfaces etc.)

The configuration of the QoS means a node tree is created (QoS tree), where each node corresponds to a queuing discipline and its configuration parameters (see analysis). The data format of this tree is already prepared – the universal data storage (see Data Management). The client uses the same data format as the server. This is an advantage, because there is no data format conversion necessary in the communication between the client and the server. In addition, the client and the server work with the data similarly, i.e. there is no need to create two different accesses, there is no confusion and it is better arranger. Having described the data format of the QoS configuration, the client also needs a graphical appearance for the configuration. This is done using a few standard widgets of the GUI (frame, table, button, toolbar, splitter, canvas...). These widgets have to be somehow connected with their data representation. This is achieved using the *TreeNode* and *QOSTreeManager* objects. *TreeNode* represents one node of the tree and controls the set up of the data in its queuing discipline by the user. *QOSTreeManager* manages a whole tree of QoS configurations, makes sure it is correctly diplayed, ensures connections between individual nodes and takes care of global actions on the tree itself.

The QoS set up configuration consists of creating/deleting tree nodes and setting their parameters. There are filters in the QoS configuration that decide which data flow from the given node goes to which node. All is provided by the GUI widgets, see figure 16.

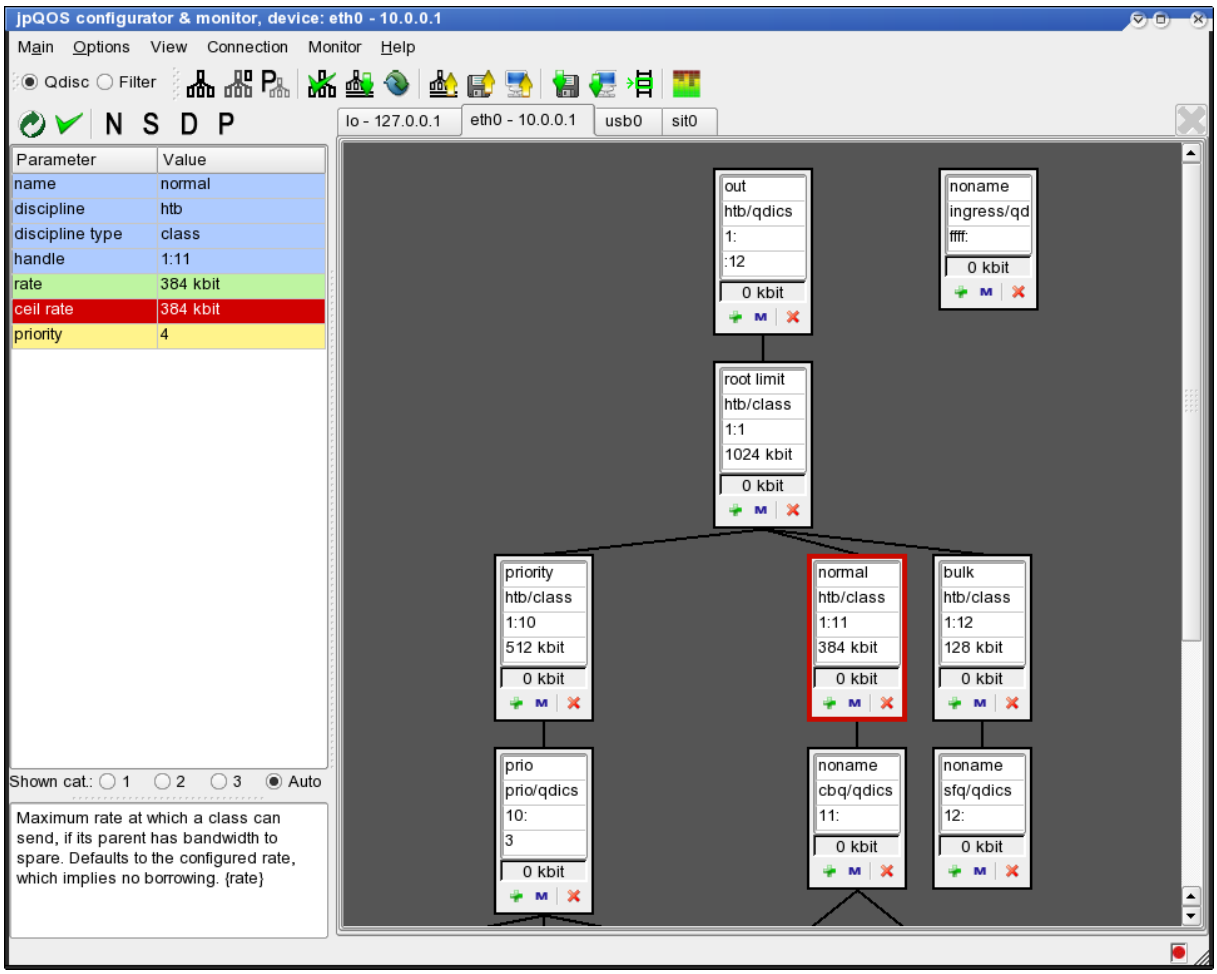


Figure 16: QoS tree configuration tool

The well-arranged depiction of the individual tree nodes onto a 2D picture is implemented in the *QOSTreeManager* and *TreeNode*. A very effective modification of the Walker Tree algorithm for running in linear time [BUC02] is used. The algorithm complies with the following aesthetic requirements for lucidity according to Wetherell and Shannon [WET79]: y-coordinate of the node corresponds to its level, so that the hierarchical structure of the tree is displayed; the left child of a node is placed to the left of the right child. i.e., the order of the children is displayed; finally, each parent node is centered over its children. These requirements correspond precisely with the requirements for a well-arranged display of the QoS configuration structure, which is why they were used.

In addition to a well-arranged depiction of the tree nodes the client has to ensure a intuitive and easy-to-use adjustment of the parameters of the individual nodes (queuing disciplines and filters). This is achieved using a QTable Widget – *PropertyTable*. *PropertyTable* uses all the previously mentioned support principles designed for the easiest possible, most effective and most user friendly adjustment of the parameters by the user.

The adjusted parameters are unknown to the client in advance (the framework is universal, it has to be able to cooperate with any queuing discipline, even ones unknown at the time of its creation), thus they cannot be “hardcoded”, there has to be a way to inform the client, what options should he offer to the user to edit. For this purpose the Component template scripts are used (for details see chapter 5.2.4 – Component template scripts), where the individual parameters of the components are described. The scripts are then converted into

a *DataStorage* (see chapter 5.2.6 – Data management) format using the *templates_xml* module, in which they are forwarded to the client. The client uses the data in the DataStorage to create a *PropertyTable* with a list of parameters in the adjusted queuing discipline, default values are set and a basic checking of user set values is performed (type match). Besides the default values the user can also turn on recommended values (see chapter 5.2.4 – Component template scripts). The adjusted parameters are also divided according to their significance (basic/advanced/special) for better lucidity (for details see chapter 5.2.4).

In addition to the type match check of the user input values a more detailed check of parameter values is also performed in accordance with the adjusted queuing disciplines rules. This check is performed in cooperation with the server via the component manager directly with the implemented validating function of the given component. This versatile system allows the checking of any type of complicated component rule system.

The basic parameter value check runs automatically after the values are input by the user. Furthermore a complete check can be initiated to check the settings of the whole component (links between individual parameters are checked) or a check on the settings of the whole QoS tree (to check links between the individual tree nodes).

To make the work with the QoS configuration trees easier, the client allows copying of whole sub-trees, including filters. While copying, new nodes are created and the settings of their original nodes are copied into them (deep copy).

Besides adjusting setting on the network interface, the tree configuration of the QoS created by the user can also be saved in the XML format into a file on the disk (for more details see chapter 5.2.10 – QoS Settings File). And of course, existing QoS configurations can be loaded from files or server network interfaces.

5.5 Logging, Error messages

The jpQOS framework contains system of logging run of the program – for server and client alike. The logs are divided into five categories: critical error, error, important info about program run, program run – brief statement, program run – detailed statement. While running the program the user can choose, how detailed the logs should be. The logs are displayed by default to the system console or stderr.

5.6 Build tool

The jpQOS is based on the qmake build tool. Qmake is a tool created by Trolltech [TRO06] to write makefiles for different compilers and platforms. Qmake takes care of all the compiler and platform dependencies, Qt's special requirements,... Qmake generate the appropriate makefiles from a simple single 'project' file.

6 Proof of concept

Practical utilization of the jpQOS framework on an Asus WL-500b router

Testing jpQOS framework on a regular linux router, like Asus WL-500b with OpenWRT Linux distribution, is a better way of checking the framework than testing it on a classic PC with a network card. It is because configuring QoS on routers is far more common and useful than on a classic computers, so it is a real test in practice.

6.1 Initial requirements

To use the jpQOS framework on the router the following requirements have to be fulfilled:

1. The router has to have the required libraries installed and configured. Specifically, it is the C++ library (libstdc++), the secure connection library (openssl) and a data serialization library (boost-serialization).
2. The jpQOS framework must be compiled and linked for the intended architecture (all components except the client). This is not a problem thanks to the concept of the program and the build tool used, since architecture independence is ensured.
3. The router must operate on a Linux with a kernel containing QoS support.
4. It has to contain the installation of the jpQOS itself, its binary files, and shared libraries, support libraries for the disciplines & filters and the configuration files divided into particular directories.
5. Have the jpQOS server access security properly configured. Create the X.509 certificate and the private key and configure the certification authority. Create and sign the certificates for clients with access rights.

After these conditions are met, the framework server can be started. This can be done from the command line using the "*jpq start / stop*" supportive script or better yet, using a configured inetd / xinetd superserver that will start the server on the clients' request, only when it is needed. It saves the routers valuable system resources.

After running the server on the router, the client has to be configured in order to utilize the created X.509 security certificates and use them to connect to the server. After establishing encrypted TCP connection between the client and the server, it is possible to start with the QoS configuration on the router.

6.2 The Situation Analysis

The situation at analysed network is following: The Asus WL-500b router contains four network interfaces. One of them eth1 (WAN) is used for the communication with the outer network (the Internet) the others are used for the communication within the local network. The eth0 interface is used for the communication via wire ethernet (LAN). The eth2 (WLAN) interface is used for the communication via wifi and the interface br0 bridges both interfaces for the local network - eth0 and eth2. The traffic control is in Linux based on individual network interfaces. Each interface configuration is created separately, therefore it is especially appreciated bound br0 interface that enables to configure the whole local network

at once. If that bound br0 interface does not exist, the situation can be solved by using IMQ (Intermediate Queueing Device) [IMQ04].

Local network within the router enables the mutual communication between computers and mainly provides the Internet connection for all computers. The eth1 interface is directly connected to the Internet via cable modem. Local communication is at a low level because of the network capacity so there is no need to treat it by the QoS. The bottleneck is the eth1 interface that transmits data at a bandwidth negotiated with the Internet provider. In this case it is 512 kbit/s upload and 4096 kbit/s download. So the best results will be achieved by setting the QoS on this interface (eth1). It is reasonable to set up the QoS where the line is overloaded and It has to be chosen what to transmit first; not where the bandwidth is sufficient for all the required data transmission.

Requirements on this particular network are these:

- First of all it is to divide the bandwidth equally among the users. Each user (a computer connected to a local network) will be guaranteed a particular section of the bandwidth and can use the rest of it in case the others do not use up their parts.
- Another requirement is the division according to the type of network services or type of the data transmitted. Some data can be delivered later (e.g. e-mail), other data has to have priority and has to be delivered immediately (e.g. video conference).

6.3 Setting up the configuration

According to the analysis and the requirements, the following configuration is set up. Firstly it is set up the output data on the eth1 (WAN) interface, which is the upload in direction to the Internet. By configuring the QoS at this interface it is ensured that the data sent by local network users to the Internet is delivered in the order that is required.

It has decided to use the HTB discipline because it satisfies the requirements: the bandwidth division possibility, the speed-level guarantee and the share of the unused capacity. Moreover, it has got better features than a similar discipline CBQ, see the measurement in the Martin Devera's document – "Princip a užití HTB QoS disciplíny" [DEV02a].

The QoS configuration is created by the nodes in a tree. Each node represents certain class of the data flow and defines how to treat the flow. The parent node defines the features that its children nodes will have in common. While creating the QoS configuration tree it is necessary to proceed from the required properties according to their priority, from the most important to the least important.

As a root of the QoS configuration tree it is set up the class (1:1) which determines the final flow of all outbound data. It is configured so that the resulting data flow won't exceed the negotiated upload limit (512 kbit/s), so long queues occurring in the cable modem is not used and it could be decided which data are sent first without waiting. 8 child classes (1:10 - 1:17) is put under this class (1:1). These child classes serve for dividing the flows from every single user. This way each user has guaranteed his particular minimal flow and moreover, sharing the unused bandwidth among the more demanding nodes is ensured. It has to be ensured that data are sorted into the right classes, It is achieved by using filters. Data is filtered according to source IP addresses. There is a slight problem - data in interface eth1 (WAN) has already passed the network address translation (NAT), so their source IP address is equal to the routers address and is the same for all network packets. But it can be solved

quite easily - by marking packets already when they reach routers local interfaces eth0 and eth2 - when the source IP address still exactly defines the user. Iptables tool is used for marking. Each user gets a unique mark. Then it is added 8 filters to the QoS tree root - one for each child node. FWMARK filters are used - every filter compare its settings with each packet mark and if these match it put the packet to an appropriate class. So the first requirement is accomplished - fair division of bandwidth among all users. See the figure 17 - the screenshot taken from the jpQOS framework client.

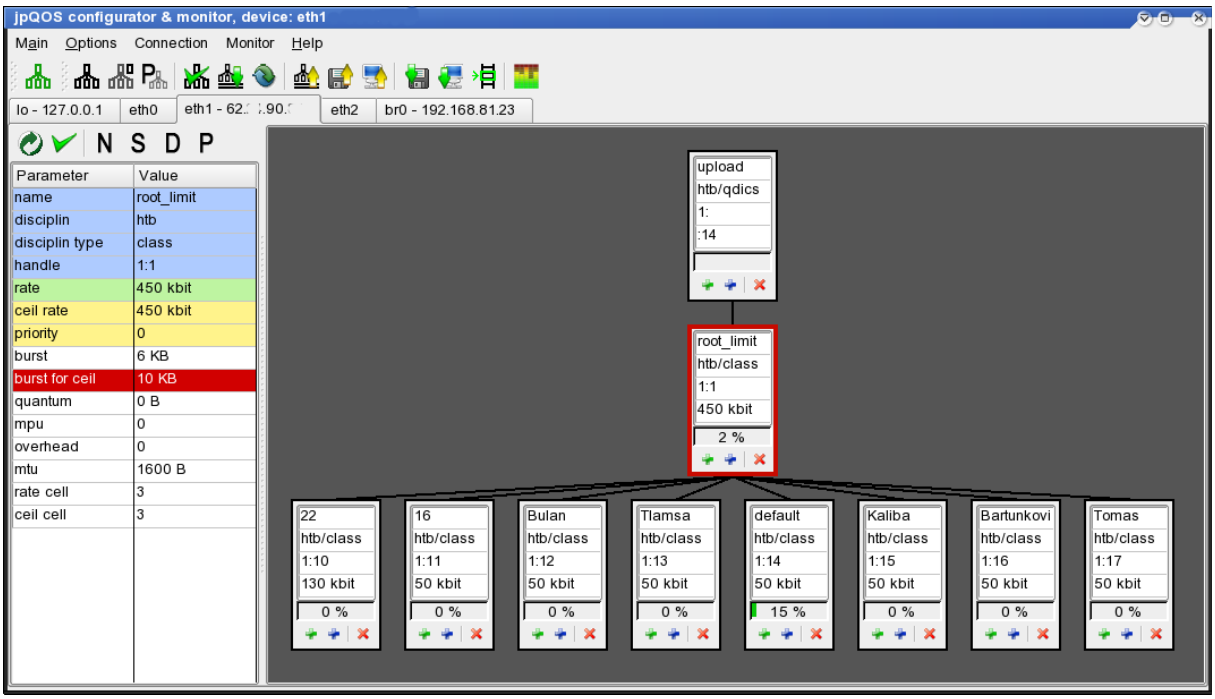


Figure 17: Division of bandwidth among users

The second requirement is division of the bandwidth according to the type of transmitted data. The data is divided into three main classes: priority one, normal one and bulk. The data which is wanted to be transmitted first - thus with a minimal delay - is filed into the priority class. It has been chosen packets with TOS (Type of Service) bit set on minimum delay, which is used by some of the services such as SSH, ICMP protocol packets, ACK confirmation packets for accelerating the download speed, packets of the telephone program Skype and web pages packets. It has been also divided the priority class into subclasses according to different types of services - this way it has been ensured the possibility of defining the guaranteed bandwidth for each separate service as well as finer division according to priority. Data which has no time limits for delivery is filed into the bulk class - it is sent when the bandwidth is free (for example the p2p site packets). The rest of the data, that are packets which do not qualify into any of the previous classes, are filed into the normal class. That data is sent after the priority class data and before the bulk class data. Qdisc of the discipline SFQ is put under all classes. This will guarantee fair division of the bandwidth, among all programs whose data belongs to the particular class.

After the hierarchy of nodes has been created, it has to been added filters which sort the data into the proper classes. The filters are added into the root class. U32 filters are used, which can classify data flows into the particular classes according to above-mentioned requirements by analyzing the IP header and the IP packet data.

The new QoS classes tree has emerged from the data type division (see the figure 18). This tree is added under each node of previous tree (1:20 – 1:17) – which was created while fulfilling the requirement no. 1. Now it has been fulfilled all requirements for upload, so it could be continued with setting the download.

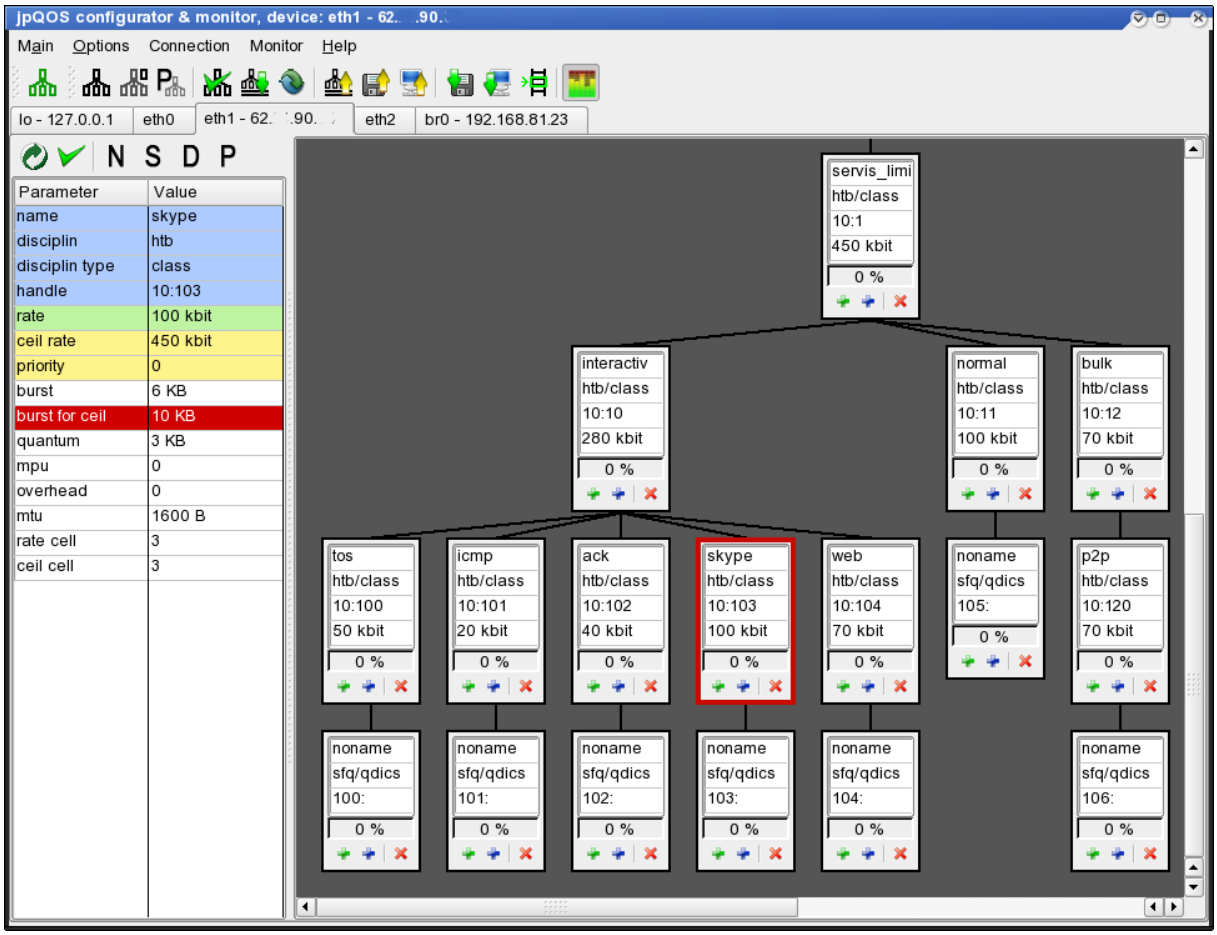


Figure 18: Division of the bandwidth according to the type of service.

The best way to configure QoS for download would be to set it up right away at the ISPs (Internet Service Provider) router, which is placed just before the bottleneck of the line. However, this is not possible due to missing access right to ISP router, so a detour have to be made. By limiting slightly the download speed at the router via the TCP data flow speed management, decreasing the speed of packets to the desired level is ensured – which allow to choose which data flows have precedence. Thanks to the slightly decreased speed the data is not delayed in long queues on the ISPs side. Otherwise these queues would prevent from preferring some of the data flows, they send the data according to the time it arrived, FIFO, possibly merged in large blocks.

Download traffic from the Internet into the local network can be set up at two interfaces. The first one is the interface eth1 (input stream – ingress qdisc). The second one is the interface br0 (output stream – egress qdisc). Configuration of the QoS at the interface eth1 has two disadvantages. The first one is that configuration at the ingress qdisc is very limited – it is only possible to limit the flow at the entire interface by using the policing filter, no specification is available. This problem could be avoided by using the IMQ [IMQ04]. The second disadvantage is that it would be difficult to distinguish the packets according to users because the destination IP address would be set to the routers IP address in this case. To avoid

these problems it's better to configure QoS for download at the interface br0. Packets have already passed the NAT so their IP addresses are set up to the local network users addresses. Interface br0 connects interfaces eth0 and eth2, so by configuring it, it can be set up the whole network communication in direction towards the local network. Configuration is done similarly as setting up the upload traffic at the interface eth1 because the requirements are same. The only difference is it have to be set up different speed rates because the download is faster (4096 kbit/s).

Conclusion:

Several measurements have been done after configuring the QoS on the router - partly a "sense" comparison of the speed of reactions in the interactive programs during the maximum load of the bandwidth. In these cases the positive effect of the data preference settings became evident. The Internet telephony, SSH reaction and loading of web pages during the maximum load of the bandwidth caused by downloading data via p2p protocol, was very fast, fluent and without any dropouts. Moreover, thanks to the jpQOS framework it is possible to monitor everything: how the individual data flows are being classified into the configured classes; how much of the allocated bandwidth they use; whether the bandwidth is used completely or whether the unused bandwidth is being shared.

Then several more exact measurements of the reaction speed, delay and rate have been done. Tests confirmed almost precise adherence to the speed rates guaranteed for particular data flows, decrease of priority flows delay. The results of the tests of the maximum upload or download speed are worse than before the configuration of the QoS. However, this was expected because there was need to decrease the speed slightly – under the maximum limit – so as it could avoid long queues at the cable modem and at ISP.

Thanks to the simulation of the jpQOS at the Asus WL-500b router it have been also managed to verify the independence from the hardware architecture – it is possible to use the server at a different architecture than x86. The mutual communication with a client running at a different architecture than the server (x86) is no problem as well.

HW configuration of the Asus WL-500b router is: MIPS32 BCM4710 processor at 125MHz, 16 MB RAM. The router meets the HW requirements of the jpQOS server; however, regarding the slow processor it is necessary to expect longer responses during the communication with the client than when the usual PC is concerned (in the case of larger QoS configuration). Memory requirements are also fulfilled, even though the jpQOS server is quite demanding. It uses many libraries (std++, boost, ssl, tc, xml) and also the code itself isn't negligible, moreover, it contains buffers for data income and outcome in the network. Resulting size of the required memory is approximately 7 MB. (3.8 MB std. libraries, 1.5 MB own libraries, code, 1,5 MB program data). However, the server must run only whilst configuring or monitoring the QoS, after the configuration is set it is possible to switch it off.

Using the framework jpQOS at the Asus WL-500b router turned out well regarding all the aspects, so I consider the test to be successful.

7 Evaluation

The aim of this thesis was to analyze the infrastructure offered by the Linux kernel for the administration and configuration of Traffic control. Then, based on this analysis, design and implement a universal and extensible framework, which would allow an effective and user friendly configuration and monitoring of the QoS settings in Linux (see chapter 1.2).

An in-depth analysis of the infrastructure offered by the Linux kernel was done and described in chapter 2. Low level infrastructure were analyzed – rtnetlink communication interface, kernel modules with queuing disciplines, as well as existing tools for the configuration of Traffic Control by the user – TC and TCNG. The analysis was done both by studying the documentation as well as by studying the source codes and practical testing. Thanks to looking into the problem from several different viewpoints a large number of discrepancies between the functions described in the documentation and the behavior observed during testing have been discovered. This is due to the fact, that most of the documentation is not written by the actual author of the application but a user with no in depth knowledge of the problem. Or it is caused by bugs in the Traffic Control resources that were not fixed yet. The list of main discrepancies is in Appendix C. Insufficient documentation and errors in the traffic control system caused many problems during the creation of this application. Considering the problems with the rtnetlink interface (see chapter 2.9.1 – analysis, netlink socket), it might be advisable to rewrite this interface or at least to expand it with a more detailed system of error messages and input parameter checking.

A thorough analysis gave the idea what the current status of traffic control management in Linux. Based on this analysis a solution was proposed (chapter 3), that would remove the main problems. The ideas of this solution were then implemented in the jpQOS framework.

One of the main objectives was the universality and extensibility of the framework. All of the framework components were designed with this objective in mind. This is why it was successfully managed to remove all dependence on a specific queuing discipline or filter, using dynamically loaded shared libraries and a component manager to work with them. This ensures good extensibility, further supported by the jpLibMaker utility tool which makes it easier for the author of the new component to create a support library. All the other parts of the framework work with each component universally, which allows extensibility without the need to rewrite the framework code. On the other hand this universal approach is not that effective, and it requires more memory and is more complicated to implement. Due to this universal approach the adjustment of some parameters is not as intuitive as it could be if we would have tuned the configuration dialogs to suit a specific queuing discipline.

Another main objective was to create a unified, easy-to-use, well-arranged and effective QoS configuration system. This was achieved by a sophisticated system of GUI widgets, a well arranged display of the nodes using the Walker tree algorithm and by using all the designed supportive principles for the QoS setup – Component template script (categorization of the parameters, help), Data Management , QoS Setting file, etc. The ease of use of the GUI elements is a subjective feeling; everybody has his own preferences. We have tried to create an easy to use and intuitive way to achieve a goal, however there still is much to perfect. The possibilities of improving the GUI are unlimited; this is a task that is never entirely finished.

Another one of the objectives was to make QoS configuration possible via remote settings. This would allow users to easily configure even those devices that are physically difficult to access (e.g. routers). This was achieved by dividing the framework into a client and a server part. All the tasks connected to this we also taken care of – data transfer on the network, authentication and authorization, platform specifics. We made use of the Boost and OpenSSL libraries for this purpose. There are some minor difficulties with the encrypted connection of the client and the server on devices with lower HW performance and slow processors, because the operations' requirements for computing capacity are high and they can burden the process and even slow down communication. The choice of the X.509 certificate for authentication and authorization ensures security, but is limiting in some cases. The client has to have his X.509 certificate every time he wants to access the server, so if the wants to use a different computer to access the server, he cannot do so without his certificate.

One more goal that was set was the run-time monitoring of the QoS. This was achieved by scheduling regular measurements of data transferred by individual tree nodes of the QoS configuration. The data are acquired, sent from the server to the client, which then represents them as data flow graphs of the individual classes. The graphs only display the current sent data flow. It probably would be useful to expand the monitoring capabilities of the application by adding long term measurement and saving the measured data, which then in turn could be used to compile graphs of data flows for the individual classes for example for the whole day. It would also be useful if we could not only monitor sent data but also queues of packets waiting to be sent in the individual classes. Unfortunately such information are not supported by the queuing components.

7.1 Future work

1. To allow running the client part of the framework under MS Windows. All modules are designed and ready for this, they are platform independent. However the license of the Qt library used, the version 3 for MS Windows is paid. The Qt version 4 is free, thus it is necessary to rewrite the client to use version 4 of Qt.
2. To allow encryption, authentication (OpenSSL library) and authorization to be turned off. Currently it is an integral part of the communication module. In some cases the use these is unnecessary, or secured by another device (external).
3. Tuning of the component parameter adjustment using the PropertyTable. Widen the possibilities for the user – choice of rate for the node in percentage. Expand the interface of the component by adding the possibility to interactively modify the property table while it is being edited by the user. Optimized display of large QoS configuration trees. The possibility to create several tabs with different views of the same QoS.

8 Related Work

The jpQOS framework is very hard to compare with related applications, since there are no works carried out on the same level. Other works from the field of Traffic Control in Linux only deal with the problems partly, not in a complex fashion as the jpQOS framework.

The main tool for the configuration of Traffic Control in Linux is the **TC** by Alexey Kuznetsov [KUZ03]. Analysis of this tool is described in chapter 2.9.2 – Analysis, TC. The jpQOS framework is designed to solve the problems found during this analysis. The extensibility and the possibility of remote settings are on the same level in both TC and the jpQOS framework. The jpQOS framework does however offer a far higher level of comfort, lucidity and ease of use via the configuration and monitoring using GUI.

A large number of problems during configuration using the TC were the reason for the creation of **TCNG** (analysis see chapter 2.9.3) [ALM02]. This tool, as well as the TC, is based of configuration using text scripts, but offers a far more lucid and easier way of notation and a higher level of abstraction. But unlike the TC and jpQOS framework it has many problems with extensibility regarding new components, because the component specifications are hard coded in the TCNG implementation.

Another tool for Linux Traffic Control administration is the **TCAPI** [OLS02]. However the TCAPI is only a library allowing the configuration of the QoS via functions in the C programming language. The analysis of the TCAPI can be found in chapter 5.2.1 – The configuration of Traffic Control Primitives. Hence, the TCAPI is not a tool for the user, moreover its architecture is not designed to be universal, i.e. all the queuing components are hardcoded making extensibility very difficult.

A variation of the TCAPI is the newer **LIBNL** library. The LIBNL is a library for applications dealing with netlink sockets. The library provides an interface for raw netlink messaging and various netlink family specific interfaces. The LIBNL therefore only solves part of the problems associated with QoS configuration – it creates a better API for communication via rtnetlink. The LIBNL library roughly corresponds to the TC module in the jpQOS framework taking care of low-level configuration of the QoS.

ALTQ: Alternate Queuing for BSD UNIX is a tool of comparable level to the jpQOS framework, it solves all the problems associated with QoS configuration, from the low-level to the high-level user configuration. Unfortunately it is built on BSD, thus it not usable on Linux because of the different network architecture.

The only QoS configuration tool allowing graphical configuration like the jpQOS does, is the **TCGUI** tool. It is an application written in java, which allows the user to create a QoS configuration via a GUI, which is then converted into TC scripts. TCGUI can be extended with new component using XML descriptive scripts. Unfortunately this application is only in its early point of development, further development does not continue and currently it only supports the HTB and SFQ disciplines. The QoS administration is only one-way, the TCGUI can only create a new configuration via the GUI and convert it into TC scripts, it cannot detect the current configuration, display a graphical representation from the data obtained from the TC scripts etc. Monitoring is also not implemented into the TCGUI application.

9 Conclusion

The result of this work is the jpQOS framework, which is an application allowing users to configure and monitor Traffic Control Facilities in Linux. The main benefit of this application is its universal approach to the configuration of the QoS and good extensibility with new queuing disciplines and filters. Another benefit of the jpQOS framework is the sophisticated tool for the configuration and monitoring QoS using the GUI. Using a unified and intuitive graphical interface makes the QoS configuration easier, more effective and more lucid for the user. Thanks to the client/server architecture, emphasis on platform independence and minimalist server requirements, use of encryption and authentication, it is possible to easily and securely use the jpQOS framework on routers and to configure the QoS remotely.

Thanks to all these benefits, an application has been created that has all the prerequisites to become widespread between the users. It has prerequisites for allow further development and prerequisites for using in practice also in the future.

References

[ALM02] - Werner Almesberger: *Linux Traffic Control – Next Generation*, Linux-Kongres, 2002

[ALM98] - Werner Almesberger: *Linux Traffic Control – Implementation Overview*, Technical Report, EPFL, 1998

[BAL03] - Leonardo Balliache: *Differentiated Service on Linux HOWTO*, 2003, http://www.opalsoft.net/QoS/DS.htm

[BLA98] - Blake S., Blake D., et al.: *An Arichtecture for Differentiated Services*, RFC 2475, 1998

[BRA94] - Braden, Clark, Shenker: *Integrated Services in the Internet Architecture*, RFC 1633, 1994

[BUC02] - Ch. Buchheim, M. Junger, S. Leipert: *Improving Walker's Algorithm to Run in Linear Time*, Universitat zu Koln, 2002

[COX06] - Mark J. Cox, et al.: *OpenSSL Project*, http://www.openssl.org/, 2006

[DEV02a] - Martin Devera: *Princip a užití HTB QoS disciplíny*, Linux & Tex Seminar, SLT, 2002

[DEV02b] - Martin Devera: *HTB Linux queuing discipline manual*, user guide, 2002

[DHA99] - Gowri Dhandapani, Anupama Sundaresan: *Netlink Sockets – Overview*, Technical Report, The University of Kansas, 1999

[FLO95] - Sally Floyd and Van Jacobson: *Link-sharing and Resource Management Models for Packet Networks*, IEEE/ ACM Transaction on Networking, 1995

[IMQ04] - Linux IMQ - Intermediate Queueing Device, http://www.linuximq.net/

[LAR04] - Linux Advanced Routing and Traffic Control, http://lartc.org

[KUZ03] - Alexey Kuznetsov: *Iproute2*, Documentation, 2003

[OLS02] - David Olshefski: *TC API*, Technical Report, 2002

[OMG02] - OMG: *Corba, Common Object Request Broker Architecture*, 2002

[MOR06] - Andrew Morgan, Thorsten Kukuk: *Linux-PAM (Pluggable Authentication Modules)*, http://www.kernel.org/pub/linux/libs/pam/, 2006

[MYE97] - John Myers: *Simple Authentication and Security Layer (SASL),RFC 2222, 1997*

[PET06] - Jiří Peterka: *Rodina protokolů TCP/IP*, verze 2.3, Lecture, 2006

[RAD99] - Saravanan Radhakrishnan: *Linux - Advanced Networking Overview*, Technical Report, The University of Kansas, 1999

[TRO06] - Trolltech: *Qt3 C++ toolkit*, http://www.trolltech.com/products/qt, 2006

[WET79] - Charles Wetherell, Alfred Shannon: *Tidy drawing of trees*, IEEE Transactions on Software Engineering, 1797

Appendix A - User Documentation

Installation

Requirements:

To successfully install, run and use the jpQOS framework the following system requirements have to be met:

- Linux kernel with QoS support, including kernel modules with queuing disciplines and filters
- Installed **OpenSSL** library (development version)
- Installed **Boost Serialization** library (development version)
- Installed **QT3** GUI toolkit (development version) with qmake build tool

Installation:

The jpQOS framework is distributed in the form of a package containing the source codes and configuration files for the qmake build tool. The compilation and installation is done in several steps:

1. Get jpqos package – from CD-ROM / current available version on Internet: http://jpqos.podu.net/jpqos.tar.gz
2. Unpack source tarball – *jpqos.tar.gz*
3. Setup *jpqos.pri* file – optional
   (in this file there is the overall configuration of the compilation – the type of compilator & linker used, the optimization level set, paths to the libraries used in case of a non-standard installation)
4. Run **qmake** from jpQOS root directory
5. Run **make** from jpQOS root directory
6. For installation to standard system directories, run **make install**
   (installation is not necessary to run the application, all can be run from the *bin* subdirectory)

Before run:

The jpQOS framework uses X.509 certificates for authentication and security of the network connection. Certificates and keys for the server and the client have to be prepared before the jpQOS can be used (see security section for details). To enable quick testing, the jpQOS framework is distributed with pre-created certificates, however their use for common jpQOS usage is not safe!

Settings, paths, certificates:

The jpQOS framework is installed into these directories by default:

Binary of the client jpmanager	- /usr/bin/jpmanager
Binary of the server daemon jpqosd	- /usr/sbin/jpqosd
Binary of the support tool jplibmaker	- /usr/bin/jplibmaker

Configuration files editable by the user:
(to allow proper run of the daemon, if the framework is installed into different paths than the given default ones, it is necessary to set the corresponding paths in the environmental variables)

/etc/jpqos/templates - component template config files (jpQOS_TemplateConfigPath)
/etc/jpqos/settings - QoS settings files (jpQOS_SettingsPath)
/etc/jpqos/demoCA/cacert.pem - certification authority certificate (jpQOS_SSL_CA)
/etc/jpqos/certs/server_cert.pem - server certificate (jpQOS_SSL_Server_Certs)
/etc/jpqos/keys/server_privkey.pem - server private key (jpQOS_SSL_Server_Key)

Starting script for the daemon (distribuce SuSE) - /etc/init.d/jpqos

Shared Components library path - /usr/lib/jpqos

Documentation - /usr/share/doc/jpqos

Dynamic Linked libraries - /usr/lib/ (libjpData.so, libjpComponent.so, libtc.so, libtinyXml.so)

To run the server, it needs to have access to its public X.509 certificate, private key and certification authority, which is used to verify certificates of connecting clients.

Prior to connecting, the client has to set the path to his public X.509 certificate, private key and the directory containing the certificates with the certification authorities of the server that he has rights to connect to.

Run

The **Server** is run by the following command:

```
jpqosd    [p port# (std=6891)]
          [d debug_mod# (0-5)]
          [i == inetd mod]
          [s == set QoS on start]
          [q == quit after setting QoS]
```

debug mod - Debug logs have 5 categories (0-4), debug mod 0 means no log will be written, debug mod 3 means log category 0..2 will be written, (debug mod 3 is set by default). See chapter 5.5 – Logging, Error messages for details.

inetd mod – This is a special mod of running the server, where it is started via a inetd superserver. In this case connection with the client is already established and passed on in stdin,stdout descriptors. After connection with the client is terminated, the server is shut down automatically (for more details see chapter 5.3.3, TCP Wrappers).

set QoS on start – The server saves one set of default QoS settings for each network interface. When running the server with this option (-s), these default QoS configuration are set for each network interface.

quit after setting QoS – the automatically quits after setting the default QoS configuration (this option is typically used when the system is booting).

The server is stopped by sending the SIGUSR1 signal.

It is recommend to use the pre-made script for running and stopping the server:

```
jpq start / stop
```

The **Client** is run by the following command:

```
jpmanager [-d debug_mod# (0-5)]
```

The debug mod has the same meaning as in the server. All the other settings are done using the options dialogs in the application.

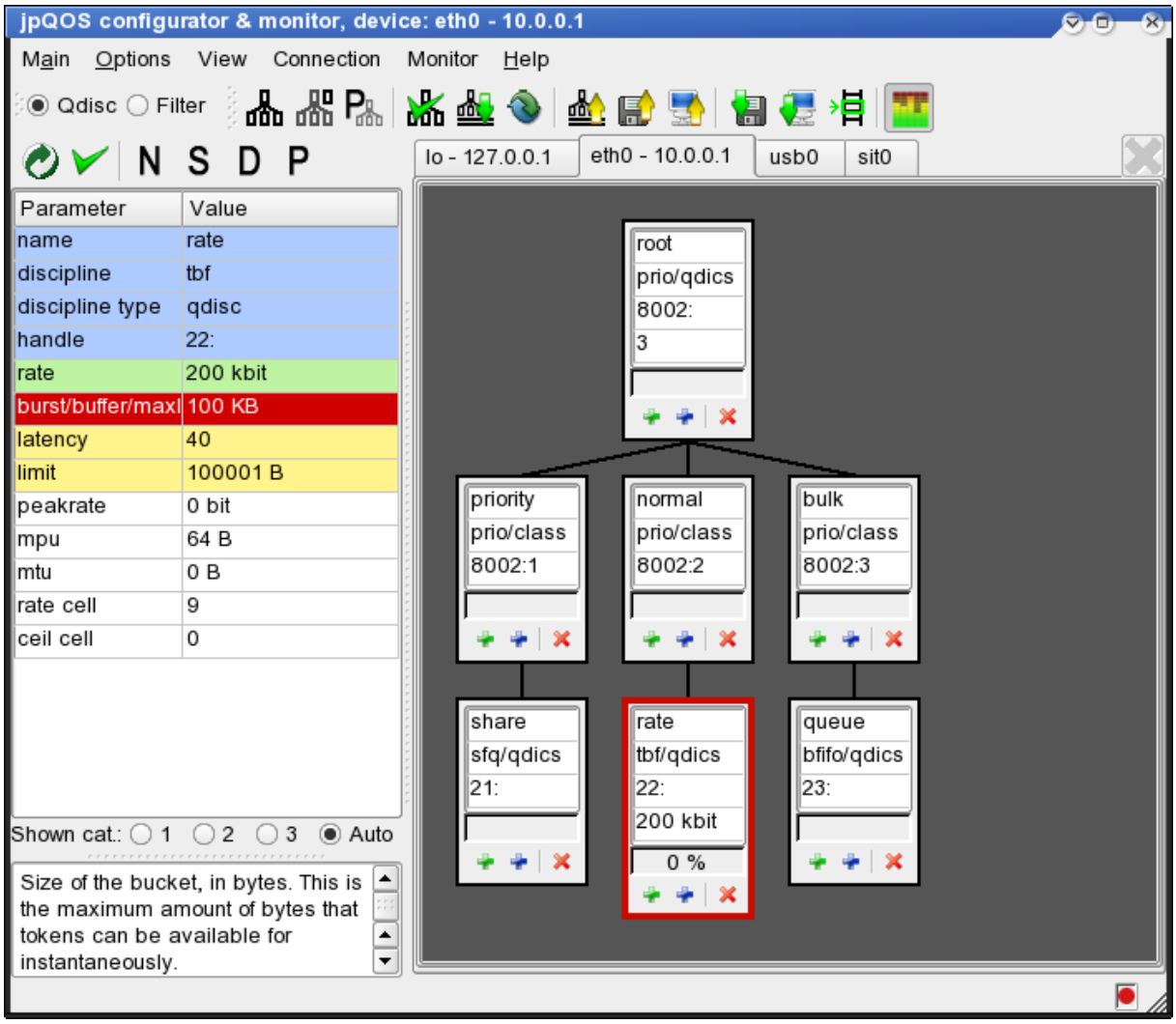


Figure 19: Client program

The QoS configuration is done using the GUI elements of the client. The client first connects to the server it wants to configure. The user than opens the tab representing the network interface he wants to configure. A window containing the currently used QoS configuration is displayed. This tree can now be modified by adding or deleting node, or if desired by deleting the whole tree and starting from blank. The parameters of individual nodes can be modified in the property table, after choosing the desired node. The QoS configuration runs in two modes – qdisc and filter. Depending on which mode is chosen either the queuing disciplines (qdiscs and classes) of the filters are configured.

The configuration is based on the user adjusting the settings for the QoS configuration on the client according to his requirements. This configuration is then sent to the server together with a command to set it up. The resulting configuration, which is set on the server, is then sent back to the client and displayed to the user. The created configurations can also be saved as files, on client and server alike. On the server, each network interface has a directory containing its saved configurations. The user can choose one of these and mark it as the default one. This configuration is then loaded for that individual interface when the server starts.

The main functions offered by the client:

1. create a new QoS tree (ALT+N)
2. create a new ingress qdisc (ALT+I)
3. check validity of the whole tree (ALT+V)
4. use the given configuration on the server (ALT+U)
5. set QoS to default – delete QoS settings
6. load QoS settings from server, currently used (ALT+L)
7. save QoS configuration into a file on the client
8. save QoS configuration into a file on the server (into the directory given by the network interface) (ALT+S)
9. select default QoS settings file on server (ALT+D)
10. load QoS configuration from file on client
11. load QoS configuration from file on server
12. use rate monitoring (ALT+R)

Node functions:

1. Create node
2. Delete node (include whole subtree)
3. Reset node
4. Reload node
5. Validate node
6. Copy/paste node (subtree)

Filter functions:
(they are available after switching to filter mode and choosing the node, in which we want to adjust filters)

1. Create Filter (F)
2. Delete Filter
3. Delete all node filters
4. Validate filter (V)
5. Reset filter(s)
6. Reload Filter(s)
7. Copy/paste filter

Reloading a node or a filter means, that their settings are changed back to the ones they had when they were loaded from the servers configuration. The reset of a node or a filter means, that their configuration is set to the default or recommended values (depending on option settings).

The client allows copying nodes, filters, parts of trees or whole trees. The copied node is selected and marked, either on its own or with his whole subtree. Then a new parent is chosen, under which the node (subtree) is copied. The node can also be copied as a root node of the QoS configuration of a network interface – paste as new QoS tree root (ALT+P).

The client allows loading of QoS configuration from the server or a file into a another new created window, a different one than the configuration window designated for network interface. This function allows viewing QoS configurations in files without the loss of the newly created interface settings. This configuration can also be used for example when copying nodes, etc. Loading into a new window is done by holding the SHIFT key down a pressing the button corresponding to the load QoS configuration on the toolbar or via the main menu.

The actual setting of component parameters is done using the property table. The parameter can be set here either by choosing from a list of possible values (combobox), directly editing the value (editbox) or choosing the node using the mouse (e.g. select handle). Each node can be named to avoid confusion.

When adjusting the values of the parameter, it is possible to go back to the original value using the ESC key, the ENTER key is used to confirm the values. The values are checked immediately for correct input type (string, handle, integer, rate, size..). Rate or size can be input in several ways, depending on the units used (kbit, mbit...). The value is then converted according to units, in which it is input into the kernel and then displayed to the user (it can differ slightly – rounded up).

Filters are set in a similar way as the queuing disciplines, using the property tables. What filter is being configured is given by choosing the node, which contains the filter and choosing the filter in the tabs above the property table. Of course the settings must be done in filter mode.

In addition to configuration the QoS the client also allows **monitoring**. Traffic rate in the individual nodes of the QoS configuration tree is monitored. Monitoring has two modes. In the first one the traffic rate is displayed in the nodes in the QoS tree, in which configuration is taking place. It is displayed in the form of a progress bar and percentage of the set flow rate. In case of nodes with no set guaranteed or maximum flow rate the rate is only displayed in kbit/s. Traffic rate monitoring only works on the tree with the currently used QoS configuration for the given network interface. If the configuration changes, monitoring is turned off. It can only be enabled again if the configuration is synchronized with the server (i.e. applying the modified configuration). The second mode of rate monitoring is the detailed monitoring displayed in the form of graphs for each node. It is enabled by the ALT+M keyboard shortcut or from the main menu and it is displayed in a separate window.

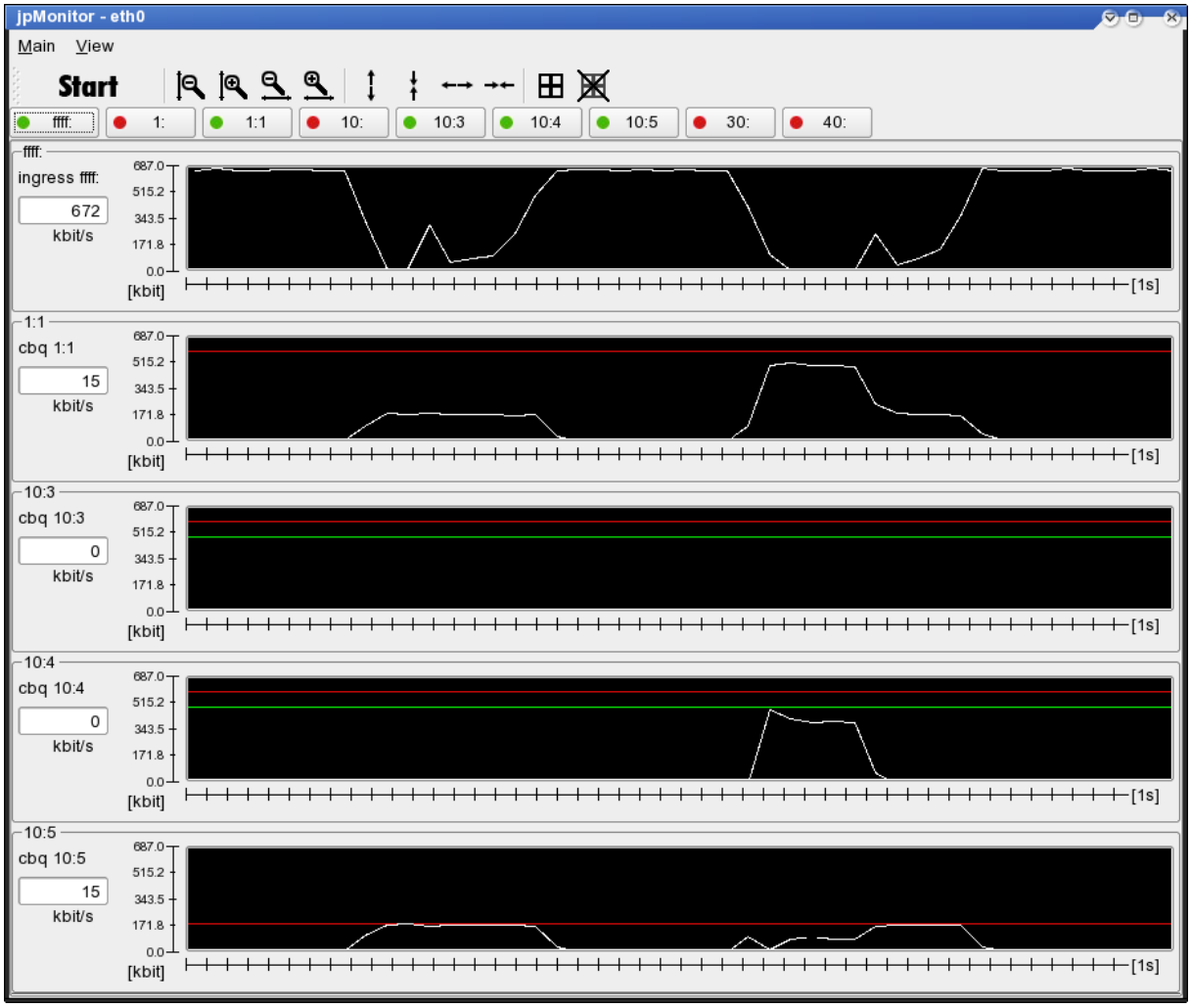


Figure 20: Real-time monitoring

Advanced issues

Editing the QoS settings file

QoS settings files can be created and modified using the client (GUI), but also manually, by directly editing the file. This possibility was preserved in case it is necessary to change QoS configuration and the client cannot be used (e.g. there is no access to X Window). The QSF configuration file is written in XML, has a predefined structure defined in *config.dtd* (for more details see chapter 5.2.10 – QoS Settings File). One configuration file contains the settings for one network interface. The configuration consists of a list of objects (qdisc/class) and filter, their order is insignificant. Each object is defined by listing its parameters and their value. To determine which parameters are available for which component the user can consult the component template scripts.

Setting the Recommended values file

The principle behind the Recommended values is described in chapter 5.2.4 – Component template scripts. For each component we can set a recommended value for each one of its parameters. The recommended values can also depend on the network interface used; hence there is one set of default ones (independent of the interface) and one set for each interface. These values are saved in the DRV files (discipline recommended values) in the *templates* directory and in subdirectories according to the interface. The format of the XML file is defined in the *recommend.dtd*. Each file contains the recommended values for one component. There is a list of structures of the given component (data storage) and each structure contains a list of its parameters with a recommended value.

jpLibMaker

It is support tool for the new QoS component creator. (for details see chapter 5.2.5 - Supportive tool jpLibMaker) Using the jpLibMaker it can be created the source code for a library to support disciplines/filters in jpqos system.
use:

```
jplibmaker <component template file name>
```

(creates a directory "component_name"QoS with prepared source files and makefile)

In requires the component template config file to work. The default component template config path is /etc/jpqos/templates/ (which could have been changed by environment variable – jpQOS_TemplateConfigPath).

Creating your own support library for a new component

If an author of a new component (queuing discipline or filter) wants to allow its setting up using the jpQOS framework, he has to create a library to support it.
First, the component template script has to be created (see chapter 5.2.4 – Component template scripts). In this script he has to describe all the component's structures and parameters. This script is then saved into the *template* directory, which ensure automatic loading of the given component into the framework. Furthermore it is possible to create a file with the recommended values for the given component – discipline recommended values (this is optional).
Based on the component template script and using the jpLibMaker support tool the source code of the library is generated. These source codes are finished if needed and the library is compiled and installed using the prepared makefile.

Managing the X.509 certificate using the OpenSSL tool

Generating a private key:

```
openssl genrsa -out privkey.pem 2048
```

(creates a private RSA key of the length of 2048 bits and saves it into the privkey.pem file; it will not be password protected (if it is password protected, the password has to be entered each time the key is used))

Generating the certificate (public key) based on the private key:

```
openssl req -new -key privkey.pem -out cert.csr
```

(generates a certificate request, which has to be signed by the certification authority; it is possible to generate several certificates from one private key)

Generating a self-signed certificate from the private key:

```
openssl req -new -x509 -key privkey.pem -out cacert.pem -days 1095
```

(such a certificate is signed by itself, thus his validity is not verified by any certification authority)

Working with the certification authority using the CA.pl script:
(a facilitating script, can also be done directly via the OpenSSL)

Creation of a new certification authority:
(according to the rules set in the /etc/ssl/openssl.cnf)

```
CA.pl -newca
```

Signing a certificate request:
(signs the request in the newreq.pem file and saves it in the newcert.pem file)

```
CA.pl -signreq
```

After loading the certificate of the certification authority of the server into the proper directory on the client the following is called:

```
c_rehash .
```

(creates a hash index in the given directory, which is used when loading certificates before verification; the directory can contain several certificates, one from each server to whom the client can connect)

Using the superserver and TCP Wrappers

Configuring the superserver:

The superserver has to know, what services listen on which ports. This is specified in the following file: /etc/services (/etc/rpc)

```
jpqosd      6891/tcp   # jpqosd server
```

Next it has to be defined, what action should the superserver perform after a new connection arrives on that port. This is specified in the /etc/inetd.conf file, or in the /etc/xinetd.d/ directory, each one of the services has its own configuration file. Here it can be specified, which daemon should be run, its parameters, etc.

```
jpqosd   stream  tcp   nowait  root   /usr/sbin/jpqosd
```

When using TCP Wrappers, the tcpd tool is run instead of the daemon. It performs a check and only is this is successful, it runs the required daemon (the name of the daemon is give to the tcpd as a parameter)

```
jpqosd    stream  tcp   nowait  root    /usr/sbin/tcpd     /usr/sbin/jpqosd
```

The set up of the check performed is done by specifying allowed and banned hosts in files: /etc/hosts.allow and /etc/hosts.deny

If using xinetd and TCP Wrappers at the same time, the service (daemon) has to have a flag configured:
`flag = REUSE NAMEINARGS`
otherwise it will not work!

example of xinetd
file /etc/x.inet.d/jpqosd:

```
service jpqosd
{
        flags       = REUSE NAMEINARGS
        socket_type = stream
        protocol    = tcp
        wait        = no
        user        = root
        server      = /usr/sbin/tcpd
        server_args = /usr/sbin/jpqosd -i
}
```

After changing the configuration of the inetd/xinetd the server has to be restarted using the SIGHUP signal:
`kill -HUP inetd`

Appendix B – Tutorial

This chapter describes the basic functions and control of the jpQOS framework in the form of an example. Its contents are running the server and the client, creating a QoS configuration on the client, setting up the server, saving the data into a file and choosing them as default (configuration will be set on system restart)

1. On the machine, where the QoS configuration takes place we run the server
   `jpq start`
2. On the machine on which the QoS is configured we run the client
   `jpmanager`
3. When first running the client we have to adjust the paths to the users X.509 certificate, the private key and the directory with the certificates of the certification authorities of servers (for the purpose of testing all this is prepared in the *bin/client_security* directory).
4. The client has to connect to the server by typing its name/IP address.
5. We choose the network interface tab that we want to configure (e.g. eth0)
6. In the window we will see the current QoS setup for the chosen interface (eth0). If QoS was never set, it is typically one node with a queuing discipline – pfifo_fast (packets are enqueued into one queue, they are sent in order of arrival, fifo, SW speed unlimited)
7. Now we can begin with the configuration. This tutorial will describe the creation of a simple configuration with one priority data flow (TV streaming), normal dataflow of the rest, maximum flow limited to 1 mbit and sharing of unused dataflow.
8. By pressing the create new QoS settings button on the main toolbar or pressing the keyboard shortcut ALT+N, the current configuration is deleted and a new root node of the new configuration is created.
9. The configuration of individual tree nodes is done in the property table, separately for each chosen node. We can choose nodes by clicking on them on the tree.
10. The queuing discipline of the root node is set to HTB in the property table -> parameter discipline -> choosing from the list of usable discipline and confirming the choice (if the queuing discipline of the root is initially HTB, we can leave out this step).
11. If the root node does not have a handle set (for example the default queuing discipline pfifo_fast does not have it set), we need to set it (e.g. 1:0)
12. By pressing the "+" key on the root node we create its child class.
13. This class is used as the root class of the configuration, setting the overall maximum allowed data flow. It also ensures the distribution of the dataflow between its sub-classes. We name the root class "root_limit", to make things more transparent. Its "rate" parameter is set to 1 mbit.
14. By pressing the "+" key after configuring the root class we create two sub-classes in that root class.
15. The first sub-class (handle 1:10) is used for the prioritized data stream – TV streaming. Its "rate" is set to 600 kbit.
16. The second sub-class (handle 1:11) is used for the remaining data, we set its "rate" to 400 kbit.
17. Since the 1:11 class contains data flows from many applications, it is good to ensure its even distribution using the SFQ queuing discipline. We thus create a new child node under the class (1:11) and change it from HTB to the SFQ qdisc (in the property table, discipline item).

18. Now the node tree is ready, we now have to ensure correct data stream distribution between the nodes using filters.
19. By pushing the radio button “Filter” we switch to the filter editing mode (on the main toolbar) (Tab key).
20. For placing the filter we choose the root qdisc HTB (1:0)
21. By clicking the “+” button on the filter toolbar (above the property table) or pressing the keyboard shortcut “f” we create a new filter.
22. We configure the filter in the property table as follows:
    a) set the filter type to u32
    b) set the destination handle to the node, to which the data is sent (1:10); this can be done by selecting the node from the tree using the mouse
    c) we set the selector to select the desired packets; in this case we have the TV streaming application, which sends packets on the port number 2000, so the selector is set to filter based on IP (match item), type destination port (ip item), key 2000 and mask FFFF (a port is a 16 bit number, masking all the set bits of the key)

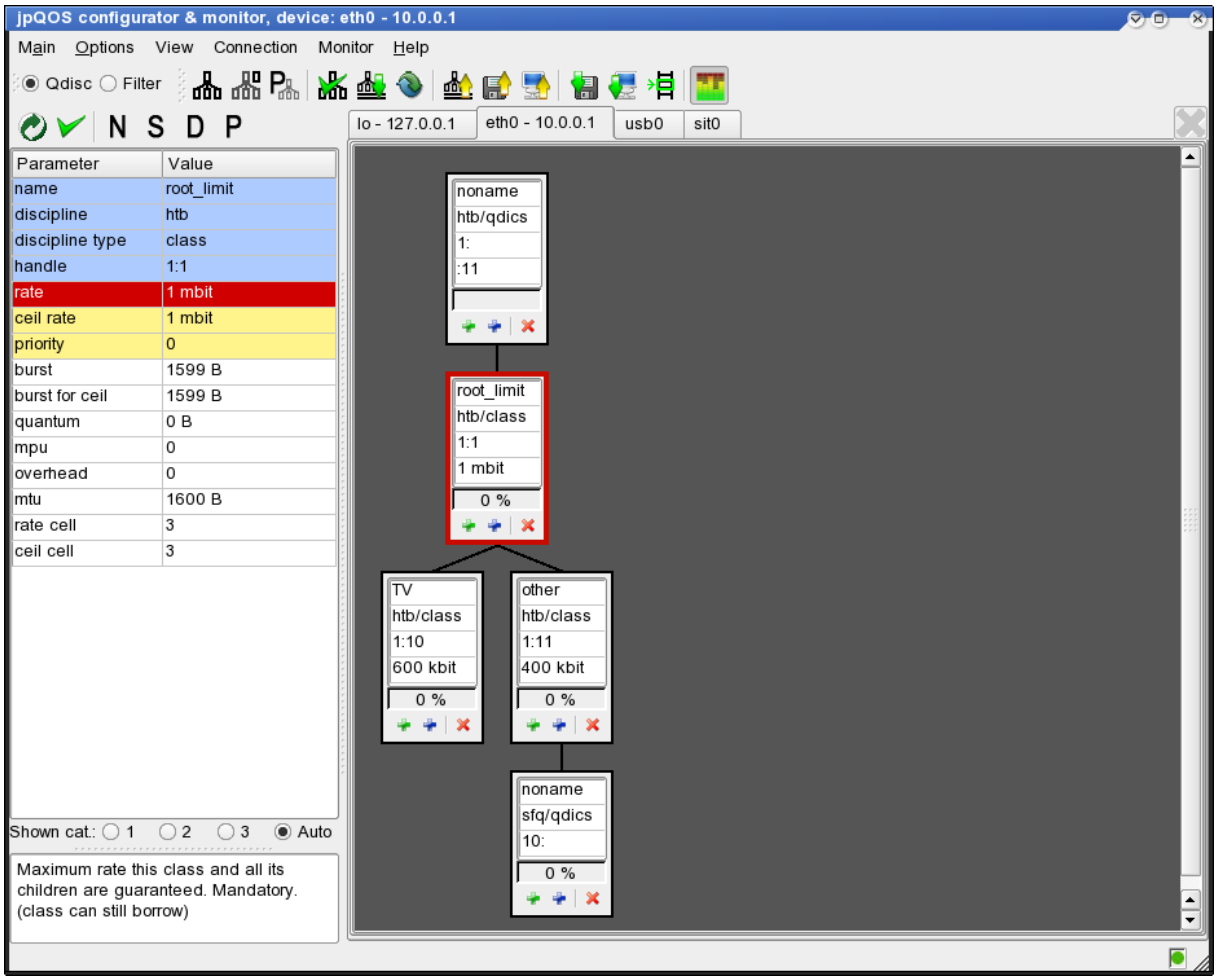


Figure 21: Tutorial QoS configuration

23. We still have to set the remaining data streams to go to class 1:11; this is easily done by setting the default class at the HTB root qdisc (1:0) (We first have to switch back to the qdisc edit mode)
24. The completed QoS configuration is checked using the toolbar button “validate QoS settings” (ALT+V)

25. Any errors reported are removed, until the check is successful
26. A successfully validated QoS configuration will be applied after configuring it on the server using the toolbar button “use QoS settings” (ALT+U)
27. After configuring the server the currently used QoS configuration is loaded from the server and displayed (due to various reason in can differ from the originally created one).
28. According to the settings, the rate monitoring of the individual nodes is displayed (ALT+R).
29. In order to allow use of the configuration after restarting the machine, it has to be saved and set as default.
30. Saving to the server is done by pressing the save QoS settings to file (server) button (ALT+S) we have to name the file which will contain the configuration.
31. Several configurations can be saved for each network interface, but only one can be set as the default one. To choose the default configuration we use the select default QoS settings file on server toolbar button (ALT+D)

Appendix C - Bugs & „features“ of Traffic control in Linux

While analyzing the tools for Traffic control administration in Linux, a few discrepancies with the facts stated in the documentation and the practical test results have been discovered. The main ones are described below in an appendix intended to help users (QoS administrators).

FILTERS:

- filter destination handle – if it points to a class containing the filter itself (cycle), the kernel goes into a infinite loop, there is no check at the point where the filter is configured

- delete filter – it not only deletes the filter with the input handle, but also any other filters in the QoS tree which have the same priority

- it is not possible to change filter parameters, just add new/delete

- **U32 filter**
  - separator U16 – offset has to be multiple of 2 (bytes)
  - separator U32 – offset has to be multiple of 4 (bytes)

DISCIPLINES:

- qdisc can only be added as new or deleted, it is not possible to change parameters

- **HTB**
  - only qdisc is allowed to have filters; if they are also added under a class they are automatically copied under qdisc
  - the parent class itself does not limit dataflow of classes under it, it only takes care of the possibility of sharing the unused data flow; the maximum possible flow is given by the sum of the guaranteed data flow rates of all leaf nodes in the QoS tree

- CBQ
  - it is necessary to have two qdiscs to limit the root flow
  - filters can only point to classes in the same qdisc as the filter is in

Appendix D – CD ROM

This thesis is accompanied by the CD ROM containing package **jpqos.tar.gz** with source code of the jpQOS implementation, makefiles, documentation, set of QOS settings examples and prepared X.509 certificates for testing purpose. See Appendix A – User Documentation for installation details.