Quality of Service Support by Active Networks

Inauguraldissertation
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Florian Baumgartner

von Deutschland

Leiter der Arbeit:

Prof. Dr. T. Braun

Institut für Informatik und angewandte Mathematik

Quality of Service Support by Active Networks

Inauguraldissertation
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Florian Baumgartner

von Deutschland

Leiter der Arbeit:

Prof. Dr. T. Braun

Institut für Informatik und angewandte Mathematik

Von der Philosophisch-naturwissenschaftlichen Fakultät angenommen.

Bern, 7. Februar 2002

Der Dekan:
Prof. Dr. P. Bochsler

Preface

The following work was performed during my employment as research and lecture assistant at the Institute for Computer Science and Applied Mathematics (IAM) of the University of Bern. I would like to thank Prof. Dr. Torsten Braun, head of the Computer Network and Distributed Systems group (RVS), for supervising this work and for his insightful advises. Prof. Dr. Braun encouraged and motivated me to publish my research results and he provided me the opportunity to present the work on various conferences, for which I thank him.

Also, Prof. Dr. Hanspeter Bieri who was willing to be the co-examinator of this work deserves many thanks.

Many thanks go to my colleagues of the RVS group and of the IAM for our various interesting discussions, and fruitful collaboration on organisational and research issues. Special thanks go to Roland Balmer, Silvia Bechter, Peppo Brambilla, Dr. Manuel Günter, Dr. Johannes Schneider and Günter Stattenberger for support in many areas and Ruth Bestgen for administrative support.

The work presented here mainly was performed in the framework of a project financed by the Swiss National Science Foundation.

Contents

Chapter 1

Introduction

The influence of the Internet on classical kinds of communications is enormous. Originally designed to allow a small number of academic or military users to access remote mainframes or to exchange data it has evolved to a nearly omnipresent communication system. In the 90's with computers getting cheaper and cheaper and especially with the development of the World Wide Web nearly everybody in the western world got the possibility to be connected.

With the increase of available bandwidth even for users at home, the Internet got also interesting for more classical communication systems like telephony or television. More and more of these services are using Internet technology. In future decades specialised networks for telephony or television might disappear and leave one single and universal medium used for all kinds of communication, the Internet. Unfortunately, the Internet was never designed to provide support for applications like telephony or video transmission.

A fundamental design rule within the Internet architecture is to leave as many tasks as possible to the end systems. Because of scalability issues it is smarter to put the more complicated algorithms into the computers at the edge with only a small number of users, instead of building network devices, that have to apply complicated mechanisms to the data of thousands or millions of users.

The Internet is a packet based network. Each computer connected to the Internet has a unique address. If a computer wants to send data to another computer, the data is fragmented into packets of a certain length and put to the network. Each packet gets a header containing the address of the destination computer and some additional information. The packet is forwarded through the network and is finally received by the computer with the destination address. Since a device within the network could receive more data than it can process, packets can be dropped and get lost.

Beyond sending and receiving packets to and from the network, there is no way for the end system to influence the way the packet is treated during transmission or whether at all a packet reaches its destination at all. To provide a reliable data transport, the end systems run rather complicated protocols, which retransmit packets if there is no acknowledgement from the receiver. Even if the end systems run complex protocols to assure a reliable transport of packets, there is no guarantee for a certain transmission speed or a time the network requires to forward a packet from the sender to the receiver.

However, for a lot of Internet services this lack of a guaranteed quality is acceptable. Neither the time it takes for an email to be transmitted nor whether there were changes in the network delay during the transmission is important. Things look different if video or audio streams have to be transmitted in real time as it is needed for television, telephony or audio/video streaming. Insufficient bandwidth or large and changing delays may disturb a transmission. These applications have to rely on certain performance parameters like bandwidth or delay to work properly. These requirements have led to the development of several approaches to add Quality of Service support to the Internet.

The Internet offers currently only one type of service based on the best-effort model. With this model and FIFO queueing deployed in the network, any non-adaptive source can take advantage to grab high bandwidth while depriving others. One can always run multiple web browsers or start multiple FTP connections and grab a substantial amount of bandwidth by exploiting the best effort model. Therefore, the Internet is unable to support applications relying on certain traffic parameters like a minimum bandwidth. Real time applications like audio or video transmission are especially affected by the best effort model.

On the other hand this ”simple” packet based technology used by the Internet is also one of the reasons for its success. Therefore any new technology providing Quality of Service not only has to be compatible with current Internet standards but has also to comply the basic design rules. It must be capable to provide a service independent from the network size and the from number of persons using this service. Besides the question of the underlying mechanisms to provide Quality of Service the management of these services is an open issue as well. Since Quality of Service requires a proper reservation of resources, mechanisms are required to also perform this tasks in a scalable manner.

Recently two approaches have been developed to add these Quality of Service capabilities to the Internet. The first one Integrated Services, based on the Resource Reservation Setup Protocol (RSVP) focuses the setup of appropriate resources per flow, while the second approach – Differentiated Services – tries to provide services not on a per flow basis but for large aggregates of flows. Both approaches and their underlying mechanisms are presented in detail in Chapter 2.

While RSVP is mainly based on well known mechanisms, Differentiated Services is a rather new concept and introduces new services and technologies. Since some of these techniques are fundamentally new, their impact and their performance have to be evaluated. Within Chapter 3, a network simulator is extended to be capable for Differentiated Services and is then used to evaluate these new aspects. The evaluations cover various parameter sets and scenarios.

However, simulators always lack a certain realism and tend to neglect small but sometimes important details. Their rather abstract view of a network is an advantage during the evaluation of huge topologies. For small and medium size networks a more realistic tool would be advantageous. To combine the advantages of network simulation and real test beds, a new concept for the emulation of Internet topologies was developed and implemented. The new approach allows the integration of real end systems and therefore the ability to directly evaluate any application available for these end

systems. Chapter 4 describes this architecture. Section 4.4 proves the power of this new approach by presenting several experimental results. A special focus is put on the emulation of Differentiated Services networks and the comparison of the results to those obtained by the network simulator.

The basic mechanisms studied by simulations and emulations are only one part of a network wide support for Quality of Service. Using the best effort model, Internet service providers only had to ensure that packets reach their destination, while in a Quality of Service enabled network they also have to provide certain guarantees or at least probabilities for a certain throughput or a maximum delay.

Both approaches for Quality of Service within the Internet have their own advantages and drawbacks. An integration of both concepts would be advantageous. While RSVP has certain aspects supporting the end user in access networks in particular, the strengths of Differentiated Services is the good support for Internet backbones. Chapter 5 illustrates the problems arousing from Quality of Service enabled networks. Within section 5.3 a concept of integrating both approaches combining their advantages is presented and evaluated.

Services like the mapping between Integrated and Differentiated Services show the capability of new services added to the network. Unfortunately, the implementation of new services is an extremely complicated and time consuming process. One problem are the specific and incompatible configuration interfaces router manufacturers provide, making management rather complicated and limited. The other more severe problem is the size of the Internet. A central instance like a management station cannot supervise thousands of network nodes.

In Chapter 6, a lightweight mechanism is presented allowing the installation of new services quite rapidly. This includes new functionalities on network devices as well as simple mechanisms to implement new protocols for signalling purposes. Active Networks are a possible solution providing the required flexibility. They allow to inject active packets into a network. These packets carry executable code and can be executed by network devices. An Active Network environment was designed and implemented for Linux and the network emulator mentioned above. This allows to set up and evaluate large active networks. In contrast to other approaches the environment is based on the python language allowing to implement new services in a reusable and understandable programming language.

To evaluate and demonstrate the capabilities of such a service and network management approach, different new services were implemented and are presented in section 6.5.1. A presentation of an ”active” approach to integrate different Quality of Service concepts conclude the presentation of the active network environment.

Chapter 7 summarises the results and gives an outlook on further research topics.

Chapter 2

Quality of Service in the Internet

The incredible rapid growth of the Internet has resulted in massive increases in demand for network bandwidth guarantees to support both existing and new applications. In order to meet these demands, new Quality of Service (QoS) functionalities need to be introduced to satisfy customer requirements including efficient handling of both delay critical and bandwidth greedy applications. QoS, therefore, is needed for various reasons:

- Better control and efficient use of networks resources (e.g. bandwidth).
- Enable users to enjoy multiple levels of service differentiation.
- Special treatment to high priority applications like remote surgery while allowing others to get a fair treatment without interfering with this sensitive traffic.
- Business Communication.
- Virtual Private Networks (VPNs) over IP.

2.1 Adaptive Internet Applications

A pragmatic approach to achieve good quality of service (QoS) is an adaptive design of the applications to react to changes of the network characteristics (e.g. congestion).

On protocol level this is already done by the TCP protocol. TCP monitors round trip time and packet loss and reduces bandwidth according to the actual available network resources. TCP tries to avoid congestion within the network. It is not able to react to scarce resources in an intelligent manner, but provides only a rather generic reliable data transmission service without any support for a certain packet delay or a certain bandwidth.

To provide a more intelligent congestion control an application may reduce the transmission rate by increasing the compression ratio or by modifying the Audio/Video coding algorithm. For this purpose functions to monitor quality of service are needed. For example, such functions are provided by the Real-Time Transport Protocol (RTP)

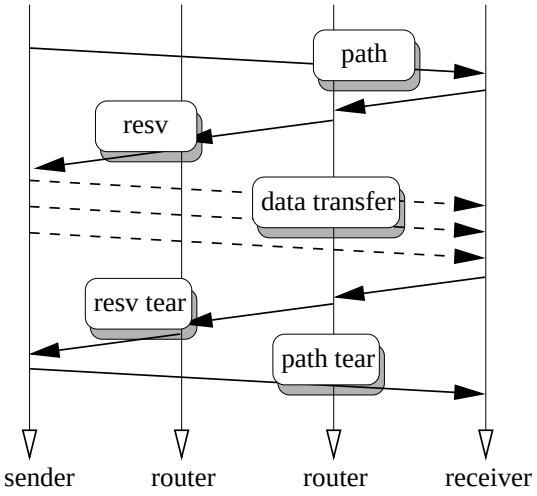


Figure 2.1: *RSVP signals during resource reservation set up*

[SCFJ96] and the Real-Time Control Protocol (RTCP). A receiver measures the delay and the rate of the packets received. This information is transmitted to the sender via RTCP. With this information the sender can detect if there is any congestion in the network and adjust the transmission rate accordingly. This may affect the coding of the audio or video data. If only a low data rate is achieved, a coding algorithm requiring less bandwidth has to be chosen. Without this adaptation the packet loss would increase, making the transmission completely useless. However, rate adaptation is limited since many applications need a minimum rate to work reasonably.

2.2 Integrated Services

The most obvious solution to provide a certain QoS for a specific packet stream is to configure all network devices between sender and receiver to guarantee a certain amount of bandwidth or a maximum delay to packets of this flow. This approach is called Integrated Services and is based in the Resource Reservation Protocol (RSVP) [BZB+97]. The RSVP protocol is used to signal QoS requirements to all RSVP capable routers between a source and a destination. Figure 2.1 shows the setup of a resource reservation.

A RSVP session starts with a *path*-message being sent from the host, that wants to transmit data to the destination. This message has the following tasks:

- determination of the route the data will take later on through the network.
- information about the destination, the traffic characteristics and perhaps the costs
- initialisation of information in each RSVP capable router on the path. Each router has to know its neighbour routers.

To record the route properly, the *path* message has to be sent to the final receiver of the data. Unfortunately an intermediate router would simply forward such a packet,

without the special treatment necessary to record the route. To apply this special processing of the packet, the Router Alert option [Kat97] is used, signalling a router to check a packet more intense.

If the receiver agrees the advertised flow, he returns a *resv*-message, which is transported from hop by hop via RSVP capable routers towards the sender of the *path*-message. A RSVP router allocates resources and forwards the *resv*-message if he can meet the flows requirements, Otherwise he sends a *resv-err*-message to the sender of the *resv*-message.

If the receiver gets the *resv*-message, the resources are reserved and the data can be transmitted. To terminate a reservation, an *resv-tear*-message is transmitted to remove the resource allocations and a *path-tear* message is sent to delete the path states in every router on the path.

RSVP has the ability to guarantee a certain QoS, is supported by several user applications like Microsoft's Netmeeting and is implemented in most modern routers.

Unfortunately RSVP requires each router to store the flow state for the reservation. This works great in small LANs with a small number of end systems. In the backbone, however, it would be extremely difficult if not impossible to store millions of flow states, even with very powerful processors. Another disadvantage is the time it takes to set up a reservation and the protocol overhead. For setting up a video transmission a certain delay to establish the reservation is acceptable, but for short-lived HTTP connections this causes too much overhead.

2.3 Differentiated Services

To avoid the scalability problems of RSVP, the concept of Differentiated Services (DS) was developed. In contrast to RSVP Differentiated Services work on an aggregation of flows by marking each packet and invoking some differentiation mechanism within the nodes on the packet's path. There, depending on the marking, the packet is handled differently. Therefore depending on how the packets of a specific stream are marked they get a certain Quality of Service [BBH98], [BBEK99].

Such a concept does not require the storage of flow states within each backbone router as RSVP does, but simply treats each packet according to its mark. As will be described later the algorithms for the different packet treatments are not very complicated and can be applied even at high packet rates.

It depends on the network and the application where packets are marked. It might make sense to mark packets directly within an application running on the end system to provide different kinds of Quality of Service for different data. Therefore a video conference application might send the packet containing the audio data with a better Quality of Service than the less important video data.

In another scenario the marking may be done by the customers or the Internet Service Providers (ISPs) border routers. This makes sense if all traffic of a customer should get a certain Quality of Service. Of course the ISP has to control that the customer does not send more packets than he is allowed.

As can be seen Differentiated Services are very flexible, allowing the support of a wide range of scenarios.

The rather static agreements between a customer and his ISP are a disadvantage. Therefore, one reservation may exist for several, possibly consecutive connections.

The probability to be able to provide the requested quality of service depends essentially on the dimensions and configuration of the network and its links, i.e. whether individual links or routers can be overloaded by high priority data traffic. Though this concept cannot guarantee any QoS parameters as RSVP can, it is more straightforward to be implemented than continuous resource reservations and it offers a better QoS than mere best-effort services.

2.3.1 Popular Services of the Differentiated Services Approach

During the work of the IETF's Differentiated Services working group, several services were proposed. Even if there were several proposals for different services, there are currently only two standardised in RFC documents.

Expedited Forwarding: The Expedited Forwarding (EF) service was designed to provide a kind of leased line service for the Internet. This should allow ISPs to sell virtual leased lines with a fixed bandwidth and a limited delay to customers. As more and more companies are using the Internet instead of the originally leased lines to connect, there is a big demand for such a service.

Negotiating such a service with his ISP allows a customer to send packets at a specific rate to the ISP's network. If a packet exceeds this rate it is – in contrast to Assured Service – dropped. Like a leased line this service defines an upper limit for the allowed bandwidth but also guarantees the forwarding of packets up to this limit at a constant delay.

Assured Forwarding: Assured Forwarding (AF) was based on the initially proposed Assured Service by Clark and Wroclawski [CW97]. The Assured Service proposed, should allow a customer to define a service profile with his ISP, defining a certain packet rate he is allowed to send higher priority packets with. Packets exceeding this rate are forwarded with higher drop precedence. Based on this initial proposal Heinanen defined the Assured Forwarding Service. Assured Forwarding defines four independent traffic classes, each with three drop precedences. Heinanen writes in [HBWW99a]:

> [...], if traffic conditioning actions at the ingress of the provider DS domain make sure that an AF class in the DS nodes is only moderately loaded by packets with the lowest drop precedence value and is not overloaded by packets with the two lowest drop precedence values, then the AF class can offer a high level of forwarding assurance for packets that are within the subscribed profile (i.e., marked with the lowest drop precedence value) and offer up to two lower levels of forwarding assurance for the excess traffic. [...]

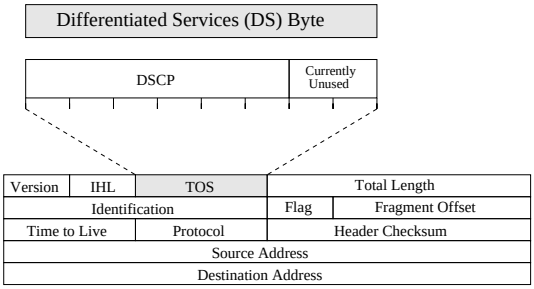


Figure 2.2: *The fields of the DS byte in IPv4 [NBBB98]*

Obviously this service does not define a QoS as clear as Expedited Forwarding does. The end-to-end service depend crucially on proper network provisioning. On the other hand this service is capable to use free network resources if available.

2.3.2 Differentiated Services Marking

Differentiated Services are based on marking single packets. As the IP header can not be changed due to compatibility reasons, the eight bit wide Type of Service (ToS) field in the IP header was used to mark packets differently. Six bits of the ToS byte are used as Differentiated Service Code point (DSCP) as can be seen in Figure 2.2.

Within each Differentiated Services capable router there is a mapping between a DSCP value and a per hop behaviour (PHB), which defines how a packet is treated within the router. If a PHB specifies to forward a packet preferential to all others and that PHB is applied within all routers of a network this would result in a service providing noticeable better throughput and low delay to all packets with the appropriate DSCP.

The six bit wide DSCP allows to differentiate between $2^6 = 64$ different PHBs. To allow an Internet Service Provider to provide own services only a small number of DSCPs and PHBs are specified. An ISP might map DSCPs at his border routers to provide a similar PHB within his own network. Router implementations should support the recommended code-point-to-PHB mappings. The default PHB, for example, is 000000, other recommended PHBs will be introduced later.

2.3.3 Per Hop Behaviours

An introduction to Per Hop Behaviours (PHBs) has already been given while discussing DS byte marking 2.3.2. Further [BW98] writes: *"Every PHB is the externally observable forwarding behaviour applied at a DS capable node to a stream of packets that have a particular value in the bits of the DS field (DS code point). PHBs can also be grouped when it is necessary to describe the several forwarding behaviours simultaneously with respect to some common constraints."*

However, there is no rigid assignments of PHBs to DSCP bit patterns. This has several reasons:

- There are (or will be) a lot more PHBs defined than DSCPs available, making a static mapping impossible.

Drop Precedences	AF Code points			
	Class 1	Class 2	Class 3	Class 4
Low Drop Precedence	001010	010010	011010	100010
Medium Drop Precedence	001100	010100	011100	100100
High Drop Precedence	001110	010110	011110	100110

Table 2.1: *The 12 different Assured Forwarding code points*

- The understanding of good choices of PHBs is at the beginning.
- It is desirable to have flexibility in the correspondence of PHB values and behaviours.
- Every ISP has to be able to create/map PHBs in his Differentiated Services domain.

This is why there are no static mappings between DS code points and PHBs. The PHBs are enumerated as they become defined and can be mapped to every DSCP within a Differentiated Services domain. As long as the enumeration space contains a large number of values (2^{32}), there is no danger of running out of space to list the PHB values. This list can be made public for maximum interoperation. Because of this interaction, mappings between PHBs and DSCPs are proposed, even if every ISP can choose other mappings for the PHBs in his Differentiated Services domain.

Until now, two PHBs and corresponding DSCPs have been defined.

Assured Forwarding PHB: Based on the current Assured Forwarding PHB (AF) group [HBWW99b], a provider can provide four independent AF classes with each class having three drop precedence values. These classes are not aggregated in a DS node and Random Early Detection (RED) [FJ93] is considered to be the preferred discarding mechanism. This required altogether 12 different AF code points as given in table 2.1.

In a Differentiated Service (DS) Domain each AF class receives a certain amount of bandwidth and buffer space in each DS node. Drop precedence indicates relative importance of the packet within an AF class. During congestion, packets with higher drop precedence values are discarded first to protect packets with lower drop precedence values. By having multiple classes and multiple drop precedences for each class, various levels of forwarding assurances can be offered. For example, Olympic Service can be achieved by mapping three AF classes to it's gold, silver and bronze classes. A low loss, low delay, low jitter service can also be achieved by using this service if the packet arrival rate is known in advance. AF does not give any delay related service guarantees. However, it is still possible to say that packets in one AF class have smaller or larger probability of timely delivery than packets in another AF class. The Assured Forwarding can be realized with AF PHBs.

Expedited Forwarding PHB: The forwarding treatment of the Expedited Forwarding (EF) PHB [JNP98] offers to provide higher or equal departure rate than the configurable rate for aggregated traffic. Services which need end-to-end assured bandwidth and low loss, low latency and low jitter can use EF PHB to meet the desired requirements. One good example is premium service (or virtual leased line) which has such requirements. Various mechanisms like Priority Queueing, Weighted Fair Queueing (WFQ) or Class Based Queueing (CBQ) are suggested to implement this PHB since they can preempt other traffic and the queue serving EF packets can be allocated bandwidth equal to the configured rate. The recommended code point for the EF PHB is 101110.

2.3.4 Per Domain Behaviours and Service Profiles

Administrators of Differentiated Service Domains may freely choose DSCP values for different PHBs and also provide specific PHBs. On the other hand the mapping of DSCPs and PHBs within a domain has to be uniform.

For the definition of an end to end service involving multiple Differentiated Service Domains usually, the packet treatment per domain is a more feasible definition than a Per Hop Behaviour. Therefore the term Per Domain Behaviour (PDB) was defined in [NC01]:

> **Per-Domain Behaviour**: the expected treatment that an identifiable or target group of packets will receive from ”edge to edge” of a DS domain. A particular PHB (or, if applicable, list of PHBs) and traffic conditioning requirements are associated with each PDB.

Specification of the transit expectations of packets across a Differentiated Services domain will support the composition of end-to-end, inter-domain services. Networks of DS domains can be connected to create end-to-end services by building on the PDB characteristics without regard to the particular PHBs used. This level of abstraction makes it easier to compose inter-domain services as well as making it possible to hide details of a network’s internals while exposing information sufficient to enable QoS.

PDBs are intended to be useful tools in configuring Differentiated Service domains, but the PDB used by a provider is not expected to be visible to customers any more than the specific PHBs employed in the provider’s network would be. The configuration of a PDB might be taken from a Service Level Specification as defined in [Gro01].

A Service Level Specification (SLS) is a set of parameters and their values which together define the service offered to a traffic stream by a Differentiated Service domain. It is expected to include specific values or bounds for PDB parameters. Any SLS includes also a Traffic Conditioning Specification (TCS), which is a set of parameters and their values which together specify a set of classifier rules and a traffic profile.

The SLS defines the service an Internet Service Provider offers to his customers and is usually part of an Service Level Agreement (SLA). In contrast to the technical description provided by SLS and TCS, the SLA is more a contract and may additionally include information about prices and administrative data.

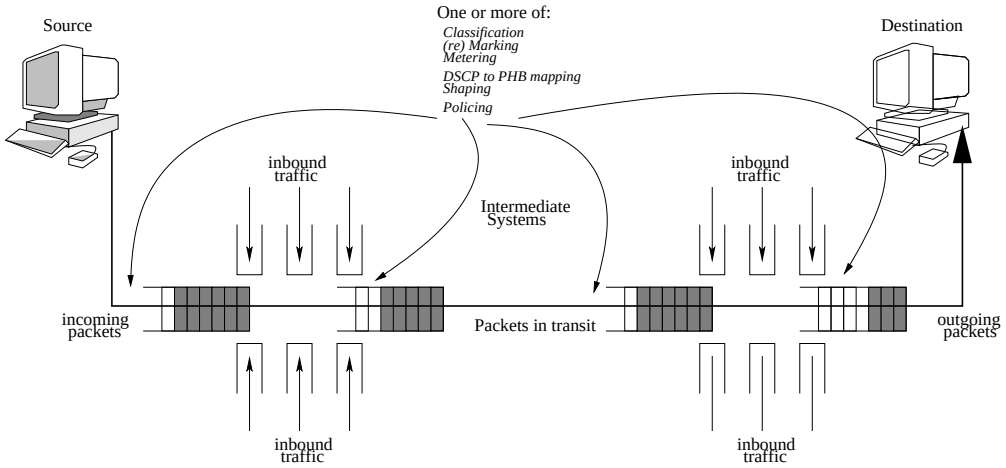


Figure 2.3: *DS Traffic Conditioning in Enterprise Network (as a set of queues)*

Only a static SLA, which usually changes weekly or monthly, is possible with today's router implementation. The TCS parameters are set in the router manually to take appropriate action. Dynamic SLAs change frequently and need to be deployed by some automated tool which can renegotiate resources between any two nodes.

Such a tool and an according protocol was developed within the CATI [SBP99] project. An application can contact a bandwidth broker using a special protocol to negotiate a certain QoS. The bandwidth broker configures the networks devices to provide the service and to police the users traffic according to the TCS.

2.3.5 Differentiated Services Traffic Conditioning

Traffic conditioners [BBC+98a] are required to instantiate services in Differentiated Services capable routers and to enforce service allocation policies. These conditioners are, in general, composed of one or more of the followings: classifiers, markers, meters, policers, and shapers. When a traffic stream at the input port of a router is classified, it then might have to travel through a meter (used where appropriate) to measure the traffic behaviour against a traffic profile which is a subset of a SLA. The meter classifies particular packets as IN or OUT-of-profile depending on SLA conformance or violation. Based on the state of the meter further marking, dropping, or shaping action is activated.

Traffic Conditioners can be applied at any congested network node (Figure 2.3) when the total amount of in bound traffic exceeds the output capacity of the switch (or router). In Figure 2.3 routers between source and destination are modelled as queues in an enterprise network to show when and where traffic conditioners are needed. For example, routers may buffer traffic (i.e. shape them by delaying) or mark them to be discarded later during medium network congestion, but might require to discard packets (i.e. police traffic) during heavy network congestion when queue buffers fill up. As the number of routers grows in a network, congestion increases due to expanded volume of traffic and hence proper traffic conditioning becomes more important.

Traffic conditioners might not need all four elements. If no traffic profile exists then packets may only pass through a classifier and a marker.

Classifier: Classifiers categorise packets from a traffic stream based on the content of some portion of the packet header. It matches received packets to statically or dynamically allocated service profiles and pass those packets to an element of a traffic conditioner for further processing. Classifiers must be configured by some management procedures in accordance with the appropriate TCA.

There are two types of classifiers:

BA Classifier: works on behaviour aggregates and classifies packets based on patterns of the DS-byte (DSCP) only.

MF classifier: classifies packets based on any combination of the DS field, protocol ID, source address, destination address, source port, destination port or even application level protocol information.

Markers: Packet markers set the DS field of a packet to a particular code point, adding the marked packet to a particular Differentiated Services behaviour aggregate. The marker can (i) mark all packets which are mapped to a single code point, or (ii) mark a packet to one of a set of code points to select a PHB in a PHB group, according to the state of a meter. The approach of Differentiated Services and especially the Assured Forwarding service with its different drop precedence levels, requires specialised components.

Meters: After being classified at the input of the boundary router, traffic from each class is typically passed to a meter. The meter is used to measure the rate (temporal properties) at which traffic of each class is being submitted for transmission. This is then compared against a traffic profile specified in TCA (negotiated between the Differentiated Services provider and the Differentiated Services customer). Based on the comparison some particular packets are considered conforming to the negotiated profile (IN-profile) or non-conforming (OUT-of-profile). If a meter passes this state information to other conditioning functions, an appropriate action is triggered for each packet.

Shapers: Shapers delay some packets in a traffic stream using a token bucket in order to force the stream into compliance with a traffic profile. A shaper usually has a finite-size buffer and packets are discarded if there is not sufficient buffer space to hold the delayed packets. Shapers are generally placed after either type of classifier. For example, shaping for EF traffic at the interior nodes helps to improve end to end performance and also prevents the other classes from being over flooded by a big EF burst. Only either a policer or a shaper is supposed to appear in the same traffic conditioner.

Policer: When classified packets arrive at the policer it monitors the dynamic behaviour of the packets and discards or re-marks some or all of the packets in order to force the stream into compliance (i.e. force them to comply with configured properties like rate and burst size) with a traffic profile. By setting the shaper buffer size to zero (or a few packets) a policer can be implemented as a

special case of a shaper. Like shapers policers can also be placed after either type of classifier. Policers, in general, are considered suitable to police traffic between DS domains (e.g. a customer and a provider) and after BA classifiers in backbone routers. However, most researchers agree that policing should not be done at interior nodes since it unavoidably involves flow classification. Policers are usually present in ingress nodes and could be based on simple token bucket filters.

Especially the Assured Forwarding requires special treatment, because of its three drop precedence mechanism Heinanen e.a. proposed two meter/marker combinations to mark packets with different probabilities. Each of this components associates a colour (red, yellow, green) with a drop precedence.

Two Rate Three Colour Marker

This marker [HG99b] uses four parameters: the peak information rate, the committed information rate and their associated burst sizes.

$$(rate_{pi}, burst_{pi}, rate_{ci}, burst_{ci})$$

The burst sizes are usually given in bytes, the rates in bits per second. A packet is marked red if it exceeds the peak information rate $rate_{pi}$. Otherwise it is marked either yellow or green depending on whether it exceeds or doesn't exceed the committed information Rate $rate_{ci}$. This marker is useful for ingress policing of a service, where a peak rate needs to be enforced separately from a committed rate. This type of marker is usually implemented using two token bucket filters with $rate_{pi}$ and $rate_{ci}$ as bucket rates and $burst_{pi}$ and $burst_{ci}$ as bucket sizes.

Single Rate Three Colour Marker

The other marker proposed by Heinanen is the "Single Rate Three Color Marker" [HG99a]. In contrast to the "Two Rate Three Color Marker" only one rate and two burst sizes are specified. A stream is metered and marked according to three traffic parameters Committed Information Rate (CIR), Committed Burst Size (CBS), and Excess Burst Size (EBS) to be either green, yellow, or red.

$$(rate_{ci}, burst_c, burst_e)$$

A packet is marked green if it doesn't exceed $burst_c$, yellow if it does exceed $burst_c$, but not $burst_e$ and red otherwise. This marking mechanism is useful for ingress policing of a service, where only the length but not the peak rate of the burst determines service eligibility.

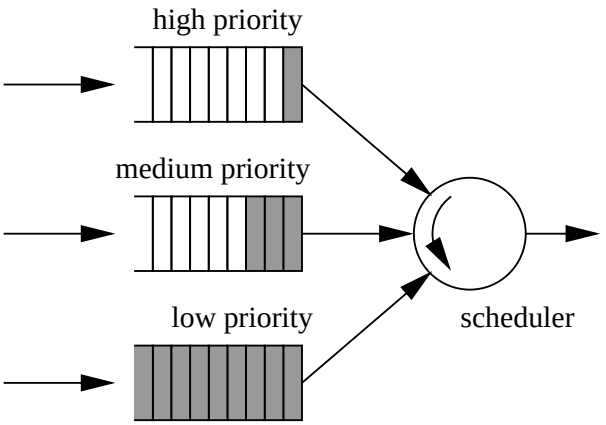


Figure 2.4: *Absolute Priority Queueing: The queue with the highest priority is served at first*

2.4 Queueing Components for Quality of Service

In the previous section the functional components of a Differentiated Services routers were described. Since several of these components are rather generic and can be used for any kind of resource management in a network, this section will also give an overview over traffic conditioning components in general.

Quality of Service is a kind of service discrimination. A network component (e.g. a router) has to handle packets in different manners in order to achieve a certain Quality of Service for a specific flow or type of packets. This differentiation between packets may be done on a per flow basis as in RSVP or for an aggregation of packets, distinguished by different DSCP values as it is done by Differentiated Services. Within the Internet the processing speed of network components is assumed high compared to the bandwidth of the links between them. Congestion occurs when a router has to transmit more packets over a link than the link's capacity allows. The router has to discard packets. All approaches to realise QoS are based on dropping the "right" packets.

To handle packets differently, an Internet router usually has a queueing system attached to his outgoing interfaces. Simple queueing systems like FIFO (first in first out) queues are capable to intercept short bursts of packets exceeding the output bandwidth. More complicated queueing systems allow to handle packets differently, putting them to different kind of queues and processing these queues with different priority.

Several mechanisms were developed for various purposes.

2.4.1 Absolute Priority Queueing

In absolute priority queueing (Figure 2.4), the scheduler prefers queues with high priority, forwarding packets from queues with lower priority only the higher priority queues are empty. Therefore, the packets put into the queue with the highest priority achieve the best service, while packets sent to queues with lower priority have to be satisfied with the left resources.

The basic working mechanism is as follows: the scheduler always scans the queues from their highest to lowest priority and transmits packets if available. Once a packet has been sent, the scheduler starts again at the queue with the highest priority.

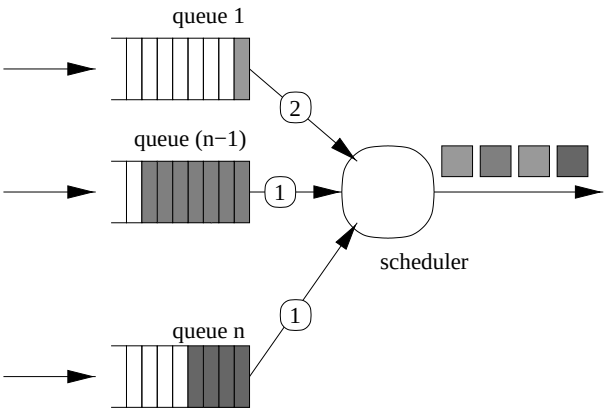


Figure 2.5: *Weighted Fair Queueing. Each queue gets a bandwidth portion according to its weight*

This queueing mechanism is useful to give a specific flow or service class the absolute priority over other traffic, which is important for bandwidth and delay critical data. As long as high priority packets do not exceed the outgoing bandwidth, the packets are forwarded with a minimal delay. Setting up Differentiated Services networks, this algorithm might be used to provide an expedited forwarding service. Of course the expedited forwarding traffic has to be policed as misbehaving EF senders would be able to completely starve the bandwidth of traffic classes with less priority.

2.4.2 Weighted Fair Queueing

Weighted fair queueing (WFQ) (figure 2.5) is a discipline that assigns a queue for each flow. A weight can be assigned to each queue to guarantee a different part of the network's bandwidth capacity. As a result, weighted fair queueing can provide protection against other flows.

WFQ can be configured to give low-volume traffic flows preferential treatment to reduce response time and share the remaining bandwidth between high volume traffic flows in a fair way. With this approach bandwidth hungry flows are prevented from consuming too much of network resources while depriving other, smaller flows.

WFQ does the job of dynamic configuration since it adapts automatically to the changing network conditions. TCP congestion control and slow-start features are also enhanced by WFQ. The result is predictable throughput and response time for each active flow.

Even if WFQ was originally designed to support one flow per queue, it can also be applied for aggregations of flows. This can be used to reserve a certain portion of the available bandwidth to each Assured Forwarding class, while another portion is left to best effort traffic.

Based on the initial work of Demers, Keshav e.a. [DS90], several improvements were developed like "Worst case fair Weighted Fair Queueing" [BZ96] and "Worst case fair Weighted Fair Queueing +" [BZ97].

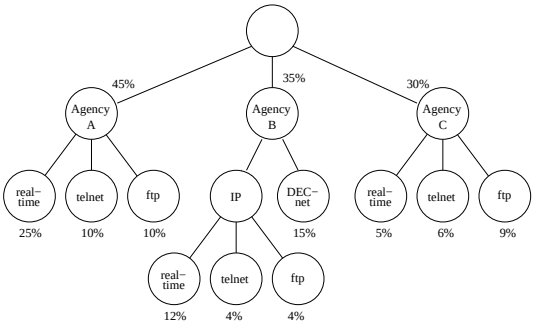


Figure 2.6: *Hierarchical Link-Sharing*

2.4.3 Priority Weighted Round Robin

The implementation of a Differentiated Services queueing system usually has to support six traffic classes: Expedited Forwarding, four classes of Assured Forwarding and best effort.

Whereas each Assured Forwarding class and the best effort usually get a certain share of the available bandwidth, the Expedited Forwarding traffic has to be handled preferential.

This requires a combination of absolute priority queueing for the EF traffic and a weighted fair queueing like bandwidth sharing mechanism. This is provided by the priority weighted round robin (PWRR) scheduler [SB00]. This scheduler is able to handle certain queues preferential as absolute priority queueing does and share remaining bandwidth between other queues.

2.4.4 Class Based Queueing (CBQ)

In an environment where bandwidth must be shared proportionally between users, CBQ [FJ95] (Figure 2.7) provides a very flexible and efficient approach by first classifying user traffic and then assigning a specified amount of resources to each class of packets and serving those queues in a round robin fashion.

A class can be an individual flow or aggregation of flows representing different applications, users, departments or servers. Each CBQ traffic class has a bandwidth allocation and a priority. In CBQ, a hierarchy of classes (Figure 2.6) is constructed for link sharing between organisations, protocol families and traffic types. Different links in the network will have different link sharing structures. The link sharing goals are:

- Each interior or leaf class should receive roughly its allocated link-sharing bandwidth over appropriate time intervals.
- If all leaf and interior classes 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 granular level of control in CBQ can be used to manage the bandwidth allocated across the departments of an enterprise or to provide bandwidth to individual tenants of an apartment building.

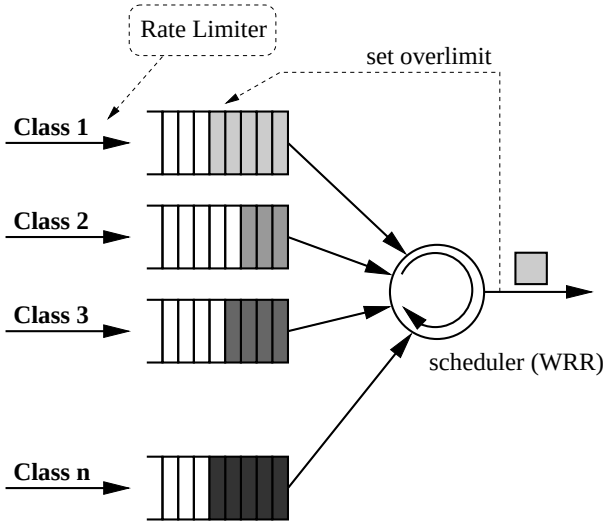


Figure 2.7: *Class Based Queueing: Main Components*

Other than the classifier that assigns arriving packets to an appropriate class, there are three other main components that are needed in this CBQ mechanism: scheduler, rate-limiter (delayer) and estimator.

Scheduler: In a CBQ implementation, the packet scheduler can be implemented with either a simple round robin, a priority round robin or weighted round robin scheduler.

While the round robin scheduler will process all classes sequentially, providing the same service to all classes, priority round robin has different priorities for the classes and treats packets from the highest priority level preferential. In weighted round robin scheduling uses weights proportional to a traffic class's bandwidth allocation. This weight finally allocates the number of bytes a traffic class is allowed to send during a round of the scheduler. Each class at each round gets to send its weighted share in bytes. If a class is underlimit [1] but the packet's length exceeds the available share, the class might borrow bytes from future rounds of the scheduler.

Rate-Limiter: If a traffic class is overlimit[2] and is unable to borrow from its parent classes, the scheduler starts the overlimit action which might include simply dropping arriving packets for such a class or it can also limit overlimit classes to their allocated bandwidth. The rate-limiter computes the next time that an overlimit class is allowed to send traffic. Unless this future time has arrived, this class will not be allowed to send another packet.

Estimator: The estimator estimates the bandwidth used by each traffic class over the appropriate time interval and determines whether each class is over or under its allocated bandwidth.

[1] If a class has used less than a specified fraction of its link sharing bandwidth (in bytes/sec, as averaged over a specified time interval)

[2] If a class has recently used more than its allocated link sharing bandwidth (in bytes/sec, as averaged over a specified time interval)

2.4.5 Random Early Detection Gateways

Random Early Detection gateways (RED) [FJ93] are designed to avoid congestion by monitoring traffic load at points in the network and stochastically discarding packets when congestion starts increasing. By dropping some packets early rather than waiting until the buffer is full, RED keeps the average queue size low and avoids dropping large numbers of packets at once to minimise the chances of synchronisation effects within the network. Thus, RED reduces the chances of tail drop and allows the transmission line to be used fully at all times. This approach has certain advantages:

- bursts can be handled better, as always a certain queue capacity can be reserved for incoming packets.
- as filled queues increase the delay of a packet and RED is able to keep the queue length reasonably short, real-time applications are better supported.

A RED queue has four parameters. A minimum threshold th_{min}, a maximum threshold th_{max}, a value w_q which is used for averaging the queue length and a factor max_p used to calculate the dropping probability. The average queue length avg is calculated by

$$\text{avg} = (1 - w_q) \cdot \text{avg}_{old} + v \cdot w_q$$

where v is the actual queue length and avg_{old} the old average value. The dropping probability p_b increases linearly from 0 to max_p.

$$p_b = max_p \frac{avg - th_{min}}{th_{max} - th_{min}}$$

Additionally the dropping probability p_b increases with the number $count$ of packets since the last dropped packet. The final dropping probability p_a is given by:

$$p_a = \frac{p_b}{1 - count \cdot p_b}$$

$$P_{RED} = (th_{min}, th_{max}, max_p)$$

and the queue weight w_q. Each parameter has a certain influence on the behaviour of the RED system.

w_q: This value determines how fast the average queue length avg follows the value of the actual queue length v. Thus it determines how fast RED reacts to changes of the bandwidth (e.g. bursts).

max_p: Figure 2.8 shows the dropping probability p_b for a RED queue. Between th_{min} and th_{max} the dropping probability does not increase linearly from zero to one but from zero to max_p. The reason for this is TCP congestion control. A TCP flow can be throttled down by dropping only very few packets of this flow. So it is sufficient to drop only with the probability max_p. A drawback of this strategy is that not only TCP conforming flows are transmitted by a RED queue.

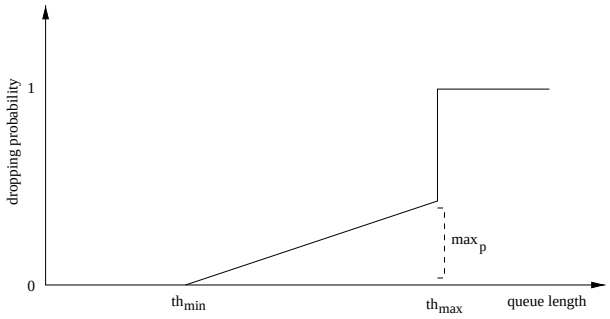


Figure 2.8: *Dropping probability for a packet in a RED queue*

th_{min} and th_{max}: These parameters specify in which interval the RED algorithm is used. Below th_{min} no packet is dropped, above th_{max} every packet is discarded.

Another element of the algorithm is to increase the dropping probability p_b with the number $count$ of transmitted packets since the last dropped packet. This way a dropping of sequential packets causing TCP to decrease its transmission rate too much can be avoided.

If a packet arrives at the RED queue and avg is smaller than th_{min} it is stored in the queue and avg is recalculated. If avg is larger than th_{max} the packet is dropped. In the case of the average queue size falling between the thresholds $th_{min} <$ avg $< th_{max}$, the arriving packet is either dropped or queued. Mathematically said, it is dropped with linearly increasing probability.

RED is very useful for TCP since it has the ability to flexibly specify traffic handling policies to maximise throughput under congestion conditions. Especially, RED is able to split bandwidth between TCP data flows in a fair way since lost packets automatically cause a reduction to a TCP data flow's packet rate. The situation becomes more problematic if data flows are involved, not conforming the TCP congestion avoiding mechanisms like UDP based real-time or multicast applications. Flows not reacting to packet loss have to be handled differently by reducing their data rate to avoid an overloading of the network.

Since RED cannot make unfair protocols like UDP behaving fair and most Internet traffic is a mixture of several types of traffic, the improvements gained by RED are hard to prove. Christiansen et al. [CJODS00] conclude their evaluation of RED with the statement, that it would be better to spend time on better network provisioning and use simple queueing mechanisms instead of wasting time for RED.

2.4.6 Random Early Detection for Assured Forwarding

To support Assured Forwarding flows with their different drop precedences, some modifications of the RED mechanisms were developed. An initial one was the RED with In and Out (RIO) mechanism proposed by Clark and Wroclawski [CW97]. They suggested to put the in and out of profile packets into different queues. Since this approach can change the packet order, it was replaced by a system consisting of one queue with a modification of the RED mechanism differentiating between the drop probabilities. With the specification of the Assured Forwarding Per Hop Behaviour, requirements for the queue were defined also.

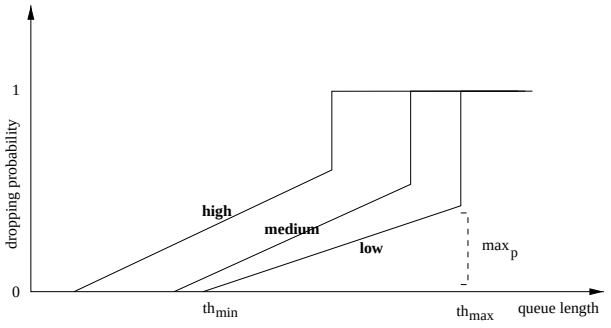


Figure 2.9: *Dropping probability for a packet in a queue for Assured Forwarding with three different drop precedences (TRIO)*

Therefore, the most important properties are also part of the Assured Forwarding RFC [HBWW99a]:

- The dropping algorithm must be insensitive to the short-term traffic characteristics of the micro flows using an AF class.
- Flows with different short-term burst shapes but identical longer-term packet rates should have packets discarded with essentially equal probability.
- The dropping algorithm must treat all packets within a single class and precedence level identically.
- The level of packet discard at each drop precedence in relation to congestion, must be gradual rather than abrupt, to allow the overall system to reach a stable operating point. The use of a common queue for the different drop precedences prevents changes of the packet order.

Heinanen also suggests a RED based mechanism to be used to for Assured Forwarding. The algorithm is similar to the RED algorithm as described in Section 2.4.5. Same as the standard RED algorithm it uses a weight w_q to calculate the average queue length but defines a parameter set for each of the three drop precedences.

$$\left\{ \begin{array}{l} (th_{min}, th_{max}, max_p)^{high} \\ (th_{min}, th_{max}, max_p)^{medium} \\ (th_{min}, th_{max}, max_p)^{low} \end{array} \right\}$$

Figure 2.9 shows a graph of a TRIO queue with the dropping probabilities for each precedence. The impact of different parameter sets on the behaviour of different flows will be evaluated in Section 3.2 using the ns network simulator.

2.5 Differentiated Service Networks

Beyond the traffic conditioning aspects Differentiates Services require policing, shaping and classification mechanisms within specific devices of a network. The amount of traffic a customer or an ISP may transmit to another ISP is defined within service level agreements.

Therefore any participant is interested and required to control whether the sent and received traffic exceeds the SLA. ISPs will not simply trust their customers to keep within the limits of the negotiated SLAs, but police incoming traffic and drop or reclassify the exceeding packets. On the other hand a customer usually wants to keep within the SLA but also profit from it as good as possible. To utilise the negotiated SLA and to keep in within the SLA the customer will apply shaping functions before the traffic is transmitted to the next ISP.

The concept of Differentiated Services defines several mechanisms usually located at specific locations within a network. Routers have to be configured to police incoming traffic to prevent customers exceeding their SLA, shape traffic to utilise a SLA as good as possible or simply forward packets as far as possible without any additional functionality.

ingress routers: A Differentiated Services ingress router receives IP packets from a neighbouring ISP or from customers. Within the SLA between the ISP and his neighbour, the traffic profile the ISP has to forward is defined. To keep within the profile policing functions are applied within the ingress router. The ingress router classifies incoming traffic according to the DSCP values or other informations like addresses, protocol ids (multifield classification) and drops or reclassifies packets exceeding the negotiated profile.

egress routers: Before traffic is forwarded to the next ISP it is profitable to apply shaping mechanisms usually. They guarantee that outgoing traffic conforms the SLA and that packets arriving in bursts won't be discarded. Usually the egress router uses a BA classifier, as any traffic to be processed has passed an ingress router and got the DSCP values set.

first-hop router A first-hop router has to perform similar functions as an ingress router. But as it can be seen in the next section, it might also be located within a customer network. A first-hop router located within the customer network has the advantage to be usually configured by the customer itself and therefore match better the Quality of Service demands of the customers network. In smaller networks with only one router, the customer's egress router may be the first-hop router, so the egress router has to apply MF instead of BA classification.

Of course it is up to the ISP to offer more advanced services to his customers and provide more first-hop router like functionalities within his ingress border routers like the marking of packets using higher protocol information or special treatment for specific end systems.

interior router An interior router (see Figure 2.10) does not perform any marking. It only (BA)-classifies packets according to their DSCP values and puts them to the appropriate queues. It is obvious, that an ISP has to ensure, that not more packets of a specific service can pass its border routers, than his interior routers can handle. Compared to first hop or ingress routers, interior routers are relatively simple. This simplicity is important, as these interior routers are used to set up an ISP's backbone and have to handle high packet rates.

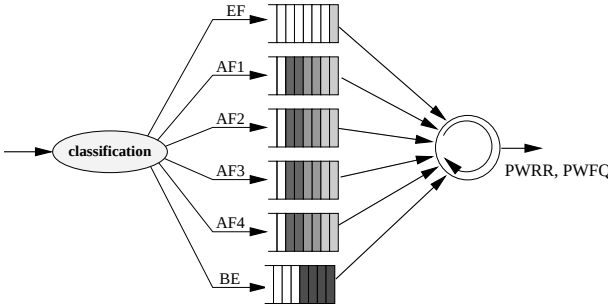


Figure 2.10: *Interior router*

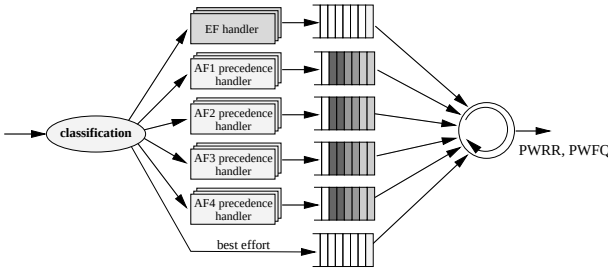


Figure 2.11: *First-Hop, Egress or Ingress Router*

Figure 2.11 shows the setup of an ingress, egress or first hop router. The classifier works either in MF (ingress, first-hop) or BA (egress) mode. After the MF/BA-classifier the packets are processed by the specialised mechanisms.

Since the queues attached to the AF classes have to handle multiple drop precedences, TRIO queues are necessary and Expedited Forwarding is processed by a RED queue. For the best effort traffic, a simple FIFO or a RED queue may be used.

2.5.1 Expedited Forwarding

Expedited Forwarding was proposed to provide a leased line service. This should allow companies to connect their LANs via a virtual private network with a certain Quality of Service comparable to the leased line service they are used to connect their head quarter with distant offices.

For Expedited Forwarding the customer negotiates with his ISP a maximum bandwidth for sending packets through the ISP's network. Furthermore, the aggregated flow is described by the source and destination addresses of the packets or by the address prefixes to support end to end connections as well as providing QoS for a connection between two networks.

In Figure 2.12 users and ISPs have agreed on a rate of three packets/s for traffic from A to B. The user configures the first-hop router in the individual subnet accordingly. In the example in Figure 2.12 a packet rate of two packets/s is allowed in every first-hop router. This is reasonable, because

- it can be assumed, that no two end systems will use the full bandwidth of two packets/s at the same time
- an end system should not be able to allocate the full bandwidth of three packets/s alone.

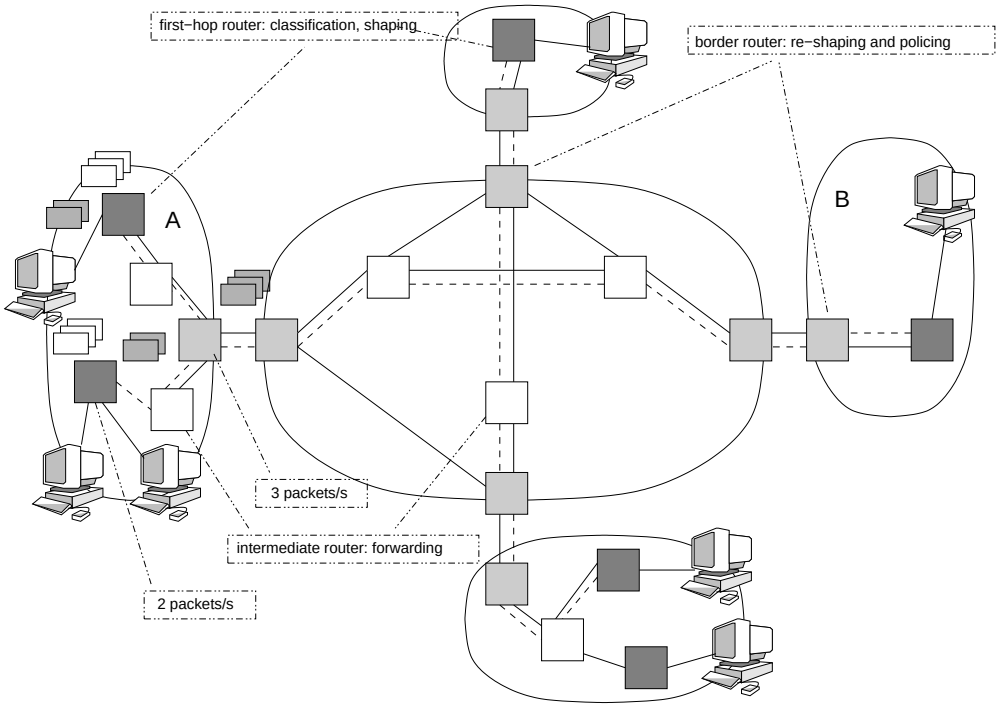


Figure 2.12: *Expedited Forwarding in a Differentiated Service network*

First-hop routers have the task to classify the packets received from the end systems, i.e. to analyse if the Expedited Forwarding Service shall be provided to the packets or not. If yes, the packets are tagged as EF and the data stream is shaped according to the maximum allowed bandwidth. The user's border router re-shapes the stream (e.g. three packets per second) and transmits the packets to the ISP's border router, which performs a policing function. For example, it checks whether the user's border router remains below the negotiated bandwidth of three packets/s. If each of the two first-hop routers allows two packets/s, one packet per second will be dropped by shaping or policing at the border routers. All first-hop and border-routers own two queues, one for EF-packets and one for all other (see Figure 2.12). If the EF-queue contains packets these are transmitted prior to others to guarantee Quality of Service.

The leased line like behaviour of Expedited Forwarding provides a service with concrete parameters independent from the networks actual state.

2.5.2 Assured Forwarding

A potential disadvantage of Expedited Service is the weak support for bursts and the fact that a user has to pay even if he is not using the whole bandwidth. Assured Forwarding tries to offer a service which cannot guarantee bandwidth but provides a high probability that the ISP transfers high-priority-tagged packets reliably. This can be compared to buying a minimum Quality of Service with the chance to get more if there is no congestion on the network. The three dropping precedences per Assured Forwarding class allow the Internet service provider and the customer to differentiate

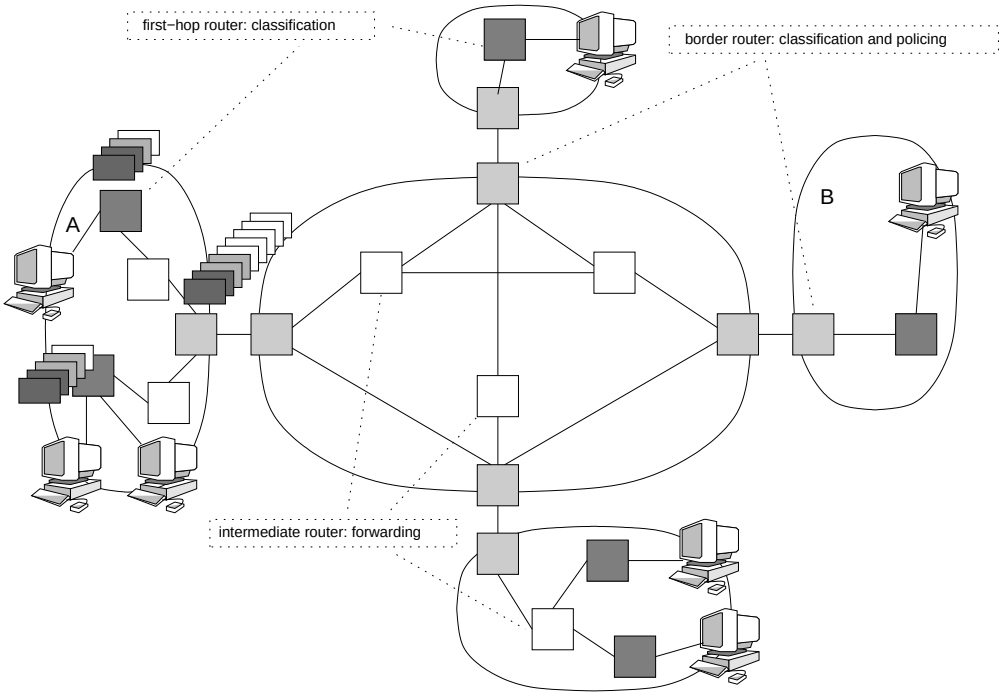


Figure 2.13: *Assured Forwarding in a Differentiated Service Network*

between two excess bandwidth values.

With the Assured Forwarding the user negotiates a service profile with his service provider. Such a service profile can contain the maximum packet rate per drop precedence.

The user may tag his packets as high, medium or low priority within the end system or the first-hop router, i.e. assign them a tag for assured forwarding (AF) (see Figure 2.13). To avoid modifications it is also possible to set up more intelligent first-hop routers. They can analyse the packets with respect to their IP addresses and UDP/TCP ports and then assign them the according drop precedence, i.e. set the AF-DSCP for conforming Assured Forwarding packets.

To ensure that the traffic conforms the profile the customer has to (re-)classify the packets in his border router.

Nevertheless, the service provider has to check if the user remains below the maximum rate for high, medium and low priority packets and apply corrective actions such as policing if necessary.

The example in Figure 2.13 shows a customer A, which is allowed to send two packets/s with low and two with medium drop precedence. The rest of the packets he wants to send is marked with high drop probability.

Bursts are supported by shaping the traffic and provide memory capacity to buffer the packets exceeding the maximal allowed bandwidth. Inside the network, especially in backbone networks forwarding thousands and millions of flows, bursts can be expected to equalise.

2.6 Conclusion

Even if the standardisation process of Differentiated Services is rather completed, the documents describe mainly the general aspects of the concepts and leave several questions open.

The proposals for Expedited and Assured Forwarded services define properties, to be provided by traffic conditioning components. They do not specify exactly which algorithms to use. Since especially Assured Service will depend crucially on proper traffic conditioning components, algorithms have to be developed or adapted and parameter sets have to be found. Heinanen proposes a RED based mechanism for Assured Service, leaving the question for proper RED parameters open. In the next chapter the impact of different parameter sets for such a RED based algorithm will be evaluated.

This question is even more interesting, since the traffic forwarded by a router within the Internet usually consists of different protocols. Even without Differentiated Services the interaction of these protocols is not easy to predict. How Differentiated Services traffic conditioning components influence the interaction of these protocols is important for providing Quality of Service. Especially the behaviour of flows under congestion conditions is important, since an ISP sells certain services to his customers and has to fulfil these contracts.

In the following chapter the network simulator ns will be used to evaluate the behaviour of Assured Services. New components had to be added to allow the simulation of Assured Service.

Chapter 3

Evaluation of Assured Forwarding using the ns Network Simulator

To provide Quality of Service to specific flows usually means a discrimination of other traffic. Any concept providing Quality of Service for the Internet has to distinguish between packets to be handled preferential and best effort traffic.

RSVP defines a protocol, that allows the end user to reserve resources along a path through a network. The RSVP protocol will also inform the user whether the desired resources were set up correctly or not. In the Differentiated Services concept no such protocol is defined and the Service Level Agreements negotiated with an Internet Service Provider are usually more static.

While each flow set up with RSVP is handled separately and requires information to be stored within each router on the flow's path, DiffServ works on aggregates. An intermediate Differentiated Services router has no information about flows or reservations at all. All the router knows is how to handle the traffic classes, depending on their DSCP values.

The focus of the evaluations in this chapter is on Assured Forwarding. In contrast to Expedited Forwarding [JNP99], [DCB$^+$01] the implementation of the Assured Forwarding PHB [BBC$^+$98b] requires fundamentally new queueing methods.

Mechanisms to implement Expedited Forwarding like Absolute Priority Queueing (see Section 2.4.1) are well known and are deployed within the Internet since years. These mechanisms are provided by almost all modern routers and are also used by protocols like RSVP.

Of course Expedited Forwarding works on a behaviour aggregate instead of dealing with single flows like RSVP does. On the other hand this makes not much difference for the traffic conditioning components. While packets of a RSVP flow are treated in a certain manner due to their addresses and port numbers, packets of an EF aggregate are treated due to their DSCP. The used components are rather the same. Additionally EF is intended to provide some leased line like quality, so congestion will only occur as a result of bad network configuration.

Assured Forwarding is different. As described in Section 2.4.6 Assured Forwarding

uses four classes with three drop precedences per class. Each class is handled by a single queue processing packets of different drop precedences.

Since protocols like TCP react to packet loss and also to changes of the round trip time[1], the reaction of such protocols to a queue treating packets differently is not clear. To make things even more complicated such a queue will probably forward protocols of different types, some reacting to packet loss like TCP, others not. Therefore the behaviour of TCP and more aggressive protocols like UDP with different types of drop precedences has to be investigated.

Another problem is caused by the RED mechanism (see Section 2.4.5) Heinanen proposes in [BBC+98b] for the implementation of the Assured Forwarding PHB. RED uses several parameters controlling its behaviour. The parameter set is even enlarged due to the adaptions necessary for the handling of different drop precedences, making the proper configuration of RED complicated.

For these evaluations the ns network simulator [ns] was used. ns is a TCL/C++ based simulation package, even allowing an evaluation of large topologies. Although if ns uses a rather abstract model of a computer network, it is very useful to evaluate traffic conditioning mechanisms. The chapter starts with a description of the ns simulator and the required modifications [BB99b] to simulate Differentiated Service networks. First tests concern the impact of different parameter sets for the RED algorithm on Assured Forwarding. To provide a better control over the time packets are delayed a new approach is presented and evaluated. Finally the fairness of Assured Service is evaluated using different protocols. A short conclusion summarises the results of the experiments.

3.1 Model, Traffic and Topology

For the evaluation a typical network situation as shown in Figure 3.1 was chosen. A number of users in a branch office will access a set of servers in the company's headquarter. Both networks are connected by an Internet Service Provider's network, which is slow compared with the local LANs in the offices. To provide a certain Quality of Service the company has a contract with the ISP. Within its ingress router the ISP has to mark packets according to the negotiated profile and forward them accordingly.

The topology of Figure 3.1 with a branch office receiving data from its headquarter's servers was implemented in ns, resulting in the abstract topology shown in Figure 3.2. The figure shows also the different types of queues being used to simulate a Differentiated Services network.

Since all links have equal bandwidth capacities, the link between I_1 and I_2 gets a bottleneck as soon as more than one of the server client pairs S_i/C_i starts to send at full bandwidth.

To simulate the behaviour of Differentiated Services, two additional components were implemented: a Differentiated Service marker and a "Red with In and Out" (RIO)

[1] the time a packet needs to its destination and back

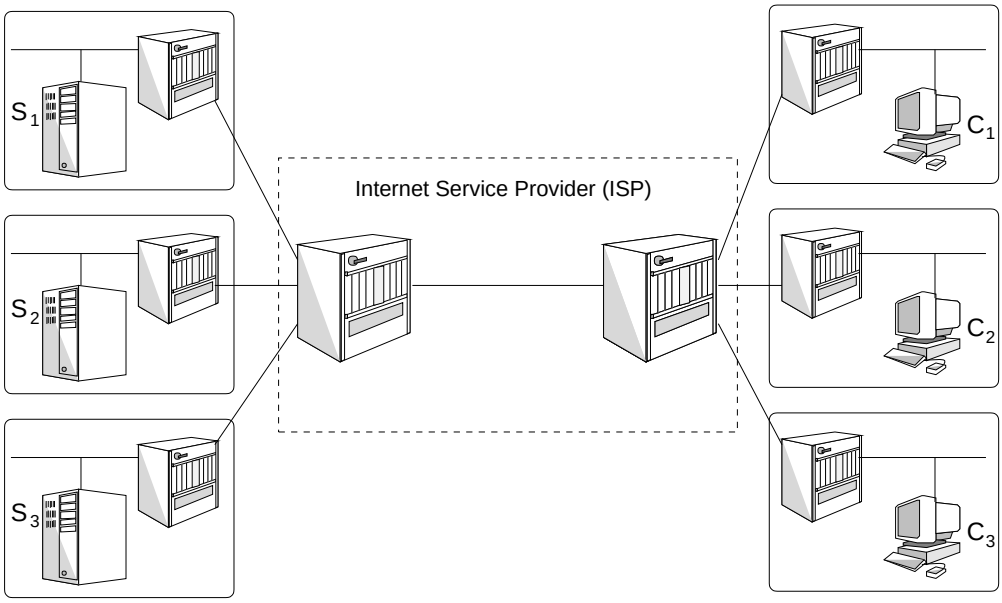


Figure 3.1: *Simulation Scenario. Six user networks are connection via an ISP network*

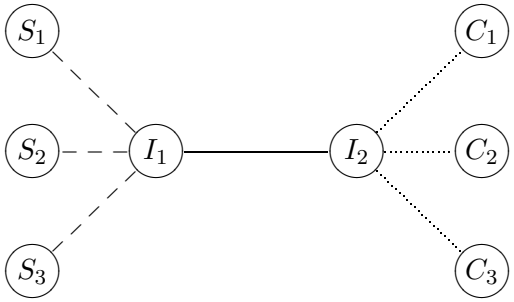


Figure 3.2: *Simulation Scenario in ns. The dashed lines represent marker queues, the solid one a RIO queue and the dotted ones the simple FIFO queues*

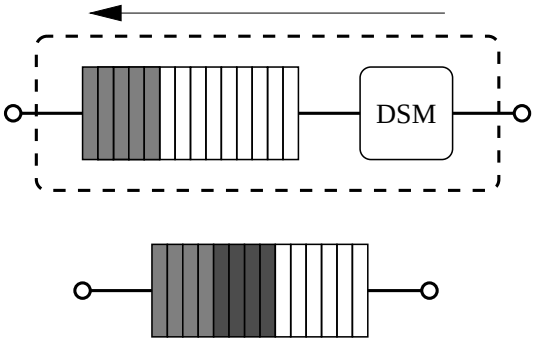


Figure 3.3: *The upper diagram shows the marker queue with a FIFO and the Differentiated Services marker algorithm. The diagram below shows the RIO queue, capable to handle multiple dropping probabilities.*

queue. In contrast to a real network these components are not located within a node, but are part of the links between the nodes. The marker is located between the server end systems (S_x) and the node I_1 (dashed lines) and the RIO queue between the central nodes I_1 and I_2. As both LANs were supposed to be fast compared to the end systems, the links between I_2 and the clients (C_x) will have no impact on the result and are simply realized by FIFO queues.

The ns code for such a FIFO queue was also used as a basis for the implementation of the new components. As can be seen in Figure 3.3 the original FIFO queue was extended by a preceding Differentiated Service marker algorithm. The RIO queue extends the usual FIFO mechanism with the capability to differentiate between dropping probabilities and process the packets according to their DSCP values. Both marker and RIO queue support two drop precedences following the initial approach of Clark and Fang [CF97] to use low drop precedence for packets within the negotiated profile and high drop precedence for the others.

In general, there are two approaches for implementing the RIO technique. Normally a common queue for in and out of profile packets is used. The higher priority of in profile packets is realized by lower dropping probabilities. In contrast earlier proposals were based on different queues for in and out of profile packets. Since multiple queues might result in changes of the packet order, the simulations presented here will use common queue for in and out of profile packet. The task of the marker component is to decide, whether a packet is in or out of the negotiated service profile or not and to set the according DSCP value. Figure 3.4 shows the probability for a packet to be marked as ”in profile” dependent on the transmission bandwidth. The graph represents the negotiated service profile.

For the decision which packet should be marked as in or out of profile two methods are proposed [IN98].

- The transmitted bandwidth can be calculated from the packet size and the inter packet time, averaging both values (e.g. exponential weighted moving average [FJ93]). For a good interaction with TCP/IP it is essential to react only slowly to changes of the current bandwidth.

- A token bucket filter can be used for tagging a certain share of transmitted packets.

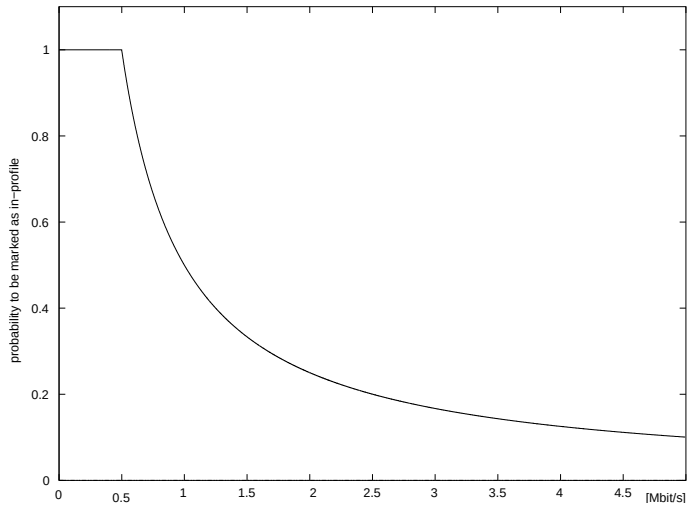


Figure 3.4: *Probability P for packets been marked as in profile at 500 kbps assured bandwidth over the sent bandwidth*

Since a comparison of both algorithms did not lead to a significant difference, the simpler Token Bucket algorithm was used. The bucket capacity allows the aggregation of one second of assured bandwidth. This allows to describe bursts very easily.

According to the assured bandwidth, certain packets are marked ”in profile”. During congestion packets are discarded with different probabilities in the RIO queue between I_1 and I_2. No packets are discarded at the link between I_2 and C_x, since they have at least the capacity of the bottleneck link $I_1 - I_2$.

3.2 Influence of RIO Parameters

As previously mentioned the impact of different parameter sets for the RIO (or TRIO) queue has to be evaluated. In Sections 2.4.5 and 2.4.6 the algorithms used for the implementation of RED, RIO or TRIO queues were described.

As mentioned there the behaviour of a RIO queue is defined by two sets of RED parameters

$$\left\{ \begin{array}{l} (th_{min}, th_{max}, max_p)^{in} \\ (th_{min}, th_{max}, max_p)^{out} \end{array} \right\}$$

and a common parameter w_q, which is used for averaging the queue length. An example for such a set is given by Yeom [YR98]. He suggests $\{50, 100, 0.02\}$ for in and $\{20, 40, 0.5\}$ for out packets. The queue weight w_q can be supposed to be the default value of 0.002 suggested in [FJ93].

Type	in	out	bandwidths	delays
2 × TCP + UDP	$\{0.5, 1.0, 0.02\}$	$\{0.2, 0.4, 0.02\}$	fig. 3.10	fig. 3.11
3 × TCP	$\{0.5, 1.0, 0.02\}$	$\{0.2, 0.4, 0.02\}$	fig. 3.6	fig. 3.7
2 × TCP + UDP	$\{0.5, 1.0, 0.02\}$	$\{0.2, 0.4, 0.5\}$	fig. 3.12	fig. 3.13
3 × TCP	$\{0.5, 1.0, 0.02\}$	$\{0.2, 0.4, 0.5\}$	fig. 3.8	fig. 3.9
2 × TCP + UDP	$\{0.5, 1.0, 0.02\}$	$\{0.2, 0.5, 0.5\}$	fig. 3.14	fig. 3.15

Table 3.1: *Overview over tested RED configurations and parameter settings. In the first line the parameter used by Yeom [YR98] are shown. Each set of values represents* $\{th_{min}, th_{max}, max_p\}$.

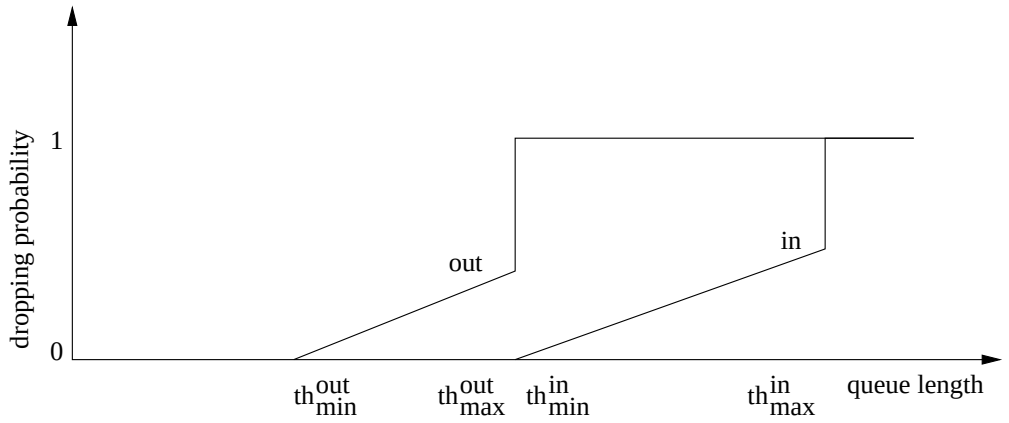


Figure 3.5: *The RIO parameters with the two dropping phases for in and out of profile packets*

The simulations use two drop precedences to investigate the impact of different parameter sets. The central queue between I_1 and I_2 is configured with different parameters. The RED algorithm is assumed to work fine for an aggregate of TCP flows. Unfortunately this does not match the situation in the Internet. Any traffic aggregate will consist of packets from several different protocols. With applications like video conferencing or streamed audio and their usually UDP based protocols, it is quite sure that some of the protocols will not react to congestion as TCP does.

Therefore, an aggregate of two congestion avoiding TCP flows and one non-responsive UDP flow is used and the RIO parameters are varied. All links are configured for a bandwidth of 1 Mbps and a delay of 1 ms and get different assured bandwidth values. The two TCP connections get 0.5 Mbps and 0.3 Mbps. The aggressive UDP flow has no reserved bit rate and is used as some kind of background noise. The values for th_{min} and th_{max} are defined in relation to the complete queue length of 50 packets.

Table 3.1 shows the evaluated configurations. First tests have been done with different settings of the max_p value for the out of profile queue. After this th^{out}_{min} was adjusted to th^{in}_{min}, as can be seen in Figure 3.5, causing the dropping of in profile packets to start directly after the dropping phase for out of profile packets.

The graphs lead to no unique result. However, a higher value for max^{out}_p seems to improve the differentiation between the achieved troughput values (see Figure 3.6 and

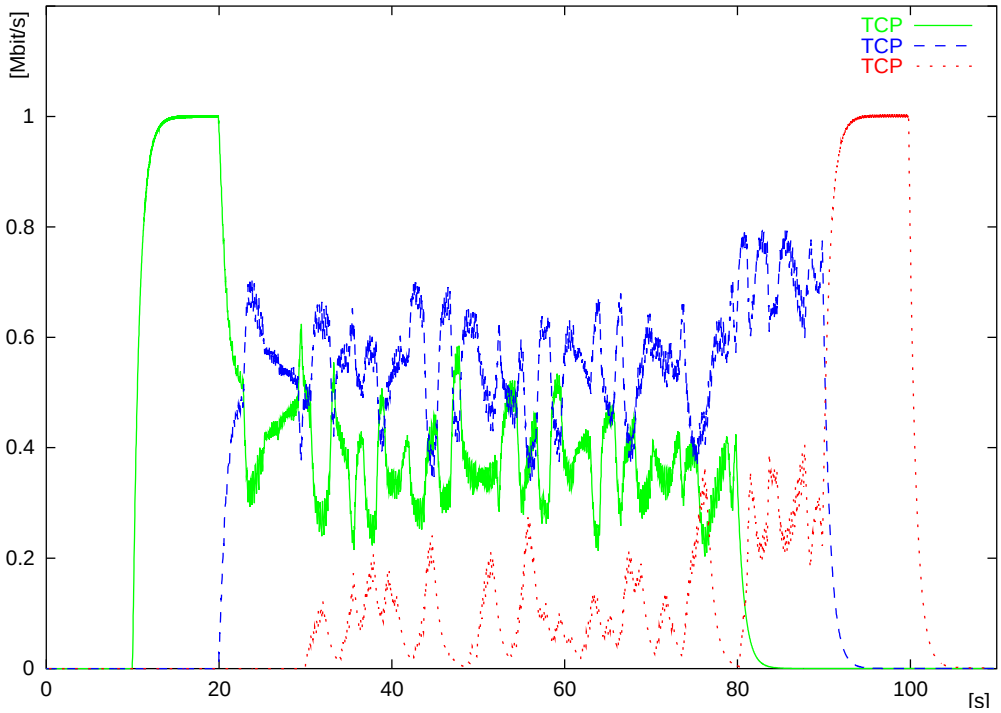


Figure 3.6: *Bandwidth of three TCP flows with different assured bandwidths. RIO parameters: in*=$\{0.5, 1.0, 0.02\}$, *out*=$\{0.2, 0.4, 0.02\}$ *and* $w_q = 0.002$

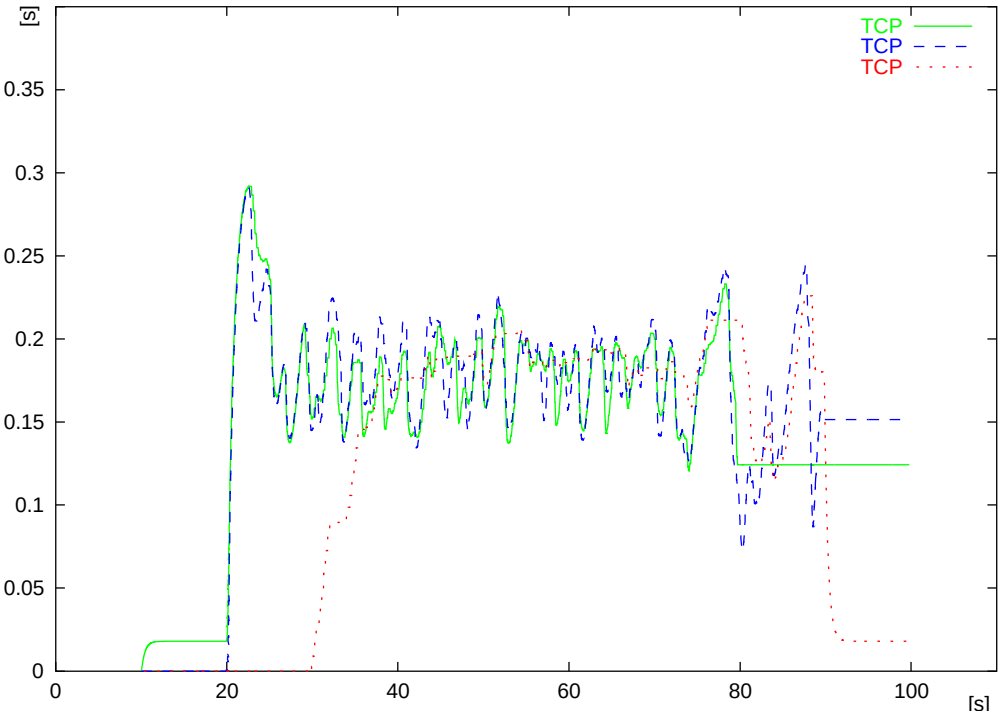


Figure 3.7: *Latency of three TCP flows. RIO parameters: in*=$\{0.5, 1.0, 0.02\}$, *out*=$\{0.2, 0.4, 0.02\}$ *and* $w_q = 0.002$

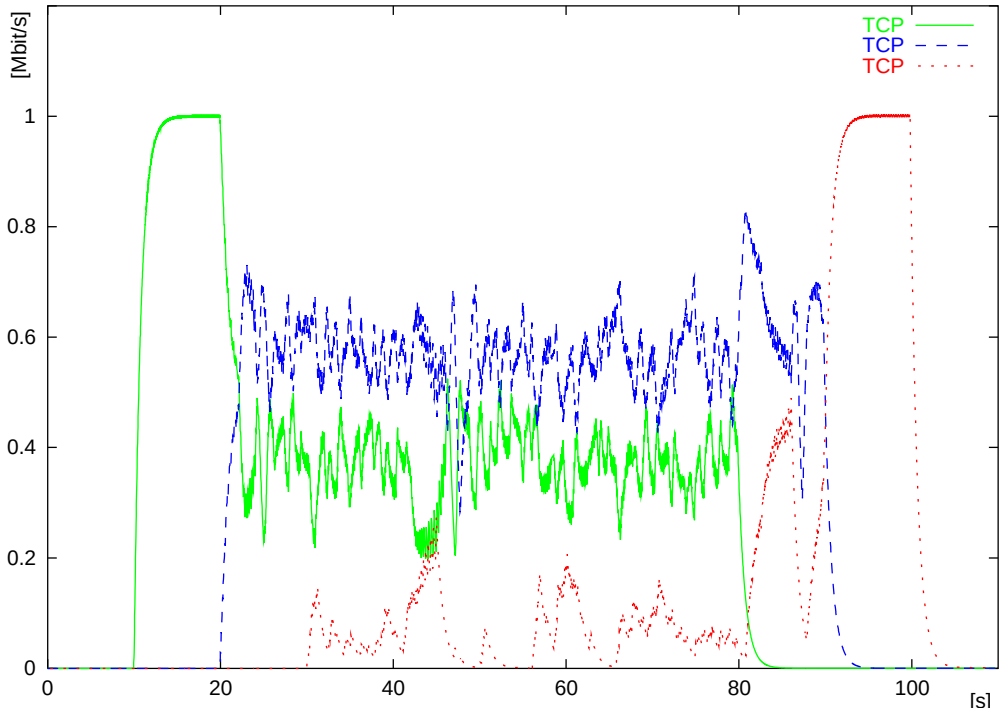


Figure 3.8: *Bandwidth of three TCP flows. RIO parameters are:* $in = \{0.5, 1.0, 0.02\}$, $out = \{0.2, 0.4, 0.5\}$ *and* $w_q = 0.002$. *Because of the suppression of best effort traffic the bandwidths of the single flows are more separate those with a lower value of* max_p^{out} *(see Figure 3.6)*

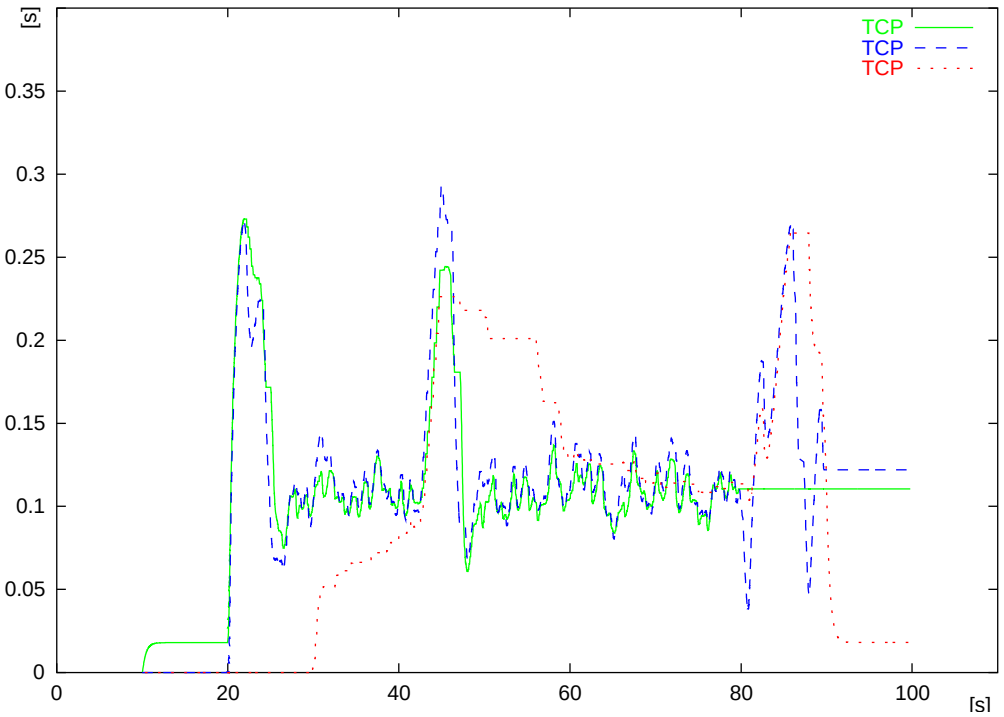


Figure 3.9: *Latency of three TCP flows. RIO parameters:* $in = \{0.5, 1.0, 0.02\}$, $out = \{0.2, 0.4, 0.5\}$ *and* $w_q = 0.002$

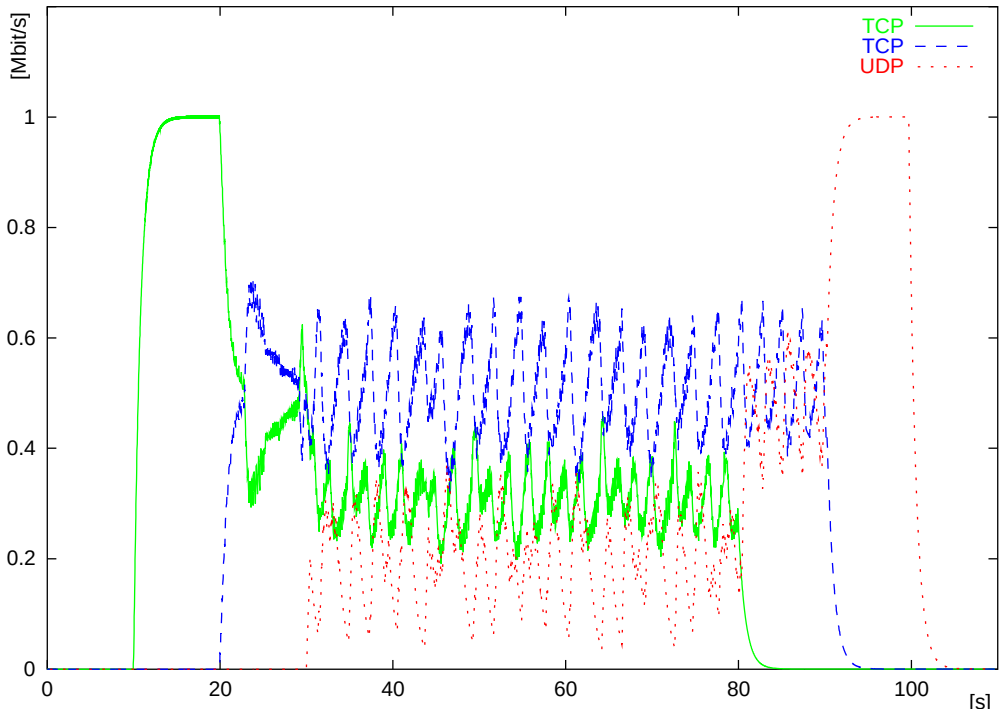


Figure 3.10: *Bandwidth of two TCP and one UDP flows. RIO parameters are:* $in = \{0.5, 1.0, 0.02\}$, $out = \{0.2, 0.4, 0.02\}$ *and* $w_q = 0.002$

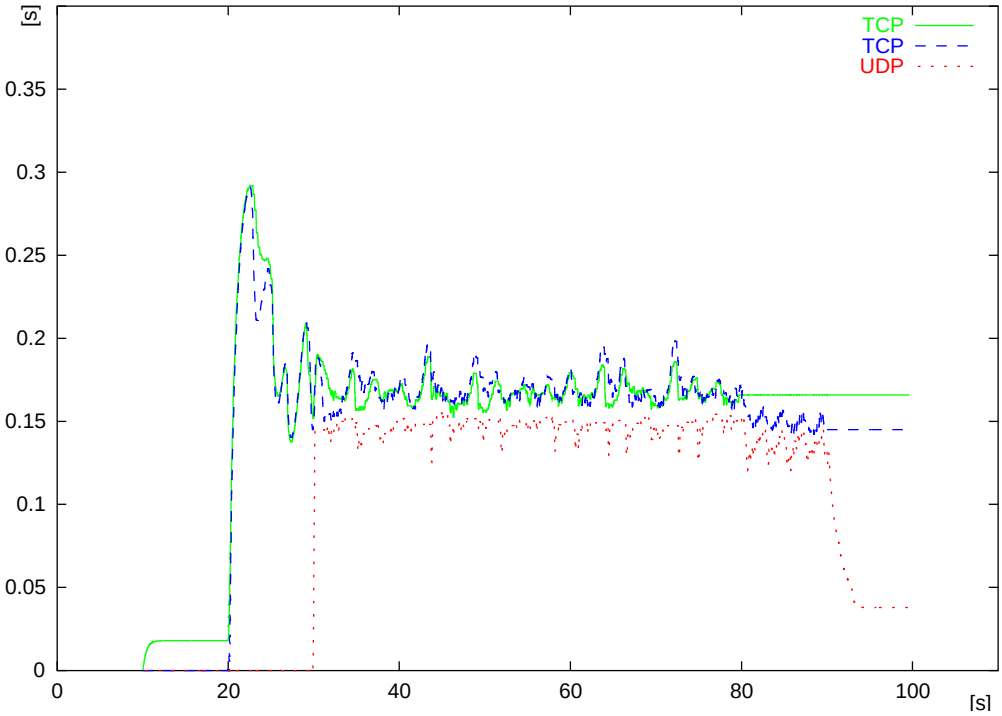


Figure 3.11: *Latency of two TCP and one UDP flow. RIO parameters:* $in = \{0.5, 1.0, 0.02\}$, $out = \{0.2, 0.4, 0.02\}$ *and* $w_q = 0.002$. *Because of low* max_p^{out} *belowth*$_{max}^{out}$ *only few best effort packets are dropped causing a drop tail like behaviour. The lower delay of the best effort UDP packets is obvious.*

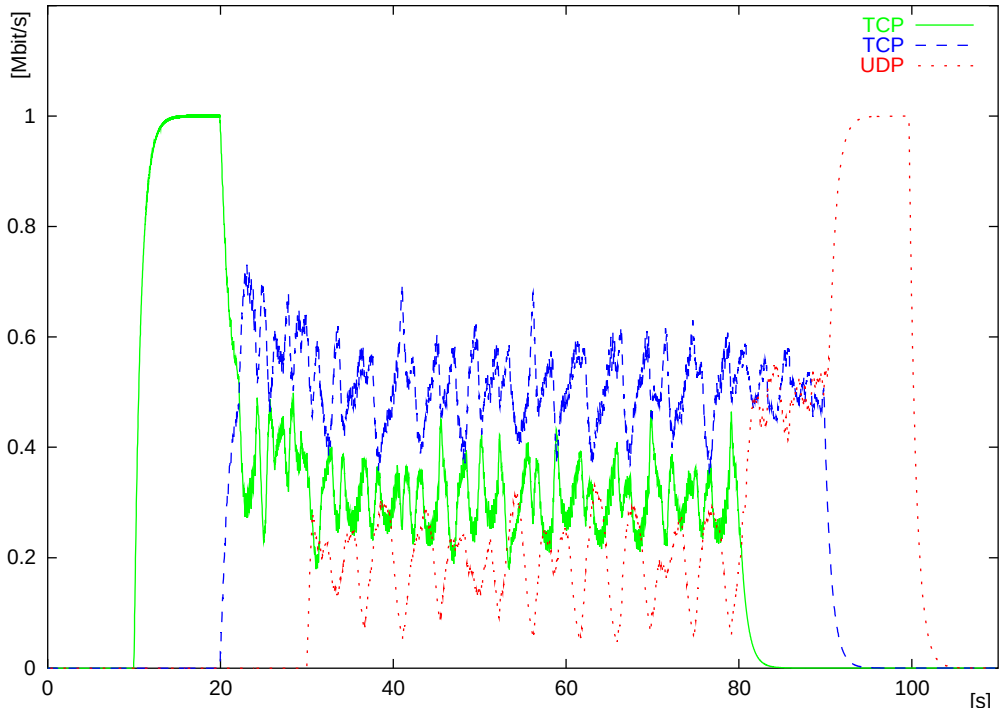


Figure 3.12: *Bandwidth of two TCP and one UDP flows. RIO parameters are:* $in = \{0.5, 1.0, 0.02\}$, $out = \{0.2, 0.4, 0.5\}$ *and* $w_q = 0.002$

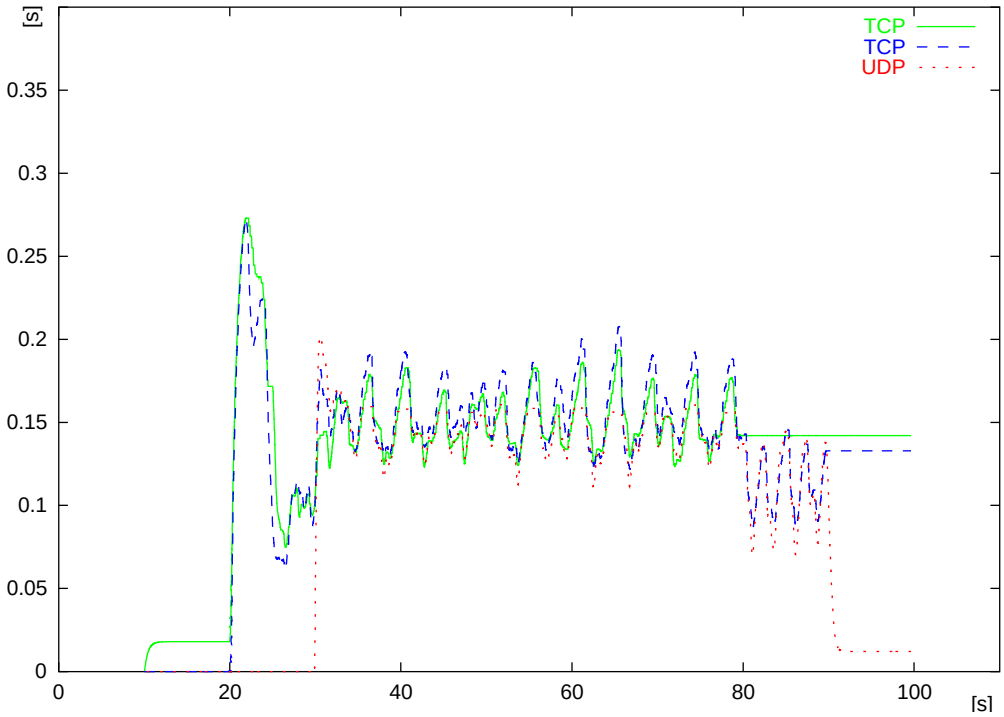


Figure 3.13: *Latency of two TCP and one UDP flow. RIO parameters: in:*$\{0.5, 1.0, 0.02\}$ *and out:*$\{0.2, 0.4, 0.5\}$, $w_q = 0.002$.

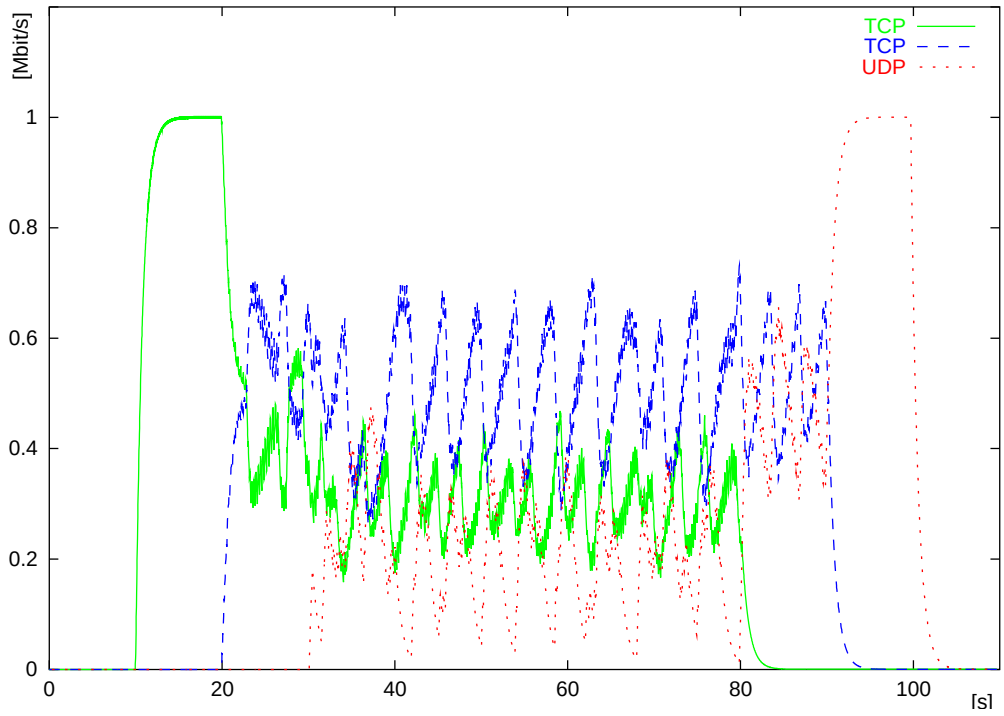


Figure 3.14: *Bandwidth of two TCP and one UDP flows. RIO parameters are:* $in = \{0.5, 1.0, 0.02\}$, $out = \{0.2, 0.5, 0.5\}$ *and* $w_q = 0.002$. *The UDP best effort flow disturbs the transmission of TCP traffic. (see Figure 3.12)*

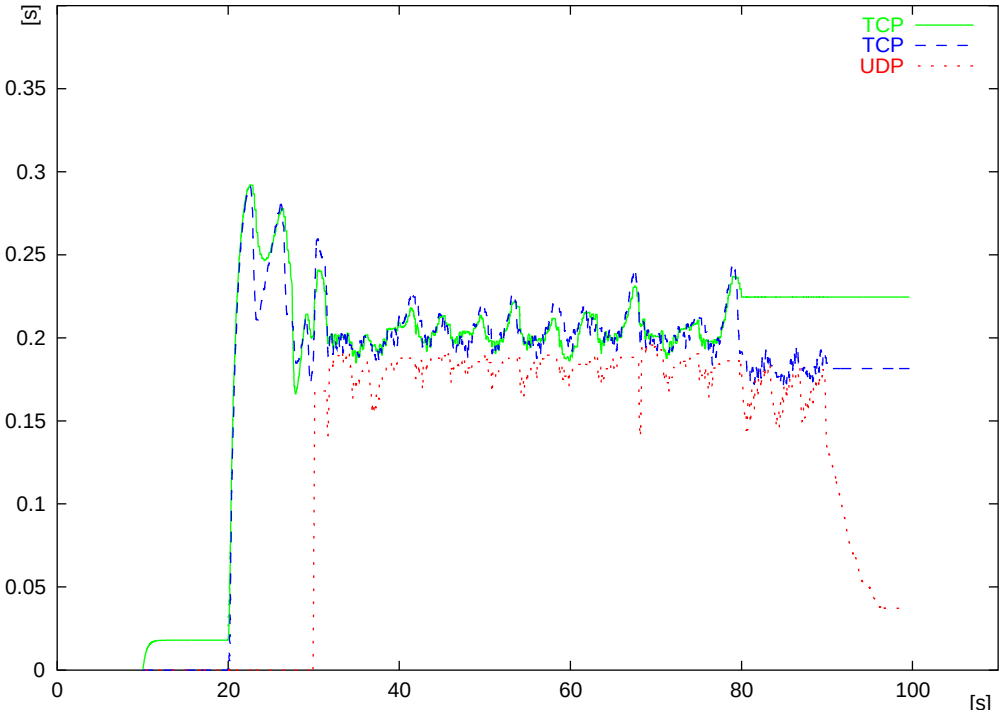


Figure 3.15: *Latency of two TCP and one UDP flow. RIO parameters:* $in = \{0.5, 1.0, 0.02\}$, $out = \{0.2, 0.5, 0.5\}$ *and* $w_q = 0.002$

3.8 for TCP and Figure 3.10 and 3.12 for mixed TCP and UDP traffic). As can be seen in Figure 3.11, a side effect of dropping out of profile packets, while in profile packets are still accepted is that out of profile packets reaching their destination have a better delay than in profile packets.

The problem of RED is the complexity of its parameter interaction and their effects on different kinds of traffic. On one hand, a small value of max_p is quite appropriate for TCP traffic, because of TCP congestion control. On the other hand it leads to a FIFO like behaviour of the queue for aggressive UDP flows.

Disregarding the complicated behaviour of the RED algorithm, one result seems obvious: Assured Service is able to protect even congestion avoiding TCP flows against aggressive UDP traffic.

3.3 Guaranteeing Delay with Assured Service

The original idea of the RIO algorithm [CF97] mentioned two queues, one for each type of packet. To avoid changing the packet order, RIO has been realized with one queue only and with different dropping probabilities for each packet type. Since the delay of packets primarily depends on the queue length the packet experiences on its way through the network, an in profile packet is delayed in a congestion situation, even if there is no overload of Assured Service [BB99a]. In reality every out of profile packet reaching its destination has a shorter delay as an in profile packet.

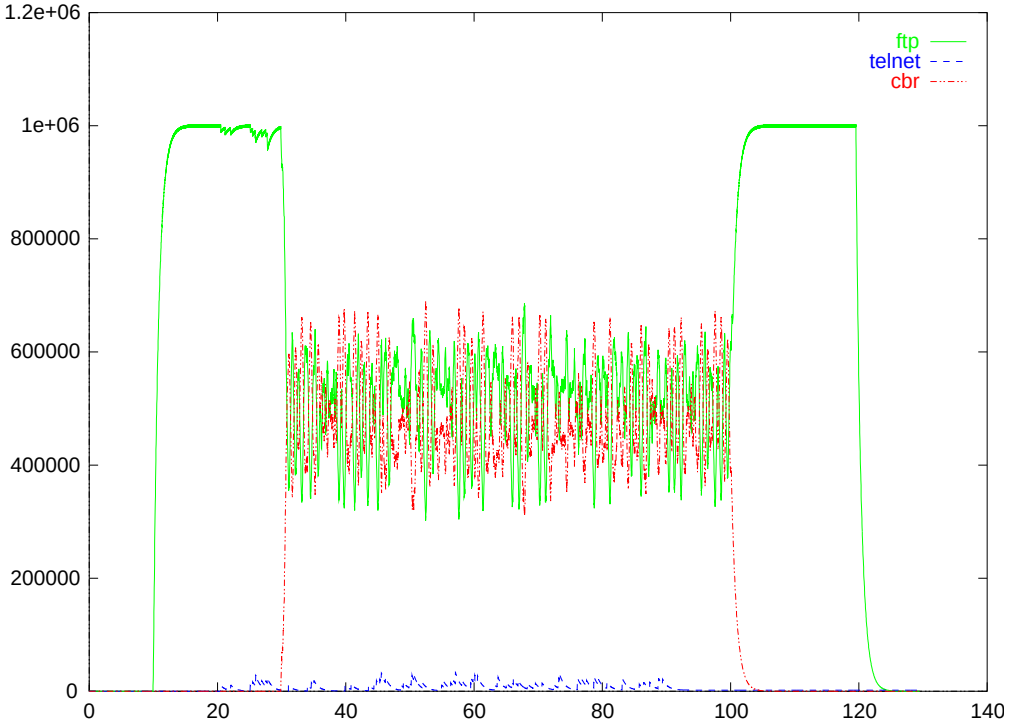


Figure 3.16: *Bandwidths with two queue RIO and Assured Service*

Since it is desired to ensure not only bandwidth but also a certain maximum delay for in profile packets better than the delay for out of profile packets, a separate queue for every packet type is implemented allowing better control over the delay of each packet type.

The scenario is similar to the one used during the evaluation of RIO parameters, except that different traffic sources are used. A FTP and a telnet session have been mixed with aggressive constant bit rate UDP traffic. The maximum bandwidth the FTP source is able to send is 1 Mbps, whereas the telnet source typically produces only very little and is primarily used for evaluation of the delay. The UDP connection sends at a constant bit rate of 0.8 Mbps. The FTP connection has an assured bandwidth of 0.5 Mbps, the telnet connection of (never used) 0.2 Mbps and the UDP traffic has no reserved bandwidth at all. Figure 3.16 shows the averaged bandwidth values and Figure 3.17 the packet delay.

The queue for in profile packets is emptied first. If there is any packet in the in profile queue, the packet is transmitted regardless how many packets are in the out of profile queue. As it can be seen clearly, the in profile packets for FTP and telnet suffer only a very short delay. In contrast, the delay for the UDP traffic increases. The drawback using this technique is the permutation of packets, causing worse performance of the TCP/IP stack.

An extension of the RIO algorithm with two queues is presented in Figure 3.18. The basic idea is to be able to control the delay for each packet type separately and minimise the number of changes in the packet order. The extended RIO queueing mechanism uses two different queues for in and out of profile packets. In addition, every

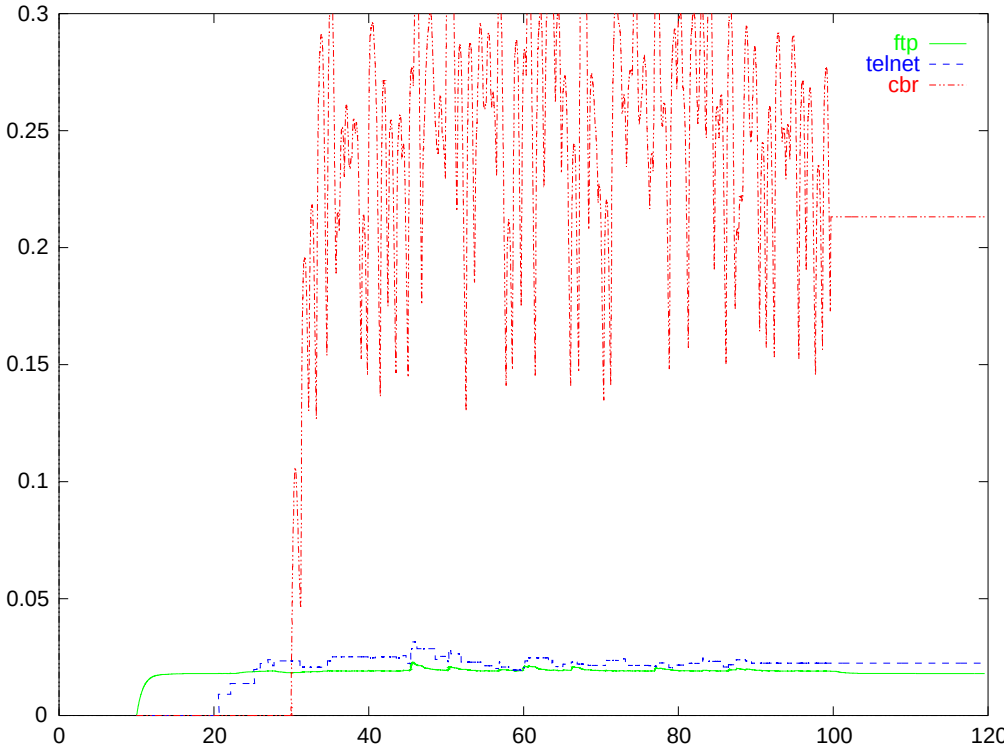


Figure 3.17: *Latencies with two queue RIO and Assured Service*

packet is marked with a ”time stamp” with the time it is received by the router, which makes it possible to reconstruct the packet order at the end of the queues. In reality this ”time stamp” is a simple number representing the order of packet arrivals at the queue. At the end of the queues a component (re-sequencer) decides whether a packet of the in or the out of profile queue will be forwarded. In contrast to the algorithms mentioned before, the router can be configured with a maximum allowed delay for in profile packets. The re-sequencer uses this allowed latency to forward between the high priority in profile packets as many out of profile traffic as possible. The re-sequencer stops transmitting out of profile packets, if an in profile packet with an earlier time stamp is available or if the next in profile packet would exceed the maximum allowed delay.

So the re-sequencer tries to keep the packet order. If an in-profile packet has to be forwarded even if does not match the packet order, but it reached the maximum allowed latency, the packet order will be disturbed. In this case the packet order may either be reconstructed by dropping all out of profile packets not matching the packet order or a certain degree of packet dis-ordering is accepted. Of course the packet order should be preserved, but on the other hand out of profile packets might be dropped, which could have been forwarded even if not matching the packet order. The following simulations do not try to preserve the packet order strictly and out of profile packets are only dropped if the out of profile queue gets full.

Figure 3.19 shows the number of changed packet orders using the traditional two queue RIO algorithm as proposed by [CF97] and the re-sequencing variant. Since the disturbance of the packet order gets higher the more congestion appears, the original

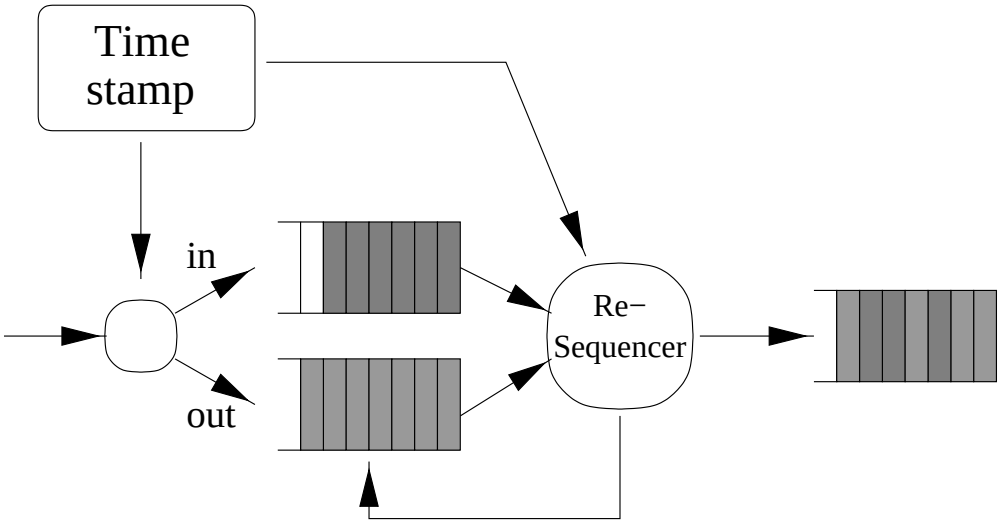


Figure 3.18: *extended RIO queueing with re-sequencing*

approach was evaluated with two different packet rates (800 and 400 kbps). As can be seen on the graph the mechanism with the re-sequencer disturbed the packet order much less than the traditional approach.

3.4 Fairness of Assured Service

After the experiments regarding the influence of RIO parameters and the presentation of a new RIO mechanism to provide better control over packet delays, the bandwidth of several flows shall be investigated, using different amounts of assured bandwidth values and different protocols.

For the simulation the setup described in Section 3.1 has to be changed slightly. Instead of three client/server pairs, ten pairs $\{(S_i, C_i), ..., (S_{10}, C_{10})\}$ are set up and all links of the simulation are configured equally to transmit 1.0 Mbps with a delay of one millisecond.

Different traffic sources have been simulated: constant bit rate UDP flows, several TCP flows and the mix of both. The ten client/server pairs are transmitting at full bandwidth causing a tenfold overload on the bottleneck link. A different assured bandwidth is assigned to every Server S_i, causing each "marker" queue S_i-I_1 to mark another percentage of forwarded packets as in profile. Every source sends with the same maximum bit rate of 1 Mbps. The single sources start delayed to each other. Although there always is an overload on the bottleneck links, we will change the amount of assured bandwidth varying the congestion of in profile traffic.

3.4.1 Heavy Overload

First evaluations have been done to simulate heavy overload of in profile traffic on the bottleneck link between I_1 and I_2. The single flows get assured throughput values

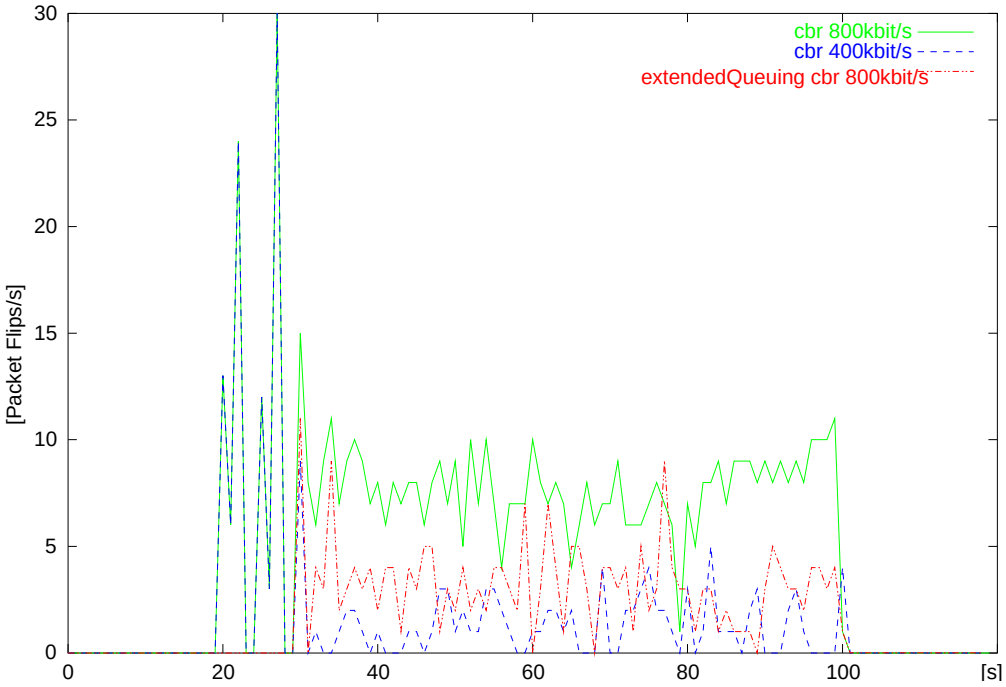


Figure 3.19: *Number of switched packets on the FTP connection at different of the CBR source*

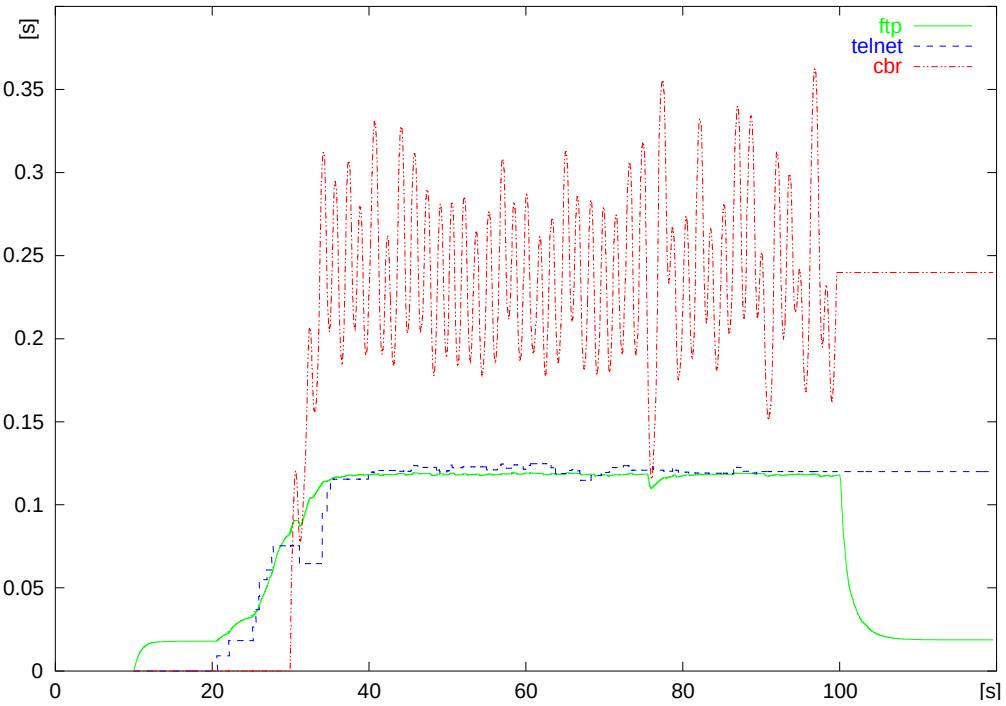


Figure 3.20: *Latencies with a maximum delay of 100 milliseconds allowed for in profile packets*

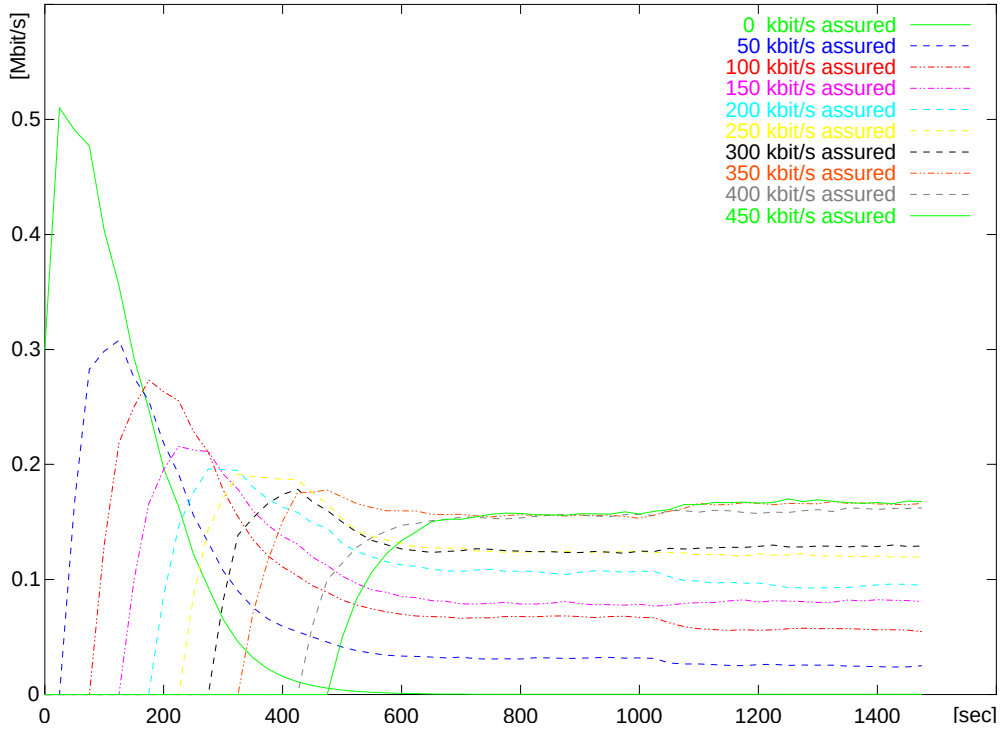


Figure 3.21: *Bandwidth of ten CBR flows causing heavy congestion*

from 0 to 450 kbps and start to send at different times. The flow starting to send as last is configured with the highest assured bandwidth. The total amount of assured bandwidth is 2.250 Mbps which is more than twice the bandwidth available.

Figure 3.21 shows results using ten constant bit rate UDP-type connections. The first flow (the one with no assured bandwidth) gets no bandwidth whereas the flow that started sending as last and with the highest assured bit rate reaches the highest throughput. Between these two extrema, all other flows share the bandwidth in a relatively fair way. The second column of table 3.2 shows the percentage of assured bandwidth actually achieved.

It can be seen, that each UDP flow can achieve about 40 to 55 percent of its assured bandwidth. Now we use exactly the same topology and assured bandwidth values but instead of the constant bit rate UDP traffic we use TCP.

Figure 3.22 shows the resulting bandwidth values. The flow without assured bandwidth is not able to transmit any data in the congestion situation, whereas the others perform more or less according to their assured bandwidth. On the other hand, the single throughput values do not differ as obvious as this was the case with the UDP flows. The reason for this is the TCP congestion control, causing a source to reduce the sending bandwidth in an overload situation. The third column in Table 3.2 shows - like before with constant bit rate UDP - the achieved percentage of assured bandwidth. The decrease of achieved bandwidth with the amount of assured bit rate is obvious: The lower the assured bit rate is the higher is the probability that a flow gets this bandwidth.

As a final aspect of the evaluation of Assured Service, the interaction of both types of

ass. bw	UDP	TCP	UDP or TCP
0	-	-	-
50	49 %	94 % †	88 %
100	55 %	86 % †	4 % †
150	54 %	86 % †	85 %
200	48 %	57 % †	<1 % †
250	48 %	47 % †	80 %
300	49 %	39 % †	<1 % †
350	47 %	37 % †	80 %
400	40 %	27 % †	<1 % †
450	37 %	28 % †	73 %

Table 3.2: *Five TCP, five UDP flows with different assured bandwidth values, causing heavy congestion. The Table shows the percentage of assured bandwidth reached. The †marks the TCP flows.*

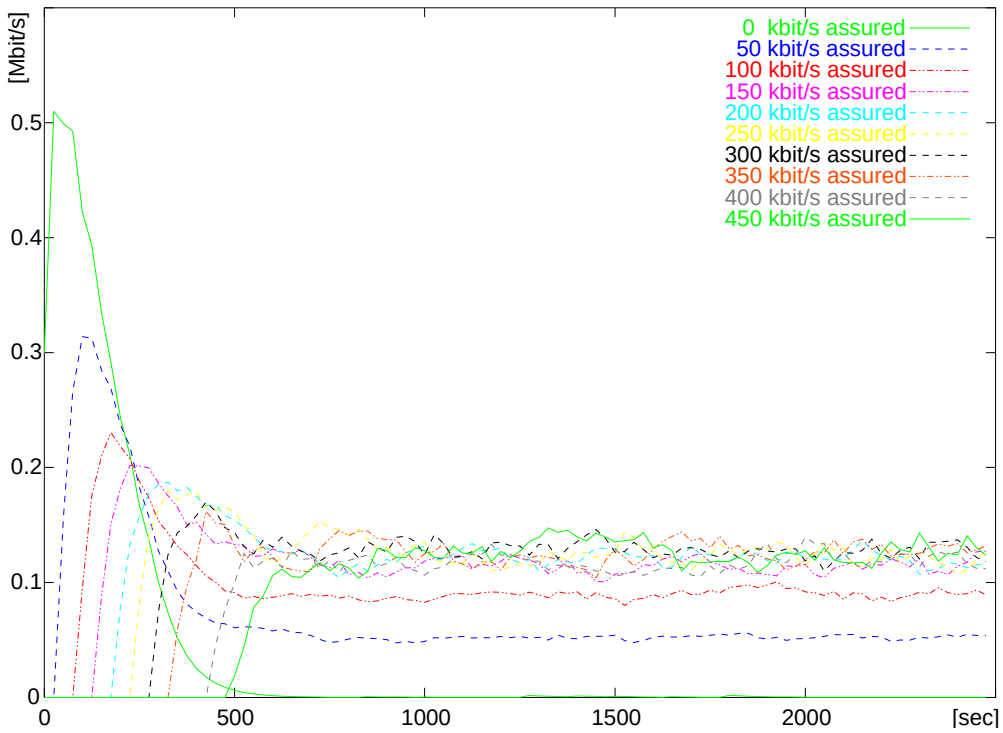


Figure 3.22: *Bandwidth of ten TCP flows causing heavy congestion*

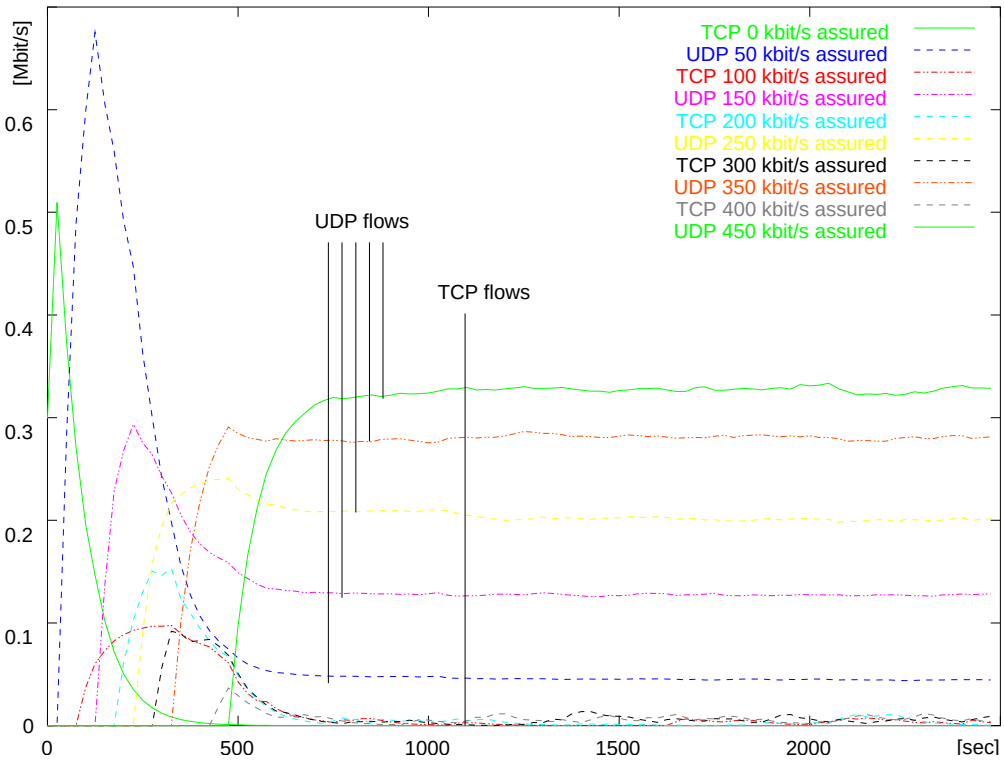


Figure 3.23: *Bandwidth of mixed CBR TCP flows causing heavy congestion*

traffic has been examined. The constant bit rate UDP traffic is supposed to suppress the TCP flows during overload. Figure 3.23 shows the graphs confirming this expectation. The total amount of assured bit rates allocated by UDP flows is about 1.250 Mbps. Therefore the very aggressive UDP flows alone lead to a congestion on the bottleneck link, leaving no bandwidth for the TCP connections (see column four on Table 3.2). The suppression of TCP by aggressive UDP flows is not a special problem of Assured Service but a general problem in the Internet. This is why there is a demand for mechanisms being able to detect and police aggressive flows.

3.4.2 No Congestion

So far the simulation showed the sharing of bandwidth during an extreme overload situation. As mentioned before the most important issue for the success of Assured Service is the proper dimensioning of the network. In the previous section simulations have been done with more than double the load the bottleneck link is capable to transmit. In this section the interaction of several flows using assured bandwidth values. The total of the reserved bandwidth must be below the networks capacities.

Figure 3.24 shows the respective graphs. Similar to the previous scenario, different assured bandwidth values have been allocated for the connections. The sum of assured bandwidth values is 675 kbps. Therefore, the bottleneck can forward all in profile packets. Table 3.3 shows the percentage of the assured bandwidth a flow was able to reach in the last 100 seconds of the simulation. For flows without assured bandwidth

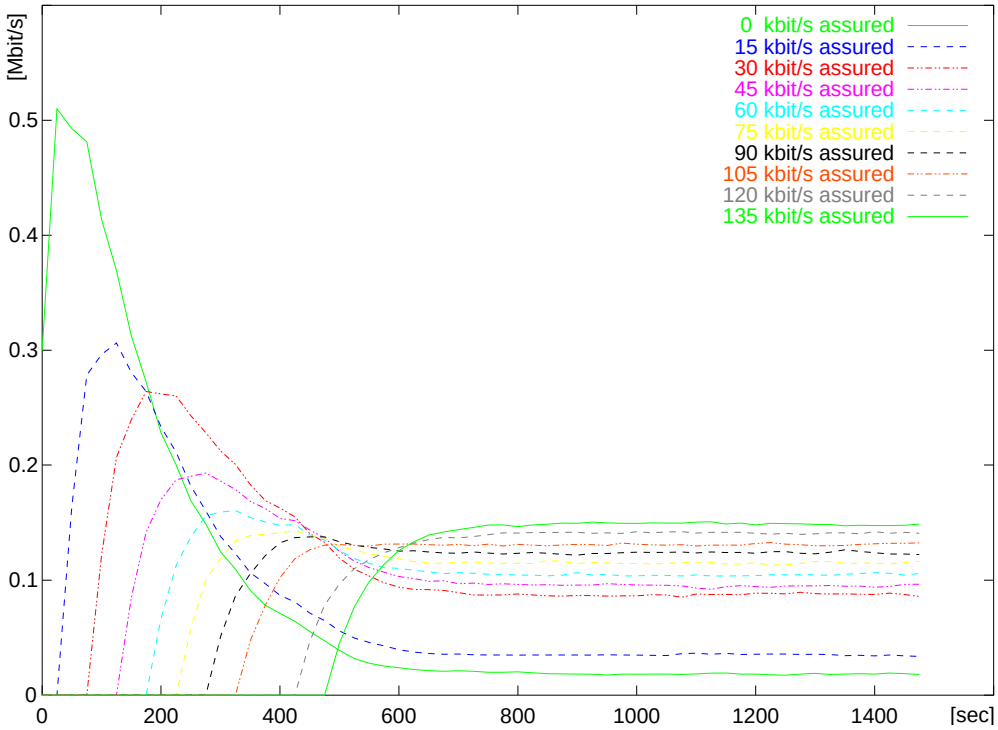


Figure 3.24: *Bandwidths of ten UDP flows causing no congestion*

the bit rate achieved is given.

In analogy to the results examining the behaviour under heavy congestion, flows with small assured bandwidth generally perform better. The constant bit rate of the UDP flow with only 15 kbps assured bandwidth is exceeded by 220 percent, whereas the flow with 135 kbps assured bandwidth gains not more than 113 percent (152 kbps).

The results of another experiment with ten TCP flows and low congestion is shown in Figure 3.25. Figure 3.26 depicts the interaction of mixed UDP and TCP traffic. The bit rates achieved in the last 100 seconds are listed in Table 3.3. In contrast to the situation with the sum of in profile traffic exceeding the capacity of the bottleneck link, every flow now gets at least the assured bandwidth. Of course, the very aggressive constant bit rate of UDP sources use nearly the whole bandwidth not allocated by assured traffic, but Assured Service is capable to protect the TCP flows in a way they can meet their profile.

3.4.3 Assured TCP Flows only

Concluding the evaluation of Assured Service with the ns simulator, the capability of Assured Service to protect TCP flows from aggressive UDP flows shall be evaluated. For this purpose we use a similar simulation scenario as before, but assign assured bit rates to the TCP flows only, while the UDP traffic is transported with best effort. In analogy to the former simulations each traffic source starts sending timely delayed. Figure 3.27 shows the results of the simulation, table 3.4 the bandwidth each flow

ass. bw	UDP	TCP	UDP+TCP
0	18 kbps	30 kbps	0 kbps †
15	220 %	379 % †	690 %
30	286 %	230 % †	100 % †
45	219 %	166 % †	273 %
60	175 %	146 % †	100 % †
75	154 %	152 % †	184 %
90	134 %	132 % †	99 % †
105	125 %	128 % †	149 %
120	117 %	125 % †	99 % †
135	113 %	129 % †	131 %

Table 3.3: *Five TCP, five UDP flows with different assured bandwidths, causing no congestion. The Table shows the percentage of assured bandwidth reached. The † marks the TCP flows.*

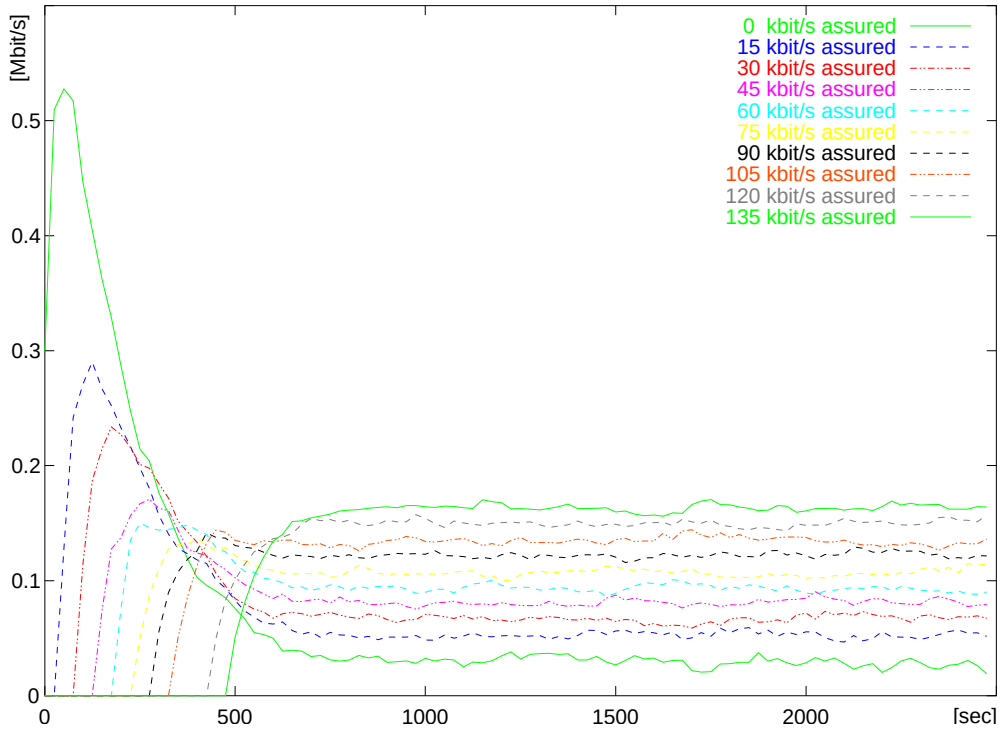


Figure 3.25: *Bandwidths of ten TCP flows, causing no congestion*

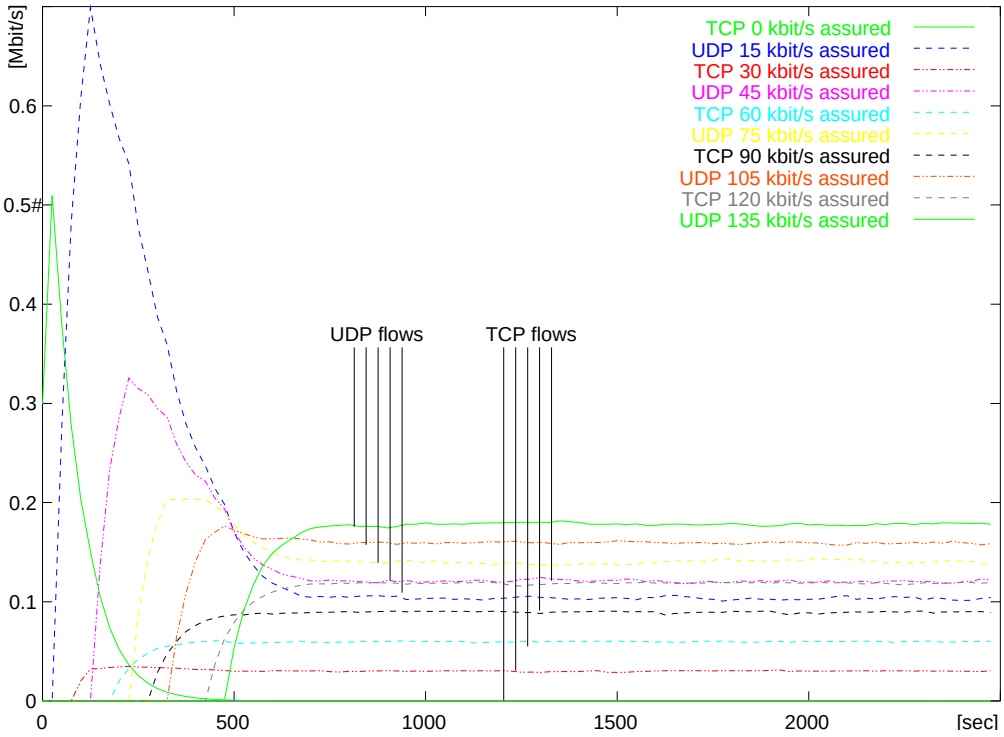


Figure 3.26: *Bandwidths of ten mixed UDP and TCP flows with different assured bandwidths*

achieved in the last 100 seconds of the simulation.

Every TCP flows is able to transmit data at least at the corresponding assured bandwidth, while the very aggressive constant bit rate UDP flows occupy the rest. As can be seen on Table 3.4 it is almost impossible for the TCP flows to transmit data with more than their assured bandwidth.

This shows, that Assured Service is surely able to guarantee at least the assured bandwidth of high priority TCP flows, while aggressive best effort traffic blocks each out of profile TCP transmission.

3.5 Conclusion

The simulations presented in this chapter showed, that Assured Forwarding is able to differentiate between different levels of Quality of Service. Especially it was proven that TCP flows can be protected against aggressive traffic.

The evaluation of queueing delays showed a tradeoff between the guaranteed bandwidth for in profile packets and their delay, because in profile packets are accepted even at large queue lengths and while out of profile packets are dropped at short queue lengths. This leads to high bandwidth and possible long delay for in profile packets and low bandwidth, but minimum delay for out of profile traffic. To provide more control over the delays an alternative queueing method was presented and evaluated,

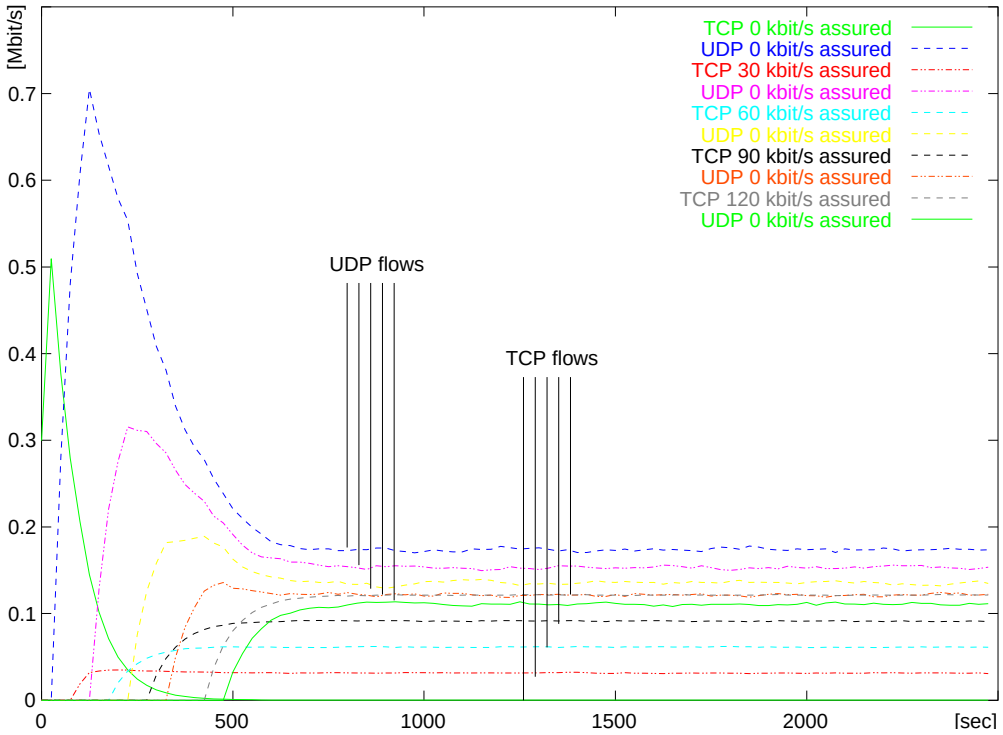


Figure 3.27: *Assigning assured bandwidth to TCP flows only*

ass. bw	Traffic Type	bw in kbps
0	TCP	0
0	UDP	172
30	TCP	30.9
0	UDP	152.6
60	TCP	61.1
0	UDP	136.6
90	TCP	91.0
0	UDP	121.9
120	TCP	121.3
0	UDP	112.1

Table 3.4: *Five TCP, five UDP sources. The TCP sources got assured bandwidth. The Table shows the reached bit rate.*

supporting short delay and high bandwidth for in profile traffic.

The initial evaluation of different parameter sets for the RED based RIO queues led to no clear result. In contrast to the recent proposals, the experiments used two dropping probabilities only, therefore a third drop precedence can be expected to make things even more complicated. As Christiansen e.a. showed in [CJODS00] the tuning of RED parameters is hard enough for TCP only flows and the additional negative impact of aggressive protocols like UDP is obvious.

The results of the experiments regarding throughput of Assured Forwarding confirm other evaluations for lower bandwidth values as performed by Ibanez [IN98] or by Yeom [YR98]. These results are even more important, since TCP acts differently for different bandwidth values. Especially with the increasing network capacity that property has a high significance.

On the other hand ns is a simulation package and does not necessarily meet the reality, as it tends to have a very idealistic view of networks. A good example for the limitations of such a toolkit are the constant bit rate senders used during the evaluations. With a simulation package working on a mathematical basis the implementation of a constant bit rate sender is simple and packets will be perfectly equidistant. Since computers in reality usually have to deal with processes, interrupts etc. a perfect timing is nearly impossible. Therefore, traffic generated by real end systems is hard to model and will hardly conform statistical distributions as used by network simulators. Additionally the statistical background of ns is not as clean as it may be supposed to, as an evaluation of the ns random number generator revealed [HE02].

Another general disadvantage of simulators like ns is the fundamental difference between a simulation and a real network with single routers and end systems, each with own routing tables, traffic conditioning mechanisms. Of course ns is very powerful to simulate the traffic flow through large networks but differs significantly from real devices regarding configuration and management. For an evaluation scenario focusing not only on the measurement of performance parameters like bandwidth, delay or jitter, but also including management related tasks, a pure simulation scenario has its drawbacks.

Since real applications and end systems are exposed to several parameters not to be calculated like human intervention, an environment allowing the simple and transparent integration of real hard- and software is required.

Of course the abstract model of ns has also advantages. Huge topologies can be set up more easily using ns than really wiring routers and network devices. Also the generation of a ns topology by writing a single setup script is much more easy than configuring real devices. However, this does not count for small and medium size networks. Here a more ”realistic” environment may be preferred.

Chapter 4

Emulation of IP Networks

The set up of experimental networks is in general an important action during development of new concepts and their evaluation. As shown in the previous chapter simulations can be used to test new concepts or queueing mechanisms quite easily and have a reduced need for hardware resources. The experiments were done with the ns network simulator [ns], which has been extended with Differentiated Service components to mark and queue packets according to their drop precedence.

The strength and drawback of such simulators like ns or also Opnet [opn] in general is the use of a mathematical model to simulate a network, a node or a link between two nodes. Since there is no relationship between the real time and the time ns uses internally, ns can be used to run huge simulations with thousands of nodes and links. The simulation simply will take longer, but is nevertheless mathematically correct. The use of a theoretic model also includes a rather abstract view of a network consisting of nodes and links.

These properties are sufficient to simulate the traffic flow of thousands of nodes within a huge network. On the other hand it is rather inconvenient if the components of the testbed have to work analogous to real devices and an integration of an emulated network with real components is favoured. Such a combination of real devices like routers and end systems with an emulated topology has several advantages for the set up of testbeds and the development of new concepts.

Of course the idea to emulate network devices is not new. So Wang [WK99] proposes to apply an address mapping scheme and to forward packets repeatedly through the kernel of the same host. Even if this is rather complicated and does not provide a real emulation of an Internet router, it is especially capable to emulate the multiple impact of a specific host on forwarded traffic. Another approach is to add special kernel components applying special characteristics of a WAN router to a single PC [nis]. This allows to emulate a set of WAN routers within a laboratory LAN.

- Real end systems allow the use of real applications with the network emulation. Using a simulator these applications have to be redesigned which usually only is possible if a proper description of the internals of an application are available. A network emulator connected to a real network can be used with any type of application without any changes.

- New traffic conditioning components can be implemented and evaluated in a convenient emulation environment. The writing of new kernel code and especially its evaluation is usually rather complicated and time consuming. Nevertheless the new components can be evaluated directly with real applications.
- The network emulation can be used to test applications for real networks. Before setting up a set of IP routers or other network devices, similar scenarios may be set up using emulated devices. In contrast to a simulation the emulation can not only be used for testing the set up with simulated sources but with the real end systems.
- The front-end of the emulator can be like the one of a real network device. This might be a drawback during the set up of large topologies yet it is a big advantage if a typical test network with a few dozens routers has to be set up. The use of a well known front-end also simplifies the handling of Virtual Routers for new users and in environments with virtual and real devices.

Some of these tasks can also be accomplished with the ns network simulator. The simulator can be connected to real networks and real packets are forwarded through the simulator. On the one hand this allows the use of real end systems, but limits also the size of the simulated network, as real traffic has to be processed in real time. Also the simulator is still some kind of monolithic block, with a complete different user front-end than an end system. Since the whole simulation is performed by a single program, the extensibility is limited, since single network devices can not be shut down.

Another disadvantage of the ns concept are the differences between the configuration of the simulator and real devices. For a complete emulation of an Internet Router it is not sufficient to apply similar packet treatments, but also to provide similar front-ends and configuration concepts.

4.1 Virtual Router Architecture

The basic idea for combining real hardware with an emulated topology is to use single small entities, each emulating an IP router and to replace the links normally used to connect routers by communication channels between these entities. Since each of these entities provides functionalities similar to an IP router but is only software, the term Virtual Router (VR) is used [BB00a], [BB00b].

To illustrate the interconnection of several Virtual Routers Figure 4.1 shows a set of Virtual Routers running on different computers. The type of the communication channels between the Virtual Routers can be different, therefore the Virtual Routers can either run on a common computer using the Unix inter process communication facilities as channels or can be distributed over multiple computers, using channels based on UDP tunnels.

Since VRs are to behave like real routers, they have to process traffic in real time. They have to receive packets, process them and forward them – anything a normal

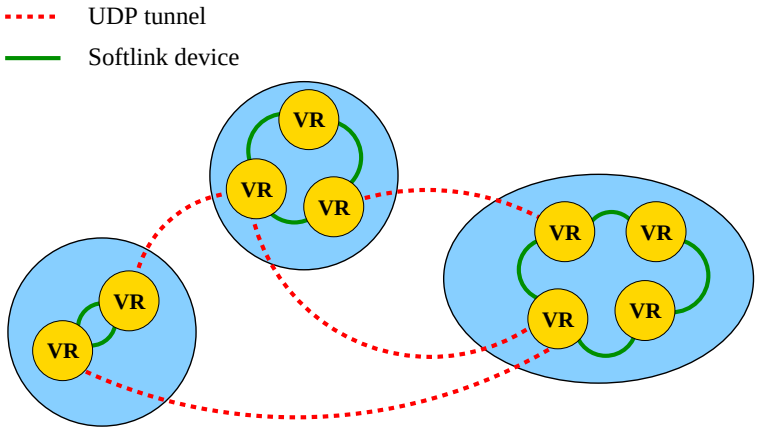


Figure 4.1: *Distributing Virtual Routers to hosts with connections by Unix IPC and UDP tunnels*

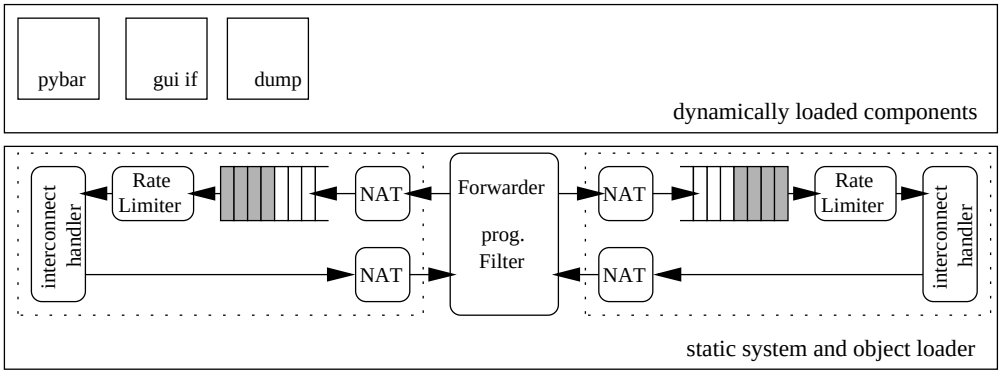


Figure 4.2: *The Architecture of a Virtual Router with two interfaces and several loaded components*

router does. This has – in contrast to network simulators – to be done in real time, limiting the number of Virtual Routers per computer due to its CPU power.

The Virtual Routers with their Unix IPC or UDP tunnel based interconnections span up a network. To connect this network to a real network, an additional special connection type exists allowing a Virtual Router to connect to a computers network layer. It therefore is possible to route traffic from a computer to and through an emulated topology and emulate multiple sources, sinks and routers on a single computer.

Each Virtual Router runs as an independent program, not interfering with other Virtual Routers and only exchanging packets by communication channels. Figure 4.2 shows the architecture of a Virtual Router. The figure shows also the capability of a Virtual Router to be extended by loadable modules like a Active Networking environment (pybar), a graphical user interface or a traffic monitor (dump). The router consists of a program providing the main functionalities and a mechanism to load further extensions dynamically during runtime to the Virtual Router. The single components will be described more in detail in the following sections.

4.1.1 Interfaces

The main work regarding packet processing is assigned to the interface components of the VR. The Virtual Router of Figure 4.2 has two of these interfaces. There are several components within each interface to receive, transmit, queue a packet or to provide some additional mechanisms like address translation.

Each interface has to receive or send packets. Usually, this is done over the attached communication channels.

connections to a softlink device: This type of connection is used to enable the communication between a Virtual Router and a real network. The interface is connected to a softlink device, provided by a Linux kernel module as described in Section 4.1.9. This allows the Virtual Router to exchange packets with a real network

connections between VRs via FIFO pipes: This connection type is used to connect two VRs running on the same computer. It allows to forward packets from one VR to another and is simply based on Unix pipes between two processes. Since Unix pipes are simplex, two pipes are used for each connection.

connections between VRs via UDP tunnels: Tunnelling based on UDP is used to exchange packets between VRs on different computers. IP packets to be forwarded to another VR are encapsulated within an UDP packet and sent to a specific port of the remote host. The opposite Virtual Router has to be configured to listen to this UDP port and read packets from there. Like any other connection it must be duplex, since the VR's interfaces on both sides of the connection have to receive and transmit UDP packets.

logical interfaces (IPIP tunnels): This mode is not no a connection like the others but is used to establish IP over IP tunnels. The interface does not send the packet over a communication facility, but encapsulates it into another IP packet and send the new packet back to the Virtual Routers for new processing. Section 4.1.5 will describe this mechanism in more detail.

Received data is processed by an IP network address translation unit (NAT). This allows to force the routing of packets through an emulated topology by modifying the destination/source address pair within a VR. Therefore it is possible to set up large networks on a single computer. The address translation mechanism is powerful but complex. A more detailed description of the address translation feature will be given in Section 4.2.2.

Data for transmission is first processed by a NAT component also and then put to a queueing system. The queueing system is rather flexible supporting Differentiated Services and other traffic conditioning methods. It will be explained separately in the next section.

The last two components within a VRs interface are the rate limiter and the interconnection handler. The interface speed of a Virtual Router is not limited like the interface of a normal router. Therefore the outgoing bandwidth of an interface has to be limited.

This is done by a token bucket filter. This filter can be configured during run time. This allos to change the speed of each interface during operation. After passing the rate limiter, the packets are sent to another Virtual Router or to a softlink device. To provide an unique mechanism interconnection handler, all functions regarding packet transport are combined within the interconnection handler.

The number and type of interfaces per Virtual Router are not predefined. Once a Virtual Router has been started, there are no interfaces. The interfaces first have to be generated, configured and connected. This can either be done by the command line interface or by an automatically executed startup script as described in Section 4.2.

4.1.2 The Queueing System

Because of its flexibility the queueing system is one of the most complex parts of the interface. It consists of a set of components like queues, filters, shapers and schedulers, that can be combined and configured during runtime. Currently the following components are offered:

- a generic classifier
- a token bucket filter
- a drop tail (FIFO) queue
- a random early detection queue (RED) [FJ93]
- a Weighted Round Robin (WRR) scheduler
- a simple Round Robin scheduler
- a Priority Round Robin (PRR) scheduler
- a TRIO queue [HBWW99a]
- a Differentiated Services marker
- a Priority Weighted Round Robin (PWRR) scheduler

Each Virtual Router interface has an own queueing system attached. It stores and processes packets put to the interface by the Virtual Router. Every time the interface is able to transmit, a packet is removed from the queue and sent over the interface connection.

The queueing system implemented within the Virtual Router consists of several small basic components as listed above. These components can be linked together to set up complicated systems favouring certain kinds of packets or limiting the bandwidth of others.

As can be seen in Figure 4.3, each queueing component has a number of input links and a number of output links. The number of input and output links depend on the type of the component. While a FIFO queue will have one input and one output link, a classifier will have one input and multiple output links. For the set up of a queueing system the following steps are required:

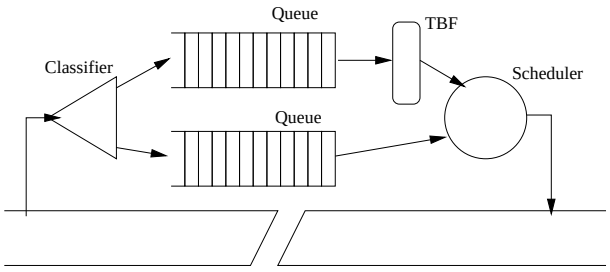


Figure 4.3: *Several small queueing components set up more complicated queueing systems*

creation of components: Since a queueing system will contain multiple queues or token bucket filters, a component first has to be instantiated. Each instance gets an unique id, that has to be used as reference to this instance during later operations.

linkage of instances: The instances (referenced by their id numbers) can be linked together. The number of links allowed for a component depends on its type.

component configuration: Each instance of a queue, a scheduler or any other component can be configured separately. Of course it is possible to define some reasonable global default values like FIFO queue lengths. But since the behaviour of the queueing system will depend on a proper set up of each single component, bandwidth values and bucket sizes for the token bucket filters can of course be set separately as well as the weights or priorities of PRR/WRR/PWRR schedulers.

Therefore, complex tasks may be achieved by an appropriate configuration and connection of components. In which way queueing components can be combined to provide a certain traffic conditioning will be demonstrated in Section 4.2.3 by a set up for a queueing system to protect TCP flows against UDP traffic.

Implementation of the Queueing System

The basis to link the queueing components is the *root component* as shown in Figure 4.4. A packet forwarded to the interface is sent to the root component. The root component will forward the packet to the next component.

The root component is also accessed if the interface can send data and has to extract packets from the queueing system. Therefore, each packet leaving the Virtual Router has to pass the root component twice. First when sent to the interface and a second time, when extracted from the queueing system to be transmitted.

The *droptail/FIFO queue* is a simple queue for buffering packets up to a certain amount. If the queue is full, new packets are discarded. The only parameter is the number of packets the queue can hold.

To merge packets coming from different components a scheduler is used. There are different scheduler algorithms available to be able to achieve different behaviours. All different scheduler modes are integrated into a *generic scheduler*, which can be switched to either

- Weighted Round Robin

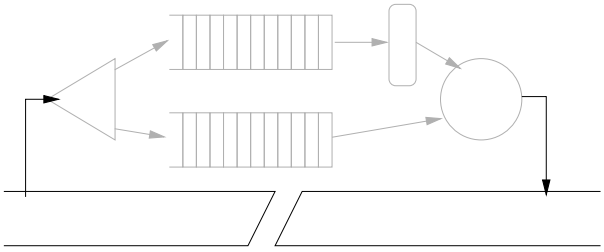


Figure 4.4: *The root component is used as first and last component within a queueing system*

- Round Robin
- Absolute Priority Round Robin
- Priority Weighted Round Robin

The different scheduler modes have been described in the traffic conditioning Section 2.4. The *classifier* is used to distinguish between packets and forward them according to a set of rules to other components. The classifier might work in MF of BA mode, checking either various fields of the IP header or only the DSCP value, depending on the rules set up. The rules consist of a set of the following specifiers:

- source address (specified by IP address and netmask)
- destination address (specified by IP address and netmask)
- Differentiated Services Code Point
- maximum packet size
- protocol type

Of course for a pure BA classifier only rules regarding the DSCP would be applied, while the other fields are interesting for a multi field classification only. Even without Differentiated Services a queueing system therefore might assure a certain share of bandwidth to TCP flows in order to protect them against aggressive protocols.

To reduce the throughput of a component to a certain maximum a *token bucket filter* can be applied. Therefore a combination of a FIFO queue and a token bucket filter can be used to shape traffic. Since the token bucket filter only acts as limiter, a queue is needed to drop the packets. Therefore a strict policer will contain a queue with a minimal queue length and a token bucket filter, to limit the rate the queue can be emptied with.

The implementation of Differentiated Services requires two additional components: a *Differentiated Service marker* and a queue capable to handle multiple drop precedences as required for Assured Forwarding. The Differentiated Service marker implemented within a Virtual Router allows to specify a set of rules. These rules include a pattern for the packets to be marked and a traffic profile. The pattern includes similar parameters like the multi field classifier.

- source address range (specified by address and netmask)

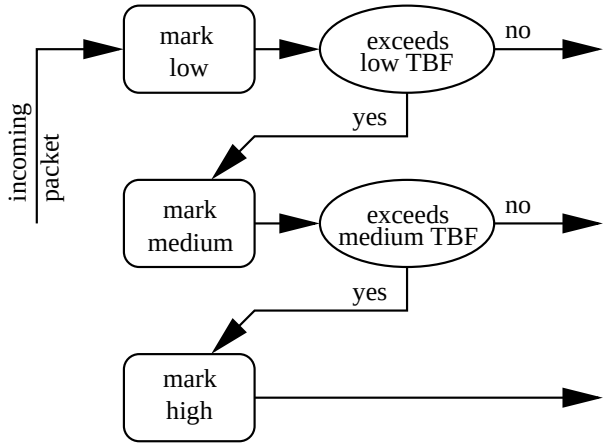


Figure 4.5: *VR implementation of a two rate three colour marker. A Packet is marked with a higher drop precedence if the packet length exceeds the number of tokens in the according bucket.*

- destination address range (specified by address and netmask)
- Differentiated Services Code Point
- protocol type

The profile determines the type of service and additionally required parameters. For Expedited Forwarding this is simply a bucket rate and a bucket size. Packets matching a pattern are marked with the EF DSCP up to the specified bandwidth. The bucket size allows to add a certain acceptance for bursts.

Within an AF class packets have to be marked for different drop precedences. Therefore a two rate single three colour marker as suggested by Heinanen (see Section 2.3.5) has been implemented as shown in Figure 4.5. Of course Assured Forwarding requires a specification about which Assured Forwarding class is to be used.

The second component implemented especially for DiffServ, is the TRIO queue, required for Assured Forwarding. The TRIO queue has to differentiate between packets according to their DSCP value and drops Assured Forwarding packets with a high drop precedence earlier than those with a low one. As shown in Section 2.4.6 the common algorithm for such a queue is based on RED. Since the benefits of RED to multi protocol flows are questionable as shown by ns simulations (see Section 3.2), the VR TRIO queue implementation supports different algorithms.

boolean This is the droptail or FIFO mode. For each drop precedence a threshold is defined. If a packet arrives and the actual queue length l exceeds the according threshold the packet is discarded. Obviously

$$th_{low} > th_{medium} > th_{high}$$

has to be true for Assured Forwarding to work properly. The probability p for the different drop precedences are:

$$
\begin{aligned}
p_{low} &= \begin{cases} 0 & \text{for } l \leq th_{low} \\ 1 & \text{else} \end{cases} \\
p_{medium} &= \begin{cases} 0 & \text{for } l \leq th_{medium} \\ 1 & \text{else} \end{cases} \\
p_{high} &= \begin{cases} 0 & \text{for } l \leq th_{hig} \\ 1 & \text{else} \end{cases}
\end{aligned}
$$

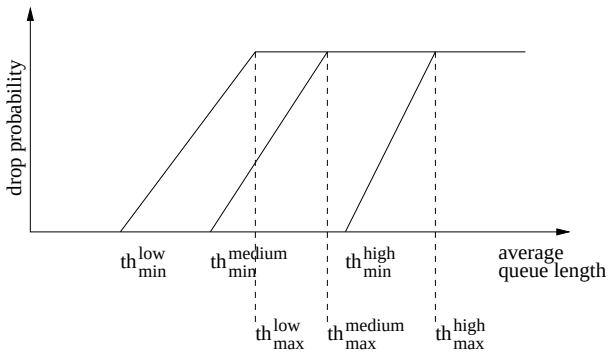
linear A pair of queue lengths (th_{min}, th_{max}) is defined for each drop precedence. Between these two queue lengths the dropping probability is increased linearly. Instead of using the actual queue length l an averaged queue length avg is calculated:

$$avg = (1 - w_q)avg + l \cdot w_q$$

The value of w_q defines how fast the queue react to bursts.

$$
\begin{aligned}
p_{low} &= \begin{cases} 0 & \text{for } avg < th_{min}^{low} \\ \frac{avg - th_{min}^{low}}{th_{max}^{low} - th_{min}^{low}} & \text{for } th_{min}^{low} \leq avg < th_{max}^{low} \\ 1 & \text{for } avg \geq th_{max}^{low} \end{cases} \\
p_{medium} &= \begin{cases} 0 & \text{for } avg < th_{min}^{medium} \\ \frac{avgl - th_{min}^{medium}}{th_{max}^{medium} - th_{min}^{medium}} & \text{for } th_{min}^{medium} \leq avg < th_{max}^{medium} \\ 1 & \text{for } avg \geq th_{max}^{medium} \end{cases} \\
p_{high} &= \begin{cases} 0 & \text{for } avg < th_{min}^{high} \\ \frac{avg - th_{min}^{high}}{th_{max}^{high} - th_{min}^{high}} & \text{for } th_{min}^{high} \leq avg < th_{max}^{high} \\ 1 & \text{for } avg \geq th_{max}^{high} \end{cases}
\end{aligned}
$$

The following figure illustrates the dropping probabilities for the different drop precedences:



As the graph shows, this mode is some kind of simplified RED. In Section 3.2 was shown, that the choice of good RED parameters is complex and the benefits of the RED algorithm in a scenario of mixed protocols is questionable. Due to the lower number of parameters this algorithm can be adapted more easily.

RED This mode provides a TRIO queue based on the original RED algorithm. The dropping probability is calculated using an exponentially weighted moving average algorithm (EWMA) as described in Section 2.4.6. This TRIO queue has to be configured with a set of ten parameters. The probability p to drop a packet of a certain precedence d between the two thresholds th^d_{min} and th^d_{max} is:

$$p_d = \frac{max_p^d \frac{avg - th^d_{min}}{th^d_{max} - th^d_{min}}}{(1 - count^d) \cdot max_p^d \frac{avg - th^d_{min}}{th^d_{max} - th^d_{min}}}$$

Obviously the impact of the different parameters is not as obvious as in the simplified RED version.

The queueing system is designed to provide flexible but powerful mechanisms to handle packets. The implementation allows the configuration, generation or connection of single components without restarting the VR or even pausing the interface. Also, an extension by additional components during runtime is possible.

Since the Virtual Router is usually connected with real network devices and hosts, the possibility to perform even massive changes to components is an absolute must. It is quite awkward and not handy switch off a router or pause it, while it is connected to a network and has to transmit data.

4.1.3 Packet Forwarding and Routing

Functions to receive and transmit packets are provided by the interfaces. Additionally, a central mechanism is needed to handle the received packets, routing them to an interface or pass them to some local function, if the destination address matches the VR.

These functions are provided by the central forwarding mechanism. Once a packet has been received by an interface, it is passed to the central forwarder. The central forwarder first checks whether the packet has to be forwarded further according to some routing rules or has to be processed locally by a protocol stack. In the latter case, packet fragments are reassembled and forwarded to the protocol handlers. Currently the VR implements two protocol stacks:

ICMP The implemented ICMP stack is rather simple and mainly provides functionalities to be able to handle services like the `ping` command. Additionally it provides a set of functions allowing the sending of the appropriate ICMP messages due to events like

- packet discarded due to exceeded TTL field.

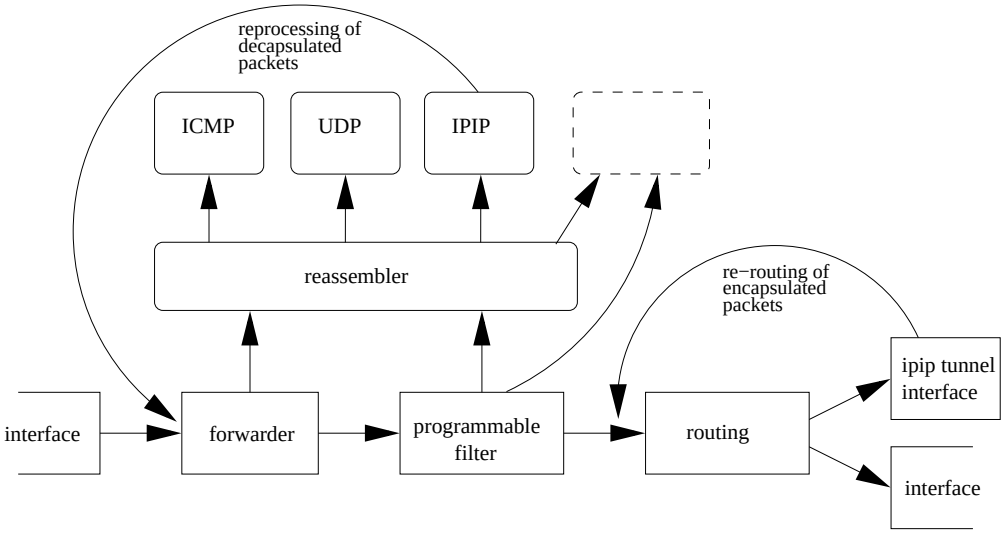


Figure 4.6: *Packet processing by the forwarder and the routing mechanisms.*

- port not reachable
- ICMP echo requests (`ping` command)

Beneath the compatibility to IP routers, these messages are also important as the `traceroute` utility, which allows to trace a packet's path through a network, relies on them. Later this utility will later be used to illustrate the set up of a network.

UDP For the implementation of routing protocols or other daemons the UDP protocol was required. Since a potentially necessary reassembling is handled by the forwarder itself, the UDP stack is simple and provides mainly functions for upper level applications to send UDP packets over a certain port or to listen to an UDP port.

The ICMP and UDP stacks allow a Virtual Router to receive, to process and to send ICMP and UDP packets. Since no TCP stack is available, it is not possible to set up TCP connections between Virtual Routers. Of course, this does not affect the transport of TCP packets through a Virtual Router. Therefore, connections between TCP capable end systems can be established over a set of Virtual Routers.

Figure 4.6 shows modules a packet has to pass while being processed by a Virtual Router. Packets to be forwarded are sent through the programmable filter and routed to an interface afterwards. The figure also shows a special kind of interface, which can be used to set up IPIP tunnels based on the "IP Encapsulation within IP" standard [Per96]. The programmable filter and the IPIP tunnel mechanism will be described in the following sections.

The routing algorithm used to decide over which interface a packet has to be transmitted works similar to any other routing mechanism. A central table stores a set of rules, which is searched for the best match. To allow more flexible routing decisions to

be made, the table does not only contain the destination address/netmask pattern, but works on multiple fields of the IP header.

- source address / netmask (e.g. 130.92.64.0 and 255.255.255.0)
- destination address / netmask
- protocol type (TCP, UDP, ...)
- DSCP value (EF, AF1, ...,AF4)

Since routing rules can be based on DSCP values, different services can be routed on different paths throughout the network. Setting up such a rule is dangerous because it has to be ensured that different drop precedences of the same service are routed in the same manner.

Like any other component of a Virtual Router, the routing table of the VR can be modified during operation. The shell like front-end presented in Section 4.1.6 provides Unix style commands for the set up and the removal of routes.

4.1.4 Programmable Filter

Since the Virtual Router supports an easy implementation of new mechanisms, flexible modules to process packets passing the program have to be provided. The Programmable Filter accomplishes this task. An application running on the VR can set up a filter and this filter forwards any packet matching the filter rule to the application. This simplifies the implementation of daemons significantly and can also be used to trigger simply a certain action, if a matching packet passes the router. A filter rule is described by a filter specification:

$$PF_{spec} = \begin{cases} IP_{dest}, Netmask_{dest} \\ IP_{source}, Netmask_{source} \\ Opt, Opt_{val} \\ protocol \\ DSCP \\ \text{filter-mode} \end{cases}$$

In addition to the usual destination and source address specifications IP options can be used to set up a filter. Therefore, packets marked with the Router Alert option as specified by RFC 2113 [Kat97] can be passed to specialised mechanisms. The concept of a central programmable filter has the advantage, that an application has to provide the filter pattern only once rather than processing all incoming data. This simplifies the set-up of applications as well as speeds up processing of normal traffic being forwarded by the VR.

There are two main filter modes: copy and move. In the copy mode a packet is forwarded normally and a copy of it is passed to the application. In the move mode the packet is not duplicated but only passed to the application. This way, an application

can process a packet and either re-inject it afterwards or completely discard it. An application can override queueing systems and forwarding mechanisms of the VR. For example, a whole queueing system may be applied for a certain type of packet, without affecting the normal processing of standard IP packets.

It can be seen in Figure 4.6 that the programmable filter can send matching packets directly to the application or send them first to the reassembler. Therefore, even if the packets pass the VR as fragments the application does not have to take this into account. In addition to forward (reassembled) packets to applications according to the set up of filters, the programmable filter can be configured to process these packets, collecting statistical information over certain packets only.

4.1.5 IPIP tunnels

In the description of the Virtual Router's interface UDP tunnels, connecting VRs running on different computers were mentioned. An IP packet that has to be transmitted over a "virtual network cable" to the next Virtual Router is encapsulated into an UDP packet and sent to the computer the other Virtual Router is running on.

There is also another type of tunnel used in the Internet. For network management reasons it is sometimes profitable to encapsulate an IP packet within another IP packet, send the new packet to its destination and decapsulate the original packet there. These tunnels have the advantage, that the "envelope" packet has the source and destination address of the start and end point of the tunnel, independently of the packets to be transported. This simplifies the treatment of packets within an interior router and supports the set up of distributed intra nets or Virtual Private Networks.

These tunnels are not a speciality of Virtual Routers. One solution to implement these tunnels is the rather simple IPIP tunnel as defined in RFC 2003 [Per96], encapsulating packets simply by adding a new IP header to the whole packet.

Virtual Routers can provide this type of tunnels as well. An IPIP tunnel start point is realized by switching an interface into IPIP tunnel mode. As it can be seen in Figure 4.6 each packet routed to that IPIP tunnel interface is encapsulated into an IP packet and the new IP packet is routed again. The new IP packet gets the IPIP tunnel interface IP address as source address. The destination address of the "envelope" packet has to be specified and determines the tunnel end point.

The routing table controls what packets are put to the tunnel interface and into the tunnel. The end point of IPIP tunnels is handled automatically. If an IPIP tunnel packet arrives at a Virtual Router it is decapsulated automatically.

To illustrate the UDP tunnels used to connect Virtual Routers and IPIP tunnels, Figure 4.7 shows five Virtual Routers connected by UDP links and a set-up IPIP tunnel between VR 2 and VR 4. The following list shows the contents of the datagram exchanged between the virtual routers and within the VR 3.

1. $IP_{udp}(VR1 \rightarrow VR2)[IP(VR1 \rightarrow VR5)[...]]$
2. $IP_{udp}(VR2 \rightarrow VR3)[IP(VR2 \rightarrow VR4)[IP(VR1 \rightarrow VR5)[...]]]$

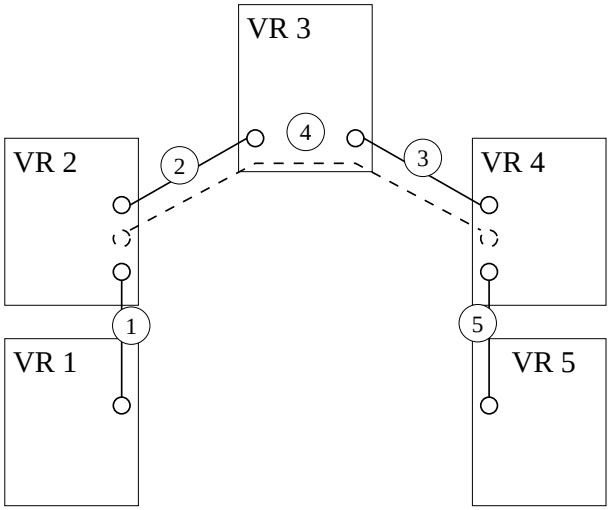


Figure 4.7: *Five Virtual Routers running on different computers, connected by UDP tunnels (solid), and an IPIP tunnel (dashed):*

3. $[IP(VR2 \rightarrow VR4)[IP(VR1 \rightarrow VR5)[...]]]$
4. $IP_{udp}(VR3 \rightarrow VR4)[IP(VR2 \rightarrow VR4)[IP(VR1 \rightarrow VR5)[...]]]$
5. $IP_{udp}(VR4 \rightarrow VR5)[IP(VR1 \rightarrow VR5)[...]]$

The transport of an IP packet through an IPIP tunnel causes a double encapsulation between two VRs. The IPIP packet is encapsulated again within UDP to be sent to the next VR. Even if this mechanism looks rather complex, it is quite simple in reality.

The IPIP tunnel mechanism can also be applied between virtual and real routers. Therefore it is possible to have an IPIP tunnel start point on a Linux host sending encapsulated packets to a Virtual Router providing the tunnel endpoint.

4.1.6 Configuration of Virtual Routers

The Application Programming Interface

To keep the Virtual Routers reasonable small and flexible, there is a strict separation between the Virtual Router core mechanisms and any kind of front-end. Any configuration of the VR has to be done by an Application Programming Interface (API). To support multiple front ends like a graphical user interface, a command line interface or an agent platform to interact with the Virtual router simultaneously, each API is represented by a bidirectional communication channel.

To configure the router a command structure has to be generated and sent over this channel. The router will send back an error code or the requested information. Appendix A lists the binary configuration commands and results.

The use of API channels allows to attach multiple front-ends as well as an application to configure the router. This way a graphical user interface might be attached by an API channel, another can be used by the command line front-end while an agent platform might use an own channel to interact with the router.

```
MicroVar Shell

>ifconfig add if0 10.42.10.1
name:                       if0
ip-address:          10.42.10.1
netmask:          255.255.255.0
broadcast:         10.42.10.255
bandwidth(bps):         1000000   bucket size(bytes):          2048
drops:                        0   errors:                         0
rx:                           0   tx:                             0
rx-t-ip               172.0.0.0   rx-t-nm                255.0.0.0
rx-t-val               10.0.0.0   rx-t-pat               255.0.0.0
tx-t-ip                10.0.0.0   tx-t-nm                255.0.0.0
tx-t-pat              255.0.0.0
connection                 none
```

Table 4.1: *Virtual Router console with the ifconfig command to set up an interface and the returned message containing interface informations*

Currently, there are two applications using these channels: the command line front-end and the active router environment which will be described later in detail. The use of communication channels has an additional advantage: The program controlling a Virtual Router has not necessarily to run on the same host as the VR itself. Therefore, remote administration is supported as well as a tool to control multiple Virtual Routers.

The Front-End

The API channels are used to exchange binary messages configuring the VRs or to query information. To provide a human readable interface, a command line driven front-end (shell) is provided. This front-end more or less does a simple translation between the human readable commands and the binary API messages. Table 4.1 shows the console output directly after starting the VR and adding an interface. In addition to this interactive mode, the shell will automatically read and execute commands from a certain file directly, once the VR is started. After this startup file has been red, the shell switches to interactive mode. These startup files have a simple syntax can therefore be generated automatically, the configurations for a large Virtual Router topology can be calculated resulting in a set of startup files. The network emulation can then be started by simply starting the Virtual Router programs with the according startup files.

As mentioned in the previous section the concept of API channels can also provide remote access to the router. Even the front-end shell can be configured to listen for incoming TCP connections. After the VR has been started and the startup script has been executed, the VR can be configured simply via the telnet command. Nearly all components of a VR can be configured, added or removed while the Virtual Router is up and running. This includes especially:

- routing tables: displaying, adding and deleting routes.

- interfaces: adding interfaces, changing link bandwidths, connecting interfaces to other VRs/softlink devices
- queueing systems: querying information, adding, modifying, removing, connecting and disconnecting components.
- general: loading new objects into the VR, querying statistical information

As far as possible the command line interface uses a Unix like command syntax. Therefore changes to the routing tables are done by a `route` command while interfaces are set up and modified by the `ifconfig` command. For a more detailed list of available commands see appendix B.

4.1.7 Event Scheduler

The core of a VR is the event scheduler (see Figure 4.1.7). Each component can register certain events to this central scheduler. This may be

timed events: The central event scheduler can trigger certain functions at specific times. It is also possible to register certain methods to be executed repeatedly.

IO events: The event scheduler monitors IO channels and reacts to their state changes. A certain function may be executed if a packet is received on a IO channel, or if a channel is ready for sending data.

Besides functions for registering and unregistering events, the scheduler also provides mechanisms to suspend events for a certain period of time.

Each event is associated with a certain method to be called. In idle state the scheduler monitors the events and when an event occurs, it executes the appropriate function. Therefore, an interface typically registers events handling its IO channels and provides a appropriate call back function to read or write packets to the IO channels. The same mechanism is used to realize the API channels or the command line interface. Events monitoring the appropriate IO channels are registered at the event scheduler.

An important property of the event scheduler is that events are processed sequentially. Once an event has been triggered and the appropriate function is executed, only this function is executed by the Virtual Router[1].

How important this property is can be seen, if the API received a command to reconfigure one of the interfaces. Since the function triggered by that API event is executed exclusively the API can not interfere with any other processes within the Virtual Router.

At a first glance this appears to reduce the performance and a parallel execution of events should be possible. On the other hand processing of a packet event is very

[1] As will be explained later, Loadable Modules can be loaded using an own thread running in parallel. But since the Loadable Objects have to communicate with the VR core via API channels or similar constructs, Loadable Objects run – from the core's point of view – also exclusively.

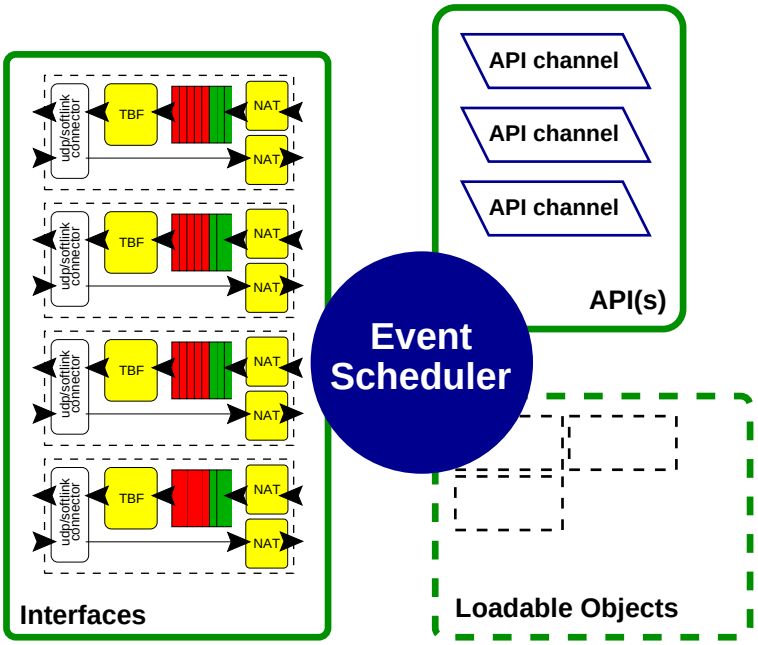


Figure 4.8: *The VR's central event scheduler*

fast and reconfigurations occur infrequently. Also even if the program would allow a parallel execution the operating system scheduler has to work analogous to the event scheduler. On computers with multiple CPUs running multiple Virtual Routers, a parallelisation is at least achieved for the whole emulation, since different Virtual Routers run on different CPUs.

Finally, even if performance is of course important an understandable and extensible design of the Virtual Router counts more.

4.1.8 Extending the Virtual Router

To allow the extension of the Virtual Router with some application level or core level functionalities, additional code can be loaded to the Virtual Router. This pieces of code are called Loadable Objects.

- Loadable Objects may be used as a simple transport mechanism to import new code into the VR core. Since some features of the VR require special functions to be called, an object containing these functions may be loaded. A filter may be set up forwarding specific packets to a dynamical loaded function. Such objects may also contain code to be called by the event scheduler.

- Loadable Objects can contain complete programs running in an own thread, independently from the VR and its event scheduler. As the event scheduler does not block these threads, any communication with the VR's core should be done via a so called API channel, providing the synchronisation between the threads.

Of course Loadable Objects are not restricted to short pieces of code, but can provide complete applications. Code for Virtual Router core callbacks might be provided, other code is executed in parallel and other parts of a Loadable Object are triggered by the Virtual Router's event scheduler.

After an object is no longer in use, it is automatically removed from memory. Three major Loadable Objects have been implemented:

dump is a small tcpdump like program, allowing to trace packets passing the VR. It was implemented due to debugging purposes and allows to specify a filter via the command line interface and print a short message for each packet passing the VR. The dump object simply adds a filter and a call back function to the VR core.

t-bone is more complicated than dump and was implemented to allow Quality of Service Measurements as proposed and evaluated in [Gün01]. The t-bone component listens to a specific TCP port to be connected by an external program. Once connected the t-bone can be configured by sending a filter description, causing the Virtual Router to forward a copy of each matching packet to the connected program.

pybar is the python based active router, which will be described later in Chapter 6.2. This loadable object implements a whole active networking environment, allowing packets to transport code. This code then can be executed by this module attached to the Virtual Router.

4.1.9 Connections to Real Networks: The Softlink Device

To connect the Virtual Router to the local computer's network system, a special network device has been implemented. This device is needed to send packets to a Virtual Router instead of forwarding them to a real network. The component is called Softlink device and was implemented as a kernel module for Linux (version 2.0.x, 2.1.x and 2.2.x).

The basic task of the kernel module is simple. It acts as an interface between user space programs like the Virtual Router and the operating system kernel, allowing user space programs to exchange network packets with the kernel like a network interface card (NIC). Once loaded into the Linux kernel the module adds a set of network devices (`sol0, ...`) to the operating system's list of network devices. For the kernel it does not make a difference whether the interface is a real NIC like `eth0` or a softlink device like `sol0`.

On the user's side the softlink module provides a set of device files `/dev/sol0, /dev/sol1, ...` as shown on Table 4.2. Each device file communicates with the according network device. If the kernel sends a packet to the network device `sol0` this packet can be read from the device file `/dev/sol0`. Accordingly, packets written to the device file are processed by the kernel as received by a NIC. The device files can simply be opened by user space programs using normal file system IO. The softlink

crw-rw- - - -	1	root	vr	63, 0	Apr 15 02:43	/dev/sol0
crw-rw- - - -	1	root	vr	63, 1	Apr 15 02:43	/dev/sol1
crw-rw- - - -	1	root	vr	63, 2	Apr 15 02:43	/dev/sol2
crw-rw- - - -	1	root	vr	63, 3	Apr 15 02:43	/dev/sol3
crw-rw- - - -	1	root	vr	63, 4	Apr 15 02:43	/dev/sol4
crw-rw- - - -	1	root	vr	63, 5	Apr 15 02:43	/dev/sol5
crw-rw- - - -	1	root	vr	63, 6	Apr 15 02:43	/dev/sol6

Table 4.2: *The softlink device files in the /dev directory. The usual Unix file permission system allows to control, which users can access the devices.*

device therefore might not only be used by Virtual Routers but is a generic tool, also supporting high level languages like Java or Python.

A softlink device has to be configured, as any NIC. On Linux this is usually done using the `ifconfig` command. Table 4.3 shows the list of network devices of a Linux system including a softlink device.

4.2 Setting up Virtual Router Networks

Using the different connection types allowing to establish connections between VRs in different manners and the softlink device to connect this topology to a computer's network system, various distributions are possible. Additional flexibility is provided by the address translation mechanism, which allows to set up topologies using only a single computer, acting both as traffic source and sink.

4.2.1 Distribution of Virtual Routers

The idea of a Virtual Router is to emulate a single router, not an end system. Real end systems are meant to be used as traffic sources and sinks.

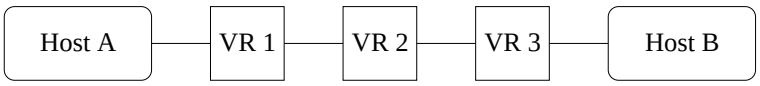


Figure 4.9: *Two end systems connected over three Virtual Routers*

The diagram above shows a typical simple set-up of two hosts (A,B) connected via three VRs (1,2,3), allowing to send traffic (UDP, TCP, ...) from host A to host B and vice versa. To establish a connection to a computer's network layer (see Section 4.1.1) a softlink device has to be used. This requires a VR running on the same computer the softlink device is installed. Nevertheless, several set-ups are possible.

Diagram 4.10 shows how Virtual Routers may be distributed. On each computer, which has an interface to be connected to a VR via softlink device, a VR has to be

```
eth0    Link encap:Ethernet  HWaddr 00:B0:D0:BF:82:98
        inet addr:130.92.66.130  Bcast:130.92.66.255 Mask:255.255.255.0
        UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
        RX packets:0 errors:0 dropped:0 overruns:0 frame:0
        TX packets:6 errors:0 dropped:0 overruns:0 carrier:6
        collisions:0 txqueuelen:100
        RX bytes:0 (0.0 b)  TX bytes:360 (360.0 b)
        Interrupt:10 Base address:0xfc00

sol0    Link encap:Ethernet  HWaddr 00:53:4F:46:54:4C
        inet addr:10.42.10.1  Bcast:10.42.10.255  Mask:255.255.255.0
        UP BROADCAST RUNNING NOARP MULTICAST  MTU:1500  Metric:1
        RX packets:513 errors:0 dropped:0 overruns:0 frame:0
        TX packets:665592 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:100

lo      Link encap:Local Loopback
        inet addr:127.0.0.1 Mask:255.0.0.0
        UP LOOPBACK RUNNING  MTU:16436  Metric:1
        RX packets:12 errors:0 dropped:0 overruns:0 frame:0
        TX packets:12 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:0
        RX bytes:968 (968.0 b)  TX bytes:968 (968.0 b)
```

Table 4.3: *Output of the Linux* `ifconfig` *command. It shows a listing of network devices, including one softlink device.*

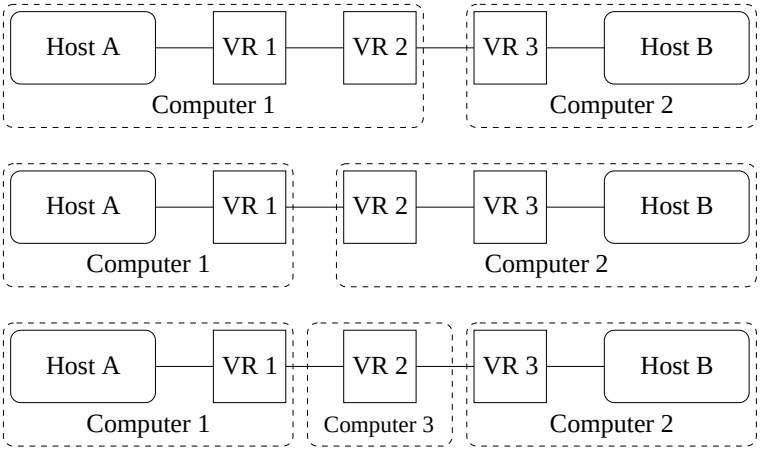


Figure 4.10: *Distributing VRs to end systems*

started. VR 2 might be placed either on computer 1 or 2 or on an additional computer 3 between.

A concrete set-up would depend on the available processing power of the computers and the bandwidth of the network the computers are connected with.

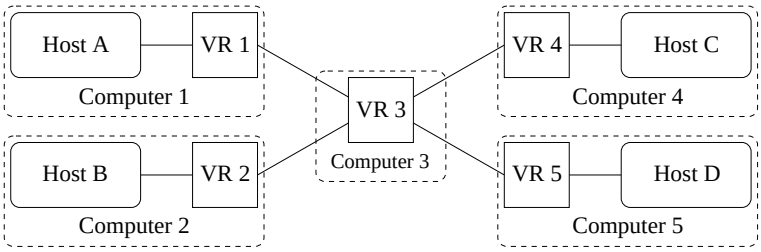


Figure 4.11: *Increasing the number of end systems*

The use of separate end systems for each traffic source or sink would cause a significant demand for computers in a topology of a reasonable size as can be seen in Figure 4.11. Fortunately, IP addresses are usually bound to the network interfaces (or logical interfaces) of that computer. As mentioned in Section 4.1.1, the softlink device is an emulation of a normal network interface and like other interfaces it owns an IP address. Since it is possible to create several softlink devices on a computer (up to 256), a computer may appear at different points in a topology as source and sink at the same time (see Figure 4.12).

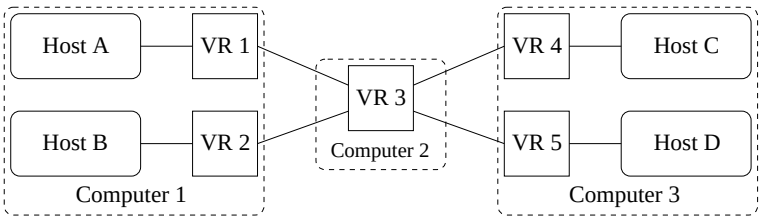


Figure 4.12: *using multiple softlink devices*

Unfortunately it is not simple to use a computer as source and sink simultaneously and forward traffic from the computer through an emulated topology back to itself. At least the Linux network layer will detect, that the destination address is one of the local (softlink) interfaces and will ignore the according entry in the routing table. Without changes to the network layer, there is no workaround for that behaviour. Fortunately, there is a way to trick the Linux router to forward the packet to the emulated topology using the address translation mechanism.

4.2.2 Address Translation

To use a single computer as multiple sources, sinks and also as location for one or a couple of Virtual Routers the address translation mechanism can be used [BB00b]. In Figure 4.13 this simple set-up is shown.

Packets on host A shall now be sent through the Virtual Router to host B. The same computer acts as two different hosts by setting up two softlink devices. The Virtual Router connecting the hosts runs on the same computer.

If a packet now is directly sent to the address IP_A of host A, the network layer of the computer will detect that IP_A is an interface of the same computer and process the packet internally, instead of sending it through the softlink devices and the Virtual Router.

To cope with this problem, an address translation within the softlink connection is performed. A packet, which is received by the Virtual Router over a softlink connection with the source, destination address pair (IP_{src}, IP_{dest}) is mapped to a packet $(IP_{src}, IP_{dest}^{mapped})$.

A packet, which is forwarded by the Virtual Router over a softlink connection to an end system (IP_{src}, IP_{dest}) is mapped to a packet $(IP_{src}^{mapped}, IP_{dest})$.

This mechanism allows to forwarding packets transparently through one or multiple VRs even, when the destination interface is placed on the same host as the source interface.

The following sequence shows the translations and the change to packet headers during the forwarding of the packet.

We use the network shown on the diagram above to demonstrate the mapping. A packet shall be sent from A_1 oder VR 1 to A_2. The packet is now not directly sent to IP_{dest} but to IP_{dest}^{mapped}.

So the computer sends a packet with the addresses

$$(IP_{src}, IP_{dest}^{mapped})$$

over interface A_1. Receiving this packet on a softlink connection, the VR translates the destination address to the correct destination address.

$$(IP_{src}, IP_{dest})$$

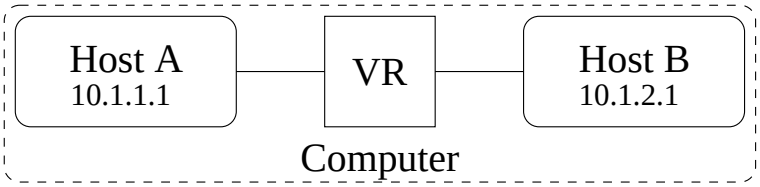


Figure 4.13: *A Virtual Router connected to two host interfaces of the same computer*

Assuming the routing tables within the VR are set up correctly, the VR will forward the packet to the VR-interface connected to A_2. Here the source address IP_{src} will be translated to IP_{src}^{mapped}.

$$(IP_{src}^{mapped}, IP_{dest})$$

The computer receiving this packet on host interface B can directly reply to the packet by using the IP_{src}^{mapped} as destination address. The same translation will take place in the other direction.

As the set-up of this address mapping scheme can be quite complicated, the VR maps in his default set-up any address fitting `172.0.0.0/8` to `10.0.0.0/8`, with only the first byte being replaced. So a packet sent to `172.1.19.22` and routed to a VR is there mapped to a `10.1.19.22`. The same is done in the opposite direction. This settings work fine, if the softlink interface of the hosts (A,B) have addresses of the `10.0.0.0` network.

To illustrate that mechanism a short example shall be given. As shown in Figure 4.13 Host A has the address `10.1.1.1` and host B the address `10.1.2.1`. Both hosts are realised by softlink interfaces of the same computer and are connected by a Virtual Router running on this computer. A packet shall be sent from host A to host B over the Virtual Router. If the packet would be sent directly to `10.1.2.1` it would not pass the VR but be processed within the computer.

Therefore, the packet is sent to address `172.1.2.1` instead of `10.1.2.1`. On the computer a route was set up sending packets for `172.1.2.*` to the Virtual Router. When the Virtual Router receives the packet, it translates `172.1.2.1` (the destination address) back to `10.1.2.1` and forwards the packet – according to its routing rules – to host B. Leaving the Virtual Router, the packet's source address is modified. The original source address `10.1.1.1` of host A is replaced by `172.1.1.1`. Being received by Host B, the packet is now addressed to `10.1.2.1` and originates from `172.1.1.1`. The translation of the source address allows a direct answer back to this address, automatically forcing a routing through the Virtual Router.

Even when this mechanism interacts smoothly with external network devices, it is complicated and mainly thought to set up small networks for development purposes. In such a scenario the capability to work on only a single computer is very important.

Since for larger topologies usually at least two computers are used, the address translation is not necessary.

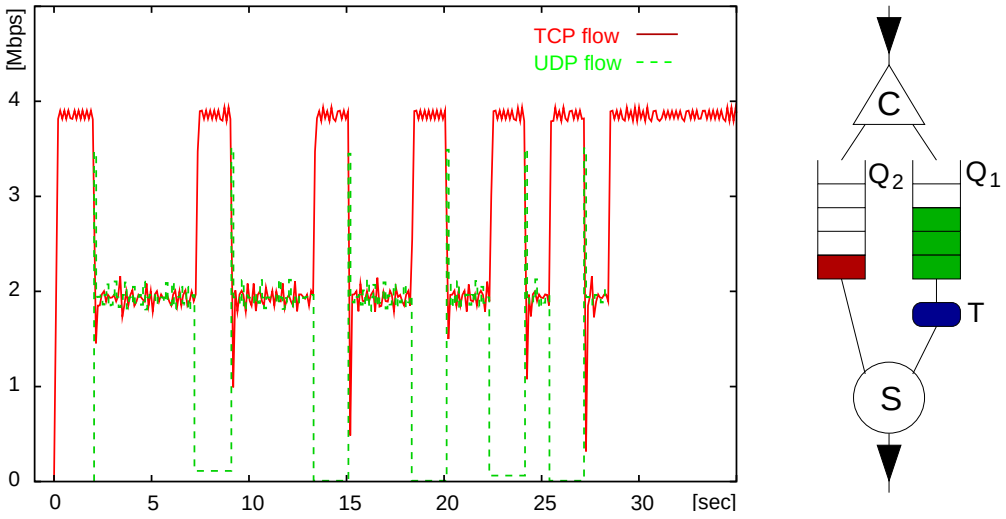


Figure 4.14: *Throughput Measurements: TCP flow protection through a VR by an appropriate queueing system.*

4.2.3 Setting up Queueing Systems

As described in Section 4.1 the queueing system consists of small components, each of them performing a well defined task. By plugging components together, various kinds of traffic handling can be achieved. In this section the concept shall be clarified by a small example. A queueing system shall be set up to differentiate between UDP and TCP flows and to prevent the suppression of congestion avoiding TCP flows by aggressive UDP traffic. TCP traffic should be guaranteed a minimum amount of bandwidth if there is aggressive UDP traffic and should also be able to use the full bandwidth if available.

The right diagram in Figure 4.14 shows the set-up. Incoming packets are processed by a classifier checking the packet's protocol id. TCP traffic is put to queue Q_2 while any other packets are put to queue Q_1. The token bucket filter T causes Q_1 to drop packets exceeding a certain packet rate. Finally a scheduler reads packets from Q_2 directly. Packets from Q_1 are read via the token bucket filter.

The scheduler is switched to absolute priority round robin mode, favouring the queue with the token bucket filter which is configured to a bucket rate of 2 Mbps.

The left graph in Figure 4.14 shows the achieved throughput values. The interface of the VR was limited to an overall bandwidth of 4 Mbps. The test starts with TCP traffic only, which is put to Q_2 and processed by the scheduler, achieving the full bandwidth of 4 Mbps. After a few seconds an UDP source starts sending. The UDP packets are put to queue Q_1 and would suppress TCP's bandwidth completely, if the token bucket filter would not limit the rate of Q_1 to 2 Mbps. TCP will then get a certain amount of bandwidth, even if the scheduler favours the queue Q_1, since it is limited by the token bucket filter. During the test the UDP source was switched on and off repeatedly, to visualise the impact of UDP on TCP.

Of course this is just an example to demonstrate the mechanism behind the queueing

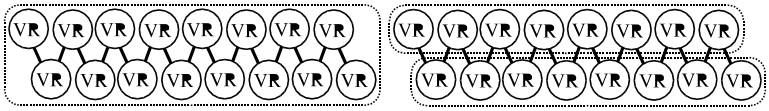


Figure 4.15: *16 VRs on one and distributed to two computers*

system. Obviously, there are simpler methods to achieve a similar or even better behaviour. Instead of the absolute priority round robin a simple round robin or a weighted round robin scheduler might be used, making the token bucket filter superfluous.

4.3 Traffic Measurements

Although the original idea of a Virtual Router was to offer a platform for the development and evaluation of distributed mechanisms like network management and Quality of Service routing, the architecture also offers a suitable testbed for traffic measurements. A first impression was given by the small example in the last section. Since the VR has to process packets in real time the number of routers emulated on a computer and the maximum of allowed bandwidth is of course limited, even if the performance may be increased significantly by multiprocessor computers.

To provide a basis for later measurements, the impact of distribution, the number of Virtual Routers and the load shall be evaluated.

4.3.1 Distribution and Packet Delay

The most problematic issue during network emulation in a distributed environment is the time a packet is delayed during forwarding and transmission [BB01]. Since the packet forwarding requires real processing and is not just simulated, the lower delay bound per hop is limited by the available processing power as well as by the available bandwidth and delay of the underlying network.

To measure the delay for an increasing number of routers a chain of 16 VRs was set up on one[2] and on two computers[3] (see Figure 4.15). To measure the round trip time (RTT) a number of pings have been sent to different routers in these router chains. These experiments have been performed on an unloaded network and with additional bursty UDP traffic of up to 50 % of the link bandwidth.

The graphs of the Figures 4.16 and 4.17 show RTTs of pings for an increasing number of hops for the local and the distributed set-up. Figure 4.16 shows the results without additional traffic in the network, Figure 4.17 the results if additional UDP traffic has been sent over the 16 Virtual Routers. All graphs show the expected linear correlation between the number of hops and the RTTs. The RTTs in the distributed emulation are much higher than in the single computer set-up. Since each VR has been connected to two VRs on the other computer, the packets suffer additional delay by the UDP

[2] dual processor 800 MHz PentiumIII, running Linux

[3] dual processor 800 MHz PentiumIII and a single processor 400 MHz PentiumII, both running Linux

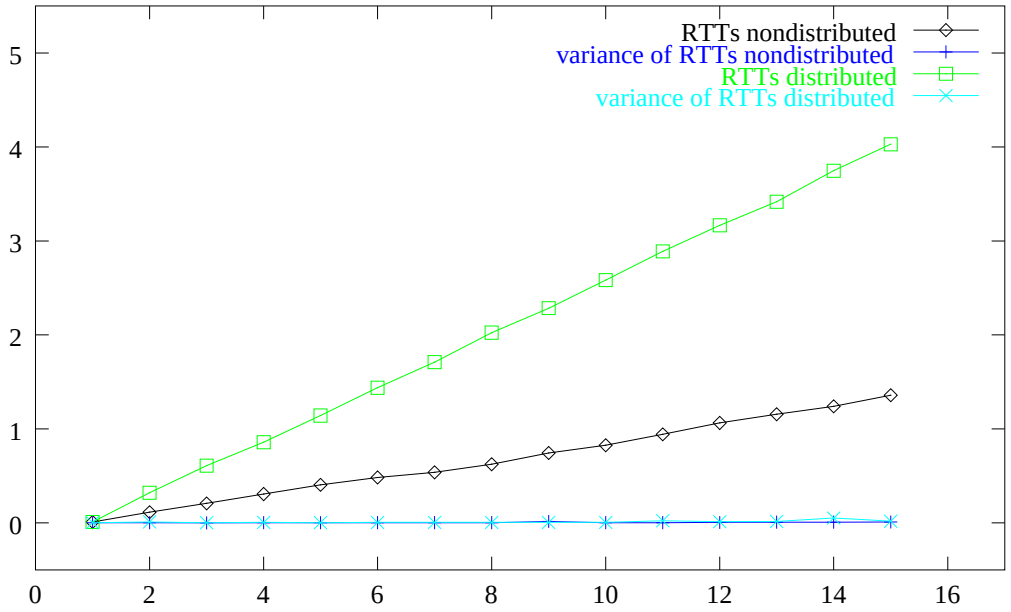


Figure 4.16: *RTTs in an unloaded VR network*

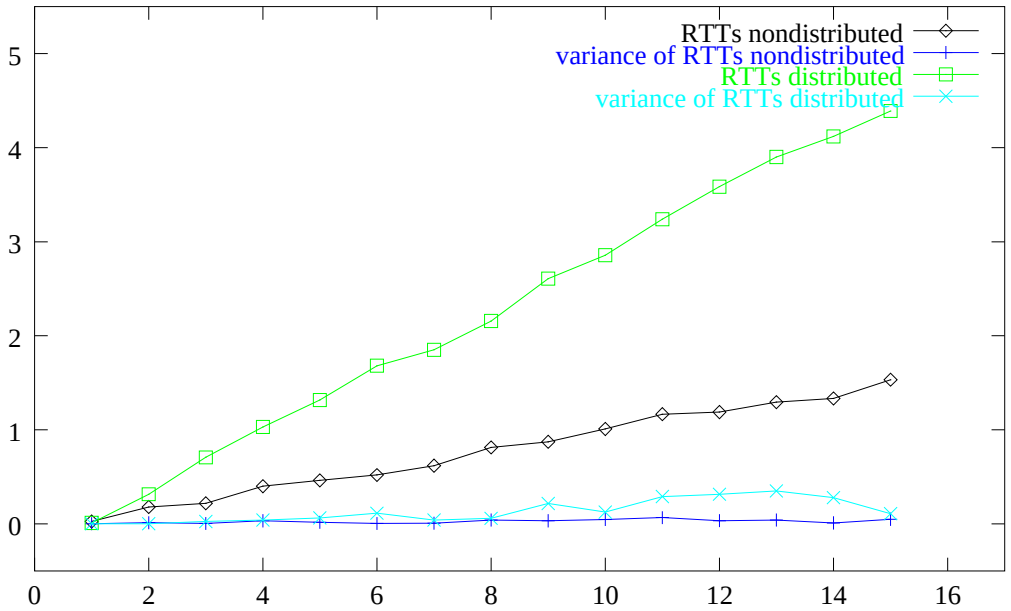


Figure 4.17: *RTTs in a a loaded VR network*

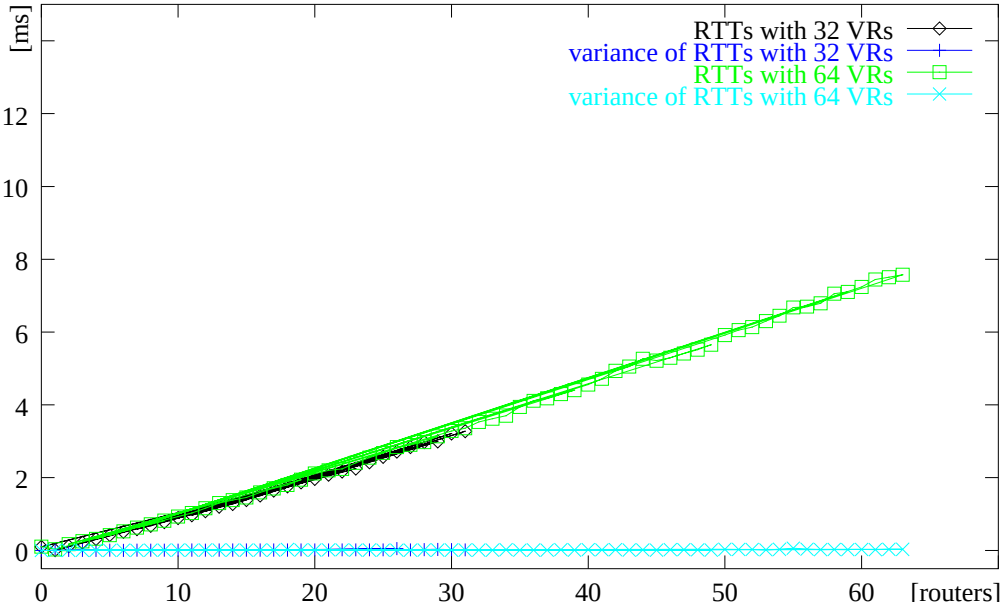


Figure 4.18: *RTTs with different numbers of VR entities in an unloaded network*

tunnels being used to connect VRs on different computers. The changing load of the UDP tunnels causes also a higher RTT variance as well.

4.3.2 Topology Size and Packet Delay

Another interesting question is the impact of the number of Virtual Router entities running on a computer to the RTTs. For this experiment 32 and 64 routers were set up in a chain on a single computer, in order to evaluate the influence of a distributed emulation. Figure 4.18 shows that as long as there is no additional traffic on the computer, the RTTs with 32 VRs correspond to the values with 64 routers.

The situation is different, if additional traffic is sent over the router chain as can be seen in Figure 4.19. The RTTs increase and the variance of the RTTs is significantly higher as well. On the other hand the graph still shows a linear increase of the RTTs with the number of hops the ICMP packets have to pass.

The measurement of RTTs gives a good impression about the impact of distribution to packet delay.

Because a VR has to process and forward packets in real time, it cannot provide a direct control over the link delay as ns can. Packets are forwarded from one VR to another by some kind of interprocess communication and therefore determined by the operating system.

Of course the behaviour of a transport medium of a certain (fixed) delay could simply be emulated by adding a queue with a certain delay between two VRs. However, this only allows to increase the delay, the smallest possible delay is determined by the computer's processing power.

Fortunately the time a computer needs to forward packets between two VRs is very

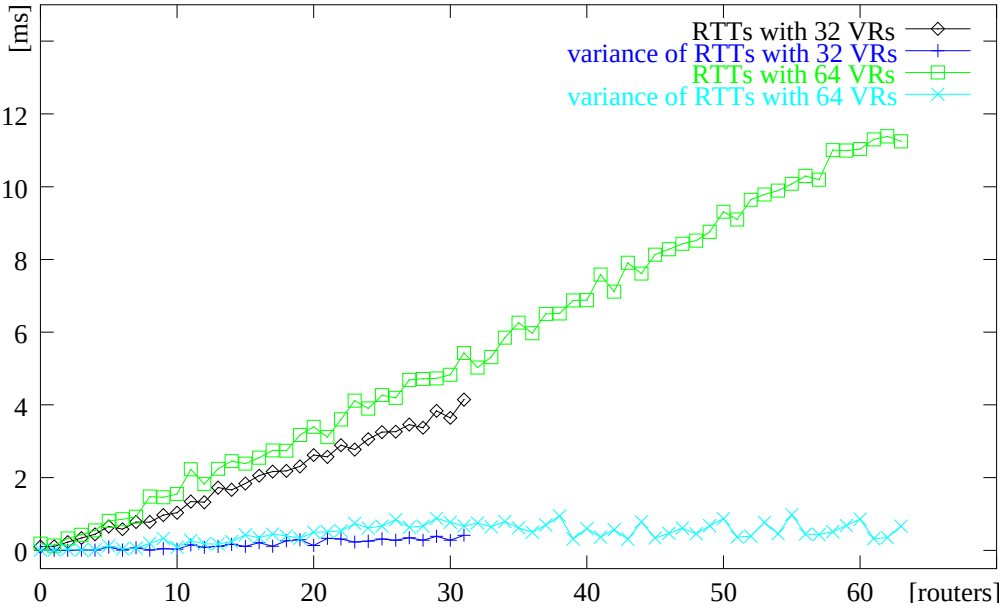


Figure 4.19: *RTTs with different numbers of VR entities in a loaded net*

small and an increased delay can be achieved rather simply by delaying packets during transmission.

Additionally, even for long distances, the delay caused by cables or fibers usually is very small and in any case constant compared to the delay caused by packet buffering within the network devices.

An evaluation of delays caused by Virtual Routers and a comparison with real network devices will be presented in the next section.

4.3.3 Impact of Queues to Packet Delay

It is quite simple to directly compare VRs to real network devices. Measurement data can be easily obtained using the ping or the traceroute programs. Table 4.4 lists some result. Each row lists a destination type: either a country (D,CH), a Virtual Router topology or our local Linux test network. A look on the results for the Linux network especially reveals the impact of the queueing delay. It can also be seen, that the Virtual Router only produces very small per hop delays, with nearly constant per hop delay for the 8 and the 16 hop network. The delays added by Virtual Routers are comparable to those of real networks as can also be seen on Table 4.5. It shows the output of two traceroute commands, the first one within the University's network and the other one over two Virtual Routers.

The Virtual Router queueing systems have the main impact on a packet's delay. Figure 4.20 shows the packet delay caused by a four Mbps link with and without congestion. Dependent on the bandwidth sent over the link, the FIFO queue has to buffer packets. An overload of five Mbps fills up the FIFO queue and causes a packet delay of approximately 40 milliseconds. If only 3.0 Mbps is sent, the queue remains empty and packets are only delayed for 2.4 Mbps.

	hops	RTT [ms]	per hop [ms]	congestion
D	15	38	2.53	unknown
D	15	32	2.13	unknown
CH	8	4.5	0.56	small
VR	8	2.1	0.26	no
VR	16	4.3	0.29	no
Linux-DS	8	75.46	9.433	heavy

Table 4.4: *Comparison of delays for different destination/networks*

---

traceroute to www.unibe.ch (130.92.9.60), 30 hops max, 38 byte packets
1 haydn66.unibe.ch (130.92.66.1) 0.644 ms 0.410 ms 0.351 ms
2 toscanini.unibe.ch (130.92.253.250) 0.794 ms 0.610 ms 0.524 ms
3 www.unibe.ch (130.92.9.60) 1.437 ms 0.850 ms 0.838 ms

traceroute to 10.43.10.1 (10.43.10.1), 30 hops max, 38 byte packets
1 10.42.10.2 (10.42.10.2) 0.142 ms 0.111 ms 0.081 ms
2 10.43.1.1 (10.43.1.1) 0.290 ms 0.274 ms 0.259 ms
3 10.43.10.1 (10.43.10.1) 0.308 ms 0.303 ms 0.282 ms

Table 4.5: *Unix traceroute within a real network (University of Bern) and through a virtual topology*

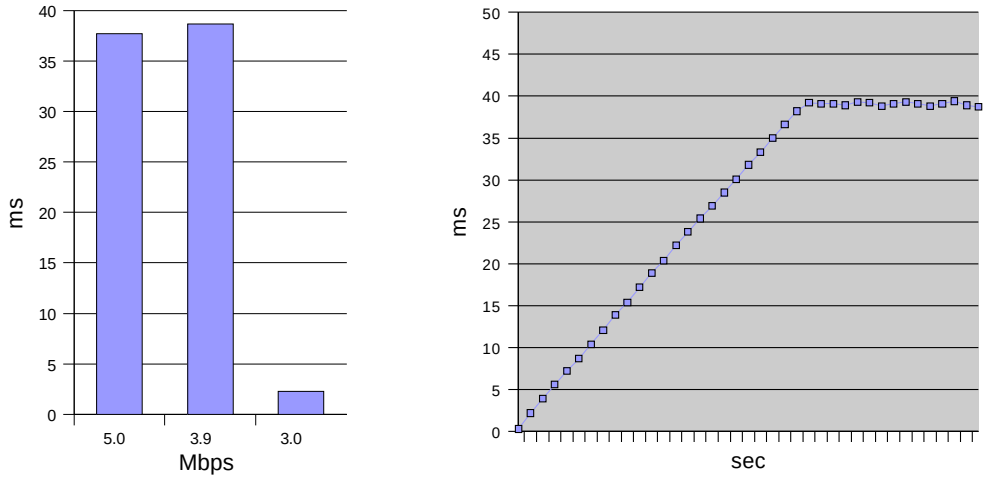


Figure 4.20: *The left graph shows the delays caused by the FIFO queue of a 4.0 Mbps link for different bandwidths. The right graph shows the increasing delay due to increasing queue length, if only slightly more than 4.0 Mbps is sent over the link and the queue is slowly filled.*

The graph on the right side of Figure 4.20 visualises this impact of queue lengths on packet delay drastically. Since slightly more packets have been sent than the queue was able to transmit, the length of the queue and therefore the delay increased slowly during the experiment. Once the maximum queue length was reached and the queue started to drop packets, the delay remained constant.

This delay caused by a FIFO queue can also be easily calculated by

$$d_{queue} = \frac{Q_{pkts} \cdot p_{avg}}{b_{bytes}}$$

with Q_{pkts} being the queue length in bytes, p_{avg} the average length of transmitted packets and b the bandwidth of the outgoing link in bytes per second. For a queue length of 20 packets and an average packet size of 1000 bytes, this results in a theoretical delay of 39 ms. The short queue length of 20 packets is a result of the TRIO queue algorithm. Even if the queue had a capacity of 64 packets, the packets with high drop precedence used for this experiment would have been dropped at a queue length of 20 packets already.

An example of the behaviour of queue lengths within a chain of routers is given in Figure 4.21. There the queue length, the delay and the throughput within a chain of 16 Virtual Routers connected by 1.0 Mbps links is illustrated.

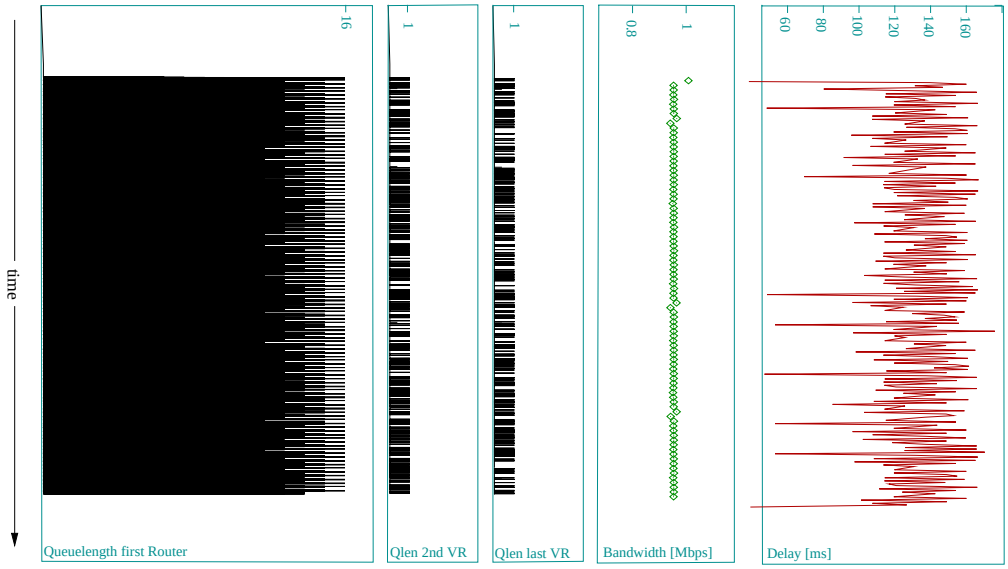


Figure 4.21: *Queue lengths, bandwidth and delay in a chain of sixteen VRs, transmitting a TCP flow. The queue lengths for the first, the second and the last Virtual Router in the chain are displayed.*

The graphs show the first queue being filled up with packets, while the queues of the following routers are empty or store one packet. Since the first Virtual Router is a bottleneck for the TCP flow, its droptail queue is filled until packets are dropped and TCP reduces it's transmission rate to less than 1 Mbps. Since all connections have the same capacity of 1.0 Mbps, the following routers are capable to transport this amount of traffic, therefore their queues remain empty.

Filled up, the first queue also increases the delay of the packets, as can be seen on the right diagram of Figure 4.21. Almost the complete delay of the packets is caused by the FIFO queue of the first router. The queue length variance and the delay are caused by the congestion avoiding mechanism of TCP.

These tests show comparable delays for Virtual Routers and real networks. Virtual Routers have usually smaller per hop delays than real network devices. If additional traffic increases the queue lengths, the delay caused by packet queueing is much higher than any delay caused by packet transmission.

If a certain transmission delay is needed as (e.g. satellite links), additional queues can delay transmitted packets and achieve the required delays.

4.3.4 Bandwidth Sharing

To conclude the evaluation of Virtual Routers core mechanisms several measurements regarding the sharing of bandwidth were accomplished.

Different flows of the same protocol have to share available resources like bandwidth in a fair manner. A Virtual Router receiving packets has to treat them in a way to achieve a realistic behaviour.

An equal processing of packets seems to be trivial but special problems arouse from a possible synchronisation between the protocol stacks and Virtual Routers running on the same computer. Since the Internet is distributed and the computers are not synchronised, such negative effects are less probable.

This is also not critical in pure simulators like the ns network simulator since they do not cope with an operating systems influence to forwarded traffic and do not work in real time. Since a simulator usually uses his own time scale and can provide exact timing, the results will match the theory. In real time this is not possible, since the operating system might be busy with the processing of routing protocols or execution of some high priority routines like interrupts, even if it is time to send a packet.

In the special case of a Virtual Router with sender, receiver and router(s) running on one computer and being therefore synchronised by the operating systems scheduler, interferences between senders, receivers, routers and the kernel can occur, leading to an unfair and unrealistic sharing of bandwidth between flows.

A proper handling of incoming packets has to be provided. Since TCP runs congestion control protocols, it reduces its bandwidth automatically if packets get lost as can be seen in Figure 4.22.

An imprecise treatment of packets can lead to completely different behaviour than expected from an IP router. In combination with negative effects of protocol stacks and routers running on the same host, results might differ significantly from real networks.

Therefore, before evaluating the Differentiated Services on Virtual Routers, some general experiments have to be performed, to evaluate the general correct behaviour.

To ensure a fair processing of incoming flows, the Virtual Router uses its own event based scheduler ensuring a proper fair treatment of multiple interfaces.

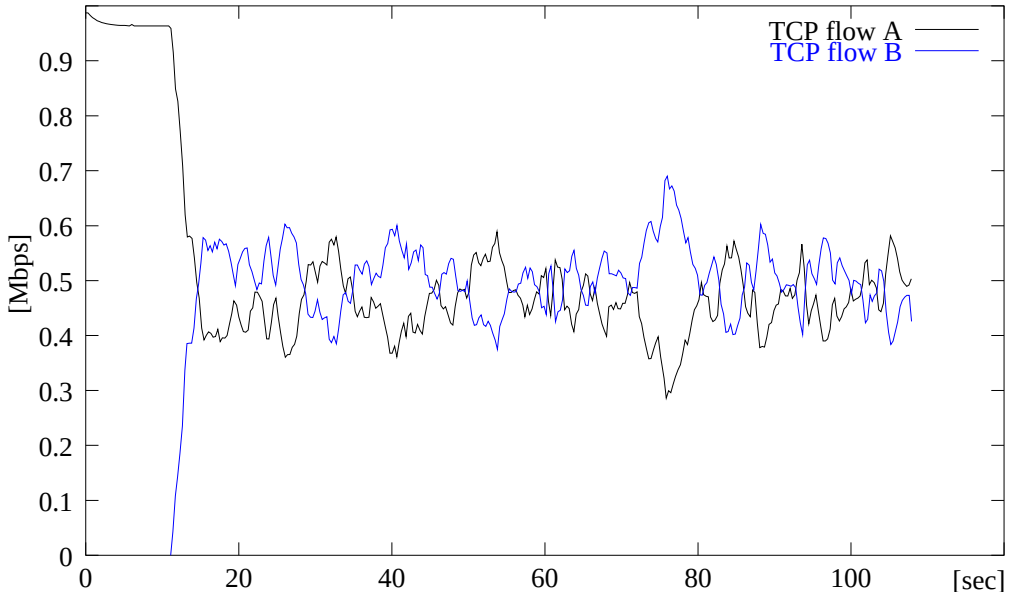


Figure 4.22: *Bandwidth sharing between two TCP flows. Both flows run congestion control mechanisms and react to packet loss, resulting in varying throughput.*

For the experiment set up ten end systems are connected over two Virtual Routers. The aggregate of all incoming traffic exceeds the bandwidth of 4 Mbps the link between VR A and VR B is capable to transmit. Five end systems and a VR are provided by one computer. Therefore, five softlink devices per computer are configured and the VRs are connected with an UDP tunnel. For the queueing systems simple FIFO queues were used, so all packets should be treated equally.

First measurements using UDP traffic have been performed. Since UDP does not use any flow control and there is no interaction between different UDP flows, the only critical point for a statistically fair bandwidth sharing is the Virtual Router. Based on the topology in Figure 4.23, five UDP flows have been measured. The flows transmitted UDP packets from:

$$10.42.n.1 \longrightarrow 10.43.n.1 \qquad \text{with } 10 \leq n \leq 14$$

During the measurements the packet rates of the UDP senders have been changed, leading to different shares of the achieved throughput values. Figure 4.24 shows three graphs, each with the measured input and the output bandwidth.

The graphs show that finally each flow achieves the same percentage of its incoming bandwidth, because the probability for a packet to be dropped is the same for all packets and interfaces. This is a realistic behaviour for a network device as long there are no mechanisms applied to favour certain kinds of traffic.

Finally the throughput values achieved by five TCP flows have been evaluated. Due to the congestion control mechanism not only the achieved throughput values are interesting, but also the reaction of TCP to changes of available bandwidth. The flows were not started at the same time, but with a delay of 60 seconds. Figure 4.25 shows

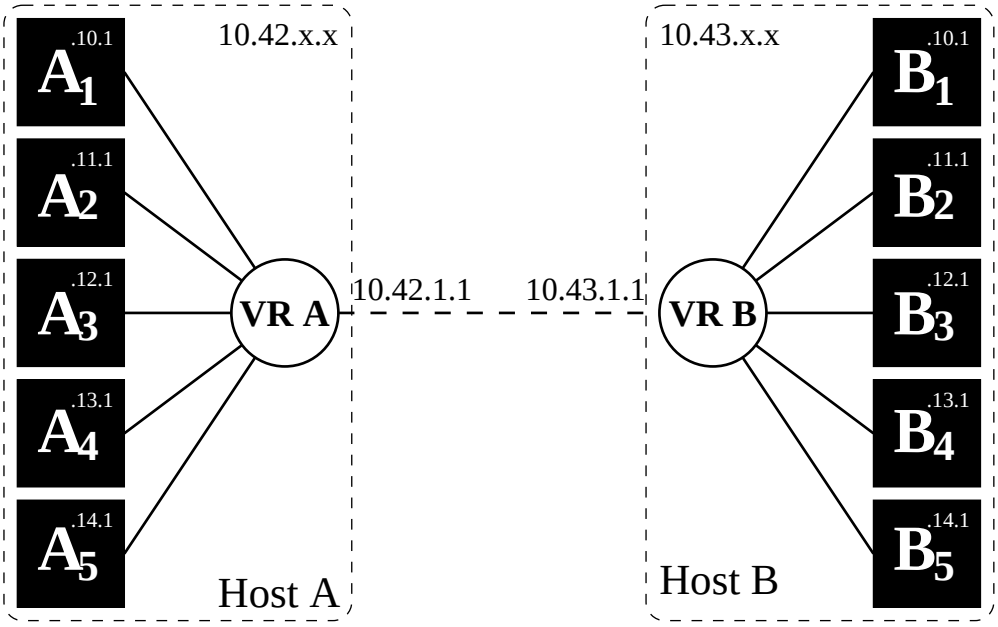


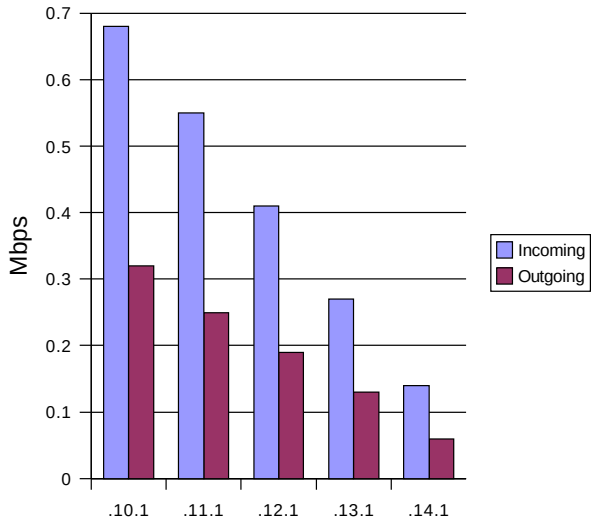
Figure 4.23: *Topology used for general Virtual Router evaluation and Differentiated Services tests. The bottleneck between the two VRs was limited to a bandwidth of four Mbps*

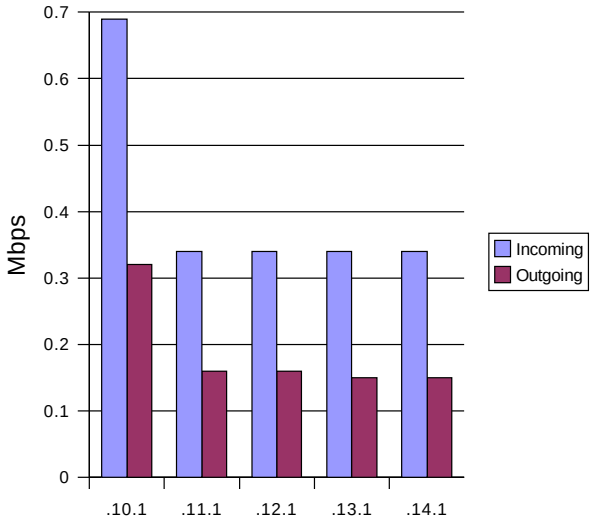
the bandwidth bahaviour and Figure 4.26 the bandwidth achieved during the last 60 seconds of the experiment.

The distribution of bandwidth between the five flows is not absolutely equal. This is because the flows were not started under equal conditions. Some flows were up for longer time than others, reacting differently to the congestion situation. However, the differences are quite small and absolutely comparable to the behaviour of real Internet scenarios.

Conclusion

In the previous sections the core mechanisms of the Virtual Routers were evaluated and compared to the behaviour of real network devices. Several experiments show the compliance of Virtual Routers with normal network devices. The available bandwidth is equally shared between incoming links and the dropping probability for different packet rates is also constant. Different flows being forwarded over a network of Virtual Routers are treated equally and achieve a fair share of the available resources. Also, the time forwarded packets is delayed is similar to the time caused by real network equipment. This allows an application to use a Virtual Router topology instead of a real network without noticing any significant difference.





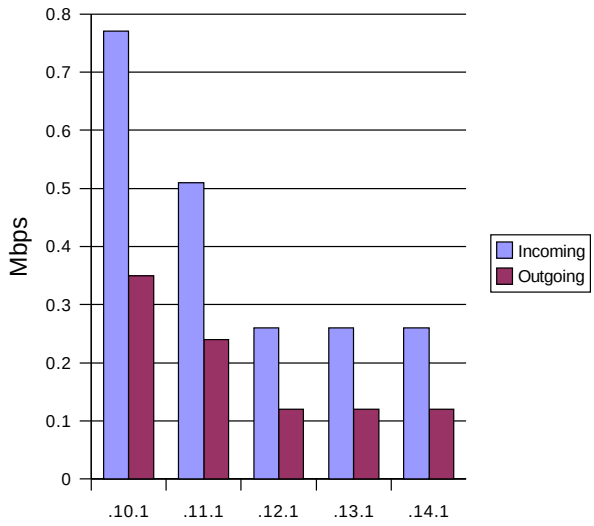


Figure 4.24: *Sharing of bandwidth between five UDP flows. The bars show the incoming and the achieved bandwidth values.*

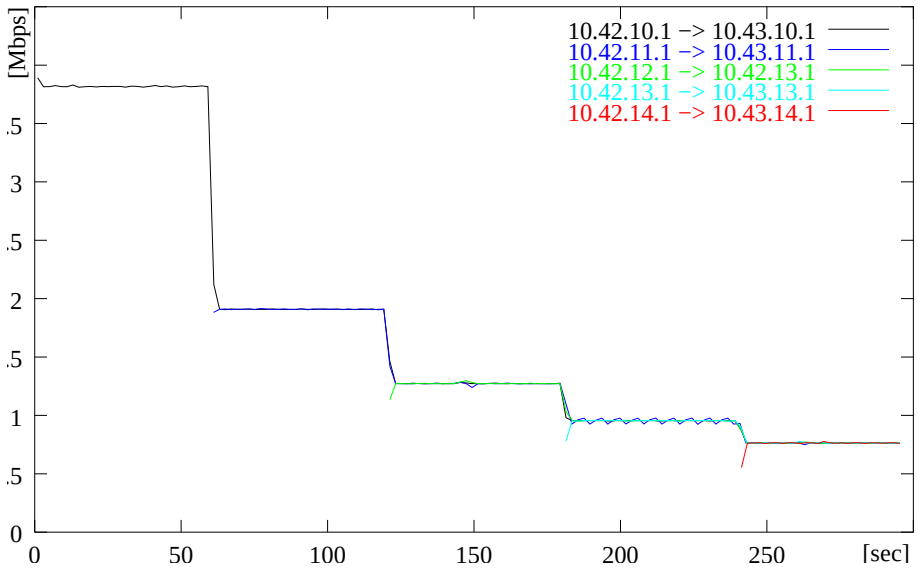


Figure 4.25: *Sharing of bandwidth between five TCP flows flows, being started with a delay of 40 seconds*

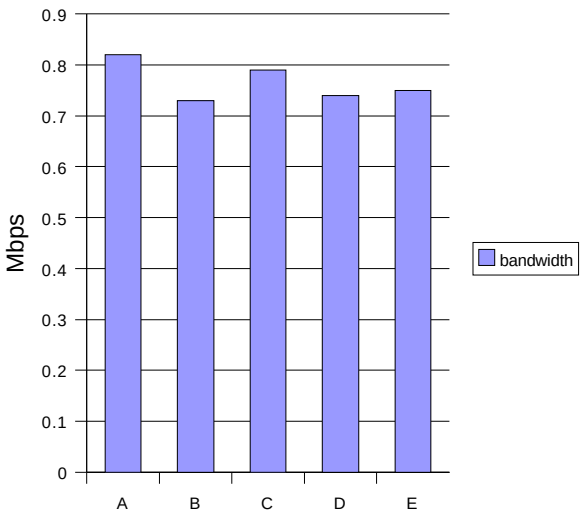


Figure 4.26: *Bandwidth reached by five competing TCP flows over a four Mbps bottleneck*

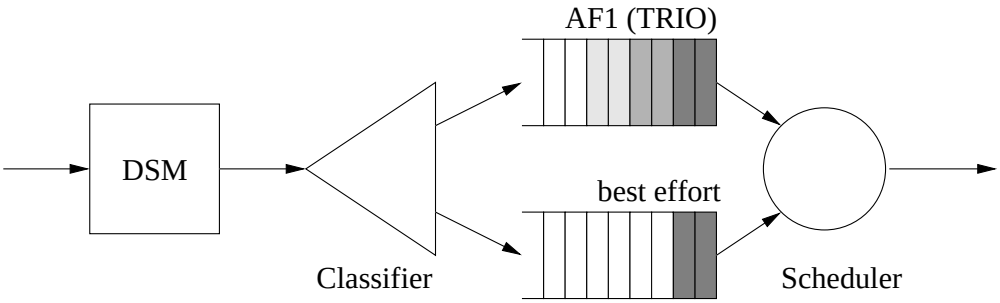


Figure 4.27: *Simple Differentiated Services queueing system with one Assured Forwarding class and best effort traffic*

4.4 Evaluation of Differentiated Services

So far the tests concerning Virtual Router were limited to show the system's ability to emulate an IP network device. The experiments covered issues like the impact of the number of emulated routers per host on the delay, questions like the behaviour during congestion and general parameters like the delay per VR or bandwidth sharing.

More advanced features like the flexible queueing system, capable to set up complex systems of queues, schedulers and classifiers will be evaluated in this section by setting up some experiments for Differentiated Services.

In Chapter 3 the network simulator ns has been used for the evaluation of Assured Service. To allow a comparison, similar experiments using Virtual Routers have been performed. As topology we use the same scenario as for the measurements of bandwidth sharing shown in Figure 4.23.

Similar to the evaluation using ns, the focus is on Assured Forwarding here as well. As mentioned before Expedited Forwarding service is no fundamentally new type of packet treatment but is similar to classical techniques providing a higher priority to certain flows as done by absolute priority queueing (see Section 2.4.1).

Nevertheless Expedited Forwarding is of course a new concept like Differentiated Service in general, but not because of special packet treatment. Differentiated Services specify a complete framework with policers and shapers at the border routers allowing to provide services like Assured and Expedited Forwarding network wide even if multiple ISPs are involved. In contrast to Expedited Forwarding, Assured Forwarding specifies not only an architecture but also new methods of per hop packet treatment.

Figure 4.27 shows the queueing system used during the evaluation. Incoming packets are checked by the Differentiated Service Marker (DSM) for their destination address and marked with DSCP values according to a set of rules (see Table 4.6).

Of course this marking needs only to be done in the first VR (A), which is acting as a kind of ingress router. The second VR (B) has no marker but simply provides a TRIO and a best effort queue. After passing the marker the packets are put to the queues by a classifier. The classifier works as BA classifier, simply checking the DSCP values. Since only the interaction within one Assured Forwarding class is interesting, the queueing system provides one AF class and also a queue for best effort traffic. The

```
Differentiated Services Marker (dsm):
—
source: 10.42.11.0/255.255.255.0     dest: 0.0.0.0/0.0.0.0
proto: 0     tos: 0
service: Assured Forwarding Class 1 with
        1.000000 Mbps and a bucketsize of 8192 bytes for low drop precedence
        0.500000 Mbps and a bucketsize of 8192 bytes for medium drop precedence
—
source: 10.42.10.0/255.255.255.0     dest: 0.0.0.0/0.0.0.0
proto: 0     tos: 0
service: Assured Forwarding Class 1 with
        2.000000 Mbps and a bucketsize of 8192 bytes for low drop precedence
        0.500000 Mbps and a bucketsize of 8192 bytes for medium drop precedence
```

Table 4.6: *Shell output as an example for marker rules for the Differentiated Service Marker within the ingress Virtual Router. Both rules mark packets according to their source address with the DSCPs of Assured Forwarding Class 1.*

scheduler runs in absolute priority mode, favouring packets from the AF TRIO queue.

The network topology with two Virtual Routers displayed in Figure 4.23 and the queueing system in Figure 4.27 were used to evaluate the behaviour of a couple of traffic flows being forwarded through that network. During this experiment only two instead of three dropping probabilities were used because of the following reasons:

- As indicated in the section about the colour markers (Section 2.3.5), the concept of three dropping probabilities mainly is meant to support bursts in a TCP environment. As our measurements try to evaluate the capacity of Differentiated Services to provide a certain bandwidth no bursty TCP or UDP traffic is used. Therefore using these mechanisms would just complicate the results.
- During the previous evaluations of Differentiated Services with the network simulator a model with only two different dropping probabilities was used. Using a similar model allows an easier comparison of the results.

4.4.1 Assured Forwarding and UDP

To evaluate the general capability to allocate a certain bandwidth for a specific flow, five UDP flows are forwarded through the network. The link of 4 Mbps between the two Virtual Routers (see Figure 4.23) is a bottleneck for the UDP senders, causing a heavy congested link, filled up queues and can be seen as some kind of worst case scenario for Differentiated Services.

The Differentiated Service Marker was configured to mark a different share of bandwidth with low dropping precedence for each flow.

Each flow is configured with a different amount of assured bandwidth, e.g. packets up to a specific bandwidth are marked with a low dropping precedence DSCP. The five

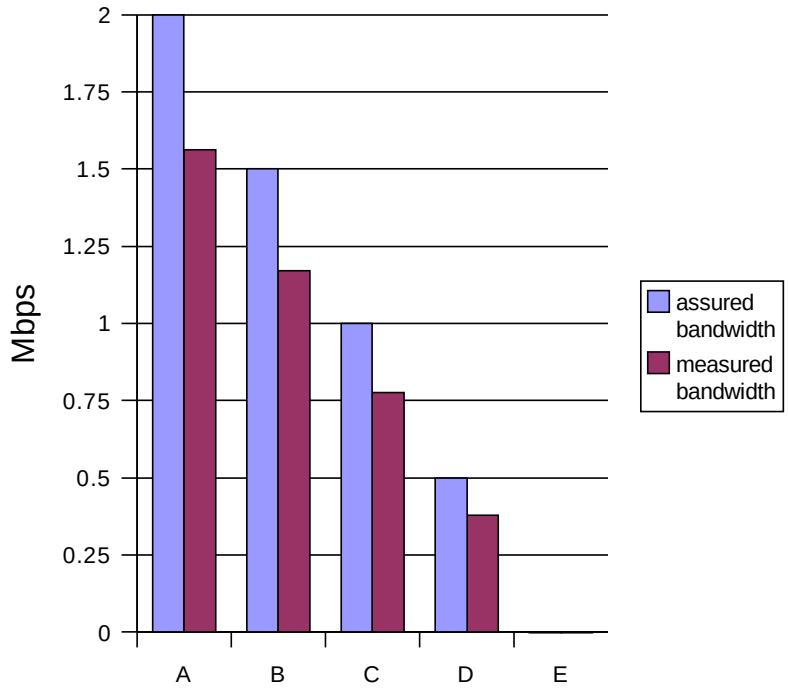


Figure 4.28: *Assured vs. measured bandwidth between five UDP flows in a Virtual Router network. Each flow had an incoming bandwidth of 2 Mbps. The bottleneck was 4 Mbps.*

flows are sent simultaneously, each with the same bandwidth of 2 Mbps, which leads to a bandwidth aggregate of 10 Mbps, congesting the 4 Mbps bottleneck link between the two Virtual Routers.

The results are shown in Figure 4.28. Each flow except E has a certain share of packets with low dropping probability. The graph shows the achieved throughput and the bandwidth up to which packets got a low dropping precedence DSCP.

The achieved bandwidth is of course smaller than the assured one, due to the congestion situation. Certainly such a set up would rarely occur in a good provisioned network. The maximum amount of traffic with assured low dropping precedence allowed should never exceed the maximum link bandwidth. Even if such a scenario is a kind of worst case scenario it confirms the ability of Differentiated Services to provide Quality of Service.

4.4.2 Assured Forwarding with Different Protocols

In Section 3.4 the capability of Differentiated Service was evaluated to protect congestion avoiding TCP against aggressive protocols like UDP. This property is important for any resource reservation mechanism. The protection of TCP against UDP by the use of different queueing systems and proper scheduling is the basis for traditional traffic conditioning mechanisms like class based queueing (see Section 2.4.4). Virtual Routers can be configured to provide such mechanisms as was shown by a simple ex-

ample for a queueing system in Section 4.2.3 which allowed to protect TCP flows from being suppressed completely by UDP.

These mechanisms differ from Assured Forwarding since Assured Forwarding does not use different queues but tries to achieve a protection of TCP by different dropping probabilities within a single queue.

The upper graph of Figure 4.29 illustrates the suppression of congestion avoiding TCP traffic by an aggressive protocol. It shows the reaction of a TCP flow to a couple of periodically activated UDP senders. This happens within the Virtual Router topology used for the Evaluation of Differentiated Services using only best effort traffic. Due to its congestion control, TCP reduces its transmission rate. This results in an unfair distribution of bandwidth between the TCP and the UDP flows. For a better readability, the upper graph only shows only the aggregate of the UDP flows. Once the UDP senders are activated, they dominate TCP completely and TCP uses the bandwidth not occupied by the UDP streams only. The situation looks different, if a certain amount of TCP packets is marked with low and medium dropping probability. The lower graph of Figure 4.29 shows the same Virtual Router experiment as before but now 1.5 Mbps of the TCP flow are marked with a low dropping and 0.5 Mbps with a medium dropping precedence DSCP, while UDP gets high dropping probability only. In contrast to the upper graph, here the throughput values of the single UDP flows are also displayed.

Since 2.0 Mbps of the TCP packets have low and medium dropping precedence, this share of the TCP bandwidth is ”protected” against the aggressive flow UDP. On the other hand the TCP flow also consists of packets with high dropping probability. Therefore, UDP is able to suppress the share of TCP being marked with a high dropping precedence and keeps on controlling this share of the TCP flow reducing the overall TCP bandwidth to 2 Mbps.

Obviously, the capacity of a service based on Assured Forwarding crucially depends on a proper differentiation among traffic with different dropping precedences. The capability to protect congestion avoiding protocols like TCP against aggressive senders by simply marking TCP packets with a lower dropping precedence is a good proof for this ability and also for the comparability of results obtained by the Virtual Router with those from the ns network simulator. A direct comparison of these and other results will be presented in Section 4.4.3.

So far the fair sharing of resources and the capacity of Assured Forwarding to protect congestion avoiding flows against aggressive traffic was shown. The next experiments will also take into account the impact of multiple dropping precedences on the achieved throughput and investigate the bandwidth sharing within a precedence level.

Assured Forwarding defines three dropping precedences. Depending on how an incoming flow is marked, the different probabilities shall allow a better support for bursts and TCP traffic. While one approach only uses one bandwidth specification with two different bucket sizes, another approach is based on two complete token bucket filters with two rates and sizes as introduced in Section 2.3.5. Because of the simpler design and the better flexibility the experiments are based on the latter one allowing to specify a bucket rate and a bucket size each for the low and medium dropping precedence.

To measure the impact of the three dropping probabilities in a pure TCP environment, five TCP flows are sent through the Virtual Router network. Per default all packets are

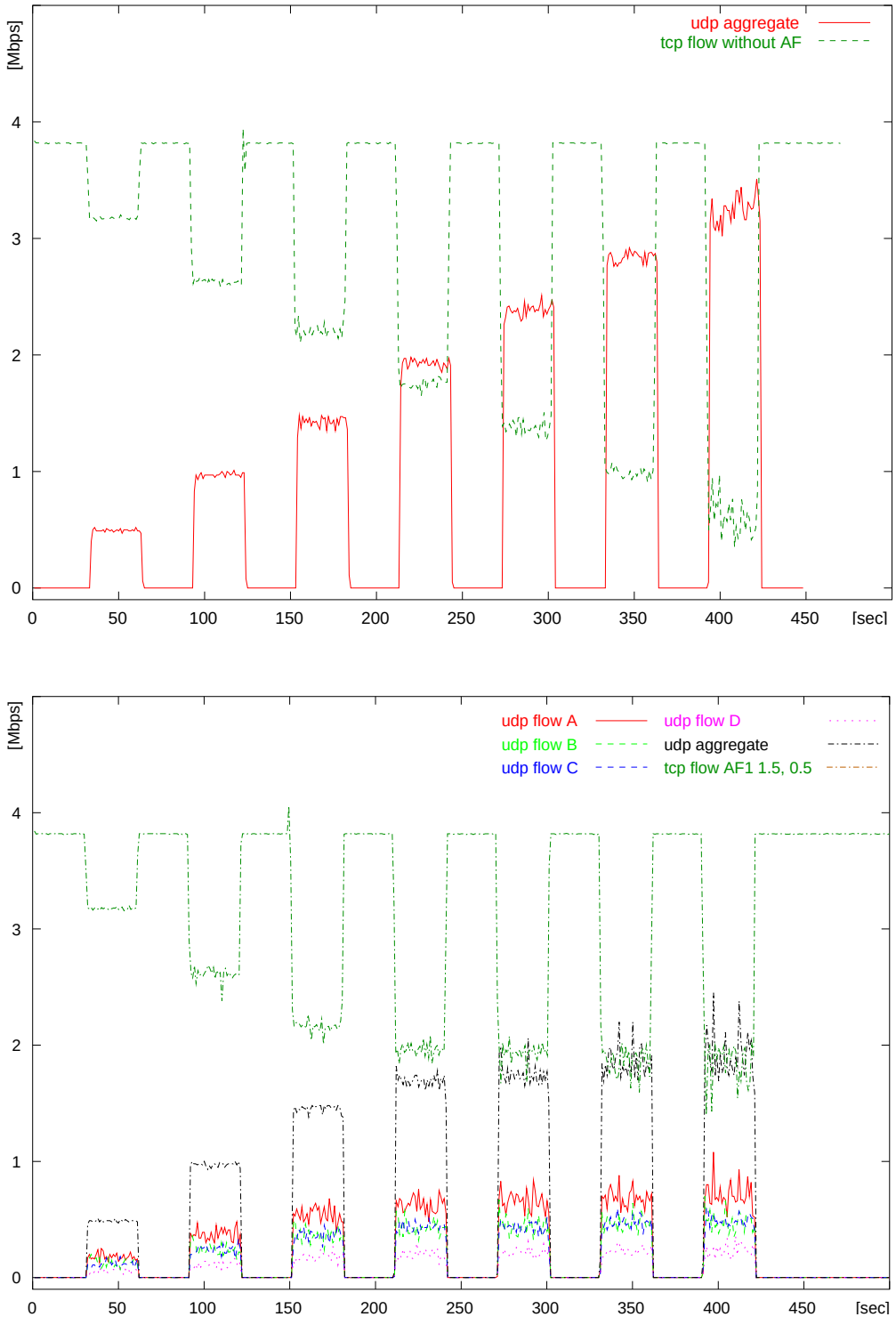


Figure 4.29: *Upper: Suppression of TCP traffic by an aggregate of UDP flows within a Virtual Router network. UDP absolutely gains control over the available resources in a bottleneck and suppresses the TCP traffic. Lower: 2.0 Mbps of the TCP packets are marked with low and medium dropping precedence DSCPs. Since UDP uses high dropping precedence only, TCP at least is able to achieve a troughput of 2.0 Mbps.*

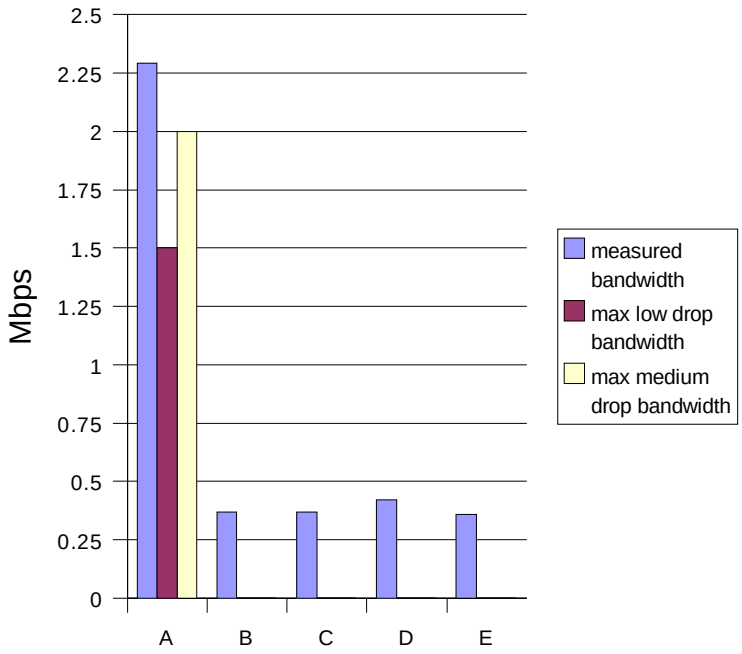


Figure 4.30: *Bandwidth reached by five competing TCP flows and Assured Forwarding in a Virtual Router Network. Flow A is supported by Assured Forwarding.*

marked with the highest drop precedence. So far this set-up is similar to the earlier measurements regarding the bandwidth sharing of Virtual Routers in Section 4.3.4. Additionally the Differentiated Service marker is configured to mark packets of flow A with lower dropping precedences.

	bw. in	bw. low	bw. medium	bw. out
A	max 4 Mbps	1.5 Mbps	2.0 Mbps	2.29 Mbps
B	max 4 Mbps	0 Mbps	0 Mbps	0.37 Mbps
C	max 4 Mbps	0 Mbps	0 Mbps	0.37 Mbps
D	max 4 Mbps	0 Mbps	0 Mbps	0.42 Mbps
E	max 4 Mbps	0 Mbps	0 Mbps	0.26 Mbps

Table 4.7: *TCP flows with assured bandwidth values. Up to 1.5 Mbps of flow A are marked with low dropping precedence DSCPS, another 0.5 Mbps are marked with medium dropping precedence (see also Figure 4.30).*

Table 4.7 shows the amount of bandwidth values of the flows. Packets are marked with either low or medium drop precedences. Any TCP packet of flow A exceeding 2.0 Mbps is marked with with a high dropping precedence like the packets of the other flow. To give a better impression of the achieved bandwidth results, Figure 4.30 shows a bar graph based on the table data. The bandwidth of flow A is above the value

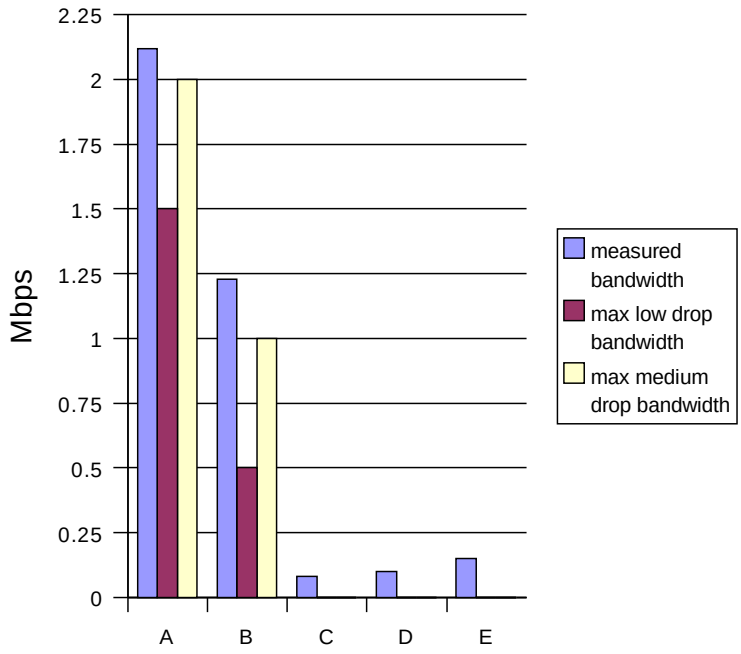


Figure 4.31: *Bandwidth reached by five competing TCP flows and Assured Forwarding*

configured for medium dropping precedence. The reason for that behaviour is obvious: Since all other flows are marked with a worse drop precedence, all the packets of flow A marked with medium drop precedence or better can be forwarded. The remaining bandwidth is distributed equally between all flows, increasing the bandwidth for flow A additionally.

The same experiment was evaluated with two instead of one flow supported by Assured Forwarding. Table 4.8 shows the measured values, Figure 4.30 the according bar graph.

	bw. in	bw. low	bw. medium	bw. out
A	max 4 Mbps	1.5 Mbps	2 Mbps	2.12 Mbps
B	max 4 Mbps	0.5 Mbps	1 Mbps	1.23 Mbps
C	max 4 Mbps	0 Mbps	0 Mbps	0.37 Mbps
D	max 4 Mbps	0 Mbps	0 Mbps	0.42 Mbps
E	max 4 Mbps	0 Mbps	0 Mbps	0.26 Mbps

Table 4.8: *TCP flows with assured bandwidth values. Packets of flow A and B are marked with low and medium dropping precedence (see also Figure 4.31).*

The results are similar to the previous experiment. Both flows A and B can achieve slightly more than the bandwidth marked with at least medium dropping precedence. The remaining bandwidth that is not used by low or medium dropping precedence packets is shared among all flows. Since only TCP is involved, the remaining band-

	bw. in	bw. low	bw. medium	bw. out
TCP	max 4 Mbps	2 Mbps	2.5 Mbps	2.14 Mbps
TCP	max 4 Mbps	1 Mbps	1.5 Mbps	1.32 Mbps
TCP	max 4 Mbps	0 Mbps	0 Mbps	0 Mbps
TCP	max 4 Mbps	0 Mbps	0 Mbps	0 Mbps
TCP	max 4 Mbps	0 Mbps	0 Mbps	0 Mbps
UDP	2 Mbps	0 Mbps	0 Mbps	0.20 Mbps
UDP	2 Mbps	0 Mbps	0 Mbps	0.19 Mbps
UDP	2 Mbps	0 Mbps	0 Mbps	0.19 Mbps

Table 4.9: *TCP and UDP flows. The TCP flows A and B are supported by Assured Forwarding, the remaining bandwidth is shared between the UDP flows. (see also Figure 4.32).*

width is shared fairly among all flows.

To conclude the evaluation of Differentiated Services the bandwidth achieved by different protocols shall be studied. Instead of using TCP flows only a mix of five TCP and three UDP flows is sent through a Virtual Router network. Table 4.9 shows the assured and measured bandwidth values.

Two of the five TCP flows were marked with low and medium dropping precedences. All other flows used high dropping precedences. Figure 4.32 displays the achieved bandwidth of the table above graphically.

The complete suppression of TCP flows without support by Assured Forwarding confirms previous experiences regarding the impact of UDP on TCP traffic. On the other hand the TCP flows supported by Assured Forwarding are able to achieve approximately their assured bandwidth values. In the previous experiment the flows with better drop precedences were also able to benefit from the bandwidth sharing of the remaining bandwidth that was not used by low and medium drop precedence traffic, achieving finally more bandwidth than assured.

In contrast in the new experiment the remaining bandwidth is used completely by UDP traffic suppressing the TCP packets with the same dropping precedence. Since TCP has to decrease its packet rate due to packet loss when it tries to exceed the medium dropping precedence bandwidth, the overall performance is less than the bandwidth allowed for medium dropping precedence traffic.

In the following section the results of the evaluations based on the Virtual Router Architecture shall be compared with results from experiments with the ns network simulator as performed in Chapter 3.

4.4.3 Comparison of Virtual Routers and ns

The scenario used for the experiments with the Virtual Routers (see Figure 4.23) is rather similar to the topologies set up for the ns network simulator (see Figure 3.1 and 3.2). The only differences are the number of nodes and the link bandwidth values. While the Virtual Router used higher link capacities (2.0 and 4.0 Mbps) the ns scenario

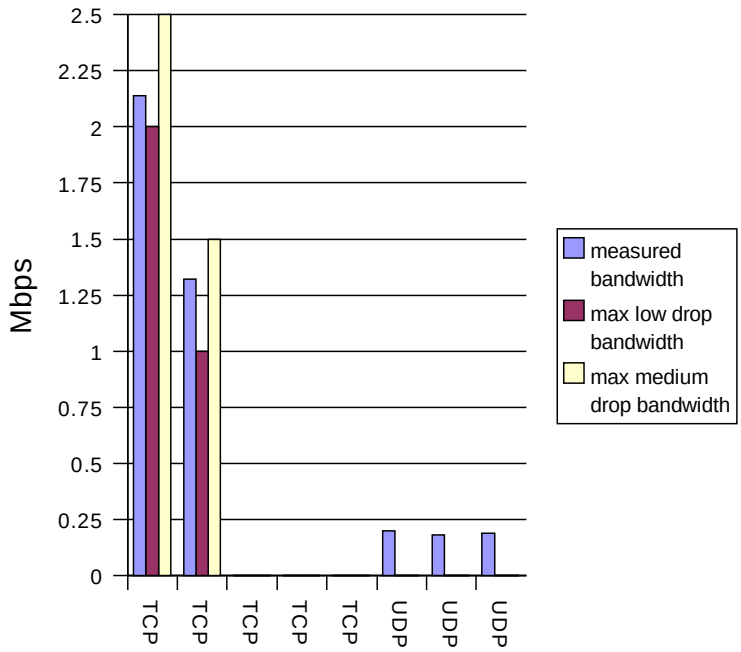


Figure 4.32: *Bandwidth reached by five competing TCP flows and Assured Forwarding*

was based on 1.0 Mbps links, but used ten instead of five end system pairs. Even if that makes a direct comparison of measurement values difficult the qualitative behaviour can be compared nevertheless.

Figure 4.33 shows two graphs. The left one was obtained by measurements with the Virtual Router, the right one using the ns network simulator. Both scenarios show the sharing of bandwidth among different TCP flows and the different assured bandwidth values they achieved. To allow a comparison even due to the different throughput values and link capacities, the ratio between the achieved and the assured bandwidth values was calculated:

$$R_{af} = \frac{\text{achieved bandwidth}}{\text{assured bandwidth}}$$

This value expresses how much percent of its assured bandwidth a flow was able to achieve. Even if the bandwidth values and the number of flows are different, the calculated value R_{af} of the Virtual Router experiment behaves similar to those values calculated from measurements with ns. In both graphs in Figure 4.33 R_{af} nearly stays constant for all flows with a trend for small flows to perform better. It can be assumed that TCP congestion control is a main reason for this behaviour. In a case of packet loss TCP decreases its bandwidth much more than it increases its bandwidth if enough capacity is available. This is why TCP flows with small bandwidth values in general perform better.

Accordingly this effect is much less significant in a scenario using UDP instead of TCP (see Figure 4.34). Theoretically R_{af} should be perfectly constant for UDP flows,

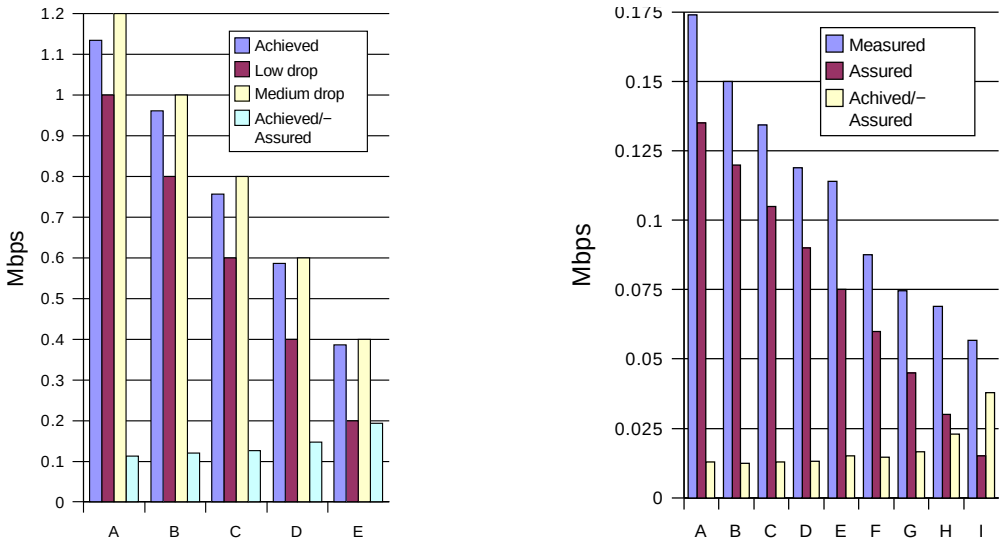


Figure 4.33: *TCP flows supported by Assured Forwarding. The left graph shows five flows using three dropping precedences and a VR topology, the right graph some results from using ns (see Chapter 3)*

since each packet should have the same probability of being discarded. Why the measurements with ns do not meet that expectations is unclear. Recent evaluations of ns [HE02] showed the weakness of the random number generator used within the ns implementation, possibly causing this variation.

To conclude the comparison of ns and Virtual Routers the protection of TCP against aggressive UDP flows shall be evaluated. Figure 4.35 shows the effect of aggressive UDP traffic on TCP. The Virtual Router set-up (left diagram) shows four UDP and one TCP flow. Each UDP flow was able achieve a throughput nearly equal to its maximum bandwidth. The TCP flow used the remaining link capacity. The ns simulator (right diagram) used five UDP and five TCP flows. In contrast to the Virtual Router network, the UDP flows transmit more packets than the bottleneck link is able to forward, causing heavy congestion. While the UDP flows in the Virtual Router set-up left a certain bandwidth to the TCP flow, in the ns experiment the TCP flows are suppressed completely by the aggressive UDP flows. However, the main effect is identical. Real UDP traffic forwarded through a Virtual Router topology is as aggressive as simulated UDP traffic using ns. Obviously an increase of the UDP packet rate would cause a complete suppression of TCP in the Virtual Router experiment similar to that using ns.

In contrast Figure 4.36 shows measurement values assuring bandwidth to TCP flows only. Similar to the previous graphs the right diagram shows results obtained by ns and the left one by an emulated network. The results are more or less identical. In both experiments TCP was able to approximately achieve the assured bandwidth while UDP suppressed traffic that was not supported by Assured Forwarding.

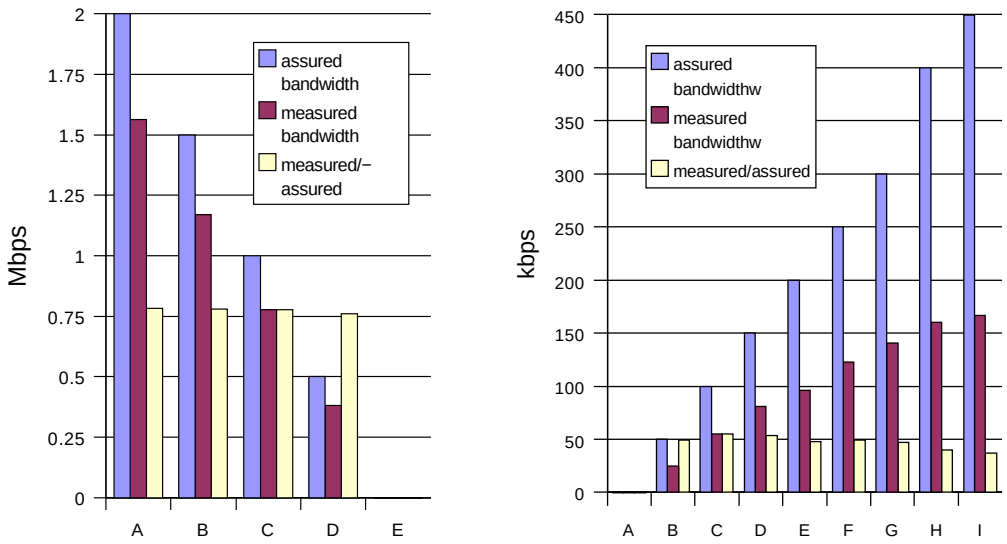


Figure 4.34: *UDP flows supported by Assured Forwarding. The left graph shows five flows using VRs, the right graph results using ns (see Chapter 3)*

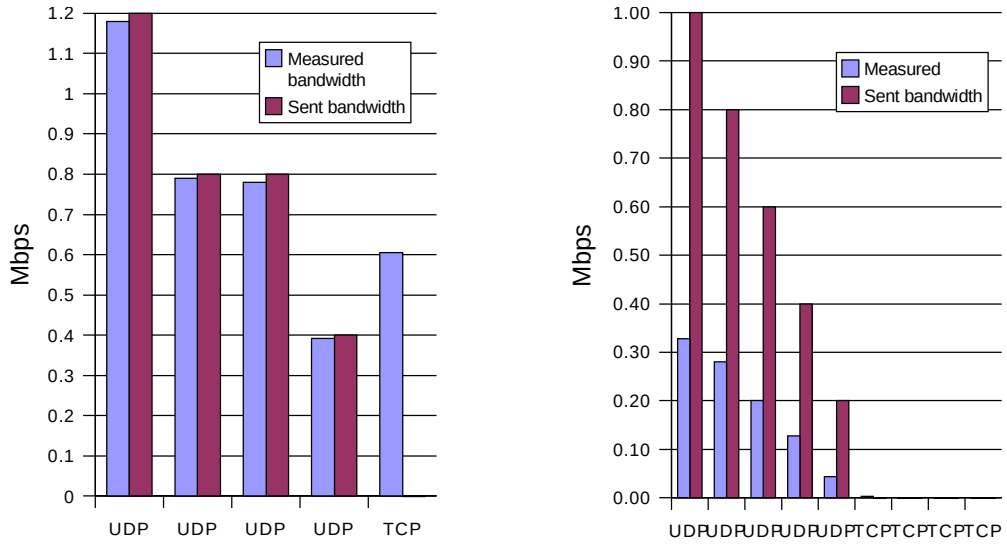


Figure 4.35: *Suppression of TCP by UDP in a Virtual Router scenario (left) and in a ns experiment (right).*

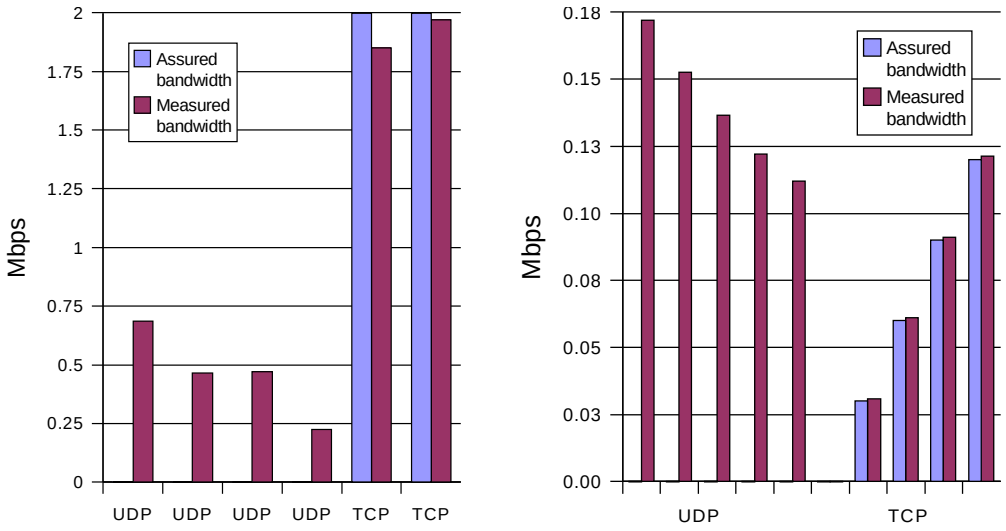


Figure 4.36: *Protecting TCP flows against aggressive UDP traffic*

4.5 Virtual Routers for Network Emulation

The Virtual Router architecture and implementation have been presented in this chapter. Virtual Routers were developed to allow the simple and quick set-up of test networks for development purposes. Using Virtual Routers, an appropriate evaluation network is available without any need for expensive hardware. Virtual Routers allow to set up large emulated network topologies on a small number of computers.

Virtual Routers do not only provide a static network emulation. In contrast to the traditional configure → run → evaluate scheme of network simulators, Virtual Routers provide a much more flexible environment. Each single Virtual Router can be started, configured and shut down independently from other network nodes. The command line front-end accepts commands like `route` and `ifconfig` that are well known from standard Unix interfaces and allows to reconfigure the router without requiring a restart of the complete emulation or even a single Virtual Router.

The emulated network can be connected to a real network. Using the softlink Linux kernel module providing a network interface card like device, a computer can route traffic to that network device and so access the emulated topology. For an application running on the computer there is no difference between the emulated network and a real one. This allows the use of standard applications which can be used with an emulated network evaluating new protocols or traffic conditioning components.

Because Virtual Routers can run completely in user space and provide simple and open interfaces for new components, VRs themselves offer a very convenient environment for the rapid implementation of prototypes, which can be evaluated directly. Usually the extension of Virtual Routers is much faster and less annoying than programming operating system code.

Since Virtual Routers have to comply in their behaviour common network simulators and of course real networks, several experiments were performed. The impact of topology size and distribution on packet delay was measured as well as more general

aspects like the bandwidth sharing between several input links.

Results from the previous chapter evaluating Differentiated Services with the ns network simulator were compared to a Differentiated Services network set up with Virtual Routers. In contrast to ns which required to process the simulation log files to get the results, the Virtual Router network allowed the direct use of standard measurement programs like ttcp. Nevertheless, a comparison of the results showed a similar behaviour.

Virtual Routers can provide a simple and quick tool and evaluation platform for the development of new protocols or traffic conditioning components. Especially the capability to provide Quality of Service by Differentiated Services offers interesting fields of applications. In the following section the Virtual Router architecture will be used to investigate concepts for the management of Quality of Service in large IP networks.

Chapter 5

Managing Quality of Service

The evaluation of Differentiated Services within the previous chapters showed the capability to provide Quality of Service but also illustrated the dependency on good network provisioning. Since DiffServ does not reserve resources per flow but for traffic aggregates and since there is no RSVP like signalling, a customer has to rely on the Service Level Agreement negotiated with its Internet Service Provider.

While for the customer RSVP has the advantage of signalling an individual flow reservation, an Internet Service Provider will probably prefer Differentiated Services because of scalability issues. This is why a combination of both services would be advantageous, providing Integrated Services to the customer by using Differentiated Services in the backbone [BBBG00].

Within this chapter a concept to integrate both RSVP and Differentiated Services will be presented and evaluated.

5.1 Network Provisioning

The amount of resources required by a certain service is defined more or less precisely. Therefore it is complicated for an ISP to exactly measure the available resources left within his network. While a service like Expedited Forwarding provides a leased line like service and usually has fixed source and destination addresses or at least networks, an Assured Forwarding like service is more likely to provide a general service improvement, independent from a particular destination address.

Therefore EF will allow to calculate precisely the amount of required resources at each point within the network, while Assured Forwarding requires some estimation by the ISP. The ISP in Figure 5.1 can sell two leased line like services each with 100 Mbps between the customers (A, C) and the customers (B, D), even if his internal link bandwidth is not capable to carry the traffic of both. Since the leased lines are using different paths through the network, they will not interfere. Obviously this is only possible if the committed service agreement defines exactly the source and destination addresses.

Without this information an ISP can either sell bandwidth only up to his internal link capacity (2 x 77.5 Mbps in our example) or rely on statistics, selling more than he can

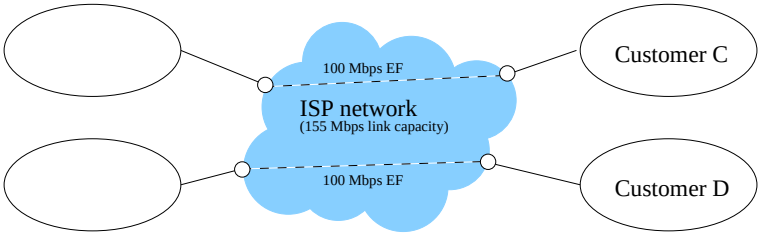


Figure 5.1: *An Internet Service Provider with his customers*

provide and hoping the overall traffic will not exceed his resources. Within a large network with a huge number of Service Level Agreements the ISP will probably use some statistics estimating the available resources for a connection.

Several proposals how to manage these resources efficiently have been made. One solution is to control the allocated resources within a network by central instance called bandwidth broker [SBP99], [SBB01]. A customer can negotiate with the broker, that configures the network devices accordingly. This negotiation of a SLA may take place using a WWW interface or some setup protocol allowing the user to automatically setup the required resources.

Another approach is to monitor the resource consumption within the network (e.g. at the ingress border routers) and to reconfigure the network automatically. Of course this reconfiguration is limited to the network capacity. Also intelligent mechanisms may be used to optimise the load sharing within the network. Packets may not only be routed according to their destination addresses but also dependent on their service.

The simplest solution to solve that problem may be to provide more resources than are required probably (over provisioning). If SLAs are changing infrequently only static resource reservations within the network can work quite well.

Since RSVP is flow based, an integration of RSVP and Differentiated Services can cause changes of SLAs and therefore requires a reconfiguration of the network. This is why the approach presented in Section 5.3 will use a central bandwidth broker to manage the resources within the Differentiated Services network.

5.2 Network Device Configuration

For the configuration of network devices the Simple Network Management Protocol (SNMP) is usually used. This works quite well as long as the network is small or the frequency of reconfigurations is low. In larger networks with frequently changing configurations a central instance like a broker faces scalability problems.

SNMP [CFSD90] provides a systematic method for the monitoring and configuration of a network. The development on SNMP started under the basic condition to create a simple protocol. Unfortunately, the final protocol does not meet this requirement. SNMP defines mechanisms to control and monitor specific variables (objects) within a network device. Also a network device can notify management stations of certain

events as link failure or hardware faults. The management station can signal the problem to the system administrator or also automatically set up backup mechanisms.

To monitor and configure network devices SNMP uses a Management Information Base (MIB) (e.g. [BCS01]) containing the variables and events, which can be controlled and configured. These MIBs are part of the network devices firmware and do not allow the creation of complete new services.

Another drawback of SNMP is the architecture based on central management stations, collecting data and reconfiguring the controlled devices. Such a mechanism is great for small networks or a low demand of reconfigurations. To provide dynamica Quality of Service to a customer network reconfiguration will be necessary more frequently increasing the management overhead.

Other frequent mechanisms to configure and access network devices are their command line interfaces. Modern devices even provide complete HTTP servers, allowing to change settings using a standard web browser. Of course command line interfaces and HTTP servers can be used not only by humans, but also by programs. Instead of using SNMP an automatic login into the network device is often the simpler solution.

Especially to communicate network traffic policy information to network devices the Common Open Policy Service (COPS) was developed. COPS is able to provide admission control and in combination with RSVP to ensure adequate bandwidth, jitter and delay bounds for time-sensitive traffic such as voice transmission [BCD+99].

5.3 Interoperation of Integrated and Differentiated Services

5.3.1 Basic Concepts

Two alternatives for inter-operation between Integrated Services and Differentiated Services were mentioned in [BYBZ98].

The first option assumes to run Integrated and Differentiated Services independently of each other. Some flows such as real-time traffic might get an Integrated Service reservation while others are supported by Differentiated Services mechanisms. This operation is simple but limits the use of RSVP [BZB+97] to a small number of flows. In this mode, each node within the Differentiated Service network may also be a RSVP capable node.

The second approach assumes a model in which peripheral stub networks are RSVP and Integrated Service aware. These are interconnected by Differentiated Services networks that appear as a single network link to the RSVP nodes. Hosts attached to the peripheral Integrated Service networks signal to each other per-flow resource requests across the Differentiated Services networks. Standard RSVP processing is applied within the Integrated Service peripheral networks. RSVP signalling messages are carried transparently through the Differentiated Services networks. Devices at the boundaries between the Integrated Service networks and the Differentiated Services networks process the RSVP messages and provide admission control based on

the availability of appropriate resources within the Differentiated Services network [BYBZ98].

This model is based on the availability of services within the Differentiated Services network. Multiple Integrated Service micro-flows which exist in peripheral networks are aggregated into a behaviour aggregate at the boundary of the Differentiated Service network. When a RSVP request for an Integrated Service arrives at the boundary of a Differentiated Services network, admission control is applied based on the amount of resources requested in the Integrated Service Flow Spec and the availability of Differentiated Service at the corresponding service level. If admission control succeeds, the originating host or the aggregating router marks packets of the signalled micro-flow according to the appropriate Differentiated Services level. RSVP/Integrated Service over DiffServ is especially suitable for providing quantitative end-to-end services.

However, for any approach integrating RSVP and Differentiated Services should meet some central requirements;

- In the access networks a parallel operation of Integrated Service and Differentiated Services should be possible. It should be left to an application which type of resource reservation is used. Especially the Assured Forwarding service with its different drop probabilities may be profitable for applications dealing with data of different importance, like multimedia services. Currently such a service can only be provided by Differentiated Services.
- The approach of mapping RSVP to a more scalable kind of resource reservation, should not be limited to Differentiated Services, because the method of resource reservation in the backbone may vary. In addition to the favoured Differentiated Services a mapping (and aggregation) of different RSVP flows to ATM PVCs may be chosen as well as a mapping to different IP tunnels.
- The used architecture should not require a modification, neither of RSVP capable applications nor of the end systems' RSVP daemons.
- The technology used for resource reservation in the backbone should conform to the standard framework for Differentiated Services [BBC+98b].

5.3.2 RSVP Signalling and its Extensions

Figure 5.2 shows a standard scenario with several ISP clouds and two access networks. In the ingress router between an Internet Service Provider and an access network the RSVP flows have to be mapped to Differentiated Services classes.

This mapping task can be split into two parts. The first one is the RSVP signalling, which is of course used for the resource reservation in the access networks and also for triggering resource reservations within the ISPs.

The second one is the technique of aggregating flows and reserving bandwidth inside the ISP's networks. We propose a central instance in the ISP called bandwidth broker, which can be queried whether there is bandwidth available within the ISP network and which supervises the ISP's resource management. How an ISP finally allocates

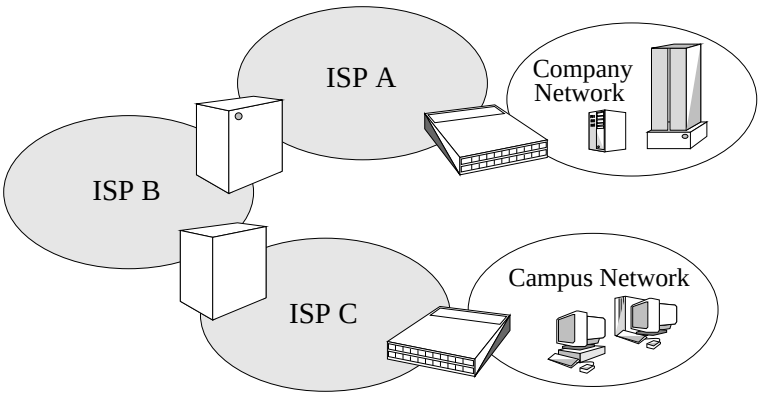


Figure 5.2: *Internet Service Provider (ISP) with Access Networks*

resources is left to the ISP. We propose Differentiated Services for the resource management in the ISP's backbone, another choice might be the mapping and aggregation of RSVP reservations within ATM VCs.

Since Differentiated Services provides scalable support for Quality of Service the ultimate goal of RSVP/Differentiated Services integration is to avoid any RSVP resource reservation between the ISP's border routers.

The information about administrative permissions and Service Level Agreements are located in the Bandwidth Brokers database. For a mapping the bandwidth broker has to be queried, whether the reservation can be set up. This requires interaction between the mapping component and the broker.

In the following paragraphs two different modifications to RSVP signalling to meet the requirements of the approach using a central bandwidth broker are presented and discussed. The methods react directly on single reservation requests using either the *path* or the *resv* message.

Bandwidth allocation using the RSVP *path*-message

As mentioned in the beginning of Section 2.2 the RSVP resource setup process includes the exchange of several messages through the network. The *path*-message is the first step of a RSVP reservation (see also Figure 2.1). The RSVP *path*-message contains information about the flow requirements. So it can be used to query the broker. This one then decides whether the resources can be allocated and will configure the backbone accordingly. The aggregation of flows can be done by mapping the different RSVP flows to IP tunnels with a certain QoS or ideally directly to Differentiated Services classes. Figure 5.3 shows the required signalling.

1. The ISPs ingress router receives the *path* message and extracts information about the requested resources, the source and destination address.
2. The ingress router queries the bandwidth broker, whether the request can be met.
3. The bandwidth broker checks its database about the available resources and informs the border router.

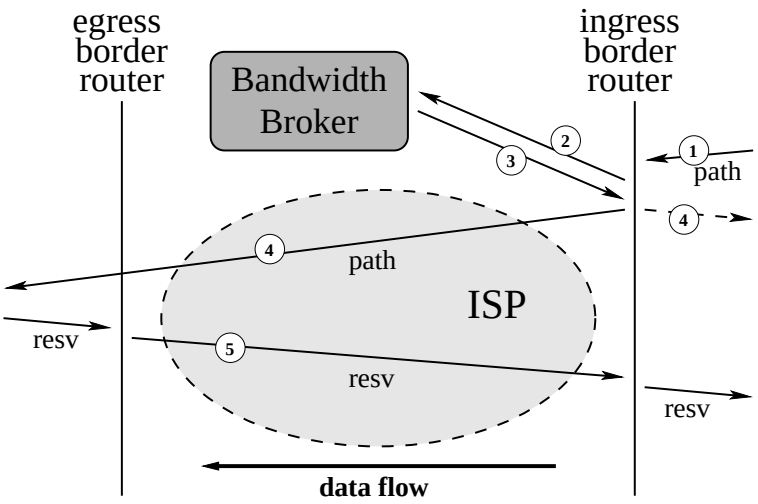


Figure 5.3: *RDG Signalling using the* path*-message*

4. If successful the path message is forwarded, if not a *resv-err* message is replied.

5. The standard RSVP *resv*-negotiation takes place.

One of the big advantages of this concept is the independent location of the component contacting the broker. It may be located in the ISP's ingress or in the egress router. Since RSVP favours a 'the receiver pays' scheme, a location on the egress border router may be preferred.

Unfortunately, the information about the requested resources included in the *path*-message are only preliminary. A user might modify this reservation or discard it completely. Another problem is the receiver pays scheme of RSVP. At the time the *path*-message has to be processed by the ingress or egress router, the receiver has not yet agreed to take over the costs for the reservation.

Bandwidth allocation Using the RSVP *resv*-message

The arrival of the *resv*-message at the ingress border router (see Figure 5.4) can also be used to trigger resource reservations in the ISP's network,

1. The ISPs ingress router receives the *path*-message, processes it and forwards it to the next router.

2. After the *path*-message has reached the receiver, a *resv*-message is generated and transported hop by hop to the sender of the *path*-message. It is assumed, that there are no RSVP capable routers in the ISP's backbone or the processing of RSVP signals is omitted by setting up tunnels between the border routers.

3. The *resv*-message reaches the ingress border router. This router is now responsible for setting up resources between the two border routers. So it queries the bandwidth broker, whether the flow conforms to the SLA or not.

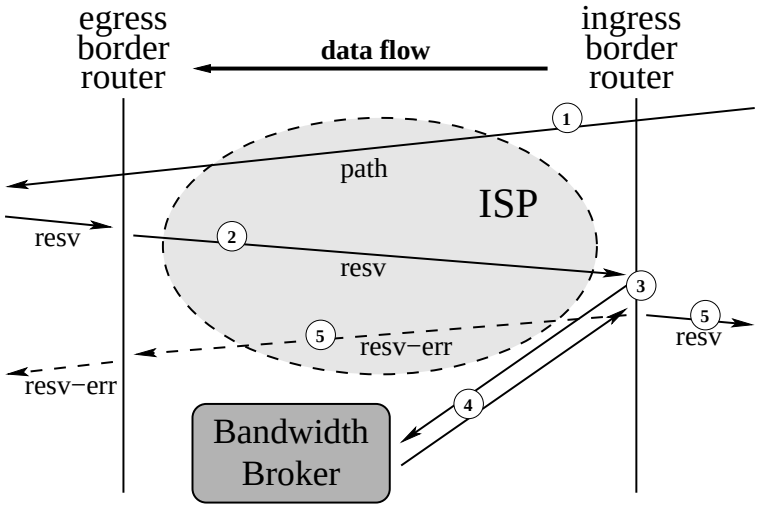


Figure 5.4: *Usage of the* resv*-message for bandwidth broker signalling*

4. The broker decides upon the reservation, replies the result to the ingress border router and sets up appropriate resources within the ISP's network.

5. If the reservation is accepted, the *resv*-message is forwarded, else an *resv-err*-message is generated and replied to the sender of the *resv*-message.

In contrast to the *path*-message as used in Section 5.3.2 to trigger the ISP's resource reservation mechanisms, the *resv*-message contains reliable information about the requested resources and is sent by the receiver, who has to take over the costs.

Even if the *path*-message is sent earlier during RSVP negotiation, the *resv*-message offers a much more reliable and facile way to trigger resource allocation in the backbone. The overhead to exchange *path*-messages, even if the broker finally rejects the resource reservation, does not carry weight. So for the implementation *resv*-messages were used to trigger reservations.

5.3.3 Prototype Implementation

For the prototype implementation commercial routers have been used. To provide QoS inside the ISP's network tunnels with a certain QoS shall be setup between the border routers. Since the RSVP daemon of Differentiated Services routers lacks functionalities to provide the required interaction with the bandwidth broker, each Differentiated Services router was supported by a Linux router running a modified RSVP daemon to manage the signalling. Figure 5.5 shows the equipment and the topology of the test and demonstration network.

The two Differentiated Services routers together with the Linux ingress routers realise the ISP's border routers. For QoS provisioning within the ISPs network the routers use VPN tunnels based on the concept of Differentiated Services as described in [BBC+98b]. The two Linux machines outside the Differentiated Service routers

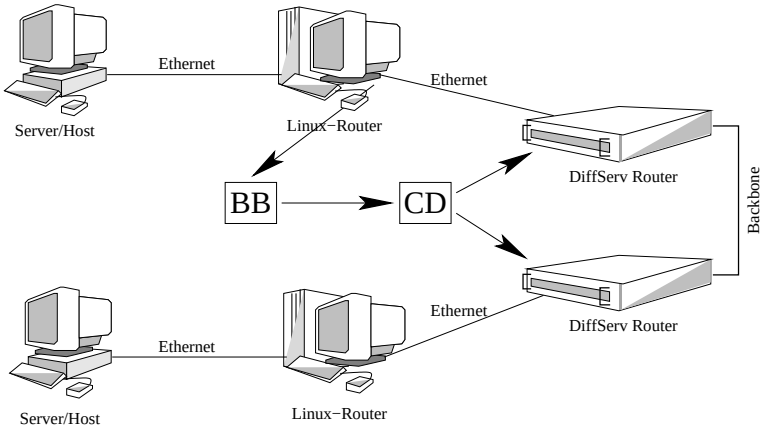


Figure 5.5: *Single ISP Scenario with two Differentiated Services and two Linux routers*

have to keep track of RSVP signalling, the reservation of local resources and the interaction with the bandwidth broker, which configures/monitors the Differentiated Service routers. The configuration daemons (CD) between the broker and Differentiated Service routers are used as some kind of adaptation layer. So the broker can use a ”platform independent router configuration language” to configure the Differentiated Service routers. This shall allow the easy exchange of the router platform (see also [GBK99], [BBB+99]).

Because of simplicity the implemented version of the RSVP Differentiated Service Gateway (RDG) directly connects to the ISP’s bandwidth broker. In reality the RDG may connect the broker of his local network, which then will negotiate with the ISP’s bandwidth broker if necessary.

The RSVP Daemon’s Extension

As mentioned in Section 5.3.2 the concept based on the use of *resv*-messages to trigger the ISP’s resource reservations was chosen. From the Linux routers point of view the whole ISP network can be treated as a huge extension of its local queueing system. The RSVP daemon provided by the Information Science Institute (ISI) and ported to Linux by Werner Almersberger [Alm] was used as a basis for this implementation. Since it was designed for a high portability to different router platforms, there is a suitable interface between the queueing system (under Linux based on the programs/libs tc and ip) and the RSVP-daemon itself.

We used this interface to add the required functionality for the interaction with the bandwidth broker. Every time local resources are reserved, released or modified also some routines are called, querying the Bandwidth Broker as shown in Figure 5.3.3. By this a reservation is only successful, if the local traffic control system and the broker agreed. Another advantage is the full transparency of the extension, because no RSVP user outside the ISP will have to change anything or even consider a RSVP to Differentiated Service mapping occurred.

Interaction between RDG and Bandwidth Broker

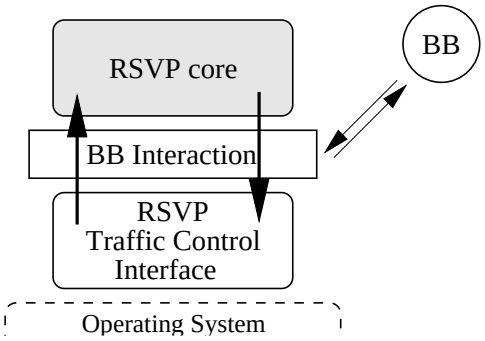


Figure 5.6: *Structure of the RSVP daemon with an additional bandwidth broker interaction layer between RSVP and the Traffic Control Components*

Boyle, Black e.a [BCD+99] propose to forward the RSVP objects are directly to a common open policy servive (COPS) without any further processing. The RSVP Differentiated Services gateway presented here extracts the required information out of the RSVP objects and uses a couple of human readable commands to communicate with the broker. This simplifies debugging during development and performance evaluation. It also enables a simple interaction between different platforms. The actual used set of commands contains terms for user authentication, setup, deletion and modification of reservations.

During resource setup the extended RSVP daemon uses these commands to negotiate with the bandwidth broker as shown in step 4 in Figure 5.4).

Commands used for the RDG/Bandwidth Broker communication	
hereis	used by the RDG to log in the broker and to exchange fundamental information like addresses and authorisation keys
newflow	tells the broker to set up the reservation for a new flow. The flow is specified by source and destination addresses or networks. The broker will return whether the reservation was successful or not.
delflow	tears down a previously set up reservation
modflow	modifies the bandwidth of an existing flow

Another advantage of such a simple protocol is the easy conversion from RDG requests to router configuration commands the Bandwidth Broker has to perform.

It should be mentioned that the RDG does no further internal processing, except the interaction with the broker. It simply waits for the answer of the broker and admits or rejects the flow accordingly. So only the broker decides how the request has to be handled. This includes:

Policy Control: Is the source or destination permitted to perform the request? The RDG may of course store some information about the SLA's of the according user to prevent broker interaction for each single reservation and do some over provisioning in allocating resources to gain more local autonomy.

Accounting: How much has the user to pay for the reservation ?

Reservation Method: In most cases the reserved flows are aggregated to large tunnels between the border routers, but it is also possible to map a reservation to an own tunnel.

Resizing of tunnels: The RDG might have different strategies in allocating resources at the ISP. So the RDG might not query the bandwidth broker for every single reservation, but allocate more bandwidth than actually needed. So he can meet reservation requests without negotiating with the broker.

Bandwidth Reservation in the Backbone

The mechanisms used to communicate between the RDG and the broker are completely independent from the concepts used for the final bandwidth reservation in the backbone. In the prototype IP tunnels with a certain QoS are established between the two routers. These tunnels are set up once, but reconfigured due to the actual requirements. The resource reservation for the tunnels is accomplished by Differentiated Services.

5.3.4 Evaluation

The graph in Figure 5.7 shows the achieved bandwidth of an UDP flow, transmitted over a RDG through a Differentiated Service domain as shown in Figure 5.5. The resource reservation was triggered by RSVP requests with several modifications of the reservation. The flow got the reserved bandwidth despite a parallel UDP flow causing heavy congestion. The two pictures in Figure 5.8 show the influence of bandwidth reservation on a motion jpeg video transmission. The left picture was transmitted with, the right one without bandwidth reservation. The lower picture quality, resulting from missing UDP packets is obvious.

5.4 Limitations of Classical Service Management

Integrated Services allow per micro-flow resource allocation, but do not scale to the core Internet. Resource reservation in the core is more likely to be deployed using the Differentiated Services architecture. However, the requirements of user applications are better met by Integrated Services. To address this conflict mechanisms can be installed within the network mapping RSVP to Differentiated Services reservations. This requires an extended RSVP daemon (the RSVP-Differentiated Service gateway - RDG) monitoring the RSVP signals and interacting with a central bandwidth broker to set up the requires resources. The measurements showed such a mechanism is able

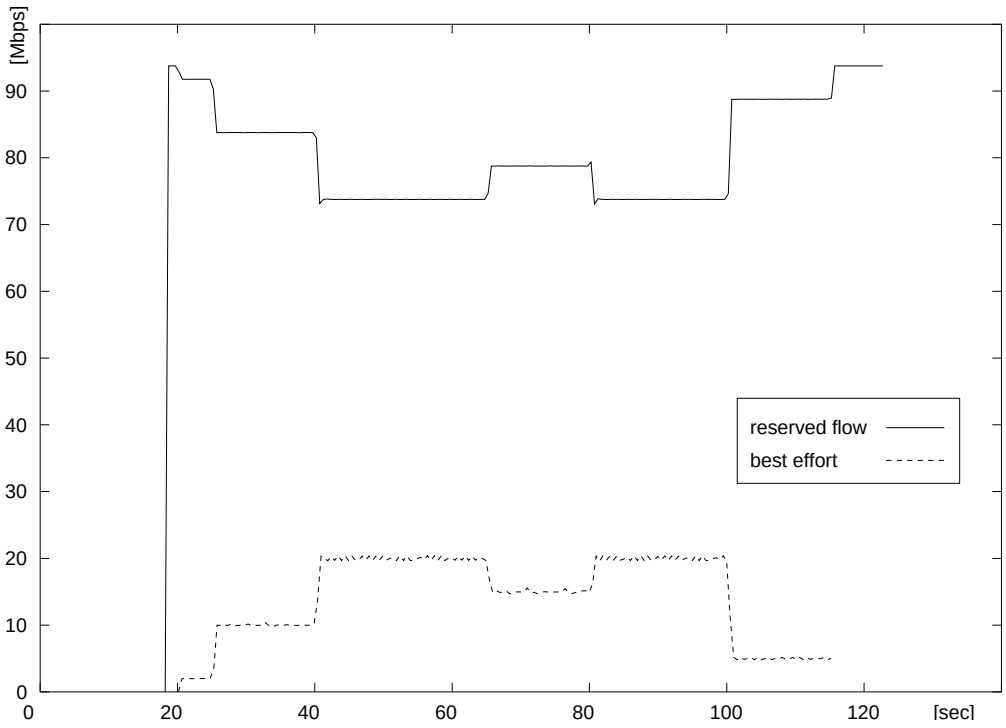


Figure 5.7: *Bandwidth of UDP flows with and without RSVP triggered bandwidth reservation*

Figure 5.8: *motion jpeg video transmission with and without bandwidth reservation. The picture errors occur from dropped UDP packets*

to map and aggregate incoming RSVP to scalable Differentiated Services. However, such an approach raises some problems:

- A central bandwidth broker does not scale well. Within larger networks a central management might become impossible. A distribution of brokers is complex, because network reconfigurations might interfere.
- Even if the frequency of tunnel reconfigurations may be decreased by over provisioning, at least periodic adaptions of the tunnels are necessary. Since even Differentiated Services supported tunnels might require adaptions of a core router's configuration, the overhead would be tremendous. A broker would have to calculate the path a tunnel takes through the network and configure the involved routers.
- To determine the current resource consumption at a certain router on a path through the network, the traffic would have to be measured. Current mechanisms are heavy weighted and increase the overhead further by transporting measurement data over the network.
- It's not trivial to distribute a central instance like a bandwidth broker, since concurrent configuration requests have to be coordinated to prohibit interferences between multiple brokers.
- To cope with the scalability behaviour a more autonomous behaviour would be convenient. Several activities could be solved locally and a central instance only queried, if absolutely necessary.

To configure large networks a mechanism is required, allowing to configure devices along a specific path, without negotiating with a central instance. Such a mechanism should be able to support heterogeneous platforms and allow to set up functionalities required to map reservation types or to set up tunnels automatically with no knowledge of the entire network.

Chapter 6

Active Quality of Service Management

The limitations of current management mechanisms to provide an adaptable system to configure and control services within an IP network enforce the development of new strategies of network management.

The supervision of a network from a central point is not feasible for networks of a large scale. In a network with several thousands nodes a higher degree of autonomous behaviour from the network itself would be desirable.

Especially to process tasks as local as possible without or with a minimum of control from a central instance only would simplify the supervision of networks significantly. Of course not all tasks can be performed on a local level, since a high degree of control may be necessary or a network wide synchronisation is inevitable.

On the other hand in large networks necessary adaptations may only affect a rather small part of the network, reconfigurations for different parts can therefore be done in parallel. A central instance would either have to perform configuration requests sequentially or to determine which adaptations might interfere to allow a parallel execution. In addition to the high computational overhead, a central topology database would be necessary, which is not feasible for large networks.

Furthermore a less central management of resources, mechanisms are needed to add new services dynamically to the network. For competing Internet Service Providers the time needed to provide a new service to their customers is essential. This is why an architecture allowing the dynamic and quick establishment of new Internet services is important.

To be more concrete the following points show the tasks to be covered by such a system. Of course this is just a short list focused on Quality of Service issues. A general approach for Quality of Service Management will also support other management functionalities.

Information collection During SLA negotiation and admission control it is necessary to get concrete information about the available resources at certain nodes within a network. Since the data usually depends on the current load and of course on

the network topology, a central database can only be used within small networks. A central approach would also require continuous monitoring causing a large overhead due to the transmission of measurement data.

The Service Level Agreement for a leased line like service provided by Expedited Forwarding contains usually the source and destination addresses of the connection. Since only a small part of an Internet Service Provider's network has to be checked, a mechanism collecting information along a specific path through an ISPs network would be profitable.

Network Configuration As described in the previous chapter the setup of tunnels can be a convenient way for the aggregation of traffic simplifying the resource provisioning in the network's core. The setup of tunnels and their occasional reconfiguration also takes place only along a certain path through the network. Therefore either the tunnel start and end points interact with a bandwidth/tunnel broker configuring the network devices or become active by themselves by signalling directly the necessary reconfigurations along the tunnel path. While the use of a central broker requires a database with the network topology and a lot of top down communication between broker and the network, signalling along the tunnel path can provide a much more light weight solution.

Even if the Resource Reservation Setup Protocol is also based on device configurations along a specific network path, there are several important differences:

- RSVP stores flow state information within each router of the core network. For millions of flows RSVP will not work properly. Using Differentiated Services no flow information within the intermediate routers has to be stored. The signalling just takes place only occasionally to adjust settings within the core routers, independently from the number of flows.

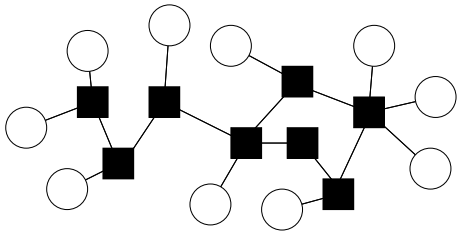
- RSVP is rather inflexible. For the creation of new services or a smart mapping between different reservation types, a more flexible mechanism should be chosen, that allows to add new functionalities without changing network devices. A flexible and extensible approach could integrate signalling, tunnel set up processes and also provide the algorithms used in the involved systems.

The most obvious way to provide such flexible systems is the exchange of program code between the network's nodes. The program code is executed by a network device, reconfiguring the router, collecting information or sending data and program code to other devices. Networks providing such mechanisms are called "Active Networks". In the following chapter an overview over the different types of Active Networks shall be given. Such an approach is even more powerful since the differences between classical network management and signalling get smaller. Active Networks can not only be used to provide a high degree of control over the network but can easily be used to implement lightweight signalling mechanisms, needed for the control of large networks.

6.1 Active Networking

From the Internet end systems point of view the actual design of the Internet forms some kind of black box. End systems simply transmit and receive passive data packets. Network devices between the end systems check the destination addresses of the packets and forward the packets according to an internal routing table. There is no control whether a packet gets lost or is delayed.

Of course there are protocols like OSPF or BGP4 to synchronise the routing tables between the Internet routers, but there are either very limited mechanisms to control packet forwarding within the core network or none at all. The treatment of packets is rather static and there are no simple mechanisms to add new types of services. Therefore even due to the complicated protocols running on core routers, the treatment of single packets is rather ”dumb”.



Following this concept most of the ”intelligence” is located in the end systems of the network. If a reliable connection has to be set up both end systems have to care about duplicated or lost packets, retransmission timeouts, connection setup or tear down.

This has a lot of advantages regarding scalability as there are a lot of end systems and a very limited number of core routers. Unfortunately, the lack of flexibility of core routers complicates the creation and establishment of additional services.

In contrast to todays Internet, Active Networks do not just forward packets, but may apply very specific packet treatments. There are several approaches for active networking with several degrees of ”activity”.

The capsule approach as presented by Tennenhouse [Ten97] uses packets consisting of a short piece of code and of additional payload. An active router receives the capsule and executes the code. This code may simply be used to route the capsule to the destination or to configure the active router on the path. Also new functionalities can be added to the device.

On the other hand approaches like the ”programmable switch” focus mainly on mechanisms for better signalling and device configuration [Ale97].

The execution of code on network devices causes several problems. Especially the capsule approach raises two main problems, that are described in the following sections.

6.1.1 Performance

In the current Internet the core network devices are optimised to perform a couple of very simple functionalities at a very high speed. Therefore a core router allows to

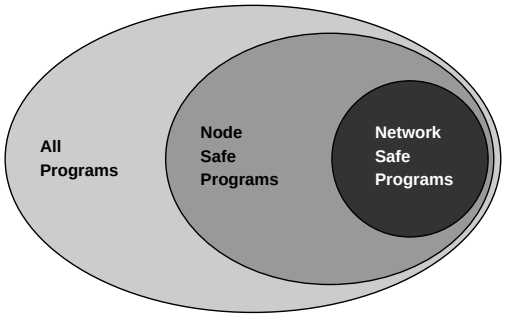


Figure 6.1: *Different levels of safeness for programs in an Active Network*

handle multiple interfaces and speeds of several millions packets per second.

Even if there are approaches to build high speed active networking switches as described by [DPC+99], such a device has to perform much more generic calculations than a common switch, increasing the complexity and therefore the price for such a device.

As an alternative, a mixture of active and passive packets can be transported. The passive packets are forwarded by the switch using the highly optimised mechanisms, whereas active packets are processed separately. The processing of active packets may even be done outside the switch itself by an additional device. This approach has two advantages:

- Active Networks offer unique methods of network management. This can make current signalling and management traffic superfluous, because such tasks can be handled smarter and simpler by Active Networking methods.
- If only a small part of the data to be forwarded by an Internet router is active, the overhead of processing these active packets will be acceptable.

This hybrid approach has also the advantage, that active packet processing does not need to be performed within the forwarding device, but can be accomplished by an additional component. Also this active components may only be located at important points of the network.

6.1.2 Security

Another, more sensible problem is the security of a node and of the whole network.

There is a wide range of Active Networking concepts. Some allow end users to inject mobile code, while others apply AN for management tasks only. Dependent on the degree of ”activity”, the security problems vary. Figure 6.1 illustrates different types of safe programs and therefore emphasises of security problems arising within an Active Network.

node security: A node has to control the amount of memory and CPU load a active packet requires for its execution. Additionally, both authorisation and authentification of a capsule have to be checked.

network security: Misbehaving code may damage the network even if this code behaves correct on the node itself. Therefore a distributed permission and resource control mechanism has to be installed on the network level.

capsule security: A capsule has to be save against manipulations by the node itself. Capsules may contain encryption keys or other confidential information. In contrast to normal packets, which might be encrypted to prevent eavesdropping, active packets need to be executed making encryption complicated.

Of course the impact of these problems depends on the type of AN. In a ”pure” active network with end systems sending active packets through the network, the security issues are much more crucial than in a scenario where AN technology is just applied for network management and therefore only accessible for network administrators or very privileged users.

However, several approaches to cope with the security issues exist. Brunner suggests in [BS99] to create Virtual Active Networks on top of a network. As in virtual private networks (VPNs), a customer can ”rent” such a virtual active network. Within its active network, the customer (e.g. a company) can exploit AN technology but has no access to the other virtual networks.

A similar but even more general approach is the X-Bone [Tou00]. The X-Bone is a software system that configures overlay networks also known as VPNs. It uses a web-based user interface to discover, configure and deploy an overlay network. The X-Bone installs routes, configures interfaces, updates DNS entries and installs encryption keys. Similar to the approach of Virtual Active Networks, the X-Bone allows to set up a virtual network topology that can be used as a basis for an Active Network.

A secure active network environment (SANE) is proposed in [AAKS98], focusing on a trust architecture covering even more fundamental security issues like a secure bootstrap mode. It checks also a node’s runtime system to provide a trustworthy active node.

An application of Active Networking which is interesting yet today is the application of AN methods for Network Management and especially for Quality of Service issues. There are several good arguments for the use of active networks in that area.

Performance: The overhead required for active networking is compensated by saving network resources currently used for management. Also, the ratio ”active/passive” traffic is very small.

Scalability: Active Networks can provide scalable solutions, for which traditional Internet techniques can not be used.

Security: Since active code is used only for management and is therefore restricted to authorised people, simple security mechanisms can be applied to restrict access to vulnerable functionalities.

Flexibility: The active components can be used for adding new functionalities to the network. Therefore the network can evolve during operation.

ANEP Header

Version	Flags	Type id
ANEP Header Length		ANEP Packet Length
Options		
Payload		

ANEP Header Options

FLG	Option Type	Option Length
Option Payload		

Table 6.1: *Header and option format of the Active Networking Encapsulation Protocol (ANEP)*

Heterogeneity: Since active packets can adapt to different platforms, different devices can be integrated and managed by the same network.

6.1.3 Active Packet Formats

The choice of an appropriate packet format is very important for the capabilities or to extend the system later. The following paragraphs will present the Active Networking Encapsulation Protocol (ANEP) [Ale], which was designed to support different Active Networking Systems.

Table 6.1 shows the header format used by the Active Networking Encapsulation Protocol.

The ANEP header usually is encapsulated in an IP header during transport. A router may receive ANEP packets directly sent to it or packets forwarded with the router alert option. The format of the options is obvious. The FLG field in the options header signals that the option is an implementation specific option an shall be ignored by other systems.

Even if the ANEP protocol is rather flexible it contains several header fields, that require a special parsing. Since the header was designed to allow a simple processing by native code written in C or C++, ANEP is not very convenient for the use within high level active networking languages. Therefore, the active network system presented later in this chapter will use a different header format, providing more flexibility, but it can be extended to support also ANEP.

6.1.4 Interpreted and Native Code

As already mentioned in the introduction to this chapter, the system allows the transport, installation and execution of native code.

The dynamic installation of code is based on loading pre-compiled C++ classes to the runtime system, similar to the proposal of Hjálmtýsson [HG98]. In addition to speed and performance, the use of native code can offer platform dependent programs that are required to add specific frequently used mechanisms to network devices.

The fundamental idea is not only the configuration of routers on specific routes through a network or the configuration of certain hosts. Because of performance reasons it is important to offer a possibility to enhance network devices with functionalities that can only be provided by native code.

One or more binary objects can be encapsulated within a packet, which may be transported through the network. A short piece of code checks, which type of program is required and installs the appropriate module. It is of course also possible to provide a platform independent default version of such a module. This way it is possible to provide a solution for any active device within a network. Either a proper native piece of code is available or the slow platform independent version is used.

6.2 The Python Based Active Router

In this chapter a lightweight but powerful Active Networking platform shall be presented. The architecture is based on the object oriented language Python, offering flexible and powerful mechanisms for a good integration into a device's operating system.

6.2.1 The Python Programming Language

The properties of a language suitable for active networking are still under discussion. A well known and frequently used language for agent systems is Sun Microsystems Java. Several Active Networking platforms like the Active Network Transport System (ANTS) [WGT98] have been written in Java. There are several reasons for Java to be used as a platform dealing with mobile code.

Security: In contrast to many other languages, Java provides internal security mechanisms. Programs are prohibited to damage the local machine. The access to the local computer programs can be restricted. Especially for a system that has to execute foreign code, access restriction is absolutely fundamental. Java provides such mechanisms allowing to block the access to certain resources like file systems.

Platform independence: Java is based on a virtual machine executing the programs and controlling the access to system resources. This allows any Java program to be executable without any re-compilation on any other Java capable platform. Unfortunately this portability suffers from version updates and not perfectly compatible virtual machines causing some Java programs to run only on certain platforms with special versions of Java virtual machines. Fortunately this lack of compatibility usually does not affect the Java core mechanisms providing good portability for standard programs.

Network Support: The need of a language with direct support of computer networks was one of the reasons leading to the development of Java. All standard Internet protocols are supported by Java.

Unfortunately Java has also a couple of drawbacks, which cause problems especially for tasks concerning the management of network devices.

Platform Interaction: Because of the system independence as a central design rule, a direct interaction with the operating system is complicated. Even if the Java Native Interface (JNI) allows to combine native code with Java programs, the interaction is not as seamless as desired and does not allow to change the behaviour of Java itself. For an active networking platform intended to configure network devices a simple and flexible interaction with the router hardware itself is required.

Network Support: Since the domain of Java is more the application area, Java was not especially designed to handle low level data like raw IP packets. Java supports higher level protocols very well, but is limited if a procession of packets on IP level is required.

Resource Consumption: Another drawback of Java is the excessive resource consumption. The comparison in [Lef00] especially criticises the memory consumption of Java requiring lots of memory even for small programs. Such a behaviour is not acceptable, especially if small pieces of code within Internet devices with probably rather scarce resources have to be executed.

C++ like Syntax: Even if user friendly interfaces might exist, network management will require humans to at least adapt short programs. Java uses a C++ style syntax, which is great for writing large applications but is not suited to create short configuration scripts.

Fortunately another programming language called Python is available. Python combines several aspects of Java with much more flexibility and more possibilities for the developer to adapt the language to her specific needs. In the following, a short overview over the capabilities of Python will be given (see also [WRA96]).

Interpreted: Python [pyt] is a dynamically interpreted language that supports byte compilation. Python code is platform independent, allowing to exchange applications or code between different platforms.

Object Orientation: Python is an object oriented language. It supports class structures with multiple inheritance and late bindings. Python also offers an application mechanism to modify Python's own fundamental mechanisms. Therefore an application might define an alternate object model or add and replace new implementations for each fundamental component.

Dynamic: Dynamic typing and dynamic name resolution allows to reduce program size, simplifies debugging and makes code reusable.

Orthogonal structure: In contrast to other languages Python just uses a very small set of powerful constructs. These constructs allow programmers to understand foreign code more easily. Of course this also allows to learn the Python language very quickly.

Extensibility: One of the biggest advantages of Python is its extensibility. Python allows to integrate external libraries or data types by modules. These modules may either be written in Python or in an other programming language like C or C++. There are hundreds of modules available providing special functionalities like WWW or database access, mathematical calculations, etc.

Even if just one module written in Python itself preserves platform independence most of the modules written in other languages can be made available by a simple compilation on the new platform.

On the other hand these platform specific modules also have two significant advantages:

Platform Access: Other languages using virtual machines prohibit access to the local computer or at least make it very complicated. Python allows to be extended by native modules allowing any access to the computer a language like C or C++ can give.

Speed: For time consuming calculations, modules providing native code can increase the performance dramatically. Python also offers the flexibility to use native modules if available and use pure Python modules as a fall back.

Embeddable: Python offers an application programming interface which allows the programmer to embed Python easily in own applications and use it as a general purpose scripting language or control tool.

Security First of all, Python code is executed by a Virtual machine, limiting the damage a Python program can do to the machine. Additionally, Python support restricted execution environments, allowing to define a ”cell” and execute foreign code within this environment. Interaction with the local computer has to follow rules the designer of the environment has defined in advance.

Compatibility: Even if the development of Python is ongoing, the changes of the language's core are always compatible to programs written for older versions of Python. Also, most of the updates do not primarily affect the core.

Portability: Python itself is written in C and is easily portable to all platforms using Posix conventions. Currently, Python is available for all major operating systems including systems with very limited resources like palm tops.

Freely available: Regarding the copyright, Aaron Watters writes in his book ”Internet Programming with Python” [WRA96]:

The Python copyright essentially protects the authors from legal jeopardy and prevents malicious users from attempting to hijack the copyright. Aside from that, Python programmers and users may use Python in source or binary form just about anyway they please. In particular, programmers may create products that use Python and release the product in binary-only form with all Python modules in byte-compiled-only form, and they may sell or give away the result in any manner they think will make them the wealthiest, or the most famous.

Of course there are a lot of similarities between Python and other platform independent languages like Java, TCL, Perl or shell scripts and there is probably no particular reason to use Python. However, there are some arguments favouring Python:

- Java is object oriented only, while TCL cannot provide any object orientation without extensions. Python allows both. While for smaller programs script like programs may be used object oriented structures are also provided.

- Python forces the programmer to use a special source code layout, as whitespace characters and text indents are used to mark program blocks and data structures as the short "Hello World" program demonstrates.

  ```
  def HelloWorld():
          print 'Hello World'


  HelloWorld()
  ```

  The short program defines and calls a function printing the 'Hello World' string. At a first glance the use of whitespace characters might appear strange to programmers used to other languages. On the other hand the format expected by Python fits mostly the style experienced programmers will use anyway and increases the readability of the source code significantly. Therefore the source code of other programmers can usually be understood and reused quite easily.

- The embedding and extension of Python is simpler, more flexible and nevertheless rather straightforward than the Java Native Interface or the mechanisms provided by TCL. The overhead to use C, C++, libraries or program code with Python is minimal. This is also the reason, why there are a large number of C or C++ bindings already available for Python, reaching from image manipulation and mathematical libraries to graphical front-ends. These bindings allow to perform even time consuming data processing with Python.

- Besides the strange source code format, the Python uses a more scripting language like style but nevertheless provides all functionalities of a modern language.

  The architecture proposed here uses the active networking concept to distribute small programs performing mainly controlling and monitoring tasks. Light weight short scripts as possible with Python are perfectly suited for that tasks, while Java with its rather C++ like syntax is too over sized.

- The final, perhaps most striking reason to choose Python is its comparable low resource consumption. On a computer running the Solaris operating system even the start up of the Java virtual machine consumes more than 32 Megabyte of memory, while Python only requires five Megabytes of memory. The availability of Python for palm top devices with only 2 Megabytes of total memory shows the capacity to minimise Python's resource consumption.

A detailed performance comparison between Python and Java is presented in [Lef00]. Since Python is an interpreted language like Java, both languages can provide a very limited performance only. The evaluations of [Lef00] revealed that unfortunately the IO performance is one of Java's weakest points. Of course there are other points like the general speed of code execution or object serialisation, where Java slightly outperforms Python.

Since especially fast input and output is important for network devices, the Python language seems to be the better choice, especially if high level functions frequently used by active packets are provided by native code.

During the design of the system, the central goal was neither the development of a multi purpose architecture allowing any participant in a network to distribute code and implement special services, nor the implementation of a high level Mobile Agent platform. The focus of the work described here is the management of networks and especially the support of quality of service. This specialisation of course affects the design of the system.

6.2.2 Outline of the Active Network System

Before describing the system in more detail, a short overview of the capabilities and limitations of the proposed and implemented architecture shall be given.

Execution of Mobile Code: This is a self evident property of any Active Networking system. The developed system allows to transport and execute platform independent byte code mixed with native code proprietary to a certain platform. The platform dependency of native code can be handled by the provision of multiple pieces of code, each for a specific platform. The different code segments can be included within the same active packet, providing efficient code for different platforms. Such a packet may also include a platform independent Python version of the same algorithm, which can be used as default.

Authentication of Code by Encryption Mechanisms: To block access for not authorised users, the system allows the use of encryption mechanisms. This encryption is provided by a special module allowing the flexible use of high speed built in functions. The transported code itself has influence on which parts of the code has to be encoded or might be seen by third persons on the network. The system provides both encryption and authentification by digital signatures.

Access to System Resources for Network Management: To accomplish its tasks, executed code requires access to the network device internals. Even if the program code is portable between platforms, the design of network equipment might differ significantly. Because of the evolving capabilities of network devices, a unified interface seems to be quite impossible. Instead of providing such a heavy weight programming interface, specialised libraries can be installed by mobile code. For example a library

for the simple set up and control of Differentiated Services flows can be installed on heterogeneous platforms, providing a unique interface to other programs.

Low Level IO Functionality for Portability and Performance: Since the setup of connections during the forwarding of a packet is time consuming, the transport of code is done connectionless either using raw IP or a protocol like UDP. The risk of packet loss can be minimised by the use of a specialised Differentiated Services traffic class with high priority. The maximum allowed packet size of 64 Kbyte is absolutely sufficient because Python code is reasonably small. Larger programs can be split up into several parts or previously installed libraries can be used.

Piggy backed Transport of Data: The active packets sent over the network can be used to transport any kind of data by appending the data to the active part of the packet. This allows the simple transport of measurement data or program code. This powerful and flexible mechanism may be used for various applications:

- Native code can easily be installed to certain hosts. The packet may contain multiple versions for different platforms.
- The executable head of the packet can contain some routing mechanism to force the packet to a special path through the network. That allows to route special data along a certain path without changing the routing tables on each router.
- Extensions for the active platform can be transported and installed.

Since the task of the system is not the secure execution of code provided by end users, but to provide mechanisms for management related issues, the system has of course some limitations.

- The system provides detailed access control to system resources. Authorised code is allowed to access any system resource. Of course it would be possible to restrict system resources to specific groups. The implemented system is thought of as a tool to allow a simple and efficient configuration of network devices and to apply special services the systems. It is not designed to be used by end users. Unauthorised access to the system is blocked by encryption mechanisms.
- The Python language provides ”restricted execution environments”, which allow to limit the (OS-) functions code can access. As the group of users allowed to access the system is equal to the group having access to the network devices itself, the examples in this chapter will not use these mechanisms.

6.2.3 The PyBAR Architecture

The design of the Active Network platform tries to separate the active components as far as possible from the conventional router functionalities. Such a separation ensures portability, scalability and also allows the integration of existing devices.

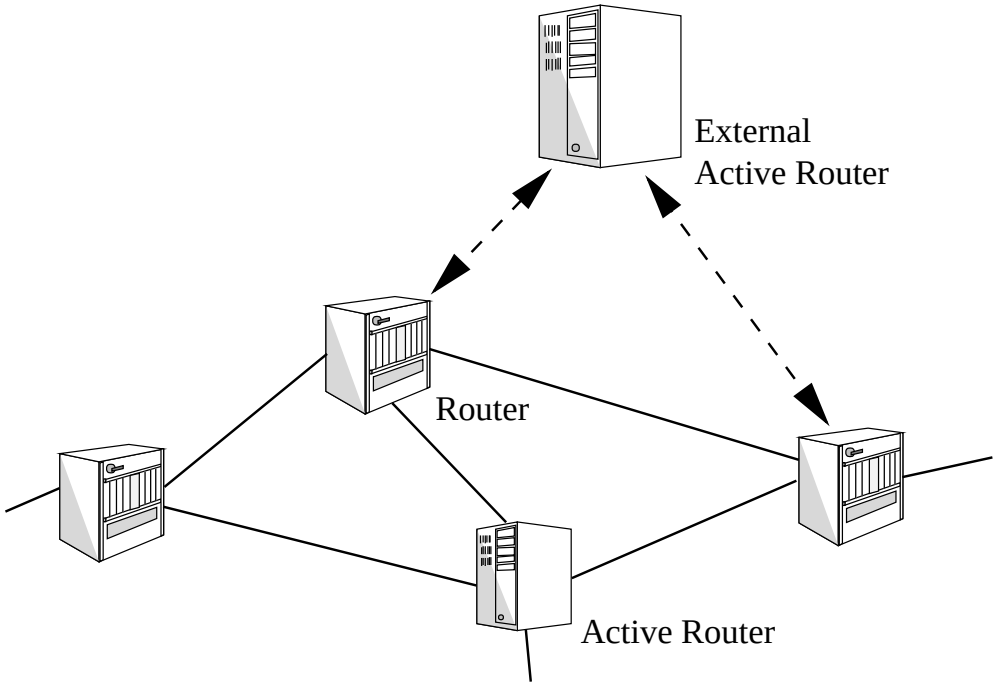


Figure 6.2: *A network with active routers. The active component of a router may either be run directly on the hardware (e.g. Linux) or be attached externally.*

Therefore the active part communicates with the routing and forwarding mechanism over a single duplex channel only. Figure 6.2 shows that the processing of active packets can either take place directly within the network device or externally in a specific host, being connected to the router with such a communication channel. This allows to set up even clustered processors, providing ”activity” to multiple routers in the network.

The PyBAR active router is designed to be portable easily for different platforms. As mentioned before the goal is not to provide a multi purpose platform but a framework or toolkit to be adapted to the actual needs.

The system consists of several parts, which will be presented in the next paragraphs. The PyBAR is based on the standard Python virtual machine. Additionally the standard Python command set was extended by several modules written in C. These extension modules provide optimised mechanisms for frequently used operations like encryption or also for packet processing. While Python provides flexibility, the C extensions add the required speed. Those extensions are not limited to C functions but can provide complete data types. Data types are important, since the programs have to be small and readable. For example an IP packet data type is provided by a C extension module, which simplifies and speeds up the analysis and processing of incoming IP packets.

For the programmer it does not matter, whether such an extension module consists of Python code or whether it is written in C. No specific initialisation is needed, neither to load the C, nor to load the Python module. If C and Python versions of the same module are available, an active packet can use these modules without taking care about

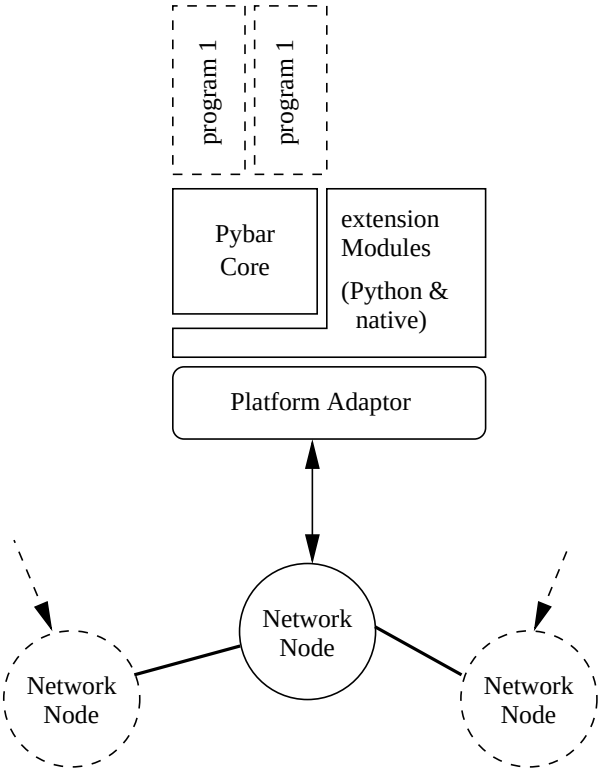


Figure 6.3: *Basic architecture of the PyBAR system.*

calling a C or a Python function. Of course such extension modules can be added dynamically to the PyBAR system, making new data types and functions available.

Figure 6.3 shows the general architecture. The platform adaptor is an extension module written in C and provides communication facilities for the PyBAR core and the extension modules. The core processes incoming (active) packets and is supported by a library of extension modules. The following paragraphs will describe the components in detail.

The Platform Adaptor

The Platform Adaptor serves as a kind of node operating system and has entirely been implemented in C. It provides a set of Python commands and data structures.

The commands provided by this layer are rather fundamental and might also be platform dependent. The platform adaptor supports the PyBAR environment with specialised data types and methods, simplifying and speeding up the processing of received data.

This layer also controls and manages the communication with the router below. The router is set up to forward specific packets to the PyBAR. Of course the PyBAR can control which packets are sent allowing to filter and process certain packet streams. Basically the system will be interested in packets containing code to be executed by the PyBAR. As a default three types of addressing will be supported:

Direct Addressing: The packet has one of the router's addresses and is forwarded

to the PyBAR for further processing. This mechanism is especially useful to address specific end systems or to send an active packet directly to a special active router.

Router Alert: The router alert option is an IP option, indicating a router to treat the packet in a special way. There is no information in the Router Alert option how the packet shall be processed in special. The Router Alert option was introduced in conjunction with the RSVP protocol and has the purpose to send a packet to a specific destination and trigger certain functionalities in the path.

An IP packet with the IP router alert option set and an ANEP payload will be forwarded hop by hop to the destination of the IP packet like any normal packet. But instead of being only forwarded to the destination, the packet is processed by each active router.

DSCP: Since the processing of IP options decreases the performance in conventional network devices, a special DSCP value can be used instead of the Router Alert option. Active routers forward these packets to the active components while conventional devices will simply ignore this DSCP. Besides performance this has another advantage. Since packets with such a DSCP can be handled preferential by Differentiated Service routers, the loss of active packets can be omitted. Of course multiple DSCPs may be used as well. As an example there might be a DSCP value for active signalling packets and another for active packets carrying data.

Of course other addressing mechanisms can be used, forcing the platform adaptor to look out for broad– or multicast addresses or for other packet properties.

The Platform Adaptor provides a set of mechanisms allowing the core to interact with the platform. Table 6.2 lists some of the implemented commands.

Of course the table does not show the complete command set, but as can be seen there are different commands to access the traffic conditioning components for Linux or Virtual Routers. Programs which want to modify these components have to check what commands are available. Another possibility is to use a library installed previously providing uniform commands.

The PyBAR Core

The core is responsible for the treatment of received packets. As it will be described in Section 6.2.7, not all active packets have to contain executable code, but may only require some processing by a module of the PyBAR.

This core is mainly a kind of framework, which can be adapted to different needs and allows to provide very different mechanisms.

Within the system type described here in more detail, the core performs the following mechanisms:

- The core checks the type of the incoming packet and handles it accordingly.

command	Linux	VR	Description
version	yes	yes	returns information about the platform the system is currently running on
getCaps	yes	yes	returns capabilities of the Internet router, the PyBAR is attached to
attach	yes	yes	connects the PyBAR system to an Internet routing device
getInterfaces	yes	yes	returns a list of interfaces.
getRoutingtable	yes	yes	returns a list of routes.
addRoute	yes	yes	adds a routing rule to the routing table
delRoute	yes	yes	deletes a route from the routing table
getQcomps	no	yes	query information about the traffic conditioning system
addQcomp	no	yes	add a queueing component
linkQcomp	no	yes	connect two queueing components
queryQcomp	no	yes	query information for a specific component
configQcomp	no	yes	configure a component
configTC	yes	no	configuration of the traffic conditioning components

Table 6.2: *Examples for commands provided by the Platform Adaptor. For the Virtual Router queueing commands see also Section 4.1.2*

- Executable packets are decrypted and decompressed. If the decryption was successful, the code in the packet is executed. For the de- and encryption a special cryptography module is used. To provide a powerful but nevertheless easy way to use the mechanism, a special Python module (PyRSA) was implemented, providing RSA based cryptography. Section 6.2.5 will provide a more detailed description of this module.
- If a received packet contains executable code, the core supervises the execution of the capsule. The core also performs a set of simple checks during the execution to avoid malicious behaviour. These checks provide no perfect security, but are merely thought to limit the damage caused by program errors.
- A small central database or stack is provided to allow capsules the storage of data for a longer time.
- Functions to install and replace modules within the library are provided.

In the description of the Python language the ”high level” of the Python language was mentioned. Even if some of the tasks like cryptography require rather complex algorithms, which usually make things more complicated, the core itself is very simply. Complex functions like cryptography or the packet analyser are implemented as an extension module (written in C) and provide a very high level interface to the user.

This separation into modules emphasises the framework character of the PyBAR system. The core itself is a kind of default only, which can be adapted or extended to the actual needs. Of course adaptions and extensions can be performed by the core itself.

6.2.4 Extension Modules

Since the hard coded commands provided by the PyBAR only cover very fundamental tasks and additionally depend on the type of platform the PyBAR is attached to, more complex issues can be covered by extension modules.

Extension modules can be written in C or in Python. Of course modules written in C are platform dependent and may require re-compilation. However, any of these modules can be added, replaced, or removed dynamically by active network mechanisms. Therefore a module can be distributed in several ways:

- An extension module may be part of the system itself and is included in the basic installation. For example the PyRSA module used for cryptography is such a module.
- Capsules processed by the PyBAR may include extension modules that are added to the system and provide functions, data types or information used by other programs.
- The core can provide mechanisms to download modules on demand and add them to the local module library.

A rather high level extension module to provide a uniform interface to setup resources will be presented in Section 6.4.

6.2.5 Security

As mentioned earlier, security is one of the biggest problems by building active networks. The security problems get extremely complicated if users are allowed to inject code into the network.

The PyBAR system is not intended to be used by the end user but by network administrators or automated monitoring systems only. This reduces the security problems significantly. Network administrators are usually authorised to have full access to network devices and can be assumed to be experienced enough to avoid configurations or activities damaging the network.

To control the behaviour of capsules the core can run a monitoring task supervising the resource consumption of capsules and terminate tasks consuming too much resources.

However there have to be some mechanisms to keep other persons from sending code to the network or inspect the content of active packets. Another important capability of such a system is to limit the damage program errors can cause.

To prevent unauthorised persons to send code to the network, a RSA based asymmetric encryption and signature system is used. Each active node will check the signature of an incoming capsule and discard the capsule immediately if the signature is not correct. Additionally the program code within the capsule may be encrypted.

For the authentification mechanism secure signatures based on the MD5 message digest algorithm and the RSA [RSA78] algorithm is used.

All those mechanisms are provided by the PyRSA [Bau00] Python extension module, which was implemented especially for the PyBAR system and provides high level access to cryptographic mechanisms. Since cryptographic mechanisms require a rather high processing speed, the module was implemented in C and uses the free RSA toolkit [Lab94]. This solution provides high usability with good performance due to the underlying C implementation.

The RSAREF library used for the implementation provides several cryptographic algorithms:

- RSA encryption and key generation, as defined by RSA Laboratories' Public Key Cryptography Standards (PKCS)
- MD2 and MD5 message digests
- DES (Data Encryption Standard) in cipher-block chaining mode
- Diffie-Hellman key agreement
- DESX, RSA Data Security's efficient, secure DES enhancement
- Triple-DES, for added security with three DES operations

Even if designed for it, the module is rather independent from the PyBAR system and might therefore also be used in other applications. The PyBAR system mainly uses authentification mechanisms provided by the system:

generate_keypair(...) Two RSA_KEY objects are returned containing the private and the public key. The only parameter `keylen` specifies the length of the keys (max. 1024 bit).

sign(...) returns a signature for the data in string, based on the private key and the digest algorithm `digest`. Currently allowed values for digest are `"MD5"`, `"MD2"`.

verify(...) checks whether a digital signature is valid.

encode(...) generates a symmetrical key and encodes the data in $string$ with this key and the specified encoding algorithm. The symmetrical key is then encoded by the RSA algorithm. Allowed values for alg are: `"des"`, `"desx"`, `"des2"` and `"des3"`. This function returns the encrypted symmetric key key_{enc} and the encoded data $string_{coded}$.

decode(...) decodes the enciphered data in $string_{coded}$ based on the algorithm alg. $enckey$ is a list containing (iv_{init}, key_{enc})

Even if the security mechanisms can not avoid programs to behave malicious, it is able to restrict secure access to a system to authorised users. Additionally the monitoring of tasks running on the active node can provide a reasonable level of stability against incorrect programs.

The implementation and design of the PyRSA module are good examples for the extensibility of the PyBAR platform. In a similar manner modules for the processing of special packet types may be developed and installed to the active routers. Instead of implementing the security mechanisms into the Platform Adapter directly a separate module is used, allowing to replace PyRSA by another module.

6.2.6 Packet Processing

Figure 6.4 shows the major steps during packet processing. Not all packets received by the PyBAR contain code, which has to be executed. Comparable to router plugins [DDPP98] certain modules can be installed for processing certain streams directly. These modules may be realised using the Python language itself or by native programs. While Python might be sufficient for monitoring, tasks requiring more processing power have to be performed by native code.

If a packet contains a program to be executed the entire packet has to pass consistence and authentification checks. While in the first step basic properties of the packet are controlled, the second one covers digital signatures and encryption related issues. After the packet has passed the consistence and authentication tests an execution environment is initialised and the code within the packet is executed.

Usually, the processing of executable packets is much slower and time consuming than the simple processing of a packet's payload. Furthermore, the treatment of such streams can be accomplished completely by the platform adaptor. If the code processing the stream packet is provided by a native module, the complete treatment of such a packet is "Python-free" and therefore reasonably fast.

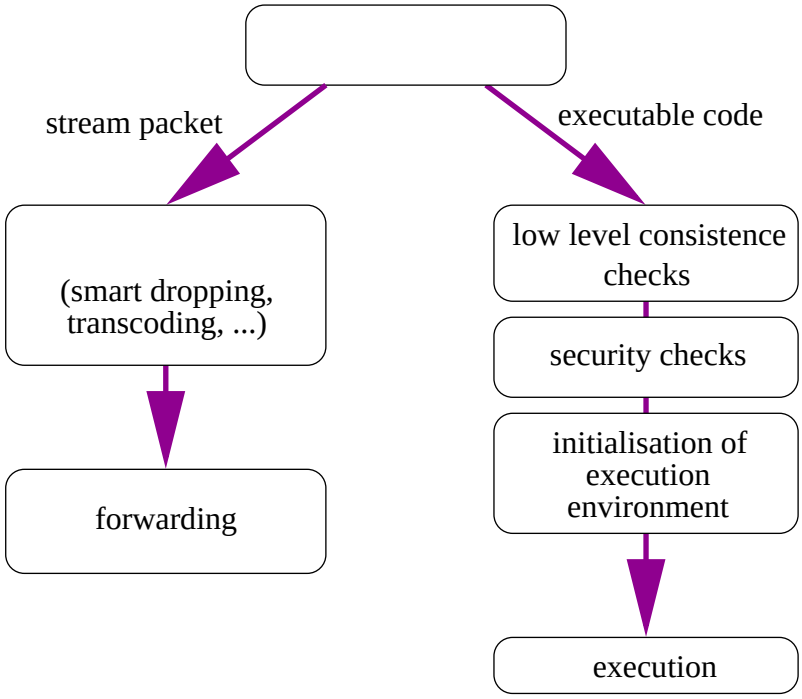


Figure 6.4: *Packet processing by the PyBAR*

A good performance is important for typical active network applications like video transcoding or reliable multicast. However, also Python based modules or modules using both native and Python code may be used for processing of certain packets. Even if the processing speed of Python is comparably slow compared to native code, it might be absolutely sufficient for less time consuming or time critical tasks. For an application specific dropping of packets (see Section 6.5.1) or network management related tasks like the set up of tunnels (see Section 6.5.3) the performance of Python is absolutely sufficient.

6.2.7 PyBAR Packet Format

In the section about active networking the Active Network Encapsulation Protocol (ANEP) format was also presented. Even if the ANEP header is flexible and may be used for different kinds of active network systems, the PyBAR system uses a different header, due to three reasons:

- The ANEP header was designed to support many different kinds of active network systems. For a rather specialised system the header does not fit very well and specific mechanisms of the PyBAR system are not supported very well.
- As mentioned before, the PyBAR system is thought as a framework or a toolkit and is therefore easily adaptable in various ways. Therefore the processing of ANEP headers can be added without great effort but should not be accomplished by the static components of the PyBAR system. For our rather specific system

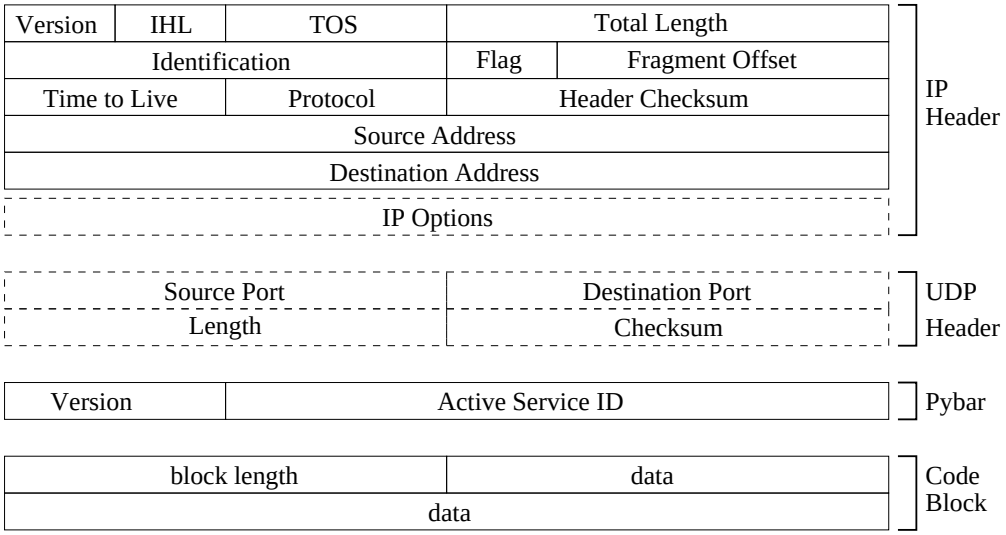


Figure 6.5: *PyBAR IP/UDP payload and a PyBAR code block. Several code blocks might be attached to the packet*

an ANEP header would cause too much overhead. A later version of PyBAR covering more general aspects might provide ANEP header processing.

- The ANEP header would be a static part of the active network system. Since most of it would be processed reasonably within the C based platform the flexibility of the active networking system would be limited. A more fundamental header leaves more possibilities about the packet format to the active networking system itself.

Therefore, the required PyBAR header is exactly four bytes long as can be seen in Figure 6.5 and can be encapsulated either in raw IP or in UDP. It only contains an eight bit wide field with version information and a 24 bit wide active service identifier. The latter one allows to notify which module of the PyBAR system shall be applied to the rest of the IP packet's payload.

A special service identifier with the value 0 is used if the rest of the packet contains executable code. Since this header is very small and simple, a stream packet has only be checked for the active service id and can be forwarded to the corresponding module responsible for this type of service. In section 6.5.1 a system will presented using this mechanism to provide smart packet dropping for a video application.

To signal the router that the packets have to be treated in a special way by the PyBAR system three mechanisms can be used:

Router Alert Option: This IP option signals a router to apply a special treatment to a packet. Packets with this option set could be checked for a packet containing an Active Service ID. Unfortunately, the Router Alert Option is already used for protocols like RSVP. Therefore the protocol id in the IP header has to be checked additionally.

Packet Filtering: The Active Router can set up filters listening to packets of specific protocols and addresses. Also some information of higher level protocols like UDP port numbers can be used. Besides the rather heavy weighted filter algorithms this does not provide a general mechanism to signal the router the special packet type, but it can be used by special PyBAR modules to process specific streams.

DSCP value: The use of specific DSCP value in the IP header can provide a unique signal for the router to check the packet for the special payload. The parsing of the special DSCP value is simple and since an ISP is allowed to use its own DSCP values, this mechanism conforms also the Differentiated Services Standard. In contrast to the Router Alert Option, this also allows to mark any packet for an ”active” treatment.

Which of those mechanisms is used is left to the PyBAR system. While for applications like smart packet dropping the use of a special DSCP value to trigger the special treatment of a packet is advantageous, the Router Alert Option can be used to exchange code or install additional modules to the router.

If the active service identifier signals an ”active” payload the rest of the payload has to consist of at least one code block. A code consists of a 16 bit wide length field and the code block's content only. Multiple code blocks may be contained within a packet.

The PyBAR system expects the first code block to contain executable code. The other code blocks are not processed. There are mainly performance and flexibility reasons for the use of multiple code blocks.

- The translation of binary data to Python byte-code is time consuming (as it is for Java byte-code). Since the packet might contain executable code and some payload a processing of the entire packet would be a waste of resources. Therefore, just the first code block is processed and executed.
- The PyBAR handles the format of executable code blocks within the python based core, providing the maximal possible flexibility.

Besides the code, the first executable code block also contains additional information like a signature. The payload format of the first code block is completely handled by the PyBAR system itself and can therefore be adjusted easily.

Of course the platform adaptor provides mechanisms allowing to access and manipulate the other code blocks in the packet. This includes the execution of code blocks and their installation as PyBAR library modules.

6.3 Injecting Active Packets: A Graphical Front-End

In the previous section the packet format required by the PyBAR system was presented. In addition to a very short header, active packets contain at least one code block. While the first code block has to contain executable data, the other code blocks can be used

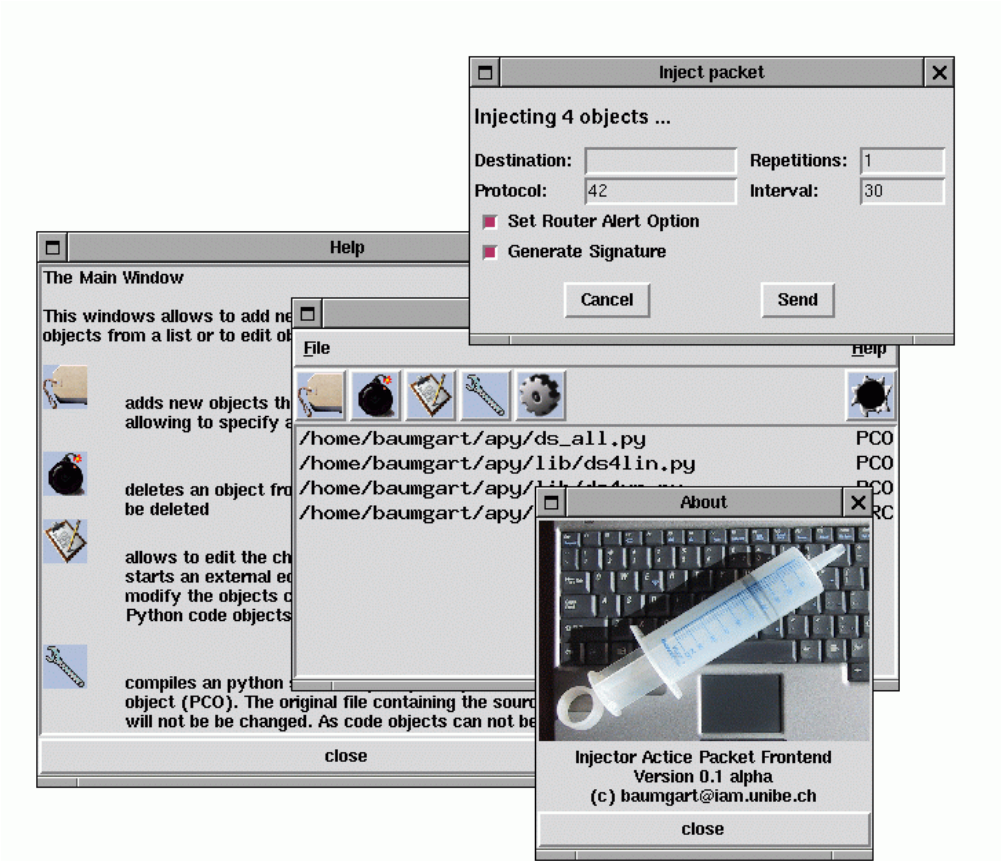


Figure 6.6: *Python and Tkinter based front-end for the injection and reception of Mobile Code*

to transport any payload. This mechanism is convenient for the installation of new library modules to active routers. Attached code blocks might contain modules written in native code for various platforms.

Of course this strategy might also be used to encapsulate other data or packets into an active packet.

To provide a convenient mechanism to compose and inject active packets with several code blocks an application based on Python and the Tkinter [tki] graphical toolkit was developed. Figure 6.6 shows a screen-shot of the program. The application allows to

- edit and compile the source code for executable code objects. Since at least the first code block of an active packet has to contain some executable code, the Injector program allows to load, to modify and to compile a piece of mobile code. The compilation is used to create platform independent Python byte-code.
- compose code blocks with extension modules. For the distribution of extension modules, the code block has to contain not only the code of the extension module but also a list of several objects, including the code and digital signatures. This ”encapsulation” of an extension module within such a structure is supported by the Injector program.

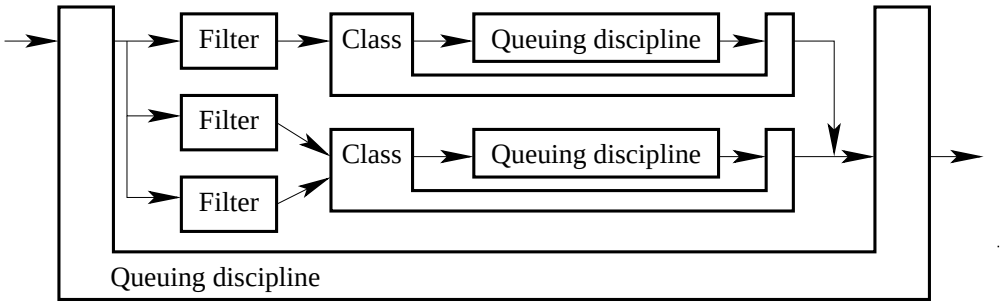


Figure 6.7: *Architecture used by the Linux tc implementation*

- add a signature to code objects to provide security. The signature of code blocks containing executable code or extension modules is checked before the code is executed or installed. These signatures can be created easily with the Injector program.
- compose a whole packet from single code blocks.
- receive and process feedback data. A capsule sent out to the network might collect data and send it data back to the Injector program. The Injector program can display this feedback data. Currently, this functionality is limited to showing some short text messages but can simply be extended to also show graphical information.
- configure the packet sender to send active packet repeatedly over the network.
- be easily adapted to other tasks. Since most of the application is written in Python and the same modules and functions are used as by the PyBAR itself, new mechanisms may be added easily.

6.4 Differentiated Services Support

A central problem within the Internet is the lack of a homogeneous configuration interface to network devices. Dependent from the vendor very different interfaces are provided.

Even more problematic than the interfaces are the fundamental design differences of devices. A good example for such different design concepts are traffic conditioning components. Linux usually uses the tc traffic control implementation [Alm99].

The architecture of the Linux traffic control system uses a system of nested boxes. Each box can contain other boxes. In contrast to such an approach other systems are more list oriented or use a graph like layout of their traffic conditioning components like the Virtual Router does.

These different concepts prohibit the use of low level configuration commands within a network. Instead of trying to design a multi purpose router configuration language the PyBAR system simply offers different commands for different types of platforms.

init()	sets up the complete traffic conditioning components required for DiffServ, with a an appropriate scheduler, EF and AF queues, Token bucket filters
setClassShares(...)	configures the bandwidth shares for the different traffic types
mark(...)	configures the Differentiated Services marker to mark specific flows with certain DSCPs
unmark(..)	removes a marker rule

Table 6.3: *Commands provided by the DiffServ module for the configuration of Differentiated Services resources.*

Since a capsule might be executed on multiple hosts and a distinction among the different command sets is not feasible, those low level commands are usually not used but library modules are installed on the system in advance.

A module providing methods to set up and maintain Differentiate Services on a router can map a command set to the low level configuration scripts. The functions provided by such a module may be on a rather high level.

- Functions to initialise and configure the basic system have to be provided. Some kind of init() function may set up the complete set of queues, schedulers and classifiers needed to provide Differentiated Services. Parameters of the init function may define whether an ingress, egress or intermediate router has to be set up.
- The marker mechanisms have to be configured to mark packets with different DSCPs. Functions to add or remove flow descriptions at the ingress routers have to be provided. Obviously, the mechanisms to apply such flow descriptions to the router may vary significantly. Therefore a high level interface would be beneficial.
- Since the Differentiated Services concept requires a separate handling of the different traffic classes, each traffic class has to be configured for a certain share of the link bandwidth. Therefore also these parameters are important for a general interface

Obviously such a description can never meet all aspects of Differentiated Services or of any other Quality of Service providing mechanism. Therefore the PyBAR does not even try to provide such a general interface or even a multipurpose module providing such a functionality.

Therefore, a module integrating DiffServ configurations for Virtual Routers and for Linux was implemented, providing a convenient small set of commands as listed on Table 6.3.

The module is written in Python and might be extended to provide more control and advanced features rather easily. This is important, because the set of commands pro-

vided by an appropriate Differentiated Services module depends on the services an Internet Service Provider wants to provide and control.

Therefore a light weight mechanism to install modules on different platforms and to adapt extension modules for new purposes is much more important than the attempt to provide a really generic interface.

The mechanisms to install such modules is provided by the PyBAR platform. An active packet may transport and install code within each suitable node of a network, on a single device only or on any machine along a certain path. Even the attachment of multiple code objects is possible, whereas each code object is to be installed on the matching router hardware.

The adaptability of the extension modules is provided by the python language itself. The combination of readable source code, a high level syntax and convenient data types allow the quick adaptation of mappings between the extension module's uniform command set provided to other programs and the router hardware dependent commands.

6.5 Adding Active Services to a Network

The active service id field in the PyBAR header provides an easy mechanism to add various packet treatments. Since the processing can completely be provided by native code, the required processing power keeps reasonably low, even for more complex algorithms.

A possible application for such active services can be the support for video applications or to provide support for management related tasks, like:

- A smart dropping mechanism using an application specific dropping algorithm for better performance.
- A lightweight multicast algorithm to send a packet to multiple destinations without transmitting one packet per destination.
- a framework to set up tunnels within a network

The first two applications have been evaluated with a video application, capable to transport real time traffic.

The video application is running on Linux, using a frame grabber device. Each grabbed frame is encoded into a JPEG picture and transmitted over the network. A client program receives the frames and displays them. Both tools were developed as a basis for any kind of experiment requiring a video stream and therefore are highly configurable. The C++ based implementation also allows to add new functions and mechanisms quite easily.

Figure 6.8 shows the front-end of the video application with its controller window. The application is able to handle television broadcast as well as other video sources like cameras. The controller window allows to adjust the video source, the television tuner

Figure 6.8: *The Video application with the controller window to set up frame rates, window dimensions and parameters for the frame grabber device*

and other parameters of the frame grabber device. Besides parameters like contrast and brightness, the resolution of the grabber can also be adjusted.

Since real time traffic is transported, the connectionless UDP protocol is used. This prevents delays due to packet retransmissions but requires a video application capable to handle missing packets.

Even at small resolutions the size of a single frame is usually larger than the maximum allowed packet size. This is why a JPEG compressed frame has to be fragmented into several packets. Similar to protocols like RTP [SCFJ96] each fragment carries a frame id and other information.

6.5.1 Application Specific Packet Dropping

To provide an adaptation of the video sender to available resources within a network an additional signalling would be required. The sender and the receiver would have to exchange information about the number of frames or packet lost during transmission. Reacting to such information the sender can adjust its packet rate or change its coding algorithm.

Another approach is to add mechanisms to the network supporting specific applications, applying a more intelligent or at least specialised dropping mechanism to network devices.

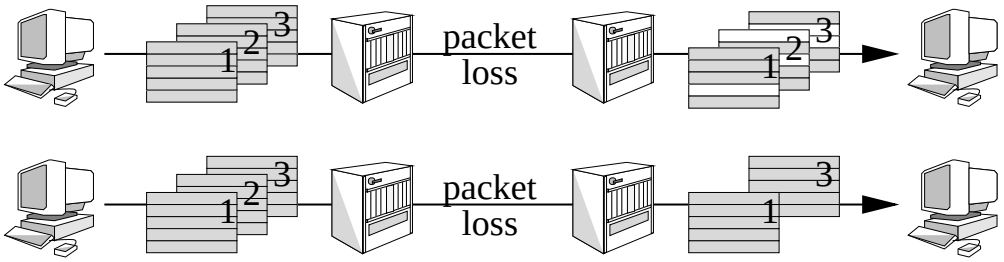


Figure 6.9: *A frame dependent dropping of packets will reduce the rate of complete frames, instead of damaging all frames*

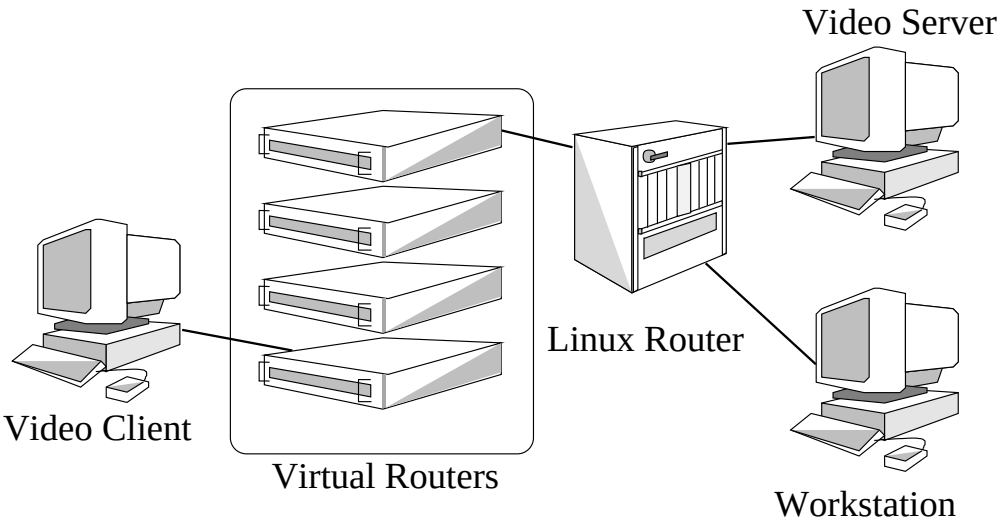


Figure 6.10: *Network with Virtual Routers used to evaluate the smart dropping algorithm*

Figure 6.9 shows the basic concept. Since each frame is distributed to several packets, a discarded packet will damage the whole frame. During heavy congestion this might result in each frame to be damaged. To allow a reassembling of packets to complete frames, either all packets must have reached the destination or a sophisticated video format is required, which is able to tolerate a certain packet loss.

In an alternative to the random dropping of packets, a more specialised component will check the frame number contained in each packet. If congestion occurs, all packets of a frame are dropped, instead of discarding only a few packets of each frame.

As can be seen in Figure 6.9, this will decrease the frame rate reaching the receiver, but can increase the number of frames reaching their destination without damage.

The capacity of such a simple algorithm has been evaluated in a network consisting of Virtual Routers and Linux systems. The setup can be seen in Figure 6.10. The video stream is sent from the video server to the video client through the network of five routers. The additional workstation connected to the Linux router is used to produce congestion within the network.

Table 6.4 shows the path of packets from the video server to the video client with the

traceroute to 10.42.14.2 (10.42.14.2), 30 hops max, 38 byte packets

1	atlas.cnds.unibe.ch (130.92.70.30)	0.220 ms	0.156 ms	0.151 ms
2	10.42.10.2 (10.42.10.2)	0.664 ms	0.172 ms	0.168 ms
3	10.42.11.2 (10.42.11.2)	0.260 ms	0.243 ms	0.234 ms
4	10.42.12.2 (10.42.12.2)	0.510 ms	0.417 ms	0.403 ms
5	10.42.13.2 (10.42.13.2)	0.520 ms	0.493 ms	0.470 ms
6	10.42.14.2 (10.42.14.2)	0.522 ms	0.514 ms	0.527 ms

Table 6.4: *A traceroute from the video server to the video client (10.42.14.2) shows the path of the packets through the network*

address (10.42.14.2). The addresses 10.42.10.2 - 10.42.14.1 belong to Virtual Routers, atlas.cnds.unibe.ch is the Linux Router. While for the links between the workstations and the Linux router 100 Mbps Ethernet is used, the links between the Virtual Routers have a bandwidth of 4 Mbps.

The Injector program of Section 6.3 has been used to create the active packet which contains a short program and the service handler. This capsule is transmitted through the network and the short program is executed on the active routers to install the service handler. The service handler monitors the number of packets in the interface queues and calculates an exponentially weighted moving average (EWMA) similar to that used by the RED mechanism.

$$avg = (1 - w)avg + w \cdot queuelength$$

The parameter w controls how fast the average queue length is adapted to the current queue length. If the average queue length exceeds a certain limit, the algorithm starts to drop packets belonging to frames with odd frame numbers. This algorithm is very simple and mainly intended to demonstrate the basic mechanism. In reality, frames would probably be discarded more smoothly instead of abruptly dropping each second frame as it is done in the experiment.

To force an active processing of the video packets, the video application has been modified to send packets in the PyBAR packet format and to set certain DSCP values to its packets. Since the PyBAR packet format is simple (see Figure 6.5), just the version and the active service id field had to be added.

The special DSCP value is used to signal that an active processing of this packets is required. The PyBAR core checks the active service id of the packets and passes them to the appropriate, previously installed service handler.

For the evaluation, a video flow has been set up and the number of correct transmitted video frames has been recorded. After 30 seconds a set of UDP senders has been activated, producing an overall UDP bandwidth between two and 3.5 Mbps. The experiment was performed twice: with and without the special dropping algorithm.

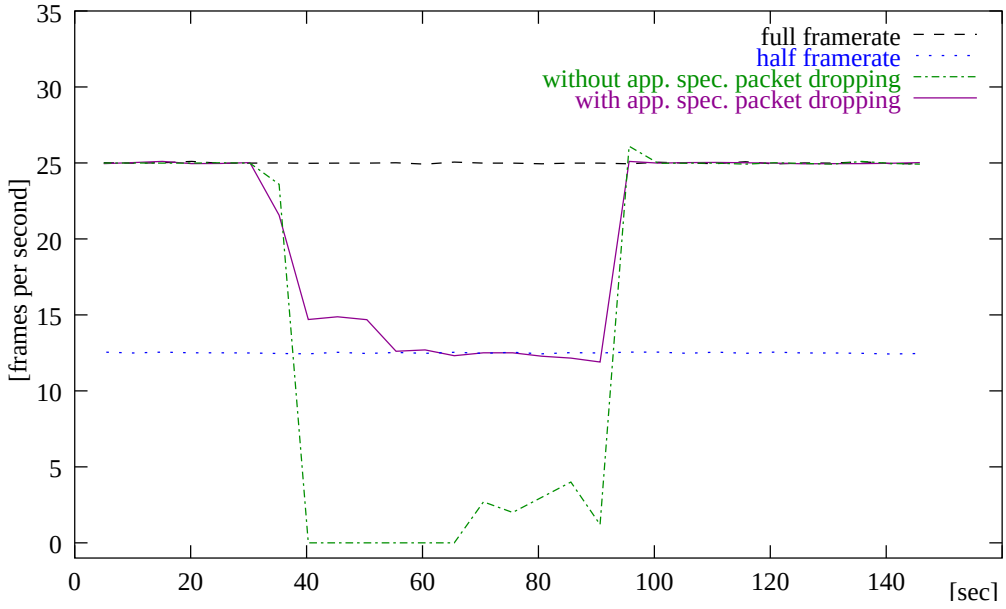


Figure 6.11: *Frame rates achieved with and without adaption to available resources.*

	bandwidth (Mbps)	frames per second
without app. spec. dropping	0.81	1.17
with app. spec. dropping	0.72	13.02

Table 6.5: *Frame rates and bandwidth with and with application specific packet dropping*

The results of both experiments are shown in Figure 6.11. It shows the rate of frames reaching the destination with and without application specific dropping. To be able to compare the results better, also graphs showing the full and the half frame rate are included.

Without special support for the video stream the frame rate during congestion nearly drops to zero, while with the help of the algorithm at least half the frame rate was able to be achieved. After the UDP senders stop the data transmission, the receivers get the full frame rate of 25 frames per second in both experiments.

Table 6.5 illustrates the difference between bandwidth consumption and frame rate. During the experiment without active support, more packets have been received, but less frames could have been reassembled. Due to the discarding of complete frames the specialised dropping algorithm uses less bandwidth than the simple best effort method but achieves a more than a tenfold rate of correct frames.

Of course the algorithm used to adapt the frame rate to the current available resources is rather simple. Therefore it is questionable whether the simple discarding of every second frame is the best solution. On the other hand TCP also reacts to packet loss by halving its bandwidth.

Also Differentiated Services can be used to mark frames with different drop prece-

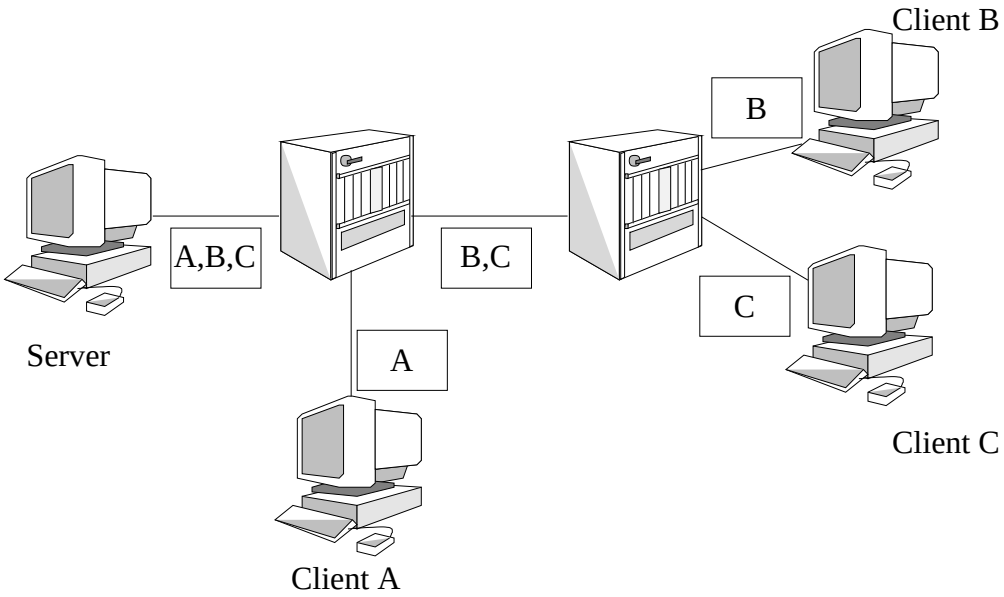


Figure 6.12: *processing of a packet with multiple addresses*

dences within an Assured Forwarding class. Even if this would allow to forward a certain frame rate with a high probability, the three drop precedences can only provide an adaptation to three different frame rates while an application specific dropping mechanism can be much more flexible.

However, the experiment shows, that even simple mechanisms specialised to certain kinds of traffic can improve the performance significantly. The algorithm used for this evaluation is therefore less complex than a standard RED queue but improves the performance of this video application tremendously. Another big advantage of such active services is that they can also be applied on the fly to quickly react to some certain network problem, since convenient methods to establish these services are provided.

6.5.2 A Simple Active Multicast Service

Another example for an application using the active services is the provisioning of a simple active multicast mechanism. Instead of signalling to drop packets according to some specific frames, the active service id signals the existence of additional IP addresses in the packet's payload.

Such an approach is similar to the Explicit Multicast (Xcast) proposal by Boivie e.a. [BFI+01] and is of course only suitable for small groups. Within large groups the number of IP addresses within each packet would exceed the packet size and also the processing of the additional addresses would also cause too much overhead. Therefore, for large groups the classical multicast approach has to be used.

Figure 6.12 illustrates the packet processing. A server has to send identical data to multiple addresses. Instead of transmitting one packet via unicast to each destination, one packet is sent, containing multiple addresses.

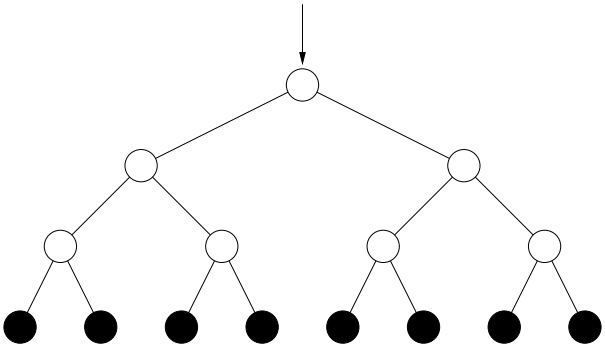


Figure 6.13: *Eight end systems (black) with routers (white) in a binary tree like network.*

An active router on the path detects the packet due to a special DSCP values checks the active service id and extracts the list of addresses. If all addresses are routed over the same interface, the original packet is forwarded, otherwise the packet is converted into several packets, each carrying the addresses of the clients routed over the same interface. The saved resources are evident.

Within a simple binary tree like network as shown in Figure 6.13, a unicast based approach would result in

$$t_{uni} = 2^n \cdot n$$

packet transmissions per datagram sent by the video server. n is the depth of the tree. This number of overall packet transmissions is reduced to

$$t_{smc} = \sum_{i=1}^{n} 2^i$$

with a multicast approach. The following table lists the numbers of packet transmissions required for each packet sent out by the source with and without the simple multicast approach.

receivers	packet transmissions		bandwidth values	
	unicast	simple multicast	unicast	simple multicast
4	8	6	4.8 Mbps	1.22 Mbps
8	24	14	9.6 Mbps	1.24 Mbps
16	64	30	19.2 Mbps	1.28 Mbps
32	160	62	38.4 Mbps	1.35 Mbps

The table also includes the bandwidth values reached by the sender in both cases with the video application described above. While in the unicast situation the sender has to transmit up to 40 Mbps to reach the 32 receivers, the bandwidth with the simple multicast algorithm is only increased slightly due to the additional addresses in the packet's payload.

```
class MiniMulticast(ARservicehandler):
    def forward(self,pkt):
        # extract address list from first code block
        alist=cPickle.loads(pkt.cb(0))
        ifcs={}
        # scan address list and create interface/address-list structure
        for i in alist:
            interface=pad.queryRoute(i)['if']
            if not ifcs.has_key(interface):
                ifcs[interface]=[]
            ifcs[interface].append(i)
        # scan interface/address-list structure and send packets
        for i in ifcs.keys():
            p=pad.Packet()
            p.cb(0)=cPickle.dumps(ifcs[i])
            p.cb(1)=pkt.cb(1)
            p.send({'dest':ifcs[i][0], 'ptype':pkt})
        return
```

Table 6.6: *The complete code of the service handler providing a simple multicast service*

Table 6.6 lists the complete Python code required for the service handler. An active packet for installing this service handler will usually consist of a short installation script to register the service and the service handler code itself. The code is rather simple and very short, since Python provides very powerful internal types.

The implementation uses the PyBAR packet format. A multicast packet consists out of two code blocks (see Figure 6.5) The first code block contains the address list, the second one the payload of the packet.

As a simplification the addresses are realised by a Python list. The addresses within this list are scanned and for each address it is checked, which interface would be used to reach that address.

With the results a dictionary is set up, storing a list of addresses for each interface. In a final step new packets are generated and transmitted.

The pad.* functions are provided by the platform adapter. Once a packet for a specific service handler arrives the forward() function of this service handler class is called with the packet as parameter. The forward() function extracts the address list from the first code block of the packet (cPickle), scans the addresses, sets up the dictionary and finally creates and sends the new packets. The destination address of a new packet for a certain interface is chosen from the list of addresses routed over that interface.

Table 6.7 shows the performance[1] of the code on a single router for different numbers of addresses contained within a packet. The bandwidth calculation was based on the execution speed of the Python code and an average packet size of 1000 bytes. For

[1] measured on a Linux PentiumII with 400 MHz

addresses	time	incoming bandwidth	max. bandwidth out
4	1 ms	8 Mbps	32 Mbps
8	1.7 ms	4.7 Mbps	37 Mbps
16	2.2 ms	3.36 Mbps	58 Mbps

Table 6.7: *Performance of the simple multicast service*

the output bandwidth it was assumed, that each packet is split up completely and each address has to be routed over a different interface.

Of course the performance can not cope with any implementation using native code. On the other hand the input bandwidth is sufficient for multiple video flows. However, the code performance can definitively be increased by using a less time consuming format for the address list. Finally there is still the possibility to provide such a mechanism by native code.

6.5.3 Active Setup of Tunnels

IP tunnels are an enabling technology for the application of new services as could be seen in chapter 5. Even if the basic mechanism, an simple encapsulation of a packet into another packet as proposed by [Per96] is simple, more complex tasks may be realised:

- The encryption and decryption of packets at the tunnel start and end point can provide end to end security. This way flows can be transmitted securely without the danger that a not trustworthy service provider might eavesdrop.
- Since all packets transmitted through the tunnel get a new header with the tunnel start and the tunnel end point addresses, traffic conditioning mechanisms can be applied quite easily for all packets in the tunnel, even if the encapsulated packets have different source and destination addresses.
- Since the original packets are within the payload of the new tunnel packets, interferences with network devices forwarding the tunnel packets can be omitted. Therefore, signals can be transported transparently through an ISPs network without triggering any mechanisms like resource setup.

Obviously for the setup of tunnels an appropriate start and end point is required if certain special services like encryption are required. But even for a simple IP in IP encapsulation, an end point needs to be capable to handle the decapsulation of packets.

As the setup of a tunnel is usually sender driven, the general problem is to detect an appropriate end point. This is even more complicated, since the end point often can not be specified easily. Therefore to set up tunnels between border routers, the ingress router usually does not know the address of the appropriate border router.

An active network allows to trigger the set up of tunnels automatically without the demand of any additional protocol. Since an active packet allows to define a kind

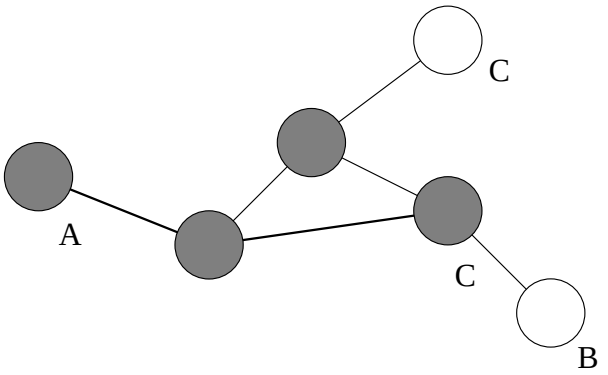


Figure 6.14: *A small network with routers capable to tunnel packets (grey) and routers without that mechanisms (white).*

of ”search pattern” for an end point and can simply be send downstream towards the destination, the setup of a tunnel can be simplified and also the exchange of keys might be accomplished.

Setup of IP tunnels

Figure 6.14 shows a small network with a set of grey routers capable to handle tunnels and white end system without this support. Node A wants a tunnel to be set up as close as possible to point B. To find out an appropriate end node, an active packet addressed to node B can be injected into the network. The packet will pass the nodes along the path and check whether the node is an appropriate tunnel end point or not. Similar to the traceroute program reporting each passed router, the active packet can report each possible candidate for a tunnel end point to node A. If simple IPIP tunnels, not requiring a specific set up of the tunnel end point, are used, node A simply uses the most appropriate candidate as end point and sends the encapsulated packets to this address.

Table 6.8 shows a example of Python code to check whether a device can handle an IPIP tunnel. The whole packet consists of the first code block containing this executable program and a second code block containing information about the tunnel start point (src_info) like the address and the port number to that the feedback packet has to be sent to.

This search for an end point is of course not only determined by the simple capability to provide the decapsulation of tunnelled packets, but may also be used to find nodes capable of certain encryption techniques.

Active tunnels

The mechanisms to provide a proper encapsulation and decapsulation may also be transported within an active packet. Of course it would be possible to add a small program header to each packet, decapsulating the embedded original packet near the destination and thereby create a tunnel. This might work for the transport of single signals, but for any reasonable flow the performance would suffer.

On the other hand active packets might provide tunnel start and endpoints. A simple tunnel end point using an active service id to trigger decapsulation is shown on Table 6.9. If an active packet with the according service id reaches an active router which

```
class DiscoverEP(ARpacket):
    def __init__(self,acpkt):
        # get a list of router properties/services
        c=pad.getCaps()
        # if IPIP available, extract information from
        # code block and send feedback packet
        if c.count('IPIP'):
            src_info=cPickle.loads(acpkt.cb(1))
            # generate and send feedback packet
            p=pad.UDPPacket()
            p.source=pad.hostip
            p.dest=src_info['tunnel_start']
            p.destport=src_info['portnumber']
            p.payload=cPickle.dumps({'service':'IPIP',
                'tunnel_end':pad.hostip, 'time':pad.time})
            p.send()
        # forward original active packet
        acpkt.send()
        return
```

Table 6.8: *Active Packet code to discover a device, able to handle IPIP tunnel endpoints*

has this service handler installed, the packet encapsulated in the single code block (see Figure 6.15) will be extracted and forwarded.

In contrast to conventional tunnels, needing a specific tunnel end point, such tunnels may even work without. For example such service handlers might be located at the border routers of an ISP and decapsulate any active packet with this service id. The processing by an active router can be triggered as before by setting a specific DSCP. Of course such a mechanism is mainly useful to keep the encapsulated packets from interacting with other network components. The advantage of tunnels supporting traffic conditioning mechanisms by their fixed start and end point addresses is lost.

```
class ANtunnelEP(ARservicehandler):
    def forward(self,pkt):
        # generate new packet from first code block
        p=pad.Packet()
        p.from_string(pkt.cb(0))
        # send the decapsulated packet
        p.send()
    return
```

Table 6.9: *Code for a service handler to provide a tunnel end point*

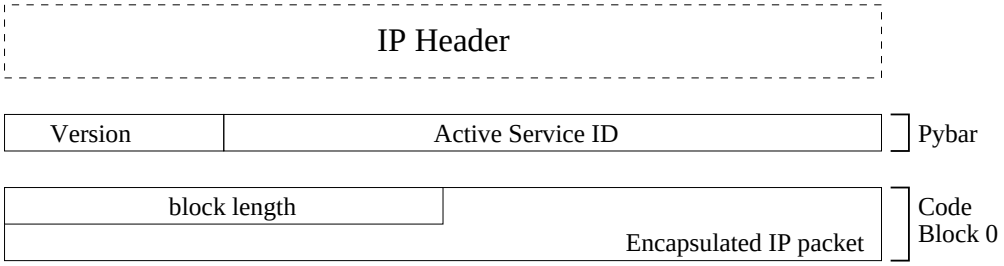


Figure 6.15: *Encapsulation of the packet into a PyBAR code block*

Of course the code on Table 6.9 will also work if active packets are addressed to the tunnel end point directly. Instead of triggering processing by each active router, no special DSCP is used causing intermediate (active) routers to forward the tunnel packet like normal IP packets. Therefore, the mechanism is similar to a normal IP tunnel, even if the code to establish the tunnel end points can be installed dynamically.

Similar to the simple multicast scenario the performance of such a high level implementation of a tunnel end point is limited. However, since the packet data type is implemented in C, this endpoint is at least able to process about 25000 packets per second[2].

Of course if tunnel endpoints with dedicated addresses have to be set up, similar problems as for the configuration of conventional tunnels would occur and active packets may have to be used to discover proper locations for end points.

Conclusion

The services described in this chapter show the power of active components within a network. Both mechanisms are simple but can increase the performance of network based applications tremendously. Of course such services can also be applied to routers and network devices statically or can be realized by specific protocols. On the other hand protocols used for video transmissions might change and new applications will require other mechanisms. Therefore, a flexible mechanism to provide such components is required.

The setup of tunnels illustrates the capabilities of mechanisms allowing to simply exchange information. Especially if used for signalling only, the performance is absolutely sufficient. But even if the performance of interpreted code is generally limited, simple functionalities like the decapsulation of packets can be provided, at least for services, which are not used very frequently or have to be established quickly.

6.6 Active Network Support for Quality of Service

In the previous chapter the capacity of the Active Network approach to create new network services has been shown. Specific applications have been supported by mechanisms dynamically distributed within the network.

[2] measured on a PentiumII, 400 MHz

Additionally to application specific services, Active Networks can also be used to support the traditional Quality of Service concepts RSVP and Differentiated Services. In chapter 5 a system for the mapping between Differentiated and Integrated Services has been presented. The concept was based on aggregating RSVP reservations to tunnels supported by Differentiated Services.

The architecture used a RSVP/Differentiated Services gateway located at the ISP border routers and a central bandwidth broker instance. The gateway negotiated with the bandwidth broker about the tunnels to be set up. The bandwidth broker configured the tunnels and the involved traffic conditioning components.

Even if such an architecture will work for small and medium size networks, the central bandwidth broker with its topology database causes scalability problems in large networks. Also a disadvantage is the additional mapping component required within border routers.

Besides the creation of new services like the application specific packet dropping, active network mechanisms can also be used to support Quality of Service. This combination has several advantages:

- Since active packets follow the same path as normal traffic, a central topology database can be omitted. The RSVP/Differentiated Services architecture presented in Chapter 5 required a central bandwidth broker to set up tunnels between border routers. Therefore knowledge about the corresponding border routers and of the intermediate network devices was required, which was stored in a central database. Using active packets, the configuration code will automatically pass all involved network devices.

- There are numerous border router pairs between tunnels that might have to be established. A central instance will either have to process the configurations sequentially or in parallel. Sequential processing can be very slow since remote devices have to be configured. For a parallelisation a central instance has to check whether the configuration requests do interfere. This requires a central topology database and rather intense computation. Since an active environment will process configurations more locally, interfering configurations are less probable and can be solved for each host separately.

- Active networks can provide more flexibility and simplify the creation of new services. As mentioned in Chapter 2, an Internet Service Provider might mark packets in his ingress routers for his customers. This is rather simple as long as IP header fields only are used. Active Networks can be used to provide more application specific packet classification. Similar to the application specific packet dropping (see Chapter 6.5.1), mechanisms marking packets more intelligently may be installed.

Of course any of these tasks could be achieved without active elements, each requiring an own specific protocol. On the other hand active networks can be used to implement these services rather easily and provide an extremely flexible platform for future services.

In this chapter mechanisms for the support of Quality of Service based on active networking technology will be presented. In contrast to the architecture of chapter 5 it will follow a more general approach, supporting different kinds of resource reservations as well as autonomous interactions between different Internet Service Providers.

6.6.1 Reservation Domains and ISP Service Mapping

In the previous chapters the Internet Service Provider networks were assumed to be rather homogeneous. Mechanisms to support different Quality of Service concepts were applied at their border routers.

An ISP's network is not necessarily homogeneous, even with a concept using mobile code to configure networks to provide Quality of Service issues and establish new services. There might be administrative or technical reasons splitting the ISP network into several regions supporting different protocols.

Of course the network devices within such a region have to provide active network support and the execution of active code is probably limited to the own active packets due to security reasons. Also regions supporting the same Quality of Service concept appear of special interest.

Therefore a network may be divided into several reservation domains, which are defined by several parameters:

Active network environment: Within such a domain a unique active network environment has to be supported, allowing to execute code and install new services at specific locations. Whether all devices have to be active network capable or only the devices at the borders of such a domain depends on the service to be applied. It might not be necessary for a network core to support active packets, if the service only requires a mapping at the border routers.

Reservation method: Within a reservation domain a unique reservation method has to be supported. The reservation domain also depends on the type of an incoming reservation. Therefore, a RSVP to Differentiates Services conversion might be desirable at an ISP's border router to prevent his backbone routers from the RSVP processing load, while a mapping to Differentiated Services is necessary when entering the ISP's backbone using some specific reservation scheme [TH98].

Authorisation: A general problem of active networking is that ISPs will not accept foreign code to be executed on their network devices. Therefore, active packets may only be used within a network owned by the same ISP or within networks of ISPs with a certain mutual trust. This is why the size of reservation domains may also be limited to certain networks.

The size and shape of a reservation domain results in the activities of the active network system itself. Code is sent to the network, installing service handlers at the edges of reservation domains. The code of course contains parameters and algorithms to detect those edges by checking whether certain services like RSVP or Differentiated Services

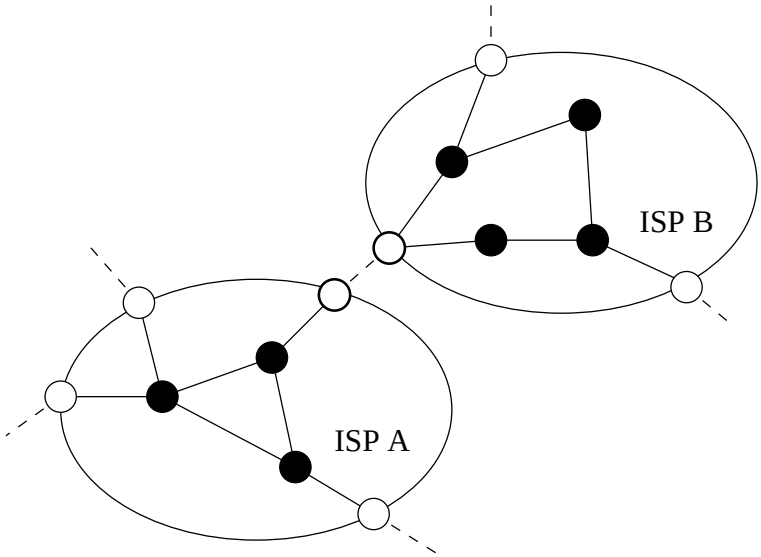


Figure 6.16: *Two ISPs as reservation domains with their border routers (white)*

are available or not. Therefore, size and shape of a reservation domain are not specified centrally, but more a result of service mechanisms distribution.

Figure 6.16 shows two Internet Service Providers as reservation domains with their interior routers (black) and the border routers (white). Typically, the used reservation methods change at the border router of an ISP, therefore a reservation for a flow forwarded from ISP A to ISP B might have to be mapped at the border routers (white). Furthermore, these methods do not have to be implemented by each Internet Service Provider, even the DSCP values for the same service may change from ISP to ISP. Therefore, at the border of a reservation domain several mappings may be necessary.

Unfortunately, the translation of a resource reservation type at the border of a reservation domain (RD) faces some general problems.

Incompatible Resource Specification: Information about the amount and type of requested resources might get lost during mapping, because of incompatibilities in traffic specifications. Therefore, as few mappings as possible should be applied to a flow or aggregate during its transport over several reservation domains. RSVP uses for example a detailed description of a single reservation, while Differentiated Services only acts on aggregates. Therefore information about a specific flow may get lost, when mapping RSVP to Differentiated Services. Unfortunately, there is no general description of a reservation being compatible with the Differentiated Services and the Integrated Services approach. RSVP uses for example quite a detailed flow specification $flow_{spec}$ [Wro97]

$$flow_{spec} = \{r, b, p, m, M\}$$

with the token bucket Rate r, the token bucket size b, the peak data rate p, the minimum policed unit m and the maximum packet size M. Unfortunately,

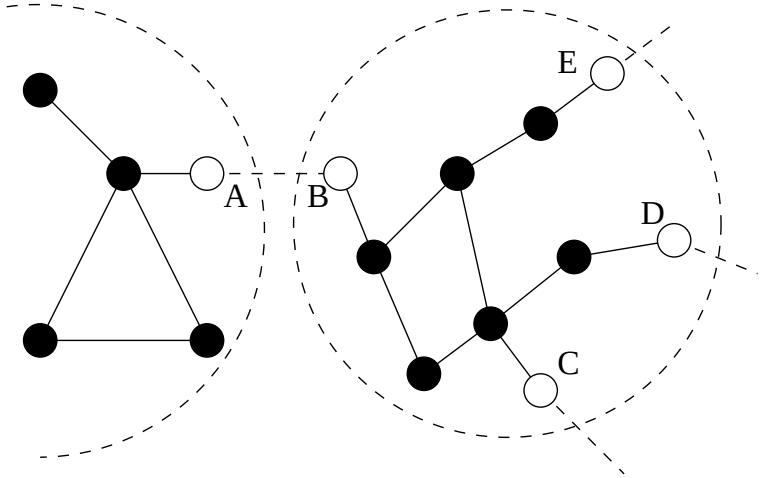


Figure 6.17: *reservation domains with Agents located at the RD borders*

the Differentiated Services framework does not describe services as detailed as RSVP. Services are defined by a per hop behaviour like Expedited or Assured Forwarding.

Router Configuration: A new mapping may require a configuration of intermediate routers as a setup of certain queueing disciplines. A central approach would have to determine the path causing a lot of management overhead, while an ingress router may simply launch a capsule to the egress point configuring the devices on its way through the network, reducing management overhead. Certain reservation mapping methods also require a specific setup of a router pair. Therefore a cooperation of two routers is necessary to set up a tunnel through an ISPs network (e.g. aggregated RSVP [GBH97]).

Of course, such reservation domains are more a construct to illustrate the architecture. Active packets containing code for a new service can simply detect the border routers where mapping functions have to be applied and will install the required mechanisms. This detection can simply be achieved by matching the router's parameters against some profile contained within the active packet. This is how active packets define the reservation domain as can be seen in Figure 6.17.

The reservation mapping itself might either be a simple configuration as required to map different DSCPs or the installation of a more complicated component as needed for RSVP signalling message processing.

However, even if the mapping only means to replace a certain DSCP value with another one, an active packet might additionally install some monitoring agent responsible for automatic reconfigurations within the network.

6.6.2 Active Mapping between Differentiated Services Domains

Probably the mapping between different Differentiated Services at the border of reservation domains requires rather a reconfiguration of a marker component than the in-

stallation of any active code. While this mapping itself is simple and can be done by the Differentiated Services router itself, the negotiation about a matching service might be complex.

The Differentiated Services framework [BBC+98b] leaves it to the ISP, which services it wants to provide to its customers and neighbouring ISPs. Therefore, it depends on the ISP whether a specific service is available or not. Even if the Differentiated Services framework recommends to provide at least the standardised set of Expedited Forwarding and Assured Forwarding, the mapping of more advanced services is an open issue. Furthermore, a standardised service like Assured Forwarding may also cause problems, since it depends on a set of not standardised configuration parameters and may behave differently for different ISPs. Therefore, a mapping does not only depend on the service type, but also on the neighbouring ISP. There are several methods to determine an appropriate translation:

encoding: The proper mapping scheme is encoded in the mobile code itself. It requires an update of the mobile code, if DSCPs are changed. Since the mapping itself is done by a router component usually, the configuration code has to set up a list of DSCP/DSCP translations. Fortunately, the translations can be assumed to be rather static.

central DSCP database: The mechanisms installed into appropriate network devices do not only set up a set of mapping rules, but also initiate mechanisms to periodically query a central database.

negotiation: The most elegant yet unfortunately most complicated approach is a negotiation between two reservation domains (probably ISPs) about the most appropriate mapping scheme. To achieve this the agents need some knowledge about the PHBs behind the DSCPs and methods to determine corresponding PHBs.

Each of this techniques has its advantages. The simplest one is probably to encode the DSCPs into the configuring capsule itself. Periodical requests at a central database cause similar overhead as re-sending an updated configuration capsule, especially as new services might also require also a reconfiguration of intermediate devices, which can be accomplished by the same capsule.

The approach using an automatic negotiation to determine an appropriate mapping would require a general description for a service. A simple description using a RSVP flow specification can work for services like Expedited Forwarding but can not describe Assured Forwarding with its multiple drop precedences.

Since Differentiated Services are able provide Quality of Service even at high packet rate, any component for the mapping of different DSCP values has to be part of the Differentiated Services router itself, rather than being provided by interpreted code. Active Network techniques can be used for the installation of such mechanisms and their configuration. Also infrequently tasks like signalling or the processing of special events can be provided by active packets.

6.6.3 Active Mapping for RSVP and Differentiated Services

The simplest active support for RSVP/Differentiated Services mapping would be the use of an active network system to install a RSVP/Differentiated Services gateway (RDG) as described in chapter 5. Of course this would not deploy the power of the active network approach, since an active network system can also be used to provide a more scalable solution for the setup of tunnels and the reconfiguring intermediate devices.

Similar to the RDG some mechanisms are required for listening to the RSVP signalling and to trigger network configurations. The RSVP/Differentiated Services gateway of chapter 5 was based on the ISI RSVP implementation, which was modified to support the required additional support for a central bandwidth broker. The use of a complete RSVP implementation would be a rather heavy weight solution. In contrast to provide the complete RSVP signalling, the Active Network will only provide a small set of functions supporting RSVP.

While in the concept presented in Section 5.3 both ingress and egress border routers provided full RSVP functionality, a more lightweight approach will only process signals required for the setup of resources in the reservation domain:

processing of *path*-message: Since the *path*-message is used to investigate the path through the network and therefore also determines the routers to be involved in the exchange of the *resv*-message, the ingress router has to process this signal. The ingress router stores the flow-id of the message and the address of the last RSVP capable router contained in it. After that it replaces this address by its own and forwards the packet. This ensures for following *resv*-messages sent hop by hop to pass this ingress router.

processing of *resv*-message: The *resv*-message contains the flow specifications of the reservation to be set up. This message is important, since it contains detailed information about the flow to be set up. If the resource setup is successful, a *resv* message is sent upstream to the next RSVP capable router formerly determined by the *path*-message.

Both mechanisms are rather light weight and do not require much processing power. Therefore the required code could be provided by mobile code itself, even if native code would of course perform better.

Table 6.10 shows the main code elements of an RSVP/Differentiated Services mapping service. Incoming RSVP reservations are used to trigger a configuration of the Differentiated Service marker.

The approach using Active Networks to manage a RSVP to Differentiated Service translation can provide much more flexibility than a static protocol based solution as described in chapter 5. Since ISPs networks are different, flexible and adaptable approaches for the integration of services are required.

The information collected within the ingress router can now be used to accomplish resource reservation within the reservation domain. Tunnels can be set up to aggregate

```
def path_process(path_pkt):
    # extract address of previous router from path
    # and get flowid
    adr=extract_path(path_pkt)
    fid=extract_flowid(path_pkt)

    # put our own address in path message
    stamp_path(path_pkt,pad.hostip)
    if flows.has_key(‘fid‘):
        # if flow is already registered, update the time only
        flows[‘fid‘][’time’]=pad.time;
    else:
        # if flow is new, remember time and prev. router
        flows[‘fid‘]={’time’:pad.time, ’prev’:adr, ’fspec’:[]}

    # forward the path message to the next router
    path_pkt.send()
    return

def resv_process(resv_pkt):
    # get the flow id drom the RSVP message
    fid=extract_flowid(resv_pkt)

    # if there is no path information ,,,
    if not flows.has_key(‘fid‘):
        return

    # if it is the first reservation message for this flow
    if flows[‘fid‘][’fspec’] == []:
        flows[‘fid‘][’fspec’]=extract_flowspec(resv_pkt);
        # set up diffserv resources
        # dsrule takes list with [source, srcport, dest, destport, flow_spec]
        dsrule=[fid[0],fid[1],fid[2], fid[3], flows[‘fid‘][’fspec’]]
        diffserv.mark(dsrule)

    # send a reservation message further upstream
    resv_pkt.dest=flows[‘fid‘][’prev’]
    resv_pkt.send()


    return
```

Table 6.10: *Processing of RSVP messages in the Python based RSVP/DiffServ mapping mechanism. For accessing the DiffServ components the DiffServ module (see Section 6.4) is used.*

traffic between ingress and egress routers or traffic simply mapped to Differentiated Services classes.

While the architecture based on two RSVP/Differentiated Services Gateways used tunnels to aggregate reservations, a direct mapping to DiffServ is favoured here.

Resource Estimation:

The RSVP protocol requires a periodical update of the resources. This involves a repetition of the setup process with a *path*- and *resv*-message. Usually this process is repeated every 30 seconds. This interval can be changed to provide better support for routes changing frequently. If no *path*- or *resv*-message has been received for at least 90 seconds, a router can tear down the reservation.

A mapping to Differentiated Services classes automatically standardises the very specific RSVP flow descriptions to a small set of services. A detailed flow description with rates, bucket sizes, peak rates, etc. has to be reduced to n Mbps of Service X. Therefore, the large set of possible different flow specifications is mapped to a very limited number of service classes.

The RSVP capable ingress router has to accumulate all RSVP flows. Besides the unique identifier for a RSVP flow (session number) the type and amount of service has to be stored. Based on this information the required amount of each Differentiated Services PHB can be calculated.

Of course an ingress router will have some policy based on a SLA limiting the amount of each service. When a reservation has to be set up, the ingress router will check whether the new reservation will exceed the maximum allowed bandwidth for this service or not and transmit an *resv-tear*-message instead of adding this flow to his internal tables.

Adaptation of Differentiated Services Core Router:

The adaptation of core routers will occur very infrequently. The ingress routers map the traffic to different service classes and this aggregates are forwarded by the Differentiated Services capable core routers. Since the core routers just handle the different traffic classes and do not have to deal with single flows, their setup can be rather static.

Any configuration of core routers faces a problem: packets with the same ingress-egress router pair do not necessarily follow the same path through the domain. This was one of the reasons, whit the RDG based approach used tunnels to aggregate RSVP flows between border routers.

Within an active network, active packets can be send along the path, the later packets will use without any need for a central topology database. In contrast to an approach using a central bandwidth broker this will have several advantages:

- there is a rather local processing of data since only network components along this path are involved.
- no knowledge about the topology is needed. The configuration packet will simply follow the path.

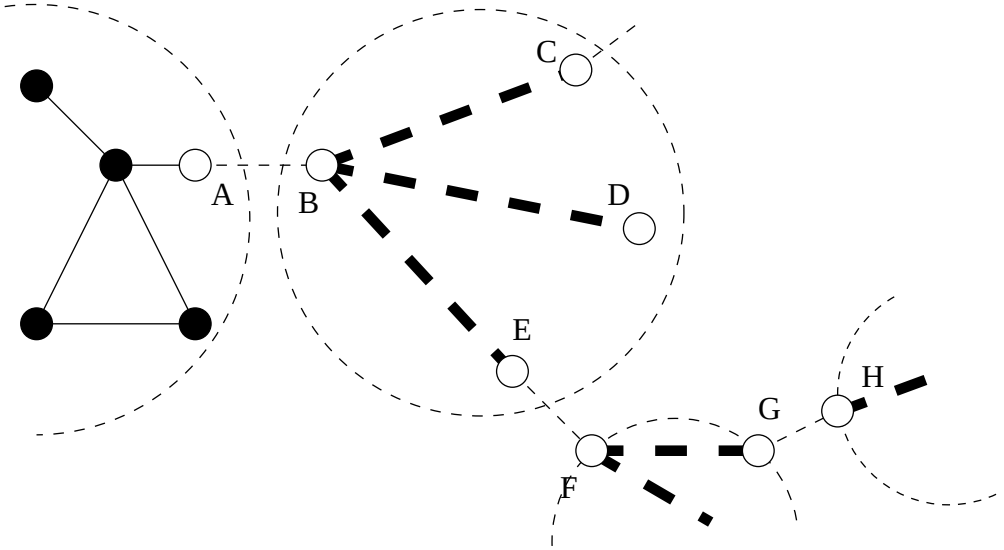


Figure 6.18: *Tunnels set up by Agents at Ingress points using capsules to configure QoS tunnels through a RD*

- a parallel operation of several ingress router is supported, since no central component has to synchronise the configuration requests.

Within the core routers no information about single flows has to be stored. Just the rather seldom passing reconfiguration packets have to be processed, which is done by the Active Network platform.

6.6.4 Resource Setup with Multiple Service Providers

In Section 5.3 tunnels were proposed to aggregate flows within a reservation domain and to provide better control about the resource consumption within a domain.

While active networks can provide functionalities to set up and maintain tunnels, these mechanisms usually are limited to a network being operated by the same Internet Service Provider [BB00c]. There are similar problems for the setup of resource reservations.

As can be seen in Figure 6.17 ideally on both sides of an Internet Service Provider (or a reservation domain) border agents based on active network technology have been installed to provide the reservation translation according to their own and to their neighbours needs. Until now any communication of agents at each side of such a border was neglected. All functionality applied to traffic was either the mapping between DSCPs or the aggregation of flows to tunnels between ISPs border routers.

Depending on the type of resource reservation mechanism used, this may cause a repeated encapsulation and decapsulation of packets at the border routers as illustrated in Figure 6.18.

To provide services spanning multiple Internet Service Provider two approaches are self-evident:

1. Interaction between central instances like bandwidth (or more general) service brokers.

2. A signalling along the path like RSVP does for the setup of resources.

The first approach was proposed and implemented as a part of the project "Charging and Accounting Technology for the Internet (CATI)" [SBP99] and works well for small and medium size networks. For larger networks a signalling along the path would be advantageous, since central components automatically limit the scalability of such an approach. In contrast to RSVP, which requires processing by each router along the path and is therefore also not suited for large networks as well, an alternative solution may only involve the ISPs border routers within such a setup process and leave it to the ISP how to provide the services.

Regarding Active Networks, such an approach has another fundamental advantage. Since security within active networks is hard to provide and an ISP will therefore not tolerate foreign code within its network, active packets have to be limited to a single ISP network.

To provide active services and simultaneously limit the activity of packets to a certain domain, interfaces at border routers have to be provided, that are accepting certain configuration commands and map them to a local management architecture.

An active packet can be sent out within an ISP to establish a tunnel with a certain Quality of Service. When the packet reaches the border router it can use such an interface to initiate the establishment of a decapsulation endpoint within the next ISP. How the next ISP provides such a service is left to the ISP.

Multiple tunnels as shown in Figure 6.18 can be avoided. If an agent at B starts to set up a tunnel, it forwards the according active packet as described in section 6.6.3. When the packet reaches the egress border router, it negotiates with an interface at F about an appropriate mapping scheme. If F considers an optimal endpoint for the tunnel to be located at G, it might proposes this to E. E transmits the new tunnel endpoint to B, setting up the proper encapsulation methods. Even if the tunnel is now spread over multiple Reservation Domains, the underlying resource reservation methods are tasks of each specific ISP.

Such a rather simple mechanism allows to set up tunnels spanning multiple ISPs as shown in Figure 6.19, completely driven by the tunnel start point. The tunnel start point even does not need to know, in which ISP network the end point is located. Also, the required negotiations between ISPs is minimised. Since an ISP has only to negotiate with its direct neighbours, an establishment of SLAs also covering the setup of Quality of Service supported tunnels would be feasible.

Tunnel Setup Process

Such an interface mechanism to set up Quality of Service enabled tunnels over multiple ISPs was implemented with the PyBAR active network system. When a Quality of Service tunnel has to be set up, the tunnel start point transmits an active packet towards the flow's destination. The tunnels to be set up comply the standard IPIP tunnels

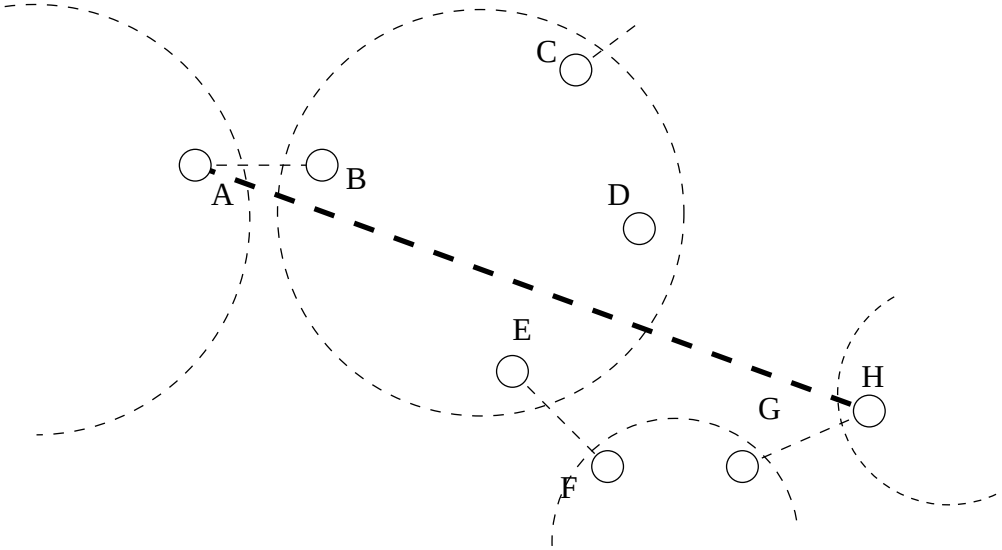


Figure 6.19: *Tunnels set up by Agents at Ingress points using capsules to configure QoS tunnels through a RD*

[Per96]. Therefore, the required encapsulation and decapsulation can be provided by the routers themselves and the active packets are used to accomplish the required setup only. The setup process follows these steps:

1. the tunnel start point transmits an active packet towards the destination of the tunnel. The active packet contains code to set up the tunnel endpoint, if the endpoint is within the authority of the active network system and some functionality to negotiate with an appropriate interface, if the tunnel endpoint lies outside the ISP.

2. (a) if the packet directly reaches a possible tunnel end point this end point is configured and an acknowledgement is sent back to the tunnel start point.

   (b) if the active packet reaches a border router, it contacts the interface to initiate the further setup of the tunnel end point within the next ISP. If the interface signals a successful setup of the tunnel endpoint, an acknowledgement is sent back to the tunnel start point.

3. if the tunnel start point receives the acknowledgement the encapsulation mechanisms are installed. Using IPIP tunnels, this requires the setup of a tunnel interface and an appropriate routing table entry.

4. optionally, the tunnel start point configures the resources along the tunnel path by sending an active packet with configuration information along the path. If the tunnel spans multiple ISPs the resource configuration of remote ISPs was already provided during the negotiation in step 2b.

Of course it would be also possible to negotiate again with the border router interface as well to set up the resources for the tunnel instead of initiating the tunnel and the

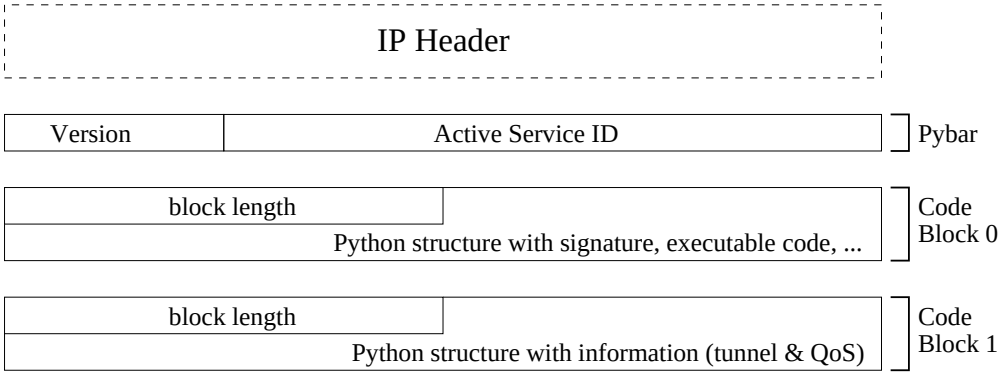


Figure 6.20: *Code blocks used for multi ISP tunnel setup*

resources simultaneously as done in step 2b. On the other hand this requires two negotiations and even if the remote ISP may agree to set up the tunnel this does not mean automatically that it is willing to provide the resources. Therefore a combined request for the tunnel and the resources is the better solution.

Table 6.11 shows the code of an appropriate active packet and Figure 6.20 the active packet with the two code blocks. Similar to the example in Section 6.5.3 an UDP packet is used to provide the required feedback to the tunnel start point. Also, a similar packet format was chosen with the first code block containing the executable code and the second code block containing information about the tunnel start point and about the desired Quality of Service.

The method query_nextISP(self,pkt) is called to accomplish the negotiation with the next ISP. How this is handled in detail depends on the mechanisms provided by the next ISP.

When the tunnel start point receives the UDP acknowledgement, it will set up IPIP components and if needed, transmit an active packet adapting configured resources between the start point and the border router.

Interface Design

The interfaces accepting the configuration requests have to provide a rather generic data format to support a broad range of services and to allow an easy extension. Due to security reasons the data sent to the interface contains no code to be executed. Therefore, a flexible solution might be the use of some text based negotiation with the advantages of platform independence, simple extensibility and human readability for control purposes.

An alternative may be to provide multiple methods for an interaction. The next ISP then may provide a central broker like interface, which can be contacted to determine the tunnel end point, or the opposite border router serves as access points.

The mechanisms used to provide tunnels spanning multiple ISPs can of course also be used to trigger any kind of setup process requiring flow specific negotiations between ISPs.

```
class MultiTunnelEP(ARpacket):
    def __init__(sefl,pkt):
        return

    def run(self,acpkt):
        c=pad.getCaps()
        m=pad.managed_ips()
        # if the destination is in the list of supervised hosts and this
        # host can handle IPIP tunnels send feedback packet
        if m.count(acpkt.dest) and c.count('IPIP'):
            self.send_feedback(self,acpkt,pad.hostip)
            return
        # is this a border router ?
        if c.count('BORDER'):
            # query result from next ISP
            result=self.query_nextISP(acpkt)
            if not result == '':
                # return feedback packet with the address of the tunnel
                # end point in one of the next ISPs
                self.send_feedback(acpkt,result)
                return
            # no answer from next ISP, the border router has to
            # handle the tunnel end point
            self.send_feedback(acpkt,pad.hostip)
            return
        # this is neither end point, nor border, forward the active packet
        # to the next host
        acpkt.send()
        return

    def send_feedback(self,pkt,t_endpoint):
        src_info=cPickle.loads(pkt.cb(1));
        p=pad.UDPPacket()
        p.source=pad.hostip
        p.dest=pkt.dest
        p.destport=src_info['portnumber']
        p.payload=cPickle.dumps({'service':'IPIP',
            'tunnel_end':t_endpoint, 'time':pad.time()})
        p.send()
        return

    def query_nextISP(self,pkt):
        # negotiate with some instance of the next ISP and
        # return ip address of end point on sucess
        # e.g. send serialised python objects
        return ''
```

Table 6.11: *An active packet looking for a tunnel end point and triggering negotiation with the next ISP*

In contrast to RSVP these mechanisms do not require core routers to store per flow information and also the active packets used to adapt settings within the core routers may only be sent if the requested resources exceed a certain limit. The system provides a high degree of flexibility.

The range of services to be supported by such an approach is endless and does not have to cope with the security issues often mentioned against active networks, since no foreign active code has to be executed.

6.7 Conclusion

In this Chapter the Python based active router (PyBAR) has been presented. Python is a modern, object oriented platform independent computer language. Python especially is able to be easily extended using either Python based components or modules written in C or C++. This capability allows Python also to access low level functionalities of an operating system, as required for an Active Network environment.

On the other hand this combination of native and high level Python code can provide good performance and high flexibility at the same time. The PyBAR system was implemented for the Virtual Router platform presented in Chapter 4 and for Linux routers.

Therefore, Active Network topologies consisting of Virtual and Linux routers can be set up, allowing the evaluation of classical network functionalities as well as experiments with active networks.

Since the PyBAR does not address end users but is intended to provide a platform for the implementation of new network services, it is rather a toolkit than a ready to use application. Special modules have been implemented for the PyBAR like the PyRSA module providing RSA based encryption mechanisms.

Based on the PyBAR platform new services have been implemented for example a mechanism for an application specific packet dropping, a simple multicast service and mechanisms realising tunnels. Of course the performance of interpreted code is limited, but nevertheless allows to provide mechanisms requiring little processing power. New mechanisms can be dynamically loaded to specific network devices, increasing the performance of applications. The ability to implement lightweight mechanisms for the detection of tunnel end points can also be exploited by other, not active components.

In Section 6.6 the capacity of the PyBAR platform for signalling has been further discussed. Since signalling takes place rather infrequently, performance issues are not as important as for packet processing. Similar to the integration architecture for RSVP and Differentiated Services in Chapter 5 an active network based architecture was presented, providing similar functionality. In contrast to the approach of Chapter 5 which requires a central bandwidth broker component, active networks can provide similar functionality without the need of central instances. This is why active mechanisms can provide a better scalability.

Another application of active networks was the establishment of Quality of Service supported tunnels spanning multiple ISPs. Even in such a scenario active network

mechanisms can be used to provide mechanisms working without the need of central instances like topology databases and therefore can work in parallel.

Chapter 7

Summary and Conclusion

The thesis focuses on mechanisms providing Quality of Service for the Internet. Besides the basic concepts like Differentiated and Integrated Services, management related tasks are also addressed. Especially the concept of Differentiated Services as proposed by the IETF plays a central role. In contrast to Integrated Services with the Resource Setup Protocol, Differentiated Services are especially suited to provide scalable Quality of Service in heavy loaded Internet backbones.

Evaluation of Differentiated Services

Since the performance of Differentiated Services depends on traffic conditioning components, the ns network simulator has been used to evaluate several scenarios. A special Differentiated Services extension of the simulator has been implemented and is described in Section 3.1 providing the needed DiffServ traffic conditioning components. The simulations performed with ns in the first place concern the Assured Forwarding service. In contrast to other Differentiated Services, Assured Forwarding requires fundamentally new traffic conditioning components (like the RIO and the TRIO queue). The components have to be evaluated appropriately. A first set of experiments focused on the impact of different parameter sets for the RIO queue (see Section3.2). Due to the design of the RED based RIO and TRIO queues (see Section 2.4.6), specific influence of particular parameters is hardly to determine. Since RED has been designed to work with TCP only, an aggregate of different protocols makes a proper choice of RED parameters nearly impossible. But even if the concrete behaviour of the RED based RIO component is complicated, the differentiation between different service levels is evident.

The proposed RIO traffic conditioning components for Assured Forwarding have some drawbacks and therefore, a new queueing mechanism has been presented in Section 3.3. The new component has the advantage of providing a better control over packet delay than the standard mechanisms and allows a simpler configuration. The new algorithm has been evaluated within different scenarios demonstrating the improved delay control. The new mechanism allows to exactly specify the maximum upper delay of a high priority packet. To minimise the changes to the packet order arising from such an approach, packet re-sequencing is applied.

Concluding the ns experiments, tests regarding the fairness of Assured Forwarding have been performed. The results show clearly the need for proper network provisioning but also prove the capability of DiffServ to provide Quality of Service. Especially DiffServ is able to protect congestion avoiding TCP flows against aggressive UDP traffic. Between identical protocols a fair sharing of bandwidth resources can be achieved.

The experiences with the network simulator initiated the development of a concept for the emulation of networks (see Chapter 4). It is based on Virtual Router (VR) programs emulating Internet routers, which can be connected to set up large topologies. The architecture allows to distribute a network emulation over several computers with multiple Virtual Routers per computer. Virtual Routers can be connected to real end systems, which allows to set up networks consisting of real and emulated network devices.

Compared to simulation programs, this approach has the advantage that no traffic generators and protocol stacks have to be implemented for a simulator, but a real end system can be integrated into the emulated network. Since the emulated network behaves similar to a real one, any application can be used to send traffic through that topology.

Since the Virtual Routers do not require any modification of the operating system kernel like other approaches, they can run completely in user space. Even the emulation of several separated networks on a single computer is possible. For the configuration of VRs a command line interface is available, providing commands similar to those of a Linux router. Of course an API is provided as well.

Because Virtual Routers have to process packets in real time, the number of VRs per computer must of course be limited. Also distributing them to multiple computers increases the delay of packets exchanged between VRs on different computers (see Section 4.3). The result of several tests shows that Virtual Routers produce similar packet delays and the same linear dependency of Round Trip Time on the number of hops as known from real networks.

Many mechanisms available in Internet routers have been implemented for Virtual Routers, for example flexible traffic conditioning components (Section 4.1.2), routing (Section 4.1.3) and tunnelling mechanisms (Section 4.1.5).

Based on the Virtual Router traffic conditioning components, Differentiated Services networks have been emulated and evaluated. A comparison of the results with the ns experiments shows a similar behaviour of both evaluation tools and confirm the capacity of the Differentiated Services approach to provide Quality of Service. Since Virtual Routers do not need any traffic generators or sinks like simulators but use real end systems, the obtained results meet the reality more closely. Due to the concept of emulating a network, the simple integration and evaluation of standard applications like web servers and browsers is also supported and provides the possibility to use measurement programs, originally designed for real networks.

Virtual Routers are a rather open environment providing a basis for various experiments. Therefore, besides the experiments presented here, Virtual Routers have also been used for the evaluation of a Software Agent based monitoring system [Gün01].

Quality of Service Management

The results of the evaluations show the dependency of Differentiated Services on good network provisioning and a proper configuration of network devices. Special techniques are required to manage and reserve resources within a Differentiated Services network and to configure network devices. Besides manual configurations using SNMP or telnet, an automatic resource management and provisioning within DiffServ based core networks can be achieved. Since the Resource Reservation Protocol provides good end user support but lacks scalability within Internet backbones and since Differentiated Services can provide exactly this scalability an integration of both concepts is proposed.

The proposed architecture is based on a RSVP/DiffServ Gateway (RDG), which supervises RSVP reservation setup requests. The RDG reacts to these signals by setting up tunnels through the DiffServ network, aggregating multiple RSVP flows. Quality of Service is provided to the tunnels by Differentiated Services. The main benefit of the RDG approach is the ability to combine the strengths of both the RSVP and the Differentiated Services concept. RSVP provides the signalling to set up the resources in the access networks and also provides feedback to the customer, whether resources have been set up correctly. On the other hand the core network does not have to process single RSVP flows but can handle the aggregate tunnels efficiently using Differentiated Services.

The set up of the tunnels and the configuration of the DiffServ devices is provided by a central bandwidth broker instance. This instance also controls, whether a reservation requests is permitted or denied.

Active Quality of Service Management

Even if the developed integration concept for RSVP and Differentiated Services works fine for small and medium size networks, the need for a central bandwidth broker instance causes scalability problems in large networks.

Due to the shortcomings of classical network management, a concept to use Active Networks for the support of new Internet services is proposed (see Chapter 6). Active Networks allow to send small programs through a network, which are executed by active routers forwarding the packets containing the programs. An active network environment using the Python programming language has been implemented. Python has a lot of similarities with Java like object orientation and platform independent code, but can be adapted more easily to specific needs and also allows the simple integration of native code elements.

The Python based active router (PyBAR) has been implemented for Virtual Routers and for Linux Routers. For the PyBAR environment, a set of modules providing convenient, frequently used mechanisms like encryption and authentification has been implemented (see Section 6.2.5). An advantage of the PyBAR system is the combination of a high level interpreted language with extension mechanisms, which can be provided by native code. The high level language provides the required platform independence, a simple syntax and flexibility, while the dynamic loadable extension modules provide the required speed.

Based on the PyBAR environment a set of active services has been implemented. These are application specific dropping mechanisms, a simple active multicast service supporting a video streaming application and an active service allowing to handle active tunnels. These services show the power of an Active Network approach to establish rapidly new services and also to provide general purpose management support. Such active services can support applications as well as reduce resource consumption within the network (see Section 6.5.1). Since active services can be installed or updated very easily, they can be adapted quickly to new applications and be installed on active network devices.

The PyBAR system was used to develop a general approach of active Quality of Service management. Such an approach can provide more autonomy and support for network management by distributing management to the network itself and decrease the need for central instances like management stations or bandwidth brokers. Configurations can be processed more locally, also allowing the management of large networks. Similar to the RSVP/DiffServ gateway, components mapping different resource reservation schemes can be established (see Section 6.6.3). These schemes do not require central instances but can configure network resources by active packets. The power of active networks for signalling purposes can also be used to establish services spanning multiple Internet Service Providers. Such a mechanism allows to dynamically set up Quality of Service supported tunnels (see Section 6.6.4). A special advantage of this approach is the capability of an active packet to behave more intelligent than standard signalling procedures can do. Additionally to the simple adaptability, reasonable default fallback solutions can be provided.

Outlook

The Virtual Router approach is still at its beginning. Within this thesis Virtual Routers were mainly used to emulate Quality of Service and Active Networks, but several other purposes are obvious:

- Since the communication channels to exchange packets may be disconnected and established during run time, node mobility can be emulated. Similar to a wireless network with an end system leaving the range of a base station, a communication channel can be disconnected and established to another VR.

  Besides Quality of Service aspects, this can also allow the evaluation of (active) routing protocols, or the evaluation of mechanisms used to establish mobile ad hoc networks.

- Virtual Routers emulate Internet routers and therefore provide similar functionalities. These similarities are not restricted to packet forwarding, but also cover the syntax of command line interfaces. This makes VRs a useful tool for student exercises, since students can set up and operate IP networks without any need for physical network devices. The capability to be configured via telnet or from a front-end via an API may also enable the set up of remote exercises.

- Of course other protocols might be ported to Virtual Routers as well, for example RSVP, Internet routing protocols, or multicast. Also, the implementation

of additional protocol stacks is a possibility to enhance the usability of Virtual Routers. Supported by the convenient development and evaluation environment provided by Virtual Routers allows a rapid prototyping and evaluation is possible.

- The Active Network system provided for the Virtual Router and Linux may also be extended and used for other kinds of purposes. The general ability to provide simple multicast services has already been shown. A possible extension may be the integration of classic and explicit multicast. Active components can map explicit multicast packets to classical multicast groups and vice versa. This way explicit multicast may be used within networks not capable to provide a classic multicast service. This idea might also be useful, since classic multicast is a rather heavy weight concept and for networks with only a small number of receivers explicit multicast may be prefered. Active components can be used to perform the required signalling and to configure the network devices.

- Another task for the Active Network environment might be the implementation of flexible and autonomous monitoring tools. Active Nodes can monitor traffic and react intelligent on exceptional traffic situations. The interaction of various nodes and the exchange of informations can be used to provide scalable solutions for tasks like the tracking of Denial of Service attacks.

Appendix A

Virtual Router API

This document describes the interface between the Virtual Router's (VR) core mechanisms and an application running on top of the VR. The application may use either the loadable object mechanism to access the Virtual Router or is run as an external process using a communication channel to interact with the Virtual Router.

A.1 API channels and VRCB handles

A Virtual Router may have several APIs each used by another application like a shell, a packet monitor or a graphical front end. Each API establishes an API channel, a duplex connection between the Virtual Router and the program.
Since only this channel is used for the communication, the program may also access a Virtual Router on a remote computer.
The communication is based on Virtual Router Control Blocks (VRCBs) and Virtual Router Result Blocks (VRRBs). The Virtual Router receives a control block, parses it, executes the command and returns an appropriate result block. There are several control and result blocks for different commands and the according results. All data types are in network byte order. A ulong specifies a four byte, a ushort a two byte wide integer.
To allow a simple parsing of the control and result blocks a hierarchy of control block exists. Each block starts with a generic header (VRCB/VRRB) and is followed by command dependent data. A specifier at the beginning of the control block defines the format of the following data. Additionally, the basic VRCB contains a handle and information about the control block lengths.

VRCB	**Control Block**
ushort	Handle of the control block, it will be referenced in the returned result block
ushort	total length of control block
ushort	VRCB command specifier

The handle is unique and referenced by the returned result block. This allows a simple mapping between commands and the according results. Currently, the following VRCB command specifiers are defined:

VRCB command specifiers	
1	add an interface to the Virtual Router
2	delete an interface
3	get list of interfaces. This will return a list of numbers used to reference an interface
4	query an interface by its name.
10	changing interface parameters (bandwidth, ...), modify the queueing system
20	list, add and delete routing table entries
21	query the event handler and return the event status of scheduled events or devices like interfaces
30	setup, list and remove filter setup
31	add a new protocol stack to the Virtual Router
40	list, add and remove loadable objects
50	send an IP packet to the Virtual Router

The Virtual Router returns a specific Virtual Router Control Block according to the received control block. Each result block starts with a basic VRRB.

VRRB	**Result Block**
ushort	handle referencing the according control block
ushort	type of the returned result block
ushort	total length of the result block

It references the according VRCB by a handle. The result block type is important to process asynchronous events. Usually a control block is directly responded by the according result block (type 0). Some configurations can cause the Virtual Router to send additional information asynchronously over the API channel. Therefore this field is set to signal the type of result block received. Currently the following values are defined:

Result Block Types	
0	synchronous RB sent as an answer to a CB
1	filtered packet
2	packet for protocol stack

If a filtered packet is sent over the API channel, the handle of the result block refers to the handle of the control block sent to set up the filter. The API guarantees that there are no asynchronous packets sent between a control block and its synchronous answer. However there might be some asynchronous packets still in the line. Therefore incoming datagrams should be checked for their handle and result block type.
This control block - result block mechanism is used for all types of configuration. The control blocks are structured. Additionally to the VRCB command specifier there might be additional command specifiers controlling certain parts of a component (e.g. control of the queueing system of an interface).

A.2 Adding an Interface

To add an interface a special VRCB has to be sent over the API. This VRCB contains fundamental information about the interface to be added.

IF_ADDINTERFACE_CB	**Control Block**
VRCB	control block with handle, length and base command specifier
byte[10]	name of the interface. The string has to be followed by a 0-byte
ulong	ip address in network byte order as returned by `inet_addr()`
ulong	netmask in network byte order
ulong	broadcast address

The name field has to be terminated by a null byte. A different name should be used for each interface. The Virtual Router will return block containing information about the interface.

IF_INFORMATION_RB	**Result Block**
VRRB	result block with the reference handle and the length
byte	the internal number of the interface within the VR. This parameter is set to 0xffff if the request fails.
byte[10]	the name of the interface
ulong	ip address
ulong	netmask
ulong	broadcast address
ulong	number of received packets (rx)
ulong	number of transmitted packets (tx)
ulong	errors during transmission/reception
ulong	packets dropped by the interfaces queueing system
ulong	bandwidth in bytes per second
ulong	bucket size of the interface's rate limiter in bytes
ulong	bucket level of the limiter in bytes
ulong	ip translation for outgoing packets: address
ulong	ip translation for outgoing packets: netmask
ulong	ip translation for outgoing packets: new address
ulong	ip translation for outgoing packets: pattern
ulong	ip translation for incoming packets: address
ulong	ip translation for incoming packets: netmask
ulong	ip translation for incoming packets: new address
ulong	ip translation for incoming packets: pattern
byte	connection type; 0: not connected, 1: softlink, 2: udp-tunnel, 3: local link (FIFO), 4: ipip tunnel (logical interface)
ulong[3]	connection parameters

On error the interface number field is set to 0xffff. See table on page A.6.2 for a description of connection parameters.

A.3 Deleting Interfaces

IF_DELETEINTERFACE_CB	**Control Block**
VRCB	the control block header
ushort	the number of the interface to be deleted

As a response to this control block the Virtual Router returns:

IF_DELETEINTERFACE_RB	**Result Block**
VRRB	result block header
ushort	number of the deleted interface, 0xffff on error

A.4 Querying Interface Numbers

To query a list of existing interface numbers, a VRCB with the according VRCB command specifier is sent to the VR. The VR will return a result block with a list of valid interface numbers.

IF_IDQUERY_CB	**Result Block**
VRRB	result block header
byte[]	returns a list of interface numbers

The length field in the result block header can be used to calculate the number of returned interface numbers.

A.5 Query Interface by Name

The special command here is to query the interface id by the name of the interface. The according control block is:

IF_IDNAMEQUERY_CB	**Control Block**
VRCB	control block header
byte[10]	name of the interface the number has to be queried

The Virtual Router will return an interface information result block as described on page 171. If there is no matching interface within the Virtual Router the number field of the returned result block will be set to 0xffff.

A.6 Configuration of a specific Interface

There are several interface specific parameters or commands. Each control block addressing a specific interface starts with an extended VRCB called IF_VRCB providing the number of the interface and an interface specific command code.

IF_VRCB		**Control Block**
VRCB	the control block header	
ushort	interface number	
ushort	interface specific command	

The interface specific command determines also the format of the rest of the control block. The following table lists the currently defined command codes.

interface specific command codes and their parameters		
1	-	query interface information (see section A.6.1)
2	-	reset all counters of an interface (rx,tx,dropped packets, errors)
101	byte[10]	set the interface name. The string should be ended by a 0 byte
102	3 x ulong	change of ip-address, netmask, broadcast (see section A.6.2)
105	2 x ulong	configure the interface bandwidth (bytes per s) and bucket size (bytes) (see section A.6.2)
107	4 x ulong	set up of translation table for received packets (ip address, netmask, new value, pattern)
108	4 x ulong	set up of translation table for packets to be sent (ip address, netmask, new value, pattern)
109	byte + ulong + 2x ushort	connect or disconnect the interface (see section A.6.2)
120	...	configuration of the interface's queueing (see section A.7)

A.6.1 Querying Interface Configuration

This interface command takes no arguments. An IF_VRCB with the according command value 1 is passed to the Virtual Router. The Virtual Router returns an IF_INFORMATION_RB as described on page 171. If an error occurs the field of the result block containing the interface number is set to 0xffff.

A.6.2 Setting of Interface Parameters

As can be seen in the previous table several parameters of the interface can be modified. Each of these configurations is done by an IF_VRCB with the appropriate command specifiers and some additional command dependent data.
Any of this requests will be answered by a returned IF_INFORMATION_RB with the interface number field set to 0xffff if an error occurred. The following list will briefly describe the commands and the required parameters.

interface bandwidth/bucket size (105): The first parameter specifies the bandwidth in bytes per second, the second one the bucket size in bytes. If the second parameter is set to 0xffffffff the bandwidth only is modified.

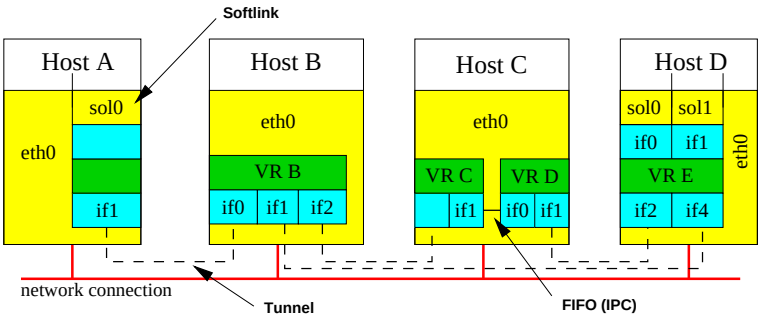
ip, netmask and broadcast addresses (102): This API call requires three parameters. If a parameter is set to 0xffffffff, the according value of the interface is not changed. Each address has to be provided in network by order as returned by the `inet_addr()` function.

(dis)connect a Virtual Router interface (109): Since a Virtual Router interface may be connected in different manners this control block takes a set of parameters. The meaning of this parameters depend on the type of the connection.

IF_CONNECTION_CB	**Control Block**
IF_VRCB	interface control block header
byte	connection type
ulong[3]	connection parameters

The connection parameters and their meaning are defined in the following table.

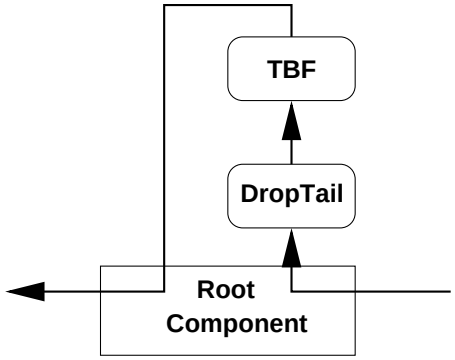
Connection Types and Parameters				
	type	[0]	[1]	[2]
disconnected	0	-	-	-
Softlink	1	-	#no	-
IPC	2	-	#no	1/2
UDP	3	destination address	tx port	rx port
IPIP tunnel	4	destination addressR	-	MTU



The Virtual Router will return an result block with interface informations as described on page 171. According to the type of error either the interface number field is set to 0xffff or the connection parameters shows a (dis)connected device.

A.7 Modifying the Queueing System

The queueing system consists of a set of components like FIFO queues, schedulers and classifiers which can be connected to set up complex and powerful queueing systems.



The root component is the basis for all other components and connects the queueing system with the interface

Each different component type has an type id as listed on the following table. This id is used to signal a components type within the control and result blocks.

Component types and ids	
component type id	
0	the root component itself
1	droptail queue, a simple FIFO queue
2	Token Bucket Filter to limit bandwidths with variable rate and bucket size
3	generic multi field classifier
4	generic scheduler with RR, WRR and PWRR support
5	trio queue for differentiated services
6	a packet marker as required for differentiated services

The components listed in the table may be created, connected, disconnected and removed. All commands for the queueing system or queueing components start with If_Q_VRCB extending the IF_VRCB control block used for interface configuration. To access the queueing system the appropriate command specifiers have to be set in the VRCB, the IF_VRCB and in the IF_Q_VRCB. The following table lists the command specifiers for the IF_Q_VRCB.

IF_Q_VRCB	**Control Block**
IF_VRCB	the interface specific control block, containing also the fundamental VRCB
ushort	queueing system specific command

The table lists the command set which can be sent to an interface's queueing system. This set covers mainly tasks concerning all components. To configure a single component the command specifier #10 provides direct access to a component. The configuration of specific components depends of course crucially on the component's type requiring special control and result blocks as will be described in the next section.

basic queueing system commands		
1	-	list all components (section A.7.1)
2	-	list all connections (section A.7.2)
3	ushort	create component (section A.7.3)
4	ushort	remove component
5	2 x ushort	connect two components (section A.7.5)
6	2 x ushort	disconnect two components (section A.7.6)
10	...	component specific configuration

A.7.1 Query List of Components

To request a list of components the IF_Q_VRCB does only contain the command specifier, and no additional data is needed. The returned result block looks like this:

	Result Block
VRRB	the result block header
ushort[]	list of (c-id/c-type) pairs

Each list element has two fields. The first one specifies the component's id, the second one the type of the component. As mentioned previously, each component in the queueing system has an unique id used to address the component. To calculate the number of components contained in the result block, the length field within the VRCB can be used.

A.7.2 Query list of Connections

A control block with the queueing command specifier #2 will return a result block with a listing of all connections between the components.

	Result Block
VRRB	the result block header
ushort[]	list of [c-id/c-type, c-id/c-type] tuples

Each list elements contains information for both components a connection. Therefore four values are included For each end point a c-id/c-type pair is provided. The first pair specifies the start point of the connection, the last one the end point. As usual the length field in the VRRB can be used to calculate the number of connections.

A.7.3 Create a new Component

A new component is created by sending the component type to the API.

	Control Block
IF_Q_VRCB	the queueing system header block
ushort	type of component to be created

The API answers with an ip/type tuple.

	Result Block
VRRB	the common result block header
ushort	the component id of the created component
ushort	the type of the created component

On error the component id is set to `0xffff`. In future releases the component type may be used for further error codes.

A.7.4 Remove a Component

For the removal of a component the component has to be disconnected first. If the component is not connected the following control block will remove the component.

	Control Block
IF_Q_VRCB	the queueing system header block
ushort	the id of the component to be deleted

The API answers with an ip/type tuple.

	Result Block
VRRB	the common result block header
ushort	the component id of the removed component
ushort	the type of the removed component

The component id is obsolete after the removal of a component and might be re-used if other new components are created. On error the returned component id is set to 0xffff, the component type field indicates the error more precisely as listed on the following table.

Error codes during component removal		
component id	component type	
0xffff	1	cannot delete root component
0xffff	2	no such component
0xffff	3	component busy

A.7.5 Connect two Components

A connection between two components allows to pass packets from one component to the next. A component may have multiple connections to other components, as for example a classifiers puts packets to different queues. Whether a component allows multiple connections or not is determined by the component itself. The TBF has one input and one output slot. Therefore a TBF can have two connections:

→ **tbf** The tbf is the end point of a connection and receives packets from another componennt.

tbf ← The tbf is the start point of a connection and sends packets to the nect component.

Any additional connection for a Token Bucket Filter would result in an error.

	Control Block
IF_Q_VRCB	the queueing system header
ushort	component id 1
ushort	component id 2

The Virtual Router returns a result block with a pair of both component ids.

	Result Block
VRRB	the result block header
ushort	component id 1
ushort	component id 2

Both component ids are also used to indicate errors as shown on the following table.

Error codes during Component Connection		
id 1	id 2	description
0xffff	1	first id invalid, no such component
0xffff	2	second id invalid, no such component
0xffff	3	connection to next failed, component busy
0xffff	4	connection to previous failed, component busy
0xffff	5	invalid length

A.7.6 Disconnect two Components

The disconnection requires a similar data format as used for the connection of queueing components. Also the error codes returned within the result block if an error occurs are similar to those used during component connection.

A.7.7 Component Configuration

The commands dealing with the queueing system presented so far covered tasks more general like the creation of queueing components and their connection.
In this section the configuration of specific queueing components will be described. As explained before each component has an unique identifier. To address a specific component the previously introduced IF_Q_VRCB control block is extended to an IF_QS_VRCB.

IF_QS_VRCB	**Control Block**
IF_Q_VRCB	queueing system control block header
ushort	component id of the addressed queueing component
ushort	component specific command

As mentioned before the control blocks follow some kind of hierarchy. Therefore, the IF_Q_VRCB header contains other control block headers. To illustrate that hierarchy, the following table lists a complete IF_QS_VRCB.

IF_QS_VRCB	**Control Block**
ushort	control block handle
ushort	length of the control block
ushort	VRCB command specifier = 10
ushort	interface number
ushort	interface specific command (IF_VRCB) = 120
ushort	queueing system command = 10
ushort	component id of the addressed queueing component
ushort	component specific command

Theoretically a component specific command code might have different meanings for different components. Since there are enough command specifiers available and a multiple usage of the same value would make things more complicated, different component command codes were defined. As can be seen in the following table, component specific codes of a value less than 100 are dedicated to functionalities common to all components like the information query or the resetting of statistical counters. The meaning of command codes over 100 may differ for different components.

Command codes for specific queueing components		
code	component	
1	all	reset the statistical counters of the component
2	all	information request
210	2	configure Token Bucket Filter
310	1	configure Droptail (FIFO) queue
410	4	change the scheduler mode
411	4	modify a scheduler weight/priority
511	3	adding a classifier rule
512	3	removing a classifier rule
611	6	add a rule to the Differentiated Services marker
612	6	delete a rule within the Differentiated Services marker
710	5	configure the Differentiated Services TRIO queue

Of course each of the component dependent commands requires a specific configuration datagram. The control blocks have only the IF_QS_VRCB part in common.

Reset the Counters of a Component

A resetting of a component affects all statistical counters of this component. The function is common to all components of the queueing system. An IF_QS_VRCB with the appropriate command code is sent to the API. As result an empty VRRB is returned.

Query a Component's Status

For a request of component parameters and statistic counters an IF_QS_VRCB control block with the appropriate command code only is necessary. Obviously, the amount and type of data returned depends crucially on the component type. However, each result block returned starts with:

QS_STAT_VRRB	**Result Block**
VRRB	the general result block header
ushort	component id
ushort	component type

The rest of the result block depends on the component. The following list described the result blocks returned by each component.

Droptail Queue: contains information about the only parameter of a droptail queue, the queue length and statistical information, how many packets passed the queue, how many packets were dropped a.s.o.

	Result Block
QS_STAT_VRRB	common statistical header
ushort	length of queue in packets
ulong	dropped packets
ulong	deque requests
ulong	deque hits
ulong	deque fails

Root Component: Since all packets sent to the queueing system pass the root component, the statistical data in this datagram gives a very global impression about the queueing system's activities.

	Result Block
QS_STAT_VRRB	common statistical header
ulong	enqued packets
ulong	dequed packets
ulong	packets stored in the queueing system
ulong	packets dropped by the queueing system

Token Bucket filter: The token bucket filter is defined by a token rate and a bucket allowing to buffer a certain amount of tokens.

	Result Block
QS_STAT_VRRB	common statistical header
ulong	bucket rate in bytes per second
ulong	bucket size in bytes
ulong	enque hits (internal)
ulong	enque fails (internal)
ulong	enque empty (internal)
ulong	deque hits (internal)
ulong	deque fails (internal)
ulong	deque empty (internal)
ulong	bucket overflows

Generic Scheduler: The generic scheduler can have multiple components connected, it receives packets from and one outgoing component. How incoming packets are treated depends on the scheduler mode. A query for statistical data reveals data for each connected component. Each component sending data to the scheduler is described by a GEN_SCHED_REC.

GEN_SCHED_REC	**Data Record**
ushort	type of preceding component
ushort	id of the preceding component
ulong	successful deque events
ushort	priority/weight (depends on scheduler mode)

These records and also additional information about the current scheduler mode are contained in the result block returned following to an information request.

	Result Block
QS_STAT_VRRB	common statistical header
ushort	type of next component
ushort	id of next component
ushort	scheduler mode (PPR, WRR, PWRR)
GEN_SCHED_REC[]	the scheduler records

Classifier: The status information of the classifier mainly contains information about the rules used to forward packets to other components. The VRRB contains a record for each outgoing component. These records are appended to the result block header.

	Result Block
QS_STAT_VRRB	common statistical header
ulong	deque events
ulong	enque events
QS_CLASS_REC[]	the classifier's records

QS_CLASS_REC	**Data Record**
ushort	component type
ushort	component id
ulong	source ip address
ulong	source netmask
ulong	destination address
ulong	destination netmask
ushort	ToS byte, use 0xffff to ignore this field
ushort	protocol, use 0xffff to ignore this field

Differentiated Services Marker: The Differentiated Services marker returns the following structure with statistical information. The structure contains overall information about the marking and the remarking the component has performed and information about each marker rule.

	Result Block
QS_STAT_VRRB	common statistical header
ulong	enqueued packets
ulong	new marks set by the marker
ulong	packet kept old mark
DSM_REC[]	the records with the marker rules

This structure is followed by a number of DSM_RECs. These records are related to the marker rules set up by preceding control blocks.

DSM_REC	**Data Record**
ulong	source ip address
ulong	source netmask
ulong	destination address
ulong	destination netmask
ushort	ToS byte, use 0xffff to ignore this field
ushort	protocol, use 0xffff to ignore this field
ushort	the service type: EF, AF, ..
ulong	service dependent parameters

The precise number of DSM_RECs depends of course on the number of set up rules. The number of records contained within the control block can be calculated by the control block's size.

TRIO queue: The datagram contains mainly data about the queue lengths set for each dropping precedence. Additionally to some statistical information the mode the TRIO queue is currently working in is returned.

	Result Block
ushort	the mode of the TRIO queue: linear, boolean, red
ushort[3]	three queue lengths
ushort	packets currently in the queue
ulong[3]	packets dropped with the different drop probabilities
ulong	deque requests
ulong	deque hits
ulong	deque fails

Configuration of Token Bucket Filters

	Control Block
IF_QS_VRCB	control block header for the queueing system
ulong	bandwidth in bits per second
ulong	bucket size in bits

This datagram changes the settings of a token bucket filter component. The two only parameter are the bandwidth and the bucket size. The bandwidth is measured in bits per second, the bucket size in bits. If a parameter is set to 0xffffffff, the API will not change the according token bucket filter parameter. The Virtual Router returns a datagram with statistical information about the token bucket filter as described on page 180.

Configuration of the Droptail Queue

	Control Block
IF_QS_VRCB	control block header for the queueing system
ushort	new queue length

This allows to modify the maximum number of allowed packets in the droptail or FIFO queue. The control block starts with the usual IF_QS_VRCB and contains an additional field for the queue length only. The Virtual Router returns the statistical result block for the droptail component.

Modify Scheduler Weights/Prios

	Control Block
IF_QS_VRCB	control block header for the queueing system
ushort	component id of the incoming component
ushort	new weight/priority

Weights and priorities of the scheduler determine – dependent on the scheduler – mode the amount of bandwidth the connected component can achieve. In weighted round robin (WRR) mode the share s of a component x is calculated by:

$$s_x^{wrr} = \frac{w_x}{\sum_{i=0}^{N} w_i}$$

In the PWRR mode the component with the heightest weight $x = 0$ is processed if a packet is available. The share s for the other components $x > 0$ can be calculated by:

$$s_x^{pwrr} = \frac{w_x}{\sum_{i=1}^{N} w_i}$$

In the Round Robin mode the weight of the components is ignored, in the Priority Round Robin mode incoming packets are processed in order of their component's weight. The Virtual Router returns a result block containing the changed record (GEN_SCHED_REC) (see page 181). If an error occurs (e.g. there is no such outgoing component) a plain VRRB will be returned.

Change Scheduler Mode

The mode of the scheduler is changed by this datagram.

	Control Block
IF_QS_VRCB	control block header for the queueing system
ushort	new mode

The VR returns statistical data for the scheduler (see page 181) but without the records for the connected components. The appropriate mode values are 0 for Priority Round Robin, 1 for Weighted Round Robin, 2 for Round Robin and 3 for Priority Weighted Round Robin. A change of the mode parameter does not affect the connected components. The weights used during weighted fair queueing are interpreted as priorities during RR, PRR and PWRR. So the mode may be switched without reconfiguring the weights/priorities.

Adding a Classifier rule

	Control Block
IF_QS_VRCB	control block header for the queueing system
ushort	id of the outgoing component
ulong	source address
ulong	netmask for source address
ulong	destination address
ulong	netmask for destination address
ushort	protocol
ushort	ToS

This datagram adds a filter rule for an outgoing component. To avoid a filtering by protocol or ToS value set according values might be set to 0xffff. The Virtual Router returns an information block for the classifier with the added record appended has explained on page 181.

Removing a Classifier rule

	Control Block
IF_QS_VRCB	control block header for the queueing system
ushort	id of the outgoing component
ulong	source address
ulong	netmask for source address
ulong	destination address
ulong	netmask for destination address
ushort	protocol
ushort	ToS

This datagram allows to delete a filter rule. The first rule matching the datagram will be removed from the classifier's table. If multiple matching rules exist, this datagram has to be used repeatedly. The Virtual Router returns a result block with classifier statistics and the deleted record GEN_CLASS_REC appended (see page 181).

Adding a Differentiated Services Marker Rule

	Control Block
IF_QS_VRCB	control block header for the queueing system
ulong	source address
ulong	netmask for source address
ulong	destination address
ulong	netmask for destination address
ushort	protocol
ushort	Differentiated Service Code Point (DSCP)
ushort	new DSCP
ulong[4]	parameters

The datagram allows to add a rule to the Differentiated Service marker. The addresses and netmasks allow to specify a set of flows. If the values for the protocol and the DSCP are to be ignored, those fields in the datagram might be set to 0xffff. The API returns a statistical block with the added rule (DSM_REC) appended as explained on page 182. The parameter block (ulong[4]) depends on the service type. If the service type is EF the parameter block might remain empty. Using the Assured Forwarding service the block has the following meaning:

Service Parameters for Assured Forwarding	
#0	max bandwidth in bytes for low drop precedence
#1	bucket size in bytes for low drop precedence
#2	max bandwidth in bytes for medium drop precedence
#3	bucket size in bytes for medium drop precedence

Removing a Differentiated Services Marker Rule

	Control Block
IF_QS_VRCB	control block header for the queueing system
ulong	source address
ulong	netmask for source address
ulong	destination address
ulong	netmask for destination address
ushort	protocol
ushort	ToS
ushort	new ToS value

The datagram allows to remove a previously set rule from the Differentiated Service marker's internal table. The API returns a block containing statistics about the marker and the record with the removed rule(see page 182).

Modifying the TRIO queue

	Control Block
IF_QS_VRCB	control block header for the queueing system
ushort	mode
ushort	queue length 1
ushort	queue length 2
ushort	queue length 3 (maximum queue length)

The TRIO queue has two ranges of drop probabilities defined by the three queue length parameters. The last queue length variable (3) also defines the overall length of the queue, while the two others define the dropping probabilities for packets with low and medium drop precedences. Also important is:

$$q_1 \leq q_2 \leq q_3$$

The value of the mode field influences the behaviour of the queue. There are several values defined:

0 (hard): This is the droptail mode. The queue lengths define a fixed limit for the packets of a certain packet type. Therefore packets with high drop precedence are not put into the queue if the queue length is longer than q_2 and packets with medium drop precedence are discared if the queue length exceeds q_1.

$$p_{low} = \begin{cases} 0 & \text{for } l \leq q_3 \\ 1 & \text{else} \end{cases}$$

$$p_{medium} = \begin{cases} 0 & \text{for } l \leq q_2 \\ 1 & \text{else} \end{cases}$$

$$p_{high} = \begin{cases} 0 & \text{for } q \leq q_1 \\ 1 & \text{else} \end{cases}$$

1 (linear): The dropping probability is increased linearly between the values for the queue lengths. This is a simple version of the RIO mechanism.

$$p_{low} = \begin{cases} 0 & \text{for } l < q_2 \\ \frac{l-q_2}{q_3-q_2} & \text{for } q_2 \leq l < q_3 \\ 1 & \text{for } l \geq q_3 \end{cases}$$

$$p_{medium} = \begin{cases} 0 & \text{for } l < q_1 \\ \frac{l-q_1}{q_2-q_1} & \text{for } q_1 \leq l < q_2 \\ 1 & \text{for } l \geq q_2 \end{cases}$$

$$p_{high} = \begin{cases} \frac{l}{q_1} & \text{for } 0 \leq l < q_1 \\ 1 & \text{for } l \geq q_1 \end{cases}$$

If certain parameters of the TRIO queue should be changed only, the other ones can be set to 0xffff. The API will ignore this fields.

A.8 Routing

This section specifies the data structures used to set up and query the routes used to forward IP packets to specific interfaces. Like the interface configuration the configuration is done by VRCBs and VRRBs as described on page 169. Similar to the interface configuration a generic control block is defined to access the routing system.

ROUTE_VRCB	**Control Block**
byte	routing specific command

The routing system needs three different commands only:

Routing Command Codes	
1	add a route to the routing table
2	delete a route from the routing table
3	list all routes of the table

A route is represented by a ROUTE_REC datagram, which is used by all control blocks dealing with routing table entries.

ROUTE_REC	**Data Record**
ulong	source address, 0x0 to disable source based routing
ulong	source netmask
ulong	destination address, 0x0 to ignore disable destination based routing (any practical use ?)
ulong	destination netmask
ushort	protocol, 0xffff to ignore this field
ushort	DSCP value, 0xffff to ignore
ushort	number of interface, the packet shall be routed over

A.8.1 Adding routes

The control block to add a route is:

ROUTE_RECORD_VRCB	**Control Block**
ROUTE_VRCB	routing control block header
ROUTE_REC	routing record

The fields to be ignored by the routing mechanism have to be set to the default values. For a normal ”unix-like” routing, only the destination based routing entries have to be set. The interface id can be queried as described in A.5. The Virtual Router will return a result block like:

ROUTE_RECORD_VRRB	**Result Block**
VRRB	result block header
ROUTE_REC	routing record

On error, the interface number is set to 0xffff. The DSCP field may then be used for more detailed error codes.

A.8.2 Deleting Routes

The deleting of routes works in the same way as the adding of routes. Only the subcommand specifier is different.

A.8.3 Querying Routes

To query the routes a ROUTE_VRCB with the appropriate subcommand specifier has to be sent over the API. The Virtual Router returns a list of ROUTE_RECs. The length field on the VRRB header has to be used to calculate for the number of returned routes.

	Result Block
VRRB	result block header
ROUTE_REC[]	records of the routing table

A.9 Filter Setup

Filters are applied to the central forwarding mechanism. Each filter has a couple of fields specifying the wanted packets, a reference to the object where matching packets shall be sent to and a priority. The priority defines in which order the filters are applied to the transported packets. This is important as a filter might remove a matching packet from the network. As a consequence the packet will not reach the following filters.
After all filters have been applied, the packet is processed by the internal routing system. The following commands are defined to setup, list, modify and remove filters.

Filter specific command codes	
0	get a list of filters
2	add a filter
3	remove a filter

A filter is represented by a FILTER_REC structure:

FILTER_REC	**Data Record**
ulong	address of an appropriate ForwarderIF class
ushort	position the filter has to be applied
ulong	source ip address, NAK is 0xfffffffffd
ulong	netmask of the source address
ulong	destination address, NAK is 0xffffffff
ulong	netmask of the destination address
ushort	Type of Service, 0xffff to ignore
ushort	protocol 0xffff to ignore
ushort	IP options type
ushort	ip options value
ushort	filter mode copy / move
byte[32]	name of the filter
ushort	handle of the control block the filter was created with

The following list describes the variables in detailed:

address is a four byte wide pointer to an appropriate forwarder class. If the filter is initialised by a loadable module, the loadable module can define a function, which is directly called which an matching packet a argument. If this pointer is set to 0, the filtered packets are sent over the API channel as asynchronous VRRBs.

filter position defines in which order the filters are applied . This can be important as filters can also remove packets. The filters are processed in the order of their position. If a filter with a low position removes a packet, this packet is lost for all filters with higher positions.

source ip and source netmask specify the type of source addresses to be matched by the filter. The addresses have to be specified in network byte order. If no source filtering has to be applied use 0 and 0xffffffff as values.

destination ip and netmask same as the above for the destination address.

Type of Service the Type of Service byte to match the filter. 0xffff disables this filter parameter.

protocol This allows to filter all packets of a specific protocol (e.g. UDP, TCP, ICMP ...). 0xffff disables this filter function.

ip option type and value allows to react on special options in the IP header. (e.g. router alert). Set both to 0 to switch this feature off.

filter mode if this field is set to 0 only a copy of the packet is put to the filter. If it is set to 1 the entire packet is removed from the forwarder and put to the filter.

The appropriate control blocks to remove, list and add filters are described in the following sections.

A.9.1 Adding and Removing Filters

To add or remove a filter an appropriate VRCB has to be sent to the API. The VRCB contains the usual handle, the length field and the command to specify whether the filter has to be removed or added.

	Control Block
VRCB	the control block header with command specifier = 30
ushort	filter specific command
FILTER_REC	filter to be removed or added

The returned structure repeats the FILTER_REC contained in the according control block. On error the call back pointer is set to 0 and the name string of the FILTER_REC contains some short error string.

A.9.2 Query List of Installed Filters

This function can be used to query the actual installed filters. The structure FILTER_CB with the command code `0` results in a list of applied filters.

	Result Block
VRRB	the result block header
FILTER_REC[]	a list of filter records

As usual the length field within the VRRB header can be used to calculate the number of attached filter records.

A.9.3 Adding a Protocol Stack

Protocol stacks are accessed like filters. Either a call back function is specified within the control block or packets matching the protocol id are passed asynchronously via the API channel.

	Control Block
VRCB	the control block header with command specifier = 31
ulong	pointer to the call back function
ushort	protocol id

The Virtual Router result block contains:

	Result Block
VRRB	the result block header
ushort	protocol

On error the protocol field is set to 0;

A.10 Loadable Objects

This type of API calls allow the integration of Loadable Objects (LOBs) into the VR core. These objects are based on a class, derived from the virtual class LOB and are stored in a separate file. For some information about the commands used to load objects from the standard shell see chapter B.4. The Virtual Router distribution also containes some examples for Loadable Objects, like two different version of a ”Hello World” program and an example for a packet filter.
The loadable object specific commands are listed in the table below:

Loadable object specific command codes	
1	load object from file
2	unload object with specific id
10	get list of information blocks

A central data structure for loadable objects is the LOB_INFO_BLOCK . This is a special structure containing all data about a specific loadable object. This structure is used for API calls querying information or is returned after the loading or unloading of an object.

LOB_INFO_BLOCK	**Data Record**
ushort	an unique id number for the loadable object
ushort	the mode of the loadable object
ulong	the size of the object code
ulong	the time in seconds since the module was loaded
byte[32]	the loadable object’s name

If an error occurs the loadable object id field contains 0xffff and the objects name field a short error string.

A.10.1 Loading an Object

To load an object the filename of the object kernel is forwarded to the API. The API then loads the object and executes its constructor. The API (and therefore also the event scheduler) is blocked as long as the constructor is executed. So no time consuming or blocking code can be executed in the constructor.
The following control block tells the API to load the according loadable object. Since it is possible to pass command line parameters to the loadable object, the control block contains a list of 0-byte separated strings, each token representing a parameter. The first token is the name of the object. If the first token contains a simple filename only, the Virtual Router automatically adds the appropriate pathname and loadable object extension. If the filename contains a ’/’ or a ’.’, the VR will assume, that an absolute pathname is given and will look for the file at the specified location.

Control Block

VRCB	the usual control block header
ushort	lob specific command, value has to be 1 for the loading of an object
ushort	flags to be passed to the object
byte[]	null-byte separated list of command line arguments for the lob, the first token is the filename

The API will answer this call by a VRRB containing a LOB_INFO_BLOCK.

Result Block

VRRB	result block header
LOB_INFO_BLOCK	the record with information about the object

Errors are signalled as described above. Since a loadable object might use the API channel it was loaded by during its construction, any loadable object has to be implemented carefully as any synchronisation problem within the object's constructor might harm the later communication on the API channel.

A.10.2 Querying LOB Information

To get some information about the currently loaded modules, a result block containing data about all loaded objects can be queried. This result block contains a list of LOB_INFO_BLOCKs.

Result Block

VRRB	result block header
LOB_INFO_BLOCK[]	list of information blocks

The length field of the result block header can be used to get the number of loaded objects as usual.

A.11 Querying Scheduler Status

This command requests information about the internal event handler. This is a read only command, therefore (so far) no modification of the central event handling system is possible. However this command at least is useful for debugging purposes.
A VRCB with the command id 21 results in a VRRB containing information about all currently registered events. The structure of the result block looks like:

Result Block

VRRB	result block header
EVENT_REC	one record per registered event

The records containing the event information have the following format:

EVENT_REC	**Data Record**
ushort	event type (READ,WRITE)
ushort	actual active flags
ushort	the time in 1/1000 seconds, the event is suspended, 0 if active
ushort	associated read file handle
ushort	associated write file handle
byte[16]	id string of the event (e.g. an interface name)
ushort	true if event execution is forced

A.12 Passing IP packets to the Router

	Control Block
VRCB	control block with id 50
ushort	flags concerning the packet handling
ushort	specifies how the packet is treated
ushort	mode dependent parameter
ip packet	the packet to be sent

The API also provides a mechanism to pass IP packets directly to the Virtual Router. For that purpose the IP packet has to be encapsulated into an VRCB. Since there are several possibilities the Virtual Router can handle the packet, a mode parameter has to be specified. Therefore a packet may either be sent using the VR's internal routing rules (see A.8) or an interface is defined to transmit the packet.

Modes for sending IP packets	
0	The packet is directly processed by the Virtual Router's routing procedures and put to the according interface.
1	The packet is analysed as any other packet that is received by the Virtual Router. The packet is processed by the Virtual Router's protocol stacks and is also analysed and processed by the filters.
2	The Virtual Router puts the received packet directly to the specified interface. The parameter field has to contain a valid Virtual Router interface number.

Additionally, the VRCB contains information about whether an IP packet shall be sent 'as is' or if the Virtual Router shall add certain information within the IP header and recalculate the CRC. The behaviour can be configured by setting certain flags in the flags parameter. The following list gives a description of the currently used flags. Multiple flags may be set.

Flags for special IP treatment	
1	The Virtual Router will control, whether the IP header of the received packet is valid or not. If the header does not comply the standard the packet will be dropped and an error will be returned.
2	The checksum of the packet header will be set correctly
4	The Time To Live field in the packet header will be set by the Virtual Router using its default value.
8	The Virtual Router's address is used as source address of the packet.
16	The received packet will be treated as not fragmented and the header fields will be set accordingly. The Virtual Router will fragment oversized packets in any case, therefore the packet sent over the API can be up to 0xffff bytes.

After the control block with the encapsulated packet has been sent to the Virtual Router a result block is sent back containing information about how the packet was processed and whether any error occurred.

Result Block

VRRB	result block header
ushort	result code

Since the type of error depends on the mode the packet was sent with, the result code field can have multiple values.

Result Codes R for the various modes		
0	R = 0xfff	routing error, no route found
0	R $<$ 0xfff	packet was routed to interface number R
0	R = 0xffff	not an IP packet
1	$0 \leq$ R $\leq$ 0xfff	packet routed to interface number R
1	R = 0xfff	routing error, no route found
1	R = 0x1000	packet processed by local protocol stack
1	R = 0x1001	no matching local protocol stack
1	R = 0x2000	packet dropped due to TTL
1	R = 0xffff	not an IP packet
2	R = 0xfff	no such interface
2	R = $\leq$ 0xfff	packet sent to interface number R
2	R = 0xffff	not an IP packet

Appendix B

The Internal Shell

In addition to the described binary format, the VR also provides a parser to process human readable commands and to produce readable ASCII format. In this chapter the available commands and their output will be presented. It should be mentioned here, that these interfaces do not provide an alternative access to the base forwarding layer. These interface functions only parse a command string and translate it to appropriate binary data. The binary data is sent to the base forwarding layer over an API channel. Vice versa output received on the API channel is translated to a human readable format. It is strongly recommended to use the binary format for automatic configuration because these human readable commands may change frequently.

B.1 ifconfig

This command covers all configurations of and VRs interfaces. Each interface has an unique name, which is used to identify the interface to be configured. The length of the name is limited to 9 chars. In the following section it will be described, how interfaces are set up and configured.

Command Syntax:

```
ifconfig help:

 ifconfig
 ifconfig add <ifname> <ip-address> [netmask] [broadcast]
 ifconfig <ifname>
 ifconfig <ifname> reset
 ifconfig <ifname> if <newid>
 ifconfig <ifname> bw <bandwidth> [buckettime]
 ifconfig <ifname> address <ip> [netmask] [broadcast]
 ifconfig <ifname> connect ...
 ifconfig <ifname> ttx ...
 ifconfig <ifname> trx ...
 ifconfig <ifname> qs
 ifconfig <ifname> qs conns
 ifconfig <ifname> qs list
 ifconfig <ifname> qs create [droptail|tbf|scheduler|classifier]
 ifconfig <ifname> connect sol <#no>
 ifconfig <ifname> connect fifo <#no> <{l|r}>
 ifconfig <ifname> connect tunnel <hostname> <dest_port> <src_port>
 ifconfig <ifname> connect ipip <hostname> [mtu]
```

```
ifconfig <ifname> qs chain <component id1> <component id2>
ifconfig <ifname> qs <component id> reset
ifconfig <ifname> qs <component id> status
ifconfig <ifname> qs <tbf-id> bw <bandwidth> [bucket size]
ifconfig <ifname> qs <droptail-id> ql <queuelen>
ifconfig <ifname> qs <scheduler-id> wt <component id> <weight>
ifconfig <ifname> qs <dsmarker-id> mark ... as ...
ifconfig <ifname> qs <dsmarker-id> erase ... as ...
ifconfig <ifname> qs <trio-id> ql len0 len1 len2
ifconfig <ifname> qs <trio-id> mode {hard|linear}
ifconfig <ifname> qs <scheduler-id> mode {wfq|prr|rr|pwfq}
```

Interface configuration, which is supported by the API can be done by this command. The command without any arguments results in a listing of all configured interfaces with their parameters.

B.1.1 Creating a new Interface

```
> ifconfig add if0 10.1.1.1
ip-address:           10.1.1.1
netmask:         255.255.255.0
broadcast:          10.1.1.255
bandwidth(bps):        1000000   bucket size(bytes):        125000
drops:                       0   errors:                         0
rx:                          0   tx:                             0
rx-t-ip              172.0.0.0   rx-t-nm                 255.0.0.0
rx-t-val              10.0.0.0   rx-t-pat                255.0.0.0
tx-t-ip               10.0.0.0   tx-t-nm                 255.0.0.0
tx-t-pat             255.0.0.0
connection                none
```

This interface was added without any parameters except the interface name `if0` and the IP address `10.1.1.1`. The rest of the parameters are the default values.

B.1.2 Connecting an Interface

Also the connection of interfaces is handled by the ifconfig command. There are three ways for connecting interfaces:

- Connections between VRs running on the same host (IPC). These connections can be established by submitting the link number and the ”end” of the cable to the ifconfig command. (see also 4.1.1)

  ```
  ifconfig <ifname> connect fifo #no {l|r}
  ```

  These connections are mapped to the appropriate FIFO queues on the system. The ”left” end of the ”cable” is associated with the end id `0`, whereas the ”right” end has the id `1`.

- Connections between VRS on different hosts.

  ```
  ifconfig <ifname> connect tunnel remotehost tx_port rx_port
  ```

  To establish a connection between two VRs on different hosts UDP tunnels are used. These connections require the specification of the remote host and two port numbers. The port is the port, where the packets are sent to, the second number specifies the port, where incoming packets from these hosts are accepted.

- A connection to a Softlink device can be established by the command.

  ```
  ifconfig <ifname> connect sol #no
  ```

  The number specifies the softlink device on the system.

- Setting up IP over IP tunnels

  ```
  ifconfig <ifname> connect ipip 10.42.10.43 1200
  ```

  The tunnel connection mentioned above is used to connect the interfaces of two virtual routers. This tunnel is comparable to an Ethernet cable connecting two real routers. In contrast there is a possibility to encapsulate IP packets within other IP packets and tunnel them through a network to another router. This mechanism is also called IP over IP tunnels. Setting up an ipip connection, packets routed to that interface are encapsulated within an IP packet and forwarded to the specified destination address. These destination address can also be a remote (not neighbouring) router. A router receiving these packets will decapsulate the packet and treat its payload as a normal IP packet.

B.1.3 Configuring an Interface's Queueing System

Each interface has an own queueing system attached. This queueing system may consist of several components connected to each other. A more detailed description about the single components is provided in Chapter 4.1.2. The minimum queueing system consists out of a single droptail queue. One from a couple of components may be chosen and instantiated using the command

```
ifconfig <if> qs create <component type>
```

where `<if>` is the name of the interface the queueing system is attached to and the name of the component type to be instantiated is given by `<component type>`. See the above mentioned Chapter 4.1.2 about the queueing system for a list of available component types. The command returns an id for the newly created component. All configuration commands will use this id as reference for the component.
The next step is to connect this component to other components. Each component gets packets from at least one other component and has at least one component it may forward packets to. Other component-types (e.g. schedulers) allow to receive packets from several other components, other ones allow to forward packets to multiple other components.
To tell a component with the id 3 to send packets to the component 17 the command

```
ifconfig <if> qs chain 3 17
```

is used. If the component 3 allows to forward packets to additional components, the same command may be used again. So an additional

```
ifconfig <if> qs chain 3 19
```

allows component 3 to forward packets to both components 17 and 19. Which packet is forwarded to which component is not affected by this configuration. This decision depends only on the internal mechanisms of number 3 and may have to be configured separately.
The other case, that a component received packets from multiple other components may be configured analogous.
As the queueing systems set up might have to be adapted these links between the components might be also removed by the command:

```
ifconfig <if> qs unchain <id1> <id2>
```

Several commands are available to maintain the instantiated components and the links between them.

`ifconfig <if> qs avail` shows a list with all available component types.

`ifconfig <if> qs list` will show a short list of all instantiated components with their component id and their type.

`ifconfig <if> qs conns` lists all connections between components. For each component the type and the id is shown.

`ifconfig <if> qs <id> status` shows the configuration and the status of the component with id `<id>`. Dependent on the component type the information printed here might be more or less detailed.

`ifconfig <if> qs <id> reset` resets all statistical counters for the element. These are mainly counters for in and outgoing packets.

As mentioned above the links between the single quite generic modules of the queueing system define only the basic layout. For final behaviour also the configuration of the single components is important. The following list gives a short description of the configurable parameters for each component.

token bucket filter (tbf): The following command allows to modify the bandwidth and (optional) the bucket size of a token bucket filter.

```
ifconfig <if> qs <id> bw <bandwidth> [bucket size]
```

The `<id>` value has to be the id of a tbf component. The command expects the bandwidth in Mbps (e.g. 2.0 for 2 Megabit per second) and the bucket size in Megabit.

droptail queue (droptail): At the moment the only parameter available, which can be set at the droptail queue is the queuelen. This value determines how many packets might be stored in the queue.

```
ifconfig <if> qs <dpt-id> ql <queuelen>
```

generic scheduler (scheduler): The generic scheduler allows more parameter to be set. Since the scheduler may operate in different modes, these mode may be changed. For a detailed description of modes see Chapter 4.1.2.

```
ifconfig <if> qs <sched-id> mode {wfq|rr|prr|pwfq}
```

Dependent on the mode an additional parameter is necessary. This parameter is called weight and might be interpreted as bandwidth share, as a priority or is simply ignored by the scheduler.

```
ifconfig <if> qs <sched-id> wt <id> <wt>
```

The weight value can be set for each component preceding the scheduler (a scheduler has usually several incoming links), so the second `<id>` value specifies which weight shall be set.

classifier: The classifier has usually multiple follow up components. The classifier allows to set rules, specifying which packet is forwarded to which component. The syntax to set up such a rule is:

```
ifconfig <if> qs <cls-id> <dip/nm> <sip/nm>
                [<p>] [<t>] to <id>
```

The `<dip/nm>` and `<sip/nm>` specify a range of destination and source IP addresses. If the netmask is omitted the netmask `255.255.255.255.` is assumed. The specification of the protocol id and the ToS byte value are optional. The `<id>` following the key word `to` specifies the component packets matching this filter will be forwarded. There may be multiple such rules pointing to the same component.

Rules defined at the classifier can be removed by:

```
ifconfig <if> qs <cls-id> del <dip/nm> <sip/nm> [<p>] [<t>]
```

DiffServ Marker

The Differentiated Services Marker component can be configured with a set of rules, specifying which flow shall be marked with which DSCP up to which bandwidth.

```
ifconfig <if> qs <dsms-id> mark [source a.b.c.d/n]
[dest a.b.c.d/n] [proto p] [tos t] as <service>
```

The command allows to specify the flow by source and destination addresses, the protocol and the ToS Byte value. The service specifies which DSCP shall be set. The service can simply be EF for the Expedited Forwarding service of a term like

```
af[1|2|3|4]:bw1[:bs1]:bw2:[bs2]
```

for Assured Forwarding. The initial af1, af2, af3, or af4 specifies the AF class. This parameter has to be followed either by two or by four other parameters, setting either the maximum allowed bandwidths for low and medium drop precedence or the bandwidth values and the bucket sizes of the according Token Bucket filters.

TRIO queue

The TRIO queue is a special queue design for the AF service by dropping packets with specific DSCPs with different probability. This behaviour can be influenced by setting different queue lengths for the dropping probabilities. The command

```
ifconfig <if> qs <trio-id> ql 10 20 30
```

will configure a TRIO queue to a maximum queuelength of 30 packets. Packets with high drop precedence are dropped if the queuelength exceeds 10 packets, whereas packets with medium drop precedence are dropped only if the queue exceeds 20 packets. There is an additional parameter to influence the algorithm used to calculate the dropping probability. This parameter can be set by

```
ifconfig <if> qs <trio-id> mode {hard|linear}
```

The different algorithms are described in section A.7.7.

B.2 route

This command allows to add, list and delete static routes in the base layer's routing table. Even when the base layer's routing rules support source, protocol and ToS based routing and the binary API also allows to set up those routes, the ASCII based command front end currently only supports the set up of "standard" routing rules.

Command Syntax:

```
route help:
route
route add <ip/nmb> <interface>
route add <ip> <nm> <interface>
```

B.3 sys

The system commands offer a possibility of access general access to system information, as the central scheduler or the module loader.

Command Syntax:

```
sys help:

sys events
sys filters
sys lobs
```

sys events: lists the registered scheduler events. A typical output for a shell and two interfaces looks somewhat like:

```
shell susp 0.0 type   R thrown - read 0 write 0 hits 6
  if0 susp inf type  RW thrown - read 3 write 3 hits 1
  if1 susp inf type  RW thrown - read 5 write 4 hits 1
```

sys filters: not yet implemented

sys lobs: this command allows to list all currently loaded objects.

```
name hellot, id 0, mode THREAD , size 10 kb, up 110.10 min
name pyan, id 3, mode THREAD , size 90 kb, up 3.10 min
```

B.4 load

This command allows the loading of additional modules, so called Loadable Objects into the VR kernel. The command syntax is simple and allows to pass additional command line arguments to the LOB.

```
load help:

 load <lob> [arg1] [arg2] [arg3] [...]
```

The shell replies with some information about the loaded object or an error code.
A typical output might look like:

```
object 'hellot' loaded, id is 0, mode is THREAD
```

Bibliography

[AAKS98] Scott D. Alexander, William A. Arbaugh, D. Keromytis, Angelos, and Jonathan M. Smith. A secure active network environment realization in switchware. *IEEE Network*, 12(3):37–45, May/June 1998.

[Ale] Scott Alexander. Active network encapsulation protocol. URL: http://www.cis.upenn.edu/ switchware/ANEP/.

[Ale97] D. Alexander. Active bridging. *Proceedings of SIGCOMM*, September 1997.

[Alm] W. Almesberger. Tc compatible linux version of the isi rsvp implementation, version 4.2a4. URL: ftp://lrcftp.epfl.ch/pub/people/almesber-/rsvp.

[Alm99] Werner Almesberger. Linux network traffic control – implementation overview. Differentiated Services under Linux, http://diffserv.sourceforge.net/, April 1999.

[Bau00] Florian Baumgartner. A python module for rsa cryptography. http://www.iam.unibe.ch/∼baumgart/pyrsa, 2000.

[BB99a] F. Baumgartner and T. Braun. Evaluierung von Assured Service für das Internet. In R. Steinmetz, editor, *Kommunikation in Verteilten Systemen (KiVS)*, Lecture Notes in Computer Science, pages 72–85, Darmstadt, Germany, 1999. Springer.

[BB99b] Florian Baumgartner and Torsten Braun. Fairness of assured service. In *Modelling and Simulation, 13th European Simulation Conference*, volume 1, pages 390–397, Warsaw, 1999. ISBN 1-56555-171-0.

[BB00a] Florian Baumgartner and Torsten Braun. Quality of service and active networking on virtual router topologies. In Hiroshi Yasuda, editor, *Active Networks, Second International Working Conference, IWAN*, Lecture Notes in Computer Science, pages 211–224, Tokio, Japan, October 2000. Springer. ISBN 3-540-41179-8.

[BB00b] Florian Baumgartner and Torsten Braun. Virtual routers: A novel approach for qos performance evaluation. In Jon Crowcroft, James Roberts, and Smirnov Mikhail, editors, *Quality of Future Internet Services, First COST 263 International Workshop. QofIS*, Lecture Notes in

Computer Science, pages 336–347, Berlin, Germany, September 2000. Springer. ISBN 3-540-41076-7.

[BB00c] Florian Baumgartner and Torsten Braun. Virtual routers supporting active networking. Stockholm Active Networks Day (SAN-day), August 2000. workshop organized by Uppsala University, Sweden, held in conjunction with ACM SIGCOMM 2000.

[BB01] Florian Baumgartner and Torsten Braun. Distributed emulation of ip networks. Speedup Workshop, University of Berne, March 2001.

[BBB+99] Roland Balmer, Florian Baumgartner, Torsten Braun, Manuel Günter, and Ibrahim Khalil. Virtual private network and qos management implementation. Technical Report IAM-99-003, Institut of Computer Science and Applied Mathematics, University of Berne, June 1999.

[BBBG00] R. Balmer, F. Baumgartner, T. Braun, and M. Günter. A concept for rsvp over diffserv. *IEEE, ICCCN'2000*, October 2000.

[BBC+98a] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weis. An architecture for differentiated services. Internet Draft draft-ietf-diffserv-arch-02.txt, October 1998. work in progress.

[BBC+98b] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weis. An architecture for differentiated services. RFC 2475, December 1998.

[BBEK99] Florian Baumgartner, Torsten Braun, Hans Joachim Einsiedler, and Ibrahim Khalil. Differentiated internet services. In G. G. Cooperman, E. Jessen, and G. Michler, editors, *Workshop on Wide Area Networks and High Performance Computing*, Lecture Notes in Control and Information Sciences 249, pages 37–60. Springer, June 1999. ISBN 1-85233-642-0.

[BBH98] Florian Baumgartner, Torsten Braun, and Pascal Habegger. Differentiated services: A new approach for quality of service in the internet. In H. van As, editor, *High Performance Networking*, pages 255–274. Kluwer, 1998. ISBN: 0-412-84660-8.

[BCD+99] Jim Boyle, Ron Cohen, David Durham, Shai Herzog, Raju Rajan, and Arun Sastry. Cops usage for rsvp. Internet Draft draft-ietf-rap-cops-rsvp-05.txt, June 1999.

[BCS01] F. Baker, K. Chan, and A. Smith. Management information base for the differentiated services architecture. Internet Draft draft-ietf-diffserv-mib-16.txt, November 2001. work in progress.

[BFI+01] R. Boivie, N. Feldman, Y. Imai, W. Livens, D. Ooms, and O. Paridaens. Explicit multicast (xcast) basic specification. Internet Draft draft-ooms-xcast-basic-spec-01.txt, March 2001. work in progress.

[BS99] Marcus Brunner and Rolf Stadler. Virtual active networks - safe and flexible environments for customer-managed services. *Proceedings of the Tenth International Workshop on Distributed Systems: Operations and Management (DSOM'99)*, 1999.

[BW98] Marty Borden and Christoph White. Management of phbs. Internet Draft draft-ietf-diffserv-phb-mgmt-00.txt, August 1998. work in progress.

[BYBZ98] Y. Bernet, P. Yavatkar, R.and Ford, F. Baker, and L. Zhang. A framework for end-to-end qos combining rsvp/intserv and differentiated services. Internet Draft draft-bernet-intdiff-00.txt, April 1998. work in progress.

[BZ96] J. C. R. Bennett and H. Zhang. Wf2q: Worst-case fair weighted fair queueing. *IEEE INFOCOM, San Francisco, CA*, March 1996.

[BZ97] J. C. R. Bennett and H. Zhang. Hierarchical packet fair queueing algorithms. *IEEE/ACM Transactions on networking*, 5(5), October 1997.

[BZB+97] B. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin. Resource reservation protocol (rsvp) -version 1 functional specification. Request for Comments 2205, September 1997.

[CF97] D. Clark and W. Fang. Explicit allocation of best effort packet delivery service. http://diffserv.lcs.mit.edu/Papers/exp-alloc-ddc-wf.pdf, 1997.

[CFSD90] J. Case, M. Fedor, M. Schoffstall, and J. Davin. A simple network management protocol (snmp). RFC 1157, May 1990.

[CJODS00] M. Christiansen, K. Jeffay, D. Ott, and F. Danelson Smith. Tuning red for web traffic. *ACM SIGCOMM'00*, September 2000.

[CMK+99] Andrew T. Campbell, H. de Meer, Micha el E. Kounavis, Kazuho Miki, John B. Vicente, and Daniel Villela. A survey of programmable networks. In *Computer Communications Review*, volume 29/2. ACM SIGCOMM, April 1999.

[CW97] D. Clark and J. Wroclawski. An approach to service allocation in the internet, work in progress. Internet Draft draft-clark-diff-svc-alloc-00.txt, Juli 1997. work in progress.

[DCB+01] Bruce Davie, Anna Charney, Fred Baker, Yean-Yves Le Boudec, William Courtney, Firoiu, and K.K Ramakrishnam. An expedited forwarding phb. Internet Draft draft-ietf-diffserv-rfc2598bis-02.txt, September 2001. work in progress.

[DDPP98] Dan Decasper, Zubin Dittia, Guru Parulkar, and Bernhard Plattner. Router plugins: A software architecture for next generation routers. *SIGCOMM'98 Conference*, September 1998.

[DPC+99] Dan S. Decasper, Guru M. Parulkar, Sumi Choi, John D. DeHart, Tilman Wolf, and Bernhard Plattner. A scalable high-performance active network node. *IEEE Network*, 13(1):8–19, January/February 1999.

[DS90] S Demers, A. Keshav and S. Shenker. Analysis and simulation of a fair queueing algorithm. *Internet Research and Experiments*, 1, 1990.

[FJ93] Sally Floyd and Van Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, August 1993.

[FJ95] Sally Floyd and Van Jacobson. Link-sharing and resource management models for packet networks. *IEEE/ACM Transactions on Networking*, 3(4), August 1995.

[FLK+99] Ted Faber, Bob Lindell, Jeff Kann, Graha m Phillips, and Alberto Cerpa. Arp: Active reservation protocol, April 1999. Presentation.

[GBH97] R. Guerin, S. Blake, and S. Herzog. Aggregating rsvp-based qos requests. Internet Draft draft-guerin-aggreg-rsvp-00.txt, November 1997.

[GBK99] M. Günter, T. Braun, and I. Khalil. An architecture for managing QoS-enabled VPNs over the Internet. In *IEEE Conference on Local Computer Networks*, 1999.

[Gro01] Dan Grossman. New terminology for diffserv. Internet Draft draft-ietf-diffserv-new-terms-06.txt, November 2001. work in progress.

[Gün01] Manuel Günter. *Management of Multi-Provider Internet Services with Software Agents*. PhD thesis, Institute of Computer Science and Applied Mathematics, University of Berne, 2001.

[HBWW99a] Juha Heinanen, F. Baker, W. Weiss, and J. Wroclawski. Assured forwarding phb group. RFC 2597, June 1999.

[HBWW99b] Juha Heinanen, Fred Baker, Walter Weiss, and John Wroclawski. Assured forwarding phb group. Internet Draft draft-ietf-diffserv-af-06.txt, February 1999. work in progress.

[HE02] Bernhard Hechenleitner and Karl Entacher. On shortcomings of the ns-2 randum number generator. In *Communication Networks and Distributed Systems Modeling and Simulation Conference (CNDS 2002)*. The Society for Modeling and Simulation International, January 2002.

[HG98] Gísli Hjálmtýsson and Robert Gray. Dynamic c++ classes - a lightweight mechanism to update code in a running program. In *Proceedings of the USENIX Annual Technical Conference*, pages 65–76, June 1998.

[HG99a] J. Heinanen and R. Guerin. A single rate three color marker. RFC 2697, September 1999.

[HG99b] J. Heinanen and R. Guerin. A two rate three color marker. RFC 2698, September 1999.

[IN98] J. Ibanez and K. Nichols. Preliminary simulation evaluation of assured service. Internet Draft draft-ibanez-diffserv-assured-evald-00.txt, August 1998. work in progress.

[JNP98] Van Jacobson, K. Nichols, and K. Poduri. An expedited forwarding phb. Internet Draft draft-ietf-diffserv-af-02.txt, October 1998. work in progress.

[JNP99] Van Jacobson, K. Nichols, and K. Poduri. An expedited forwarding phb. RFC 2598, June 1999.

[Kat97] D. Katz. Ip router alert option. RFC 2113, February 1997.

[Lab94] RSA Laboratories. Rsaref(tm) 2.0: A free cryptographic toolkit, April 1994. RSA Laboratories.

[Lef00] Glyph Lefkowitz. A subjective analysis of two high level, object oriented languages. URL: http://www.python.org/doc/Comparisons.html, April 2000.

[Mis81] Misc. Internet protocol, darpa internet program protocol specification. RFC 791, September 1981.

[MLPT00] H. de Meer, A. La Corte, A. Puliafito, and O. Tomarchio. Programmable agents for flexible qos management in ip networks. *IEEE Journal of Selected Areas in Communication*, 18(2), February 2000.

[MPT98] H. de Meer, A. Puliafito, and O. Tomarchio. Management of qos with software agents. *Cybernetics and Systems: An International Journal*, 27(2), 1998.

[MRPT98] H. de Meer, J.P. Richter, A. Puliafito, and O. Tomarchio. Tunnel agents for enhanced internet qos. *IEEE Concurrency*, 6(2), June 1998.

[NBBB98] K. Nichols, S. Blake, F. Baker, and D. Black. Definition of the differentiated services field (ds field) in the ipv4 and ipv6 headers. Internet Draft draft-ietf-diffserv-header-04.txt, October 1998. work in progress.

[NC01] K. Nichols and B. Carpenter. Definition of differentiated service per domain behaviors and rules for their specification. RFC 3086, April 2001.

[nis] nistnet. http://snad.ncsl.nist.gov/itg/nistnet.

[ns] Ucb/lbnl/vint network simulator - ns (version 2). URL: http://www-mash.CS.Berkeley.EDU/ns/.

[opn] Opnet modeler. URL: http://www.mil3.com.

[Per96] C. Perkins. Ip encapsulation within ip. RFC 2003, October 1996.

[PT00] A. Puliafito and O. Tomarchio. Using mobile agents to implement flexible network management strategies. *Computer Communication Journal*, 23(8):708–719, April 2000.

[pyt] Python language website. URL: http://www.python.org.

[RSA78] R.L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–128, February 1978.

[SB00] Günter Stattenberger and Torsten Braun. Implementation and configuration of a linux differentiated services router. Technical Report IAM-00-010, Institute for Computer Science and Applied Mathematics, University of Berne, Neubrückstrasse 10, 3012 Berne, Switzerland, November 2000.

[SBB01] Günter Stattenberger, Torsten Braun, and M. Brunner. A platform-independent api for quality of service management. In *Workshop on High Performance Switching and Routing*. IEEE, May 2001.

[SBP99] Burkhard Stiller, Manuel Braun, Günter, and Bernhard Plattner. The cati project: Charging and accounting technology for the internet. In *Proceedings of Multimedia Applications, Services and Techniques, ECMAST'99, LCNS 1629*, pages 281–296. Springer, May 1999. ISBN 3-540-66082-8.

[SCFJ96] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. Rtp: A transport protocol for real-time applications. Request for Comments 1889, January 1996.

[Ten97] D. et al Tennenhouse. A survey of active network research. *IEEE Communications Magazine*, january 1997.

[TH98] J. Touch and S. Hotz. The x-bone. Third Global Internet Mini-Conference in conjunction with Glob ecom'98, Sydney, Australia, November 1998.

[tki] Python tkinter resources. URL: http://www.python.org/topics/tkinter.

[Tou00] Joe Touch. Dynamic internet overlay deployment and management using the x-bone. In *Proceedings of ICNP*, pages 59–68, 2000.

[WGT98] D. Wetherall, J. Guttag, and D. Tenenhouse. Ants: A toolkit for building and dynamically deploying network protocol. In *IEEE Openarch*, 1998 1998.

[WK99] S.Y. Wang and H.T Kung. A simple methodology for constructing extensible and high-fidelity tcp/ip network simulators. In *IEEE INFOCOM*, March 1999.

[WRA96] A. Watters, G. van Rossum, and J. C. Ahlstrom. *Internet Programming with Python*. M&T Books, August 1996.

[Wro97] J. Wroclawski. The use of rsvp with ietf integrated services. Request for Comments 2210, September 1997.

[YR98] Ikjun Yeom and Narasimha A.L. Reddy. Realizing throughput guarantees in differentiated services networks. Technical report, Texas A & M University, November 1998.

List of Abbreviations

AF	Assured Forwarding
ANEP	Active Network Encapsulation Protocol
AN	Active Network
ANTS	Active Node Transfer System
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
ATM	Asynchronous Transfer Mode
BA classifier	behaviour aggregate classifier
BB	Bandwidth Broker
BGP	Border Gateway Protocol
CATI	Charging and Accounting for the Internet
CBQ	Class Based Queueing
COPS	Common Open Policy Service
DES	Data Encryption Standard
DiffServ	Differentiated Services
DSCP	Differentiated Services Code Point
EF	Expedited Forwarding
EWMA	Exponentially Weighted Moving Average
FIFO	First In First Out
GUI	Graphical User Interface
ICMP	Internet Control Message Protocol
IETF	Internet Engineering Task Force
IP	Internet Protocol
IPIP	IP Encapsulation within IP
IPC	Inter Process Communication
ISP	Internet Service Provider
JNI	Java Native Interface
JPEG	Joint Photographic Experts Group
LAN	Local Area Network
MD5	Message Digest 5
MF classifier	multifield classifier
MIB	Management Information Base
OSPF	Open Shortest Path First
PAD	Platform Adaptor
PDB	Per Domain Behaviour
PHB	Per Hop Behaviour
PVC	Private Virtual Circuit

PWRR	Priority Weighted Round Robin
PyBAR	Python Based Active Router
RDG	RSVP/DiffServ Gateway
RED	Random Early Detection Gateway
RFC	Request for Comment
RIO	RED with In and Out
RR	Round Robin
RSA	Rivest, Shamir, and Adelman Algorithm
RSAREF	RSA reference implementation library
RSVP	Resource Reservation Protocol
RTP	Real Time Protocol
RTCP	Real Time Control Protocol
RTT	Round Trip Time
SLA	Service Level Agreement
SLS	Service Level Specification
SNMP	Simple Network Management Protocol
tc	traffic control (Linux)
TCP	Transmission Control Protocol
TCS	Traffic Conditioning Specification
TRIO	Three drop precedence RIO
UDP	User Datagram Protocol
VC	Virtual Channel
VPN	Virtual Private Network
VR	Virtual Router
VAN	Virtual Active Network
VAR	Virtual Active Routers
WAN	Wide Area Network
WFQ	Weighted Fair Queueing
WRR	Weighted Round Robin

Curriculum Vitae

1971	Born on May, 14 in Straubing, Germany
1977-1981	Elementary School Ittling
1982-1990	Ludwigsgymnasium Straubing
1990-1997	Study of General Linguistics (Computer Science) and Physics at the University of Regensburg
1997	M.A. in General Linguistics (Computer Science) and Physics
1997-1998	European Networking Center, IBM Heidelberg
1998	Research assistant and Ph.D. student at the Institute for Computer Science and Applied Mathematics, University of Berne