

# An implementation of capacity reservation devices in IP networks

Gabriel Paues,  
Swedish Institute of Computer Science  
Box 1263, S-164 29 KISTA, SWEDEN

August 22, 2002

Bandwidth markets is an approach to achieve quality of service. By dividing capacity into shares, capacity may be traded between actors in a net. These actors are typically clients, that want to reserve capacity, and net operators offering capacity. To realize a bandwidth market, a number of components have to be implemented. This thesis describes the implementation of some of these components, those used by a net operator offering capacity on a bandwidth market. The features needed by a net operator are access control, shaping and routing. The components that implement those features are an access manager, a packet marker, a shaper and a label switch. To differentiate packets using reserved capacity from unreserved ones, parts of the IP header were used. The difficulties were to understand which parts of the IP header (TOS-field or flowlabel) and what version of the IP protocol (IPv4 or IPv6) to use.

The components were tested in a testbed. This testbed used virtual Linux machines connected together to form an IP network.

Report: T2002:11  
ISRN: SICS-T-2002/11-SE  
ISSN : 1100-3154

## Acknowledgments

First of all I like to thank Lars Rasmusson, for being my guiding hand, throughout the project. He has helped me to get back on track and seen things and aspects that I did not. I also want to express my sincere gratitude to Erik Aurell, for being an inspiring source of knowledge and academic thinking.

I would like to thank the people at Ericsson Switch lab, particularly Annikki Welin, Svante Ekelin and Carl-Gunnar Perntz, as well as Telia Skanova and Anders Rockström.

I would also like to thank Vinnova for supporting the project AMRAM, Dnr 2001-06251.

Finally, I would like to thank my family, especially my mother for her unlimited support, and Miko Paues, my dear uncle and close friend, for proofreading my report at an early stage.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Thesis introduction . . . . .	1
1.2	Task description . . . . .	1
1.3	Outline . . . . .	2
<b>2</b>	<b>Quality of Service - Background</b>	<b>3</b>
2.1	Overview . . . . .	3
2.2	Service models . . . . .	3
2.2.1	Best effort . . . . .	3
2.2.2	Integrated Services - IntServ . . . . .	4
2.2.3	Differentiated Services - DiffServ . . . . .	5
2.2.4	Discussion of the different service models . . . . .	6
<b>3</b>	<b>Bandwidth market</b>	<b>8</b>
3.1	Overview . . . . .	8
3.2	Participants of the secondary market . . . . .	9
3.2.1	Market place . . . . .	9
3.2.2	Middleman . . . . .	9
3.2.3	The clients . . . . .	10
<b>4</b>	<b>Implementation</b>	<b>11</b>
4.1	The <b>iptables</b> tool . . . . .	12
4.1.1	A small example . . . . .	12
4.2	The <b>iproute2</b> package . . . . .	13
4.2.1	Example one: Multiple routing tables . . . . .	14
4.2.2	Example two: Shaping traffic . . . . .	14
4.3	The access manager . . . . .	16
4.4	The marker . . . . .	16
4.5	The shaper . . . . .	17
4.6	The label switch . . . . .	19
4.7	Putting it together . . . . .	20
4.8	Discussion of other approaches . . . . .	22
4.8.1	Implementation with IPv6 . . . . .	23

<b>5</b>	<b>Testbed and test programs</b>	<b>24</b>
5.1	User Mode Linux - UML . . . . .	24
5.2	DBS - Distributed Benchmarking System . . . . .	25
5.3	trgen - Traffic Generator . . . . .	26
5.4	Logging and presenting the data . . . . .	26
<b>6</b>	<b>Testing and evaluation</b>	<b>28</b>
6.1	Test on capacity . . . . .	28
6.1.1	Motivation . . . . .	28
6.1.2	Method . . . . .	28
6.1.3	Hypothesis . . . . .	29
6.1.4	Results . . . . .	31
6.1.5	Discussion . . . . .	31
6.2	TOS-based routing test . . . . .	32
6.2.1	Motivation . . . . .	32
6.2.2	Method . . . . .	32
6.2.3	Hypothesis . . . . .	33
6.2.4	Results . . . . .	33
6.2.5	Discussion . . . . .	34
<b>7</b>	<b>Conclusions</b>	<b>35</b>
7.1	Overview . . . . .	35
7.2	Future work . . . . .	35
<b>8</b>	<b>Bibliography</b>	<b>36</b>

## List of Figures

1	Rules in <b>iptables</b> . The match column tells what property the rule will match, and the target what action will be performed in case of a match. . . . .	12
2	The HTB class hierarchy in the shaper. For every reservation a new child class is created. Rate stands for how much this class is guaranteed and ceil up to which capacity the class may borrow capacity from others. The default class has lower priority, and may get starved by the other classes. . . . .	18
3	The setup phase of a reservation. The client (A) wants to reserve capacity to node D in the network. The figure shows how the components and the client interacts with each other. .	21
4	The HTB setup in the capacity test. The 1:1, 1:10,1:11 and 1:12 are the labels of the classes in the hierarchy. The rates denote how much capacity the classes are guaranteed. . . . .	29
5	The expected result of the capacity test. At $t_0$ flow 1 starts, at $t_1$ flow 2 starts to be followed by flow 3 at $t_2$ . At $t_3$ flow 1 stops. Notice how the flows borrow capacity from each other. .	30
6	The results of the capacity test. Compare with the expected results and notice how the shaper works in the expected way. .	31
7	The topology of the testbed. . . . .	33
8	The results of the TOS-field routing test. The TOS value is switched around the time 30 s. . . . .	34
9	The IPv6 header with field sizes in bits . . . . .	53
10	A chain of extension headers. . . . .	54
11	Overview of the IPv6 stack in the Linux kernel. . . . .	56

# List of definitions and acronyms

## General definitions

- QoS Quality of Service describes what is measured to decide the quality of a link between two computers, for instance transmission rates, error rates and drop rates.
- Service model Defines a strategy on how to reserve resources and propagate traffic in a net.
- Best effort Describes the traffic for which capacity has not been reserved. Described in section 2.2.1.
- IntServ Integrated Services, a service model described in section 2.2.2, using path reservation by sending reservation requests through a network. The client may not use resources in the network, until a successful reservation request have been made.
- DiffServ Differentiated Services, a service model classifying traffic into different classes, and giving different levels of service to different classes. Described in section 2.2.3.
- IPv4 Internet Protocol Version 4, the Internet Protocol used today to address hosts and computers on the Internet. For a thorough description of IPv4 and its applications, see “TCP/IP Illustrated” by W. Richard Stevens [6].
- IPv6 Internet Protocol Version 6, successor to IPv4. Described in Appendix D.
- Source routing Technique used when the sender decides the whole routing path of a packet, from the sender to the receiver.

## Definitions introduced in this thesis

- Reserved packet Packet which is to be sent on a reserved link trough a net, as opposed to a best effort packet. In this text, a reservation is described by three terms: source address, destination address and destination port.

- Reserved traffic Traffic which is sent on a reserved link through a net. This is the contradiction to best effort traffic.
- Client Something that wants to use capacity in a network.
- Access node The node through which the client sends all its traffic in the net where it has reserved capacity.
- Reservation id A unique number representing a reservation.
- Access manager The component that keeps track of reservation ids and communicates them to the marker, shaper and the label switches when a client makes a reservation. Described in chapter 4.
- Marker The component that marks the packets coming from a certain client, with the correct reservation id in order for the shaper and the label switches to act appropriately on the packet. Described in chapter 4.
- Shaper The component that makes sure the client only sends as much traffic it is entitled to according to its reservation. Described in chapter 4.
- Label switch The component that routes packets, with a certain reservation id, the correct path through the net. Described in chapter 4.
- Host system A system where virtual instances of User mode linux run. Described in chapter 5.
- Virtual instance An instance running User mode linux. Described in chapter 5.

# 1 Introduction

## 1.1 Thesis introduction

During the past few years, the demand for reservable capacity has arisen. Some information, on the Internet, demands higher reliability in order to be usable, as is the case with streaming media. Other information must not arrive late, or it results in fatal consequences, like remotely performed surgical operations. Quality of Service (henceforth for QoS) refers to a group of techniques which aim to solve the problem by giving better service to certain classes of traffic through a network.

There are a number of proposed techniques on how to reserve capacity. In this thesis we will look into one approach called bandwidth markets. The idea is to let net operators measure their capacity, and sell it like shares on an independent market. On this market, a client buys shares to reserve capacity. In return the client will receive tokens from the market, which it uses to get access to the net where it have reserved capacity.

In order to realize a bandwidth market there are mainly two things that need to be done:

- We need to implement the market itself, with pricing mechanisms, transaction handling and security functions.
- The net, which uses the bandwidth market to reserve capacity, need new components to handle things like access control, shaping of traffic and source routing.

This project focuses on the second part: to implement some of the components needed in a net, which uses the bandwidth market to reserve capacity.

## 1.2 Task description

This Masters thesis consists of the following parts:

- A literature study of the QoS service models that exist today. Find out their strengths and weaknesses.
- The establishment of a testbed which will be used to verify and test the algorithms and protocols. The testbed should also serve as a framework where it is easy to manipulate the behavior of the routers.



- Implementation of the access node and the core routers. The access node should be able to shape and establish flows through the net.
- Testing of the system in action (for correctness) in order to verify that the solution with bandwidth markets is feasible.

### 1.3 Outline

First, an overview of Quality of Service and two current proposed techniques is presented. Then comes a description of the “Bandwidth Markets” model, in order to give a background to the components implemented in the project. The “Implementation” chapter describes what has actually been done, and the “Testbed and test programs” chapter describes how the test environment was set up and the programs used to generate and measure the traffic. Under “Testing and evaluation” the reader may study which tests were performed, and the results of them. The “Conclusions” chapter describes what lessons have been learned and what to do in the future. The appendices include listings of how to setup the testbed and to reproduce the tests. There is also source code of the programs made during the project, as well as an thorough description on the history and design of IPv6. A description on how to add a match module to **iptables** is included as well.

## 2 Quality of Service - Background

### 2.1 Overview

QoS refers to a group of techniques which aim to give better service to certain classes of traffic through a network. Today's data traffic is overwhelmingly routed according to the best effort model. This gives no means of reserving or guaranteeing bandwidth. Moreover, packets residing to the same chunk of data may be transferred by different routes over the Internet. This may cause the data to arrive out of order, which is bad for streaming media, and similar network services. In order to solve those problems, an alternate way of defining traffic is required: flows. The flow is an abstraction referring to a distinguishable stream of related datagrams that results from the activity of a single user, and requires the same QoS [3]. By marking packets belonging to the same flow, the routers will be able to do smarter routing decisions, and reserve bandwidth to a specific flow.

### 2.2 Service models

A service model is the term for describing the policy the routers use to decide how to propagate the traffic through a network. A network may use different service models simultaneously, mostly best effort in combination with some other model.

#### 2.2.1 Best effort

This is the simplest service model, widely used all over the Internet. This model aims to ensure maximal throughput through every single node. Best effort does not do any flow management. All traffic is handled according to the first-come-first-served principle, with some exceptions if priorities are used. This makes the model very simple and powerful since the routers may forward the traffic rapidly. A best effort router tries to forward every packet it receives until it receives too many and starts dropping them. This applies to all kinds of packets, no matter if they contain data from a streaming media application, which is sensible to packet loss, or a mail server, which may resend the dropped packet.

### 2.2.2 Integrated Services - IntServ

The IntServ model [3] is a service model proposed by the Internet Engineering Task Force [10]. This model uses a protocol named RSVP [4], Reservation Protocol, in order to ensure the capacity required by a flow before admitting the traffic into the network. This is done by first sending a request through the network, which reserves bandwidth on all routers through the path, and when this request successfully returns, one starts sending the data. In RSVP, the reservation requester sends a reservation request through the route it wants to reserve. This reservation request propagates until it reaches a node denying the request or until it reaches the destination node which returns a positive response to the reservation requester.

This kind of explicit management of bandwidth comes from the assumption that bandwidth must be explicitly managed in order to meet the application requirements. If a subnet cannot guarantee delivery of information at a certain rate, the information should not be allowed to be sent in the first place. A guarantee must be waterproof in order to be a guarantee.

IntServ key techniques are resource reservation and admission control. These are realized by four components. Before any traffic is to be sent through the network, resources have to be reserved, and when that is done, the admission control will let the traffic enter the net. The first component is the reservation protocol while the others implement the traffic control capabilities.

- A reservation protocol. The IntServ model is tightly coupled to RSVP even if in principle any reservation protocol would do.
- A packet scheduler which, with a number of queues, handles the forwarding of different packet streams. The packet scheduler decides which packet should be sent after taking the rules of reservations into account.
- A classifier which maps every incoming packet to a class. It is the classifier that decides to which queue in the packet scheduler the packet will be sent. The classifier also finds a unique key that identifies a flow.
- The admission control which implements a decision algorithm which is to judge whether a packet is to be allowed access to the domain or not. Access is not to be granted until a reservation for the flow is done.

The routers in an IntServ network are stateful. This means that the routers keep track of how much capacity they have reserved. As long as a router has more reservable capacity, it will reply positively on reservation requests. The routers being stateful may be a problem, since there may occur stale reservations not used.

The RSVP protocol provides mechanisms for specifying resources and to which packets these resources belong. Reserving a resource can be done in different manners, so called reservation styles. These styles mirror different scenarios of reservations. For instance, a reservation may be bound to the source, to the destination or both.

Reservations are triggered from application level initiated by the receivers. Reservation message is propagated through the network to the sender. Every node, where the reservation message passes, instantiates a flow specification. This specification describes the amount of reserved capacity at the node.

### 2.2.3 Differentiated Services - DiffServ

The DiffServ model[5], is a less cumbersome approach to implement QoS. DiffServ classifies packets into different types of traffic. DiffServ tries to ensure that real time crucial traffic will be served appropriately, at the expense of less demanding traffic. DiffServ can be seen as a further developed version of the IPv4 Type Of Service technique [7] which is based on marking different packets depending on what type of service the packets require. If a packet resides in mail traffic it will get a “minimize cost” label, and if it is an FTP packet a “minimize delay”. With the IPv4 Type Of Service model, there are only four labels which limits the ability to do fine grained traffic control.

DiffServ handles aggregated flows of traffic and may describe flows with service semantics. For instance, there are labels to describe “Streaming media” explicitly and not only “minimize delay”. By standardizing a large set of flow types and adding one abstraction layer, which does not directly express the way the packets should be forwarded, it eliminates the need to redefine all nodes in the network every time a new type of traffic is introduced.

A DiffServ domain is constituted by boundary and interior nodes. The boundary nodes can be divided into two groups, egress and ingress nodes. The traffic enters the domain in an ingress node and exits it through an egress node. One node may, to flows in different directions, be both an ingress and an egress node at the same time.

The functionality of DiffServ is implemented by a number of building

blocks.

- Classifiers which classify packets in a stream depending on some portion of the packet header.
- Traffic profiles which are rules using the classifiers to state what profile a packet belongs to. If a packet belongs to a profile it is called in profile, and if it is not out profile.
- Traffic conditioners specify what to do with in or out profile packets. They may contain the following elements: meter, marker, shaper and dropper.

When a packet arrives to a DiffServ domain it is first classified in the ingress node and forwarded to a meter conditioner. The output of the meter is interpreted by the traffic profile. If the packet is in profile it is sent to the corresponding traffic conditioner, which may send, delay or drop the packet.

#### **2.2.4 Discussion of the different service models**

Several arguments against the use of QoS have been stated. One is that bandwidth in the future will be so cheap, that the most cost effective service model would be not to reserve it at all [3]. According to the creators of IntServ, this is not true, at least not in the short or middle term. It is likely that the more bandwidth there is, the more the clients use traffic generating applications.

Another argument is that applications may adapt and compensate for unreliable links with buffers, which makes research in the QoS area more or less superfluous. This may be true for streaming video or other media, but it is not true for video conferences and such, which demand non buffered flows in both directions.

Even if explicit management sometimes is needed, the IntServ technology is not widely used, since it has proved to scale very badly. RSVP introduces overhead in the network. IntServ is used, but mostly on single links and smaller subnets where the overhead doesn't have such an impact.

DiffServ has a few advantages compared to IntServ. First, the nodes are stateless. This is not the case with IntServ since every node along the path of a reservation has to store information about the reservation.

Second, DiffServ does not need as much special support in the application layer. The applications that uses DiffServ must classify the packets appropriately. IntServ demands that the application sending the data implements the Reservation Protocol (RSVP).

Different service models will probably coexist in the future, since not all traffic require QoS. Best effort and other QoS techniques may be used in the same net. The effects of having QoS traffic coexist with best effort traffic is discussed by Klara Nahrstedt and Shigang Shen in [11].

## 3 Bandwidth market

### 3.1 Overview

Most research on providing QoS to the Internet involves how to set up links where the reserved traffic may be sent. This can be done explicitly, as with IntServ. The load balance in a QoS net is supposed to be managed with admission control and source routed paths through the net. The pricing of the right to send traffic with a certain amount of QoS through a network is not considered by any of the mentioned techniques. The techniques give the network operator something to sell, i.e QoS bandwidth, but not the means of selling it.

This may be solved with a bandwidth market, which is another approach to achieve QoS. A net has a certain amount of capacity between its pairs of border nodes. If we divide this capacity into pieces, and represent every such piece of capacity as a share in a market, the capacity may be traded.

There are different kinds of markets. One kind of market, is the primary market. With this market, buyers buy shares from a chosen seller, and are not allowed to trade the shares they have bought between each other. This model is proposed by Ferguson et. al. [26] where the seller is a processor selling CPU capacity to competing jobs. In our example the seller would typically be the net operator selling shares of capacity to clients. When a client wants to sell its shares back, it have to sell them to the net operator, not to other clients. Another proposed technique that belongs to this kind of market is bandwidth brokers as proposed by Zhi-Li Zhang et. al in [9]. The clients reserving capacity in a net, first contacts a bandwidth broker to be admitted access.

Another type of bandwidth market is the secondary market. With this market, all participants may trade with each other. This means that clients that have bought shares may sell them to other clients. This way, the pricing dynamics will be more like them in an ordinary stock market. If many users wants to use traffic with QoS, the prices rise, and if the demands decrease the prices go with them. A model with secondary markets is proposed by Rasmusson and Aurell, in their work on bandwidth markets [15]. The parts implemented in this project all belong to this kind of market.

The market guarantees that capacity is available to the client that have reserved it, since every share represents existing capacity in the net. If all shares are sold, there is no capacity available in the net.

## 3.2 Participants of the secondary market

In order to realize this service model, there must be a number of participants, in certain roles:

- The *market place* is where all net operators, wanting to sell capacity, register their resources.
- The *middleman* is a broker which combines capacity resources into services.
- The *clients* which trade capacity with the middleman and with each other.

### 3.2.1 Market place

The market place is where bandwidth resources may be bought and sold. Resources are divided into shares. One share may, for instance, represent a certain amount of bandwidth between two adjacent nodes in the net. The market is controlled by a market maker, a third part which always accepts offers. When congestion occurs in a route, the shares representing capacity in this route will be more expensive. This will make shares representing other routes more attractive, since they are cheaper. The load balancing instrument is driven by supply and demand.

There is one characteristic the market model must have: the price dynamics must not be affected by certain trading sequences. That is, it must be impossible to perform a sequence of trades which influence the prices, if the traded volume is zero. How this condition is fulfilled may be studied in [15].

### 3.2.2 Middleman

The middleman buys resources from the market place, and combines them into derivative contracts. These contracts are of the form “The right to use a certain amount of bandwidth between point A and point B” or “The right to have exactly one of several possible servers, being able to communicate with me at a certain rate”. With the help of the middleman the client doesn’t have to state exactly what resources it wants to reserve. As long as the client gets the wanted capacity, between the wanted nodes at the wanted time, it will be content.



The strength of the model proposed in [15] is that virtually any kind of contract may be made between the middleman and the client. “Constant rate from a certain host”, is the simplest, but the model allows the middleman to sell contracts of the form “This bandwidth between these hosts at this specific time of the day”. The client buys a contract from a middleman, and in return it receives a receipt. This receipt is used when the client connects to the net, where bandwidth has been reserved. The receipt consists of a number of tokens which all correspond to a certain capacity over a sub link in the net, together constituting the reserved path.

### **3.2.3 The clients**

The clients buy shares from the middleman, or other clients in the market place. To use the capacity, reserved with a bandwidth market, the the net needs to be prepared to handle reservations. Before the client starts sending traffic through the net, it has to contact an access node. This access node is built on a number of components. In this project, some of the components have been implemented, particularly an access manager, a shaper, a marker and label switches. These are described in the next chapter.

## 4 Implementation

As stated in the task description, the goal with this project was to implement the traffic propagating parts needed to realize a bandwidth market. The needed parts are:

- An access manager, which is the component the client connects to in order to setup a reservation through the net. A reservation consists of a source (the client), destination, port and a rate and is represented by a reservation id. The access manager keeps track of unused reservation ids. It also communicates the reservation ids to the other components (marker, shaper and label switches) whenever a reservation is made.
- A marker detects if a packet belongs to reservation, by inspecting the source address, destination address and the port in the packet header. If the packet belongs to a reservation, the marker writes the reservation id in the header. This is done in order to let the shaper know at which rate it should shape the packet, and to let the label switch know where it should route the packet.
- A shaper shapes the traffic entering the net. It uses the reservation ids written by the marker to shape reserved flows appropriately.
- A label switch which routes the traffic through the net. In this implementation, the routing is decided by the reservation id in the packet header. This is a test model, since it would be better to source route the packets when they enter the net. This is further described in section 4.6.

This chapter describes how these components were implemented. During the project a few approaches were tried. A discussion of those that were abandoned is found in section 4.8.

Before going into details we state what the implemented parts must be able to do:

1. Read and write the contents of a packet header. We want to be able to read the source and destination address and to read and write the reservation id.
2. Route the packet depending on the contents of the reservation id.

Match	Target
tcp port == 4711	LOG
tos value == 4	REJECT
dst host==[ip.adr]	TOS set value = 4

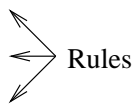


Figure 1: Rules in **iptables**. The match column tells what property the rule will match, and the target what action will be performed in case of a match.

3. Be able to set up rules to describe what to be read or written and where to route the packets.

These actions may be performed with tools found in an ordinary Linux system. What these tools are and how they work will be described in the preceding sections.

## 4.1 The iptables tool

In the Linux kernel versions 2.4.1 a new firewalling infrastructure was introduced. It is called **iptables** [12] and is the fourth generation of filtering and mangling utilities for Linux.

The principle of **iptables** is illustrated in figure 1. The package consists of a framework to define match and target modules. The match modules are designed to inspect a certain kind or property of a packet. The target modules perform an action. Some actions may change the contents of the packet (mangling) and other may just drop the package, or, for instance, send it to the system logger. These modules constitute rules, forming an entry in a **iptables** chain. This chain is hooked into a specific state in the protocol stack. Whenever a packet reaches this state, the rules in the chain are matched against the packet. If the match module in a rule detects a match, the corresponding target module is run to perform the action. By writing custom match and target modules new tasks may be accomplished, like reading or writing the contents of the flowlabel.

### 4.1.1 A small example

An example will now be presented to illustrate what **iptables** can do: The user X wants to know how many packets his computer receives with the

Minimize-Cost type-of-service property set. To accomplish this X wants to use **iptables** with the match module “tos” to match the Minimize-Cost property, and the target module “LOG” to send those packages to the system logger. This is accomplished in the following manner:

```
> iptables -A INPUT -t filter -m tos --tos Minimize-Cost \  
-j LOG --log-level 1 --log-prefix "X_testing"
```

The elements of this command line are:

- -A INPUT: Put this rule into the INPUT chain
- -t filter: The INPUT chain is situated in the filter-table. This is the **iptables**-table where filtering and logging rules are placed.
- -m tos --tos Minimize-Cost: Use the match-module tos and let it match every packet with the property type-of-service set to “Minimize-Cost”
- -j LOG --log-level 1 --log-prefix “X testing” : Use the target module LOG to send the matched packets to the system logger with the logging level 1 and the “X testing” as prefix in the system log.

This example shows some of the possibilities with **iptables**. These will later help us to implement the marker.

## 4.2 The iproute2 package

The package that provides Linux with routing capabilities is **iproute2** [17]. **iproute2** is, just like the **iptables** package, divided into two parts, one consisting of kernel modules to perform the routing and traffic control and another of user mode tools used to configure the behavior of the kernel modules. The package offers advanced routing capabilities, and traffic control, which we later will use to shape network traffic. Another feature is the ability to define multiple routing tables, and the possibility to let rules decide which of those routing tables that a certain packet will use. This is used when implementing the label switches later on.

There are two commands in the **iproute2** package we will use, **ip**, used for all sorts of management of IP related things like interfaces and rules, and **tc**, used for configuring traffic control.

### 4.2.1 Example one: Multiple routing tables

This example will illustrate how to use different routing tables and the routing rule facilities and is influenced by an example in chapter 11 of the “Linux Advanced Routing HOWTO” [20]. Assume that user X has two ways to connect to the Internet, a slow cable modem (with address A.A.A.A and device eth1) and a fast T1 fiber cable. The user X sells Internet access to his neighbors for a fee. One of the neighbors, named Y (with address Y.Y.Y.Y), will only use his access to surf occasionally and therefore wants to pay less. The default gateway points to the T1 fiber access. X wants to set up a second route pointing to the cable modem and a rule that matches user Y and sends Y’s traffic to this route. This may be done by creating another routing table with the cable modem route as default gateway and a matching rule:

```
1 > echo 200 Y >> /etc/iproute2/rt_tables
2 > ip rule add from Y.Y.Y.Y table Y
3 > ip route add default via A.A.A.A dev eth1 table Y
```

The lines in the example above does the following:

- (line 1) Add a new routing table alias, enabling us to reference the table by name Y instead of number 200. **iproute2** let us have up to 255 routing tables. By having multiple routing tables we may let different packets be routed differently.
- (line 2) Setup up a rule that will redirect all traffic coming from Y.Y.Y.Y to table Y.
- (line 3) Set the cable modem at adress A.A.A.A and device eth1 as the default gateway in table Y. By doing this, all packets that because of the rule ends up in table Y, will be routed to A.A.A.A.

### 4.2.2 Example two: Shaping traffic

The other example will show how to set up traffic control over a network interface. Traffic control, in **iproute2**, is built upon two fundamentals: queuing disciplines and filters. Queuing disciplines define the order of incoming packets, and filters are used to detect certain packets.

The user X has problems as huge downloads block his interactive traffic on his network interface eth0. Interactive traffic, refers to traffic when the user is sitting waiting for a response (browsing a page or TELNETing a

host). For simplicity, we assume that all downloads, are sent with the FTP protocol on port 21. X has the max capacity of 1 Megabits per second from his Internet transit, and wants FTP traffic never to exceed 900 kilobits per second.

The scenario is best solved with a Hierarchical Token Bucket (HTB) filter. Related examples and descriptions on how to use the HTB filter in other ways is found in [16]:

```
1 > tc qdisc add dev eth0 root handle 1: htb \  
2 default 11  
3 > tc class add dev eth0 parent 1: classid 1:1 htb \  
4 rate 1Mbit ceil 1Mbit  
5 > tc class add dev eth0 parent 1:1 classid 1:10 htb \  
6 rate 900 kbit ceil 1Mbit  
7 > tc class add dev eth0 parent 1:1 classid 1:10 htb \  
8 rate 1 Mbit ceil 1Mbit  
9 > tc filter add dev eth0 protocol ip parent 1:0 \  
10 prio 1 u32 match ip dport 21 0xffff flowid 1:10
```

The lines in the example above do the following:

- (line 1 and 2) We add the HTB queuing discipline to device eth0. We define the handle to be 1: and that traffic by default goes to class 1:11. A handle is a label by which we later may refer to this entry in the hierarchy. A class is a unit in the hierarchy to categorize traffic. Filters decide to which class a certain packet belong.
- (line 3 and 4) Here we add the root class to the queuing discipline, which defines the max capacity of the link. With HTB we have root classes and children classes. Root classes are attached to the root in the hierarchy. Child classes we call all the other classes. Root classes may not borrow capacity from each other, but children classes may borrow from their siblings. By defining a root class, and adding children classes to it, these child classes may borrow capacity from each other, which is what we want to achieve in this example. Rate means the reserved capacity for this class, and ceil means the total amount of capacity this class may use. These terms make more sense in the child classes as they may borrow capacity from each other.
- (line 5 and 6) We define the class, that will be used for FTP traffic. We set a max of 900 kilobit per second, even if the class may borrow up to

1 Megabit per second if there is spare capacity in the default class.

- (line 7 and 8) The third class we define states the conditions for the rest of the traffic. This traffic may use all available bandwidth, no matter how much FTP traffic coming in.
- (line 9 and 10) We define the filter that makes the FTP traffic enter the 1:10 class instead of the default. This assigns traffic with the ip port 21 to the highest priority queue in the class 1:10. There is a priority queue in all classes, since **iproute2** automatically attaches a default queuing discipline. This way **iproute2** knows how to handle packets in the same class. As far as **iproute2** is concerned the handle 1: and 1:0 are the same.

### 4.3 The access manager

The role of the access manager is to verify reservations and configure the other components so the traffic from the clients is handled appropriately. As this work can be done by hand, it was not implemented in this project. It is listed to clarify how the implementation should work.

When a client has got a reservation from the bandwidth market, it receives tokens as a receipt. To setup the reservation the client connects to the access manager, and sends its tokens. The access manager verify the tokens, and looks up an unused reservation id. The access manager contacts the marker with a message containing source address, destination address, port and reservation id. Whenever a packet with the matching address pair and port enters the access node the marker marks the packet with the corresponding reservation id. The access manager also connects to the shaper with a message containing the rate and the reservation id. The shaper sets up a filter which will check for the reservation id, in all packet headers, and shape the traffic at the rate. This rate is the same as the access manager sent to the shaper together with the reservation id.

### 4.4 The marker

The purpose of the marker is to inspect packet headers and mark them with reservation ids. The reservation ids will be used by the shaper to give clients, that have reserved capacity, the correct treatment. The marker inspects

the source and destination address and the port, and checks whether this combination has the right to reserved bandwidth. If so, the marker marks the package with reservation id it originally received from the access manager. See 4.7 for a scenario description.

The reservation ids are put into the Type-Of-Service-field (TOS-field) in the IPv4 header. The field is 8 bits wide, giving us the ability to differentiate 255 reservations, which for our testing purposes will suffice. This is not all true since there are restrictions on which values the TOS-field may adopt, but this may be solved by altering the **iptables** and **iproute2** packages. We will use **iptables** to mark the packets, where each reservation corresponds to an **iptables** rule. Such a rule, is defined by the following statement:

```
> iptables -A PREROUTING -t mangle -s 192.168.1.1 \  
-d 192.168.1.2 -p tcp --dport 80 -j TOS --set-tos 2
```

This rule means that if the host 192.168.1.1 sends a package to 192.168.1.2 on port 80 it should be marked with the reservation id 2. If we define a number of these rules they will form a database associating reservation ids to traffic flows in the net, that is what we want the marker to do.

## 4.5 The shaper

The task of the shaper is to force the clients to only use as much capacity as they have reserved. The shaping is implemented with the hierarchical token bucket queuing discipline. Just like in the former examples we start with a root class. The root class has a default child class where all unprioritized traffic goes. Whenever the access manager contacts the shaper, to set up a new reservation, another child class is created with a certain rate. It creates a filter to direct the packets with the reservation id, received from the access manager, to this particular class. The principle is shown in figure 2.

The behavior is implemented with the following command lines. In this example we assume that the maximum capacity in the link is 1 Megabits per second:

```
1 > tc qdisc add dev eth0 root handle 1: htb \  
2 default 10  
3 > tc class add dev eth0 parent 1: classid 1:1 htb \  
4 rate 1Mbit ceil 1Mbit  
5 > tc class add dev eth0 parent 1:1 classid 1:10 htb \  
6 rate 1 Mbit ceil 1Mbit
```



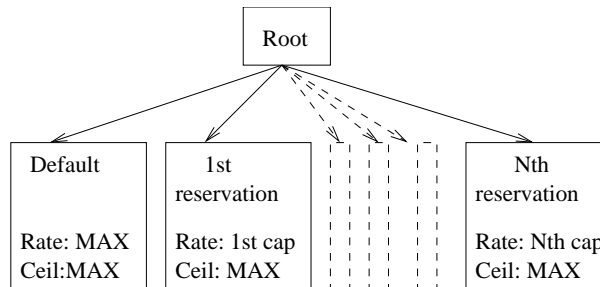


Figure 2: The HTB class hierarchy in the shaper. For every reservation a new child class is created. Rate stands for how much this class is guaranteed and ceil up to which capacity the class may borrow capacity from others. The default class has lower priority, and may get starved by the other classes.

- (line 1 and 2) We add the HTB queuing discipline to device eth0. We set the handle to be 1: and the default class (where traffic goes by default) to 1:10.
- (line 3 and 4) We add the root class 1:1 which limits the total available capacity on the link to 1 Megabits per second.
- (line 5 and 6) We add the default class, to which unprioritized traffic goes.

Whenever a reservation occurs, the access manager contacts the shaper with a reservation id and a rate. The shaper sets up a class, with the rate, and a filter to match the reservation id. In this the reservation id and rate from the access manager is 4 and 500 kilobits and the next free classid is 13.

```

1 > tc class add dev eth0 parent 1:1 classid 1:13 htb \
2 rate 500kbit ceil 1Mbit
3 > tc filter add dev eth0 protocol ip parent 1:0 prio 1 \
4 u32 match ip tos 4 0xff flowid 1:13

```

- (line 1 and 2) We add a child to our root class with classid 1:13, rate 500 kilobits per second. This class may borrow up to 1 Megabits per second from its siblings, if there is unused capacity in the link.
- (line 3 and 4) Here we add a filter that redirects all packets with reservation id 4 to the created class. By setting prio to 1 in the u32 filter we

ensure that the reserved traffic will have priority over the unprioritized one.

If the access manager wants to delete a reservation we delete the class and the filter:

```
1 > tc class del dev eth0 parent 1:1 classid 1:13
2 > tc filter del dev eth0 protocol ip parent 1:0 prio 1 \
3 u32 match ip tos 4 0xff flowid 1:13
```

- (line 1 and 2) We delete the class 1:13.
- (line 3 and 4) We delete the filter sending the packets with reservation id 4 to the class 1:13.

These commands give us the means of defining reservations which are distinguished by reservation ids.

## 4.6 The label switch

A reservation consists of reserved capacity in several links. The shaper handles the capacity and makes sure the clients does not exceed their reserved capacity. We need a component that makes sure the traffic of the specified client uses the reserved path. In this case the component consists of several units, all with the same functionality. These are the label switches.

In this implementation the label switches only route traffic depending on the reservation id. A more complete solution would be to have a component called the header writer. The access manager would, on a reservation, send the header writer a reservation id and the tokens received from the client. These tokens would constitute the path. By writing next hop instructions in the packet header, the header writer could send a packet with a reservation id on a certain path. This is called source routing. In this solution, there is no header writer, and therefore the routes have to be statically configured in the label switches.

The marker puts the reservation ids in the TOS-field. As we have seen before we may use the **ip** command to define rules. These rules may be set to match the TOS-value and send packets with different TOS-values to different routing tables. This way we implement a label switch which routes packages with different reservation ids different routes. We do an example where we

assume a certain node may route packets two separate ways. In this example we want packets with reservation id 2 sent through the host 192.168.1.2 and those with the reservation id 4 targeting host 192.168.1.4. The behavior is accomplished with the following commands:

```
1 > echo 201 route1 >> /etc/iproute2/rt_tables
2 > echo 202 route2 >> /etc/iproute2/rt_tables
3 > ip rule add tos 2 table route1
4 > ip rule add tos 4 table route2
5 > ip route add default via 192.168.1.2 table route1
6 > ip route add default via 192.168.1.4 table route2
```

- (line 1 and 2) Connect aliases to the tables 201 and 202 and call them route1 and route2 respectively.
- (line 3 and 4) Add rules to the rule table which will make all packets with TOS-values 2 and 4 go to the routing tables route1 and route2 respectively.
- (line 5 and 6) Set the default gateway in the routing tables to their corresponding hosts.

When we want the label switch to replace the former label with a new one we use **iptables**:

```
1 > iptables -A PREROUTING -t mangle -m tos --value=4\  
2 -j TOS --set-tos 2
```

This command adds a rule, resembling of the one used with the marker, to change all packets with the TOS-value 4 to 2.

## 4.7 Putting it together

It is now shown how the **iptables** and **iproute2** packages are used to implement the core components needed for the capacity reservation. In this simple example there are only two routes, but more routes could be handled in the same way. The figure 3 shows the setup.

Here is a description of the scenario. The numbers in the figure 3 correspond to the numbers listed below:

1. Client A sends a reservation request to the access manager. A asks for X Kbit over route 2 to the destination D on port P.

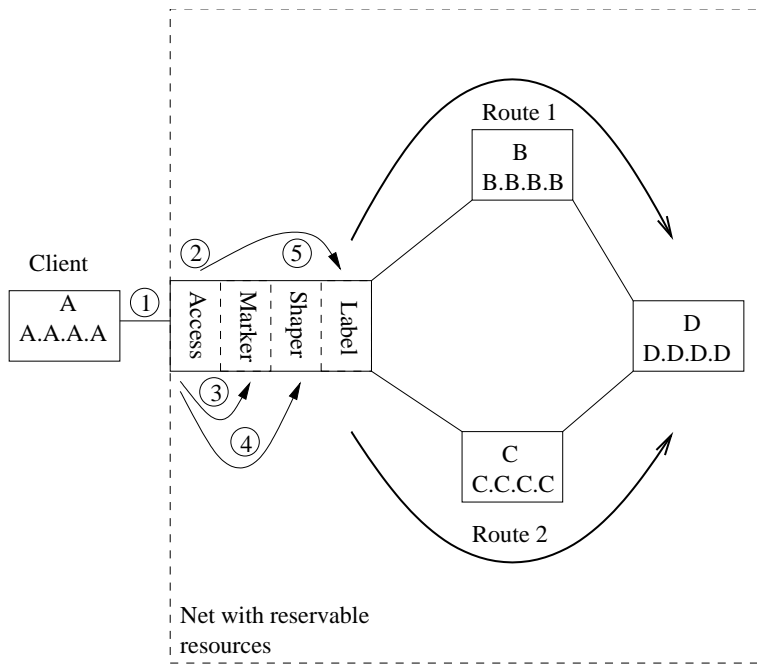


Figure 3: The setup phase of a reservation. The client (A) wants to reserve capacity to node D in the network. The figure shows how the components and the client interacts with each other.

2. The access manager checks in its internal table to find the next unused reservation id I.
3. The access manager sends the reservation id I to the marker, together with data received from A. This data is the source address (A.A.A.A), destination address (D.D.D.D) and the port value (P). The marker sets up a rule that marks all packages with the corresponding source and destination address and port with the reservation id I received from the access manager.
4. The access manager sends a message with the rate X and reservation id I to the shaper which sets up a shaper class with the rate X and a filter matching the reservation id I. This way the shaper shapes all packets with reservation id I, to rate X.
5. The access manager sends the reservation id I to the label switch together with the route request from A (route 2 over address C.C.C.C). The label switch set up a new routing table and a rule that binds the reservation id I to this routing table. The label switch also sets the default gateway in this routing table to C.C.C.C.

## 4.8 Discussion of other approaches

It is one problem with the implementation. For the routing through the net to be effective, it is important that the access node may source route the packets through the net. This is because it should be easy to reconfigure paths. This is why the header writer, discussed in section 4.6, plays an important role in a real implementation. This implementation shows the possibility to route on a reservation id. In a real world implementation a technique like MPLS [25] would probably be used for the routing instead.

One important feature of the implementation is that it stores reservation ids in the packet headers. This is done in order to give the right packets the right service. One design issue is were to put this reservation id. During the project two approaches were considered: using IPv4 and the Type-of-service-field (TOS-field) or using the flowlabel in IPv6.

Using the flowlabel was considered first and the approach was to analyze the IPv6 stack in the Linux kernel and alter it. Since the intended modifications were rather small it seemed straightforward. To make this possible the protocol stack of the Linux kernel had to be understood. This turned out

to be hard, as the IPv6 stack in Linux is very advanced software. With no experience of the internals of the Linux kernel, it would have taken long time to gain enough knowledge. As long as the whole stack is not understood, the consequences of a change are very hard to oversee. This is why another way of doing the implementation, was searched for.

Even if the approach with altering the Linux kernel was abandoned, the idea of using the flowlabel and IPv6 was not. The idea of using the **iptables** and **iproute2** packages came from an example in the “Linux Advanced Routing & Traffic Control HOWTO” [20], where the TOS-field where used to route packets differently. The features used where the TOS-match module of **iptables** and the multiple routing tables of **iproute2**. Writing a match module to the flowlabel, did not seem hard to do. The problems began when trying to use multiple routing tables as this feature was not implemented in the IPv6 stack at all. Therefore, IPv4 and the TOS-field were used instead.

#### 4.8.1 Implementation with IPv6

The IPv4 solution, using the TOS-field to store reservation ids, works but has several drawbacks. First, the TOS-field is too small to handle more reservations than 255. If we would like to realize the bandwidth market, in the real world, this is far too limited. The second drawback is to use the TOS-field to something else it is intended to. This is not a problem as long as the affected packets stay inside our net, but if we connect with others these modified packets may cause problems elsewhere.

The IPv6 flowlabel field is much better suited for the purpose of marking packets with reservation ids. It is 20 bytes wide and its use is not as standardized as that of the TOS-field. Here is a list on what needs to be done to make the IPv6 solution work:

- Marker. Write an **iptables** target extension to set the flowlabel in an IPv6 packet.
- Label switch. Add support for multiple routing tables to the IPv6 stack in the Linux kernel. Also, extend the **iproute2** package making it possible to match the flowlabel with an ip rule.

## 5 Testbed and test programs

To test and verify the implementation a testbed was set up. The testbed consists of a virtual network, with nodes running Linux and a number of test programs. To emulate a virtual network a program called User mode linux [2] (UML) was used. This program makes it possible to run several instances of Linux on one single computer, or a *host system*. By running a number of *virtual instances*, and connect them together, it is possible to simulate the behavior of a network. To test the implementation other programs are needed as well: traffic generators and packet loggers. Two programs were used as traffic generators: **DBS** [21] as in “Distributed Benchmarking System” and a program written especially for the project called **trgen**. As packet logger **tcpdump** [18] was used. The text processing programs **sed** [23] and **awk** [24] were used to parse the files generated by **tcpdump**. **octave** [22] was used to plot the results of the tests. This section describes the virtual testbed as well as the program used to test the implementation.

### 5.1 User Mode Linux - UML

UML is a program based on the Linux kernel. The kernel has been rewritten to make it possible to run it like an application instead of an operating system. Usually, the Linux kernel communicates directly with the hardware on the computer Linux is running on. The difference with UML is that it communicates with the operating system instead of the hardware, making it possible to run several virtual instances at the same host system. UML has a number of ways to communicate with the host system or other virtual instances. The easiest way is to use a program, running on the host system, called **uml\_switch** which emulates the behavior of a switch where virtual instances may connect to each other. The testbed, constructed in this project, consists of four instances of UML running on one computer as in figure 7. All virtual instances are connected with one instance of **uml\_switch** on every link. The node A also is connected to the host system, which in the tests acts as client. Appendix A describes how to set up the testbed used in the tests.

## 5.2 DBS - Distributed Benchmarking System

**DBS** is a program to test capacity of links in an IP network. It consists of a control program, measuring daemons and a report generator. The control program runs a script file which describes a test scenario and sends work orders to the measuring daemons. The measuring daemons receive work orders and generate and measure traffic between each other. When the test is done the measuring daemons send back the results to the control program which concludes them into a result file. The report generator uses the result file to generate plots, to illustrate different characteristics of the measured links.

We will now take a closer look on the script file used by the control program, to conduct the tests. One file describes one or more streams. Every stream consists of a sender, a receiver, and some entries describing the stream itself. Here is an example:

```
{
sender {
    hostname    = 192.168.1.1;
    port        = 4711;
    pattern     {1024,1024,0.01,0.0}
}
receiver {
    hostname    = 192.168.1.2;
    port        = 4711;
    pattern     {10000,4096,0.0,0.0}
}
file           = data/results;
protocol       = UDP;
start_time    = 0.0;
end_time      = 30.0;
}
```

First we define the sender. It has a hostname, a port (on which the stream will be sent) and a traffic pattern. The pattern has four fields which describes the characteristics of the traffic. The pattern in this example means: send 1024 bytes one hundred times per second. This results in a load of 800 kilobits per second. For a more thorough description of how to set up patterns, please look up the manual of DBS [21]. The receiver has the same entries as the sender, but now the pattern describes the characteristics of the receiving



buffers. This pattern means that we should have a receiving buffer with the size of 10000 bytes and be able to receive messages of the size 4096, with no time delay between the received messages. The receivers characteristics are deliberately set to larger values than the senders, to ensure the receiver always will be able to receive the messages. The entries describing the stream are the file (where to put the results), protocol (TCP or UDP), start time and the end time. If we need more streams, we add another structure in the file.

**DBS** was only used as traffic generator, since it was hard to control the way the report generator processed the data before plotting it. To measure and plot the traffic generated by **DBS**, **tcpdump** and **octave** were used instead.

### 5.3 **trgen** - Traffic Generator

Before **DBS** was discovered, a program for generating traffic called **trgen** was made. It has not the same possibilities to configure the test in detail as **DBS**, but it is easier to setup and use. It consists of a server and a client, communicating with TCP. The server is written in C and is run with a number of port numbers on the command line. The server spawns as many threads as given port numbers and starts listening on them. The client connects to the server on one of the given ports and sends a file, which name is given on the command line. The code is listed in Appendix C. Here is an example on how to run the server and the client:

```
> trgen_server 4711
> trgen_client localhost 4711 testfile
```

By these commands we start the server to listen to port 4711. Then we start the client to send the file “testfile” on port 4711 to the host localhost. **trgen** was used in test 2, because it is easier generate traffic with it. As in the case with **DBS**, **tcpdump** and **octave** were used for measuring the traffic and plot the results.

### 5.4 Logging and presenting the data

**tcpdump** is a program used to log Internet traffic. The typical output of **tcpdump** presents the packet type, source, destination, port and the packets size and the time it arrived. **tcpdump** may be used to access more

data about a packet than that, but this information was the one used in this project. The output of **tcpdump** was processed with **awk** and **sed** to make it plottable with **octave**.

**awk** and **sed** are tools found in almost every UNIX system. These tools are good for text processing. **awk** is mostly used to split lines into columns, and perform actions on these columns. **sed** is often used for altering the contents of a file or stream of characters. A common use is to do search and replace actions, or to remove certain symbols in a file.

**octave** is a free clone of **Matlab**, which is the de facto standard of numerical calculating software. During the tests **octave** was used to process, and plot the data from the tests.

## 6 Testing and evaluation

Two tests were conducted, to verify the functionality of the implementation:

- Test on *capacity*, where the shaping abilities of the system were verified.
- Test on *TOS-field based routing*. Test that routing based on the TOS-field, instead of the destination address, works. Verify that the path can be reconfigured during a session.

### 6.1 Test on capacity

#### 6.1.1 Motivation

In a net controlled by a bandwidth market, there must be ways to guarantee that clients using the net, do not exceed their reserved capacity. This is solved by letting the traffic from the clients be shaped at the access node. This is why the shaping capabilities of the solution were tested. It is also important that unused capacity belonging to reserved traffic may be used by non-reserved traffic.

#### 6.1.2 Method

Two computers were connected, one sender and one receiver. A hierarchical token bucket filter (HTB) was applied to the sender's output. The configuration of the HTB was the following: we have three classes of traffic, flow 1, flow 2, and flow 3, which all flow on the same link. The link has capacity 300 kilobits per second. All classes are at least guaranteed capacity up to 100 kilobits per second. If there is unused capacity in one of the other classes, the classes may borrow capacity from each other up to the maximum capacity of the link. The configuration is shown in the figure 4.

Each flow is distinguished by a port number. In order to verify that the different flows borrow from each other we must not send the flows simultaneously all the time. Therefore the flows were generated according to the following schemata: At  $t_0$  send flow 1 at a rate exceeding the maximum capacity. Flow 1 should use all available capacity. At  $t_1$  start sending flow 2. We should now see how the rate of flow 1 decreases because of flow 2. At  $t_2$  flow 3 starts sending, and we should see how the other two flows decrease their rates. Finally at the time  $t_3$  flow 1 stops sending and we should notice

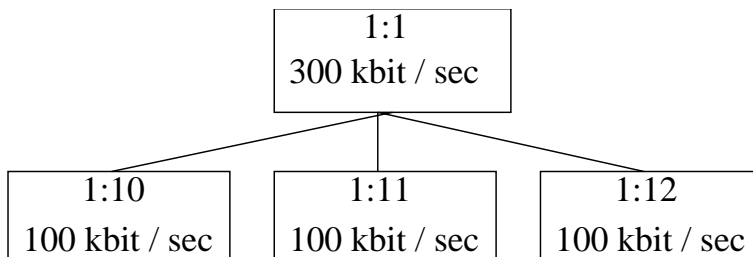


Figure 4: The HTB setup in the capacity test. The 1:1, 1:10,1:11 and 1:12 are the labels of the classes in the hierarchy. The rates denote how much capacity the classes are guaranteed.

an increase of the rate in the other flows as they use the unused capacity of flow 1. The total should all the time be 300 kilobits. The flows were generated by **DBS** and were sent with UDP. Packets with the size of 1 kilobyte were sent one hundred times per second. The size of the packets were chosen to be less than the maximum-transferable-unit (MTU) between the two hosts. The size and time of arrival of all incoming packet were measured with **tcpdump** at the receiver. The log files from **tcpdump** were parsed with **awk** and plotted with **octave**. The commands used to set up the HTB, **DBS**, **tcpdump**, **awk** and **octave** are all listed in Appendix B.

### 6.1.3 Hypothesis

The expected behavior is shown in figure 5. This qualitative graph shows how the rates change during the test.

The expected behavior of the shaper is described with the following statements:

1. The capacity of any class will always be greater or equal to 100 kilobits per second.
2. Unused capacity, in one class may be used by a flow belonging to another class.
3. The total sent rate of the three flows will never exceed the maximum capacity of 300 kilobits per second.

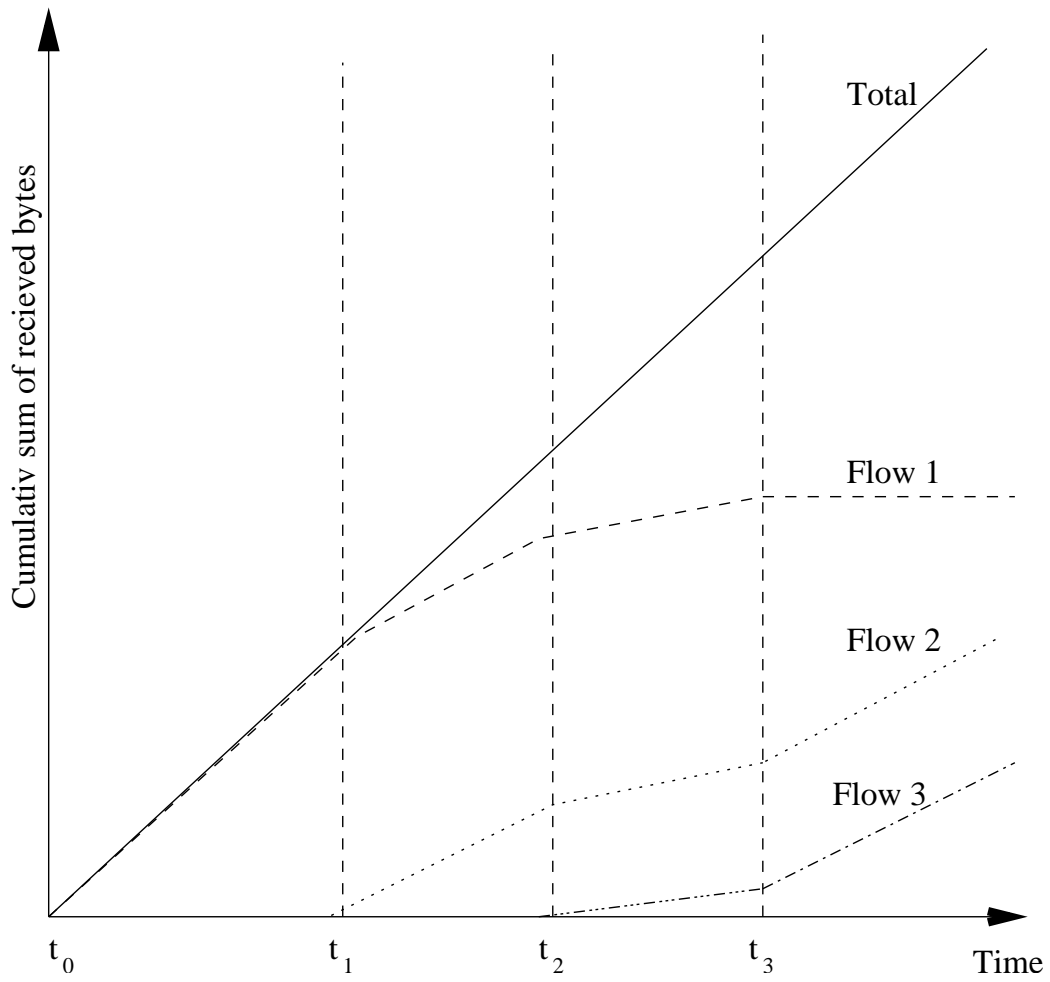


Figure 5: The expected result of the capacity test. At  $t_0$  flow 1 starts, at  $t_1$  flow 2 starts to be followed by flow 3 at  $t_2$ . At  $t_3$  flow 1 stops. Notice how the flows borrow capacity from each other.

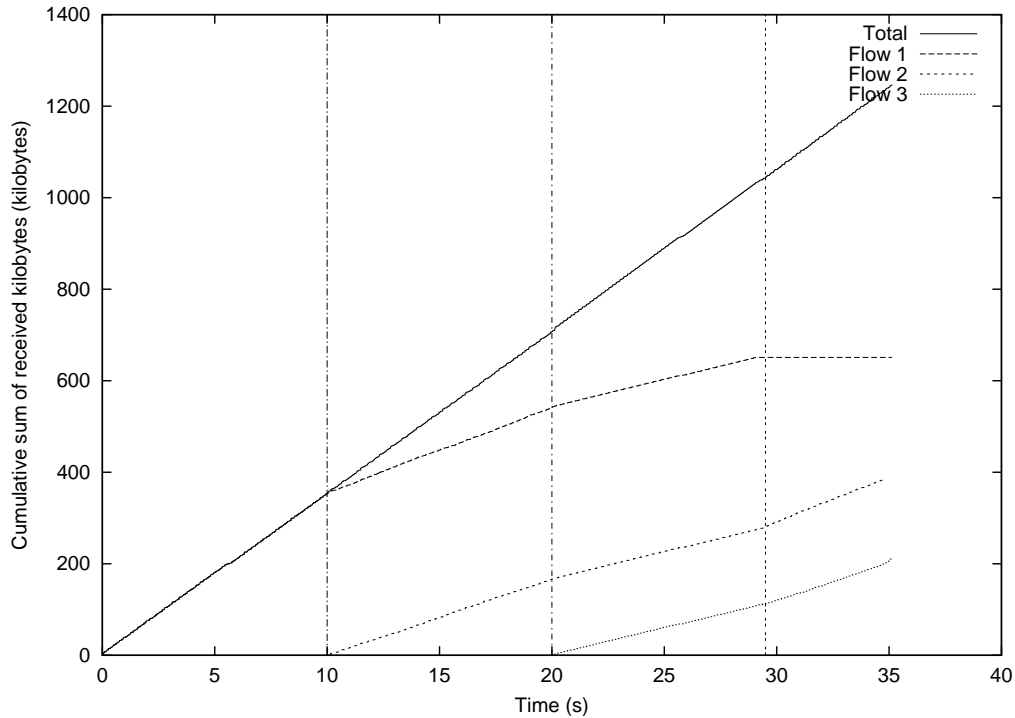


Figure 6: The results of the capacity test. Compare with the expected results and notice how the shaper works in the expected way.

#### 6.1.4 Results

The result of the test is shown in figure 6. If we compare with the expected results we see that the configuration works in the expected way.

#### 6.1.5 Discussion

At first the test were performed in the UML testbed. This did not work as the hierarchical token bucket shaped the traffic at too low rates. The reason of this phenomenon was not investigated further, but may have to do with the clock skew effects a virtual machine may suffer from. This was suggested from a conversation with Jamal Hadi at the networking development mailing list. See Appendix E to view the conversation. Therefore the test were conducted between two physical computers.

At first, the results, were unexpected as well. It looked like the total rate exceeded the maximum of 300 kilobits per second with about one fourth.

After further investigation the reason of this turned out to be the following: the test were conducted with UDP traffic, i.e with no delivery control. The size of the sent packets were 2 kilobyte, but the maximum transferable unit (MTU) between the two hosts were only 1540 bytes. Therefore, the packets got truncated, and the rate miscalculated. According to the measurements the rate was 390 kilobits per second when it was supposed to be 300 kilobits per second. Our measurements where based on the assumption that the packets where 2048 bytes, but the shaper shaped the packets on their real size, which was 1540 bytes. By dividing the real packet size (1540 bytes) with the assumed size (2048 bytes) and multiply with the measured rate we should get the real rate of the shaper:

$$\frac{1540}{2048} \cdot 390 \approx 293 \text{ kilobits / second}$$

This is approximately the expected rate of the shaper.

## 6.2 TOS-based routing test

### 6.2.1 Motivation

This test was performed to test that routing based on the TOS-field instead of the destination address works and to verify that the path can be reconfigured during a session. If it is possible to route on the TOS-field, it should be possible to route on the IPv6 flowlabel as well. That is what we would have tested if the implementation had used the flowlabel instead of the TOS-field.

### 6.2.2 Method

This test used the UML environment. The topology is shown in figure 7. The client (Cl) sends traffic to the target host D. All packets coming from the client are marked by the host A, by setting the TOS value. The host A is configured to route packets with the TOS-value set to 2 through host B, and packets with the TOS-value set to 4 through host C. With **tcpdump** the traffic is measured at the hosts A, B and C. A initially marks packets from the client with TOS-field set to 2. After about 60 seconds A is modified to mark the packets with the TOS-field set to 4. At this moment we should notice a route switch. To parse the logs, from **tcpdump**, we used **awk**, **sed** and **grep**. The formatted data was plotted with **octave**. A full description of the commands and programs used is found in Appendix B.

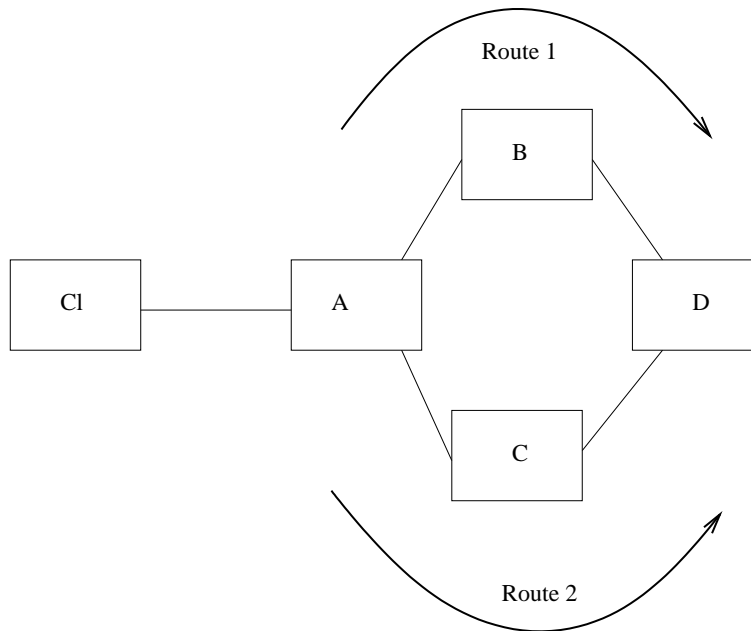


Figure 7: The topology of the testbed.

The traffic is generated by the `trgen` program sending TCP packets described in section 5.

### 6.2.3 Hypothesis

It is possible to route traffic with the TOS-field instead of the destination address.

### 6.2.4 Results

The results are shown in figure 8. The figure shows how the total number of received bytes increases constantly. It also shows that when the TOS flag is switched at a time around 30 seconds, the traffic ceases to go through host B and starts going through host C instead. It is worth noting the phenomenon near origo of the graph. Here we see how the traffic rate is very high in the beginning. The reason of this have not been investigated further but may have to do with a send buffer in the client that fills up in the beginning of the test. “Host C” curve shows the load in host C after the switch. It is not



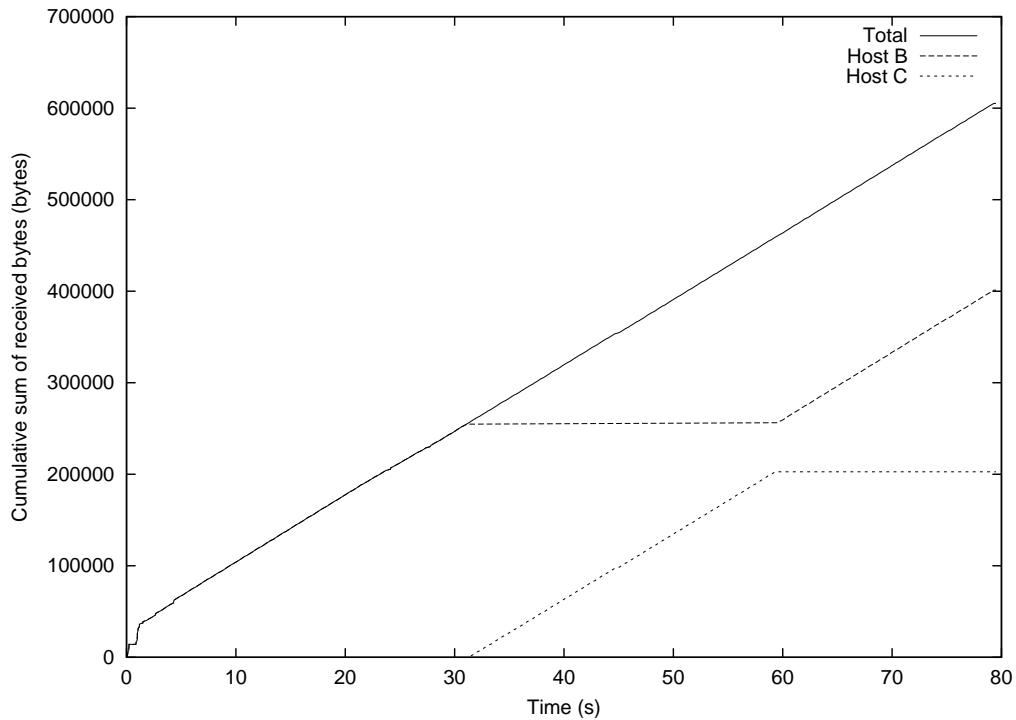


Figure 8: The results of the TOS-field routing test. The TOS value is switched around the time 30 s.

a new flow that starts sending. That is why the phenomenon does not occur in the “Host C” curve.

### 6.2.5 Discussion

The test shows how it is possible to use the TOS flag to route traffic. This test was primary a test of concept that it is possible to use the TOS-field to route traffic. If it is possible to route on the TOS-field there should be possible to do the same with the flowlabel.

## 7 Conclusions

### 7.1 Overview

The implementation, proposed in this thesis, is one possible solution on how to build the components needed in a net, in which resources are reserved with a bandwidth market. The conclusions we may draw are the following:

- The components needed to implement a net, which resources are reserved with a bandwidth market, are possible to implement with standard tools. These tools are **iptables** and **iproute2**.
- The project went too focused on details when trying to implement a solution outside a simulation environment. Altering the Linux kernel demands more resources and knowledge to be feasible.
- Linux offers a lot of possibilities when it comes to traffic control, filtering, mangling and shaping. What these possibilities are is described in chapter 4.
- In order to do the implementation with IPv6 and the flowlabel the IPv6 protocol stack needs to be extended with support for multiple routing tables. Extensions to **iptables** and **iproute2** is also needed to be able to read and set the flowlabel.

### 7.2 Future work

Proposed work in the future would be to set up a simulator environment, to test the algorithms needed to the bandwidth market. It would also be interesting to investigate the possibility of putting the meta data (tokens and switching info) in the packet payload instead of in the packet header. This way one would not interfere with IP and the way its header fields are used today. To make the model work as it is supposed to, implementation of source routing capabilities is needed. This could be solved with MPLS [25].

## 8 Bibliography

### References

- [1] S. Deering, R. Hinden *Internet Protocol, Version 6 (IPv6) Specification* RFC 2460, Internet Engineering Task Force, Dec. 1994.  
<http://www.ietf.org/rfc/rfc2460.txt?number=2460>
- [2] Jeff Dike et. al. *User Mode Linux, UML*,  
<http://user-mode-linux.sourceforge.net>
- [3] R. Braden, ISI and D. Clark, MIT. *Integrated Services in the Internet Architecture: an Overview*  
<http://www.ietf.org/rfc/rfc1633.txt?number=1633>
- [4] R. Braden, ISI, et. al. *Resource ReSerVation Protocol (RSVP)*  
<http://www.ietf.org/rfc/rfc2205.txt?number=2205>
- [5] S. Blake, Torrent Networking Technologies et al. *An Architecture for Differentiated Services*, <http://www.ietf.org/rfc/rfc2475.txt?number=2475>
- [6] W.Richard Stevens, *TCP/IP Illustrated, Volume 1, The protocols*, Addison-Wesley
- [7] P. Almquist *Type of Service in the Internet Protocol Suite*,  
<http://www.ietf.org/rfc/rfc1349.txt?number=1349>
- [8] C. Partridge *Using the Flow Label Field in IPv6*,  
<http://www.ietf.org/rfc/rfc1809.txt?number=1809>
- [9] Zhi-Li Zhang, Zhenhai Duan, Lixin Gao and Yiwei Thomas Hou, *Decoupling QoS Control from Core Routers: A Novel Bandwidth Broker Architecture for Scalable Support of Guaranteed Services*, Pages: 71 - 83 Series-Proceeding-Article, 2000 ACM Press.
- [10] *Internet Engineering Task Force*,  
<http://www.ietf.org>
- [11] Klara Nahrstedt and Shigang Shen *Coexistence of QoS and Best-Effort* in Proceedings of 10th IEEE Tyrrhenian International Workshop on Digital Communications: Multimedia Communications, Ischia, Italy, September 1998.

- [12] Rusty Russell, *Netfilter and iptables*  
<http://www.netfilter.org/>
- [13] Christian Huitama 1998. *IPv6 - The new Internet protocol (2nd edition)*  
Prentice Hall.
- [14] Robert M. Hinden, *IP next generation overview*, Communications of the ACM, volume 39, number 6, 1996, ACM Press  
<http://doi.acm.org/10.1145/228503.228517>
- [15] L. Rasmusson and E. Aurell, 2001. *A Price Dynamics in Bandwidth Markets for Point-to-point Connections*, SICS, Swedish Institute of Computer Science. SICS-T-2001/21-SE
- [16] Martin Devera *Hierarchical Token Bucket*,  
<http://luxik.cdi.cz/~devik/qos/htb/>
- [17] *IPROUTE2 Utility Suite Howto*,  
<http://www.linuxgrill.com/iproute2.doc.html>
- [18] *TCPdump homepage*,  
<http://www.tcpdump.org/>
- [19] *Classless Inter-Domain Routing (CIDR) Overview*,  
<http://public.pacbell.net/dedicated/cidr.html>
- [20] *Linux Advanced Routing & Traffic Control HOWTO*,  
<http://lartc.org/howto/>
- [21] *Distributed Benchmarking System manual*,  
[http://www.kusa.ac.jp/~yukio-m/dbs/dbs\\_man.html](http://www.kusa.ac.jp/~yukio-m/dbs/dbs_man.html)
- [22] *Octave homepage*,  
<http://www.octave.org/>
- [23] *SED - the Stream Editor*,  
<http://www.math.fu-berlin.de/~guckes/sed/>
- [24] *The AWK programming language*,  
<http://cm.bell-labs.com/cm/cs/awkbook/index.html>

- [25] *MPLS tutorial*,  
<http://www.nanog.org/mtg-9905/ppt/mpls/>
- [26] Donald Ferguson, Yechiam Yemini, and Christos Nikolaou, *Microeconomic Algorithms for Load Balancing in Distributed Computer Systems*, Proc. of the 8th Int. Conf. on Distributed Computer System, 1988.

## Appendix A - Setting up the virtual network

This section describes how to setup up the virtual testbed using UML referred to in chapter 5. All the commands listed are run in the BASH shell. Download the needed files:

```
> wget http://www.kernel.org/pub/linux\
/kernel/v2.4/linux-2.4.18.tar.bz2
> wget http://telia.dl.sourceforge.net\
/sourceforge/user-mode-linux/uml-patch-2.4.18-47.bz2
> wget http://telia.dl.sourceforge.net\
/sourceforge/user-mode-linux\
/uml_utilities_20020729.tar.bz2
> wget http://telia.dl.sourceforge.net\
/sourceforge/user-mode-linux\
/root_fs.rh-7.2-full.pristine.20020312.bz2
> wget http://luxik.cdi.cz/~devik/qos\
/htb/v3/htb3.6-020525.tgz
> wget http://www.sics.se/~gabriel/kernel-config
> wget http://www.sics.se/~gabriel/trgen.tar.gz
> wget http://www.sics.se/~gabriel/env_skel.tar.gz
> wget http://www.kusa.ac.jp/~yukio-m/dbs\
/software1.2.0beta1/dbs-1.2.0beta1.tar.gz
> wget http://www.kusa.ac.jp/~yukio-m/dbs\
/software1.2.0beta1/dbs-1.2.0beta1-src.tar.gz
```

Put these files in the same directory, called `$WORKINGDIR`. Unpack the linux kernel file and the root filesystem. Patch the kernel with the HTB and UML patches. Then compile UML:

```
> export $WORKINGDIR='pwd'
> tar jxf linux-2.4.18.tar.bz2
> bunzip2 root_fs.rh-7.2-full.pristine.20020312.bz2
> tar zxf htb3.6-020525.tgz
> ln -s linux linux-2.4
> patch -p0 htb3.6_2.4.17.diff
> cd $WORKINGDIR/linux
> cat ../uml-patch-2.4.18-47.bz2 | bunzip2 - | patch -p1
> cp ../kernel-config.config
> make linux ARCH=um && make modules ARCH=um
```

Make a directory where to mount the root root filesystem. This may be done in the “linux” directory:

```
> mkdir tmp
# Next thing must be done as root
> mount ../root_fs.rh-7.2-full.\
pristine.20020312 tmp -o loop
> make modules_install ARCH=um \
INSTALLMOD_PATH='pwd'/tmp
```

Copy the patched “tc” command to the root filesystem:

```
> cp ../tc tmp/sbin/tc
```

Unpack and compile the DBS package:

```
> cd $WORKINGDIR
> tar zxf dbs-1.2.0beta1.tar.gz
> tar zxf dbs-1.2.0beta1-src.tar.gz
> cd $WORKINGDIR/dbs/src
> make
> cp dbstd $WORKINGDIR/linux/tmp/sbin
```

Unpack the trgen-package and unmount the root filesystem:

```
> cd $WORKINGDIR
> tar zxf trgen.tar.gz
> cp trgen/trgen_server linux/tmp/sbin
> umount linux/tmp
```

Compile and install the UML utilities:

```
> tar jxf uml_utilities_20020729.tar.bz2
> cd tools
> make && make install
```

Unpack the environment skeleton and start the environment:

```
> cd $WORKINGDIR
> tar zxf env_skel.tar.gz
> ./start_env
```

A lot of windows will be opened. Look for the ones named “Host 1”, “Host 2”, “Host 3” and “Host 4”. When they have loaded log in to each of one of them as “root” with password “root” Make a directory in all the UML-instances called /tmp/host and make a file called “mounthost” with the following contents:

```
#!/bin/bash
```

```
mount none /mnt/host -t hostfs
```

and make the file runnable. Also do a symlink in all the instances like this:

```
> ln -s /mnt/host/$WORKINGDIR/host1/setup
```

Where “host1” is the name of the host you are currently in. Then you only have to type:

```
> ./setup
```

in all the instances to setup the networking so it works like in figure 7.



## Appendix B - Commands to set up the test cases in the testing chapter

### Test 1 - Test on capacity

The setup of the HTB is shown in the figure 4. There we see four classes named 1:1, 1:10, 1:11 and 1:12. We see that 1:1 is parent to the others. Child classes may always borrow from each other, but they may not exceed the max capacity of their parent, or the max capacity of themselves. In this case, this means that all three child classes may borrow up to 200 kilobits, using a total of 300 kilobits, as long as no other class is using its capacity.

The setup consists of two hosts: a sender and a receiver. The hierarchical token bucket was applied on the output (eth0) of the sending host by the following commands. They were run in the BASH shell. For information of what these commands do, please refer to the section 4 where similar examples are shown and discussed.

```
>tc qdisc add dev eth0 root handle 1: htb default 12
>tc class add dev eth0 parent 1: classid 1:1 htb \
rate 300kbit ceil 300kbit
>tc class add dev eth0 parent 1:1 classid 1:10 htb \
rate 100kbit ceil 300kbit
>tc class add dev eth0 parent 1:1 classid 1:11 htb \
rate 100kbit ceil 300kbit
>tc class add dev eth0 parent 1:1 classid 1:12 htb \
rate 100kbit ceil 300kbit
```

We also need to set up filters to redirect the two flows to the correct class in the hierarchy:

```
>U32="tc filter add dev eth0 protocol ip parent 1:0 \
prio 1 u32"
>$U32 match ip dport 4711 0xffff flowid 1:10
>$U32 match ip dport 4712 0xffff flowid 1:11
>$U32 match ip dport 4713 0xffff flowid 1:12
```

This makes the traffic to the ports 4711, 4712 and 4713 go to the different classes in the HTB respectively. In order to generate the traffic we want to measure we use DBS, using file test1.cmd. See Appendix F: It makes **DBS** send UDP packets with the size of 1024 bytes 100 times a second.

The traffic were measured by running **tcpdump** at the input (eth0) of the receiver:

```
> tcpdump -i eth0 -w logfile
```

To sort out the different flows we run **tcpdump** again on the generated logfile, together with **awk**:

```
#Traffic on the port 4711
> /usr/sbin/tcpdump -tt -r logfile "dst_port_4711" | \
awk {'print $1 "\t" $6'} > 4711.frm
#Traffic on the port 4712
> /usr/sbin/tcpdump -tt -r logfile "dst_port_4712" | \
awk {'print $1 "\t" $6'} > 4712.frm
#Traffic on the port 4713
> /usr/sbin/tcpdump -tt -r logfile "dst_port_4713" | \
awk {'print $1 "\t" $6'} > 4713.frm
#Total traffic on the ports 4711, 4712 and 4713
> /usr/sbin/tcpdump -tt -r test3.dbs.1024.dump \
"dst_port_4711||_dst_port_4712||_dst_port_4713" | \
awk {'print $1 "\t" $6'} > total.frm
```

These operations render four files which are suited to plot in **octave**.

```
# Start octave
> octave
> a=load(4711.frm);
> b=load(4712.frm);
> c=load(4713.frm);
> d=load(total.frm);
> os=a(1,1)
> xlabel("Time_(s)")
> ylabel("Cumulative_sum_of_received_bytes_(bytes)")
> plot(a(:,1)-os,cumsum(a(:,2)), \
      b(:,1)-os,cumsum(b(:,2)), \
      c(:,1)-os,cumsum(c(:,2)), \
      d(:,1)-os,cumsum(d(:,2)));
```

These operations render the diagram shown in figure 6.

## Test 2 - Test on capacity

First we set up **iptables** to mark all packets from the client (C1) with IP address 192.168.5.5, by setting the TOS value to 2 at host A:

```
#Host A
> iptables -t mangle -A PREROUTING -s \
192.168.5.5 -j TOS --set-tos 2
```

Then we set up host A to route traffic differently, depending on the TOS value. For a description on what these commands do please refer to section 4 where there are similar examples.

```
> echo 201 rt_1 >> /etc/iproute2/rt_tables
> echo 202 rt_2 >> /etc/iproute2/rt_tables
> ip rule add tos 2 table rt_1
> ip rule add tos 4 table rt_2
> ip route add default via 192.168.1.2 dev eth1 table rt_1
> ip route add default via 192.168.3.3 dev eth2 table rt_2
```

First we create two routing tables called `rt_1` and `rt_2`. Then we add a rule that sends all packets with TOS value 2 to `rt_1` and those with TOS value 4 to `rt_4`. Then we set the default gateway to host B (with IP address 192.168.1.2) in `rt_1` and host C (with IP address 192.168.3.3) in `rt_2`. Then we set up **tcpdump** on host A, B and C to record the traffic through the hosts:

```
# Host A
> tcpdump -i eth0 -w hostA.dump
# Host B
> tcpdump -i eth0 -w hostB.dump
# Host C
> tcpdump -i eth0 -w hostC.dump
```

Then we start the **trgen**, traffic generator discussed in section 5.

```
# Host D (the server plus the port to listen to)
> trgen_server 4711
# Client (Cl)
# the client program, the receiving host
# the port and the file to send
> trgen_client 192.168.2.4 4711 test
```

The file “test” is an empty file with the size of 5 MB generated in the following way with **dd**:

```
> dd if=/dev/zero of=./test bs=1024 count=5000
```

During the time the client sends the file, the following command is issued at host A in order to switch the route.:

```
> iptables -F -t mangle; iptables -t mangle \\  
-A PREROUTING -j TOS --set-tos 4
```

This command first flushes the old rule in the mangle table, and then telling **iptables** to set the TOS value to 4 instead of 2. The files generated with **tcpdump** were parsed:

```
> tcpdump -tt -r hostA.dump \\  
"dst_port_4711&&dst_host_192.168.2.4" | \\  
awk {'print $1 $6'} | sed 's/[(:)]/ /g' | \\  
grep -v ack > hostA.frm  
> tcpdump -tt -r hostB.dump \\  
"dst_port_4711&&dst_host_192.168.2.4" | \\  
awk {'print $1 $6'} | sed 's/[(:)]/ /g' | \\  
grep -v ack > hostB.frm  
> tcpdump -tt -r hostC.dump \\  
"dst_port_4711&&dst_host_192.168.2.4" | \\  
awk {'print $1 $6'} | sed 's/[(:)]/ /g' | \\  
grep -v ack > hostC.frm
```

Then we plot the formatted data with **octave**:

```
# Start octave  
> octave  
> a=load('HostA.frm');  
> b=load('HostB.frm');  
> c=load('HostC.frm');  
> os=a(1,1)  
> xlabel("Time_(s)")  
> ylabel("Cumulative_sum_of_received_bytes_(bytes)")  
> plot (a(:,1)-os,cumsum(a(:,3)), \\  
b(:,1)-os,cumsum(b(:,3)), \\  
c(:,1)-os,cumsum(c(:,3)));
```

These commands render the graph in figure 8.

## Appendix C - The trgen source code

The source code to the `trgen_server` is written in C:

```
#include <stdio.h>
#include <stdlib.h>
#include <error.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <pthread.h>
#include <time.h>
#define MAXTHREADS 10
#define BACKLOG 10
#define REPORTINTERVAL 100

int handle_session(int new_fd,
short port,
int lastconn){
    char buf[10000];
    int filesize , i , retval;

    sprintf(buf , "log-%d.%d" , port , lastconn );

    retval = recv(new_fd , buf , strlen(buf) , 0);
    if(retval <= 0){
        goto exit;
    }else{
        printf("%d: Got: %s\n" , port , buf);
        fflush(stdout);
        filesize = atoi(buf);

        for(i = 1; i <= filesize; i++){
            retval = recv(new_fd , buf , 1 , 0);
            if(retval <= 0){
                goto exit;
            }
            if(i % (1024 * REPORTINTERVAL) == 0){
                printf("%d: %d kilobytes received\n" ,
port , i / 1024);
                fflush(stdout);
            }
        }
    }
}
```

```

    }
}

exit:
    if(retval < 0)
        fprintf(stderr, "!!!ERROR!!!\n");
    else if(retval == 0)
        fprintf(stderr, "Closed_connection\n");
    close(new_fd);
    return 0;
}

void *start_listen(void *argref){
    int i, new_fd, sin_size;
    int retval, sockfd, lastconn = 0;
    unsigned short port;
    struct sockaddr_in my_addr, their_addr;

    port = *((unsigned long *)argref);
    printf("Listening_on_port:_%d\n", port);

    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if(!sockfd)
        pthread_exit(0);

    my_addr.sin_family = AF_INET;
    my_addr.sin_port = htons(port);
    my_addr.sin_addr.s_addr = INADDR_ANY;
    memset(&(my_addr.sin_zero), '\0', 8);

    bind(sockfd,
        (struct sockaddr *)&my_addr,
        sizeof(struct sockaddr));

    if(listen(sockfd, BACKLOG) == -1)
        pthread_exit(0);

    sin_size = sizeof(struct sockaddr_in);
    new_fd = 0;

```

```

    while((new_fd = accept(sockfd ,
        (struct sockaddr *)&their_addr ,
        &sin_size ))){
        handle_session(new_fd , port , lastconn++);
        printf("Session_finnished!\n");
    }

    printf("%d:_No_I_die!\n" ,port );

    return;
}

int main(int argc , char* argv []){
    int i;
    int arsize = MAXTHREADS < (argc - 1) ?
    MAXTHREADS : (argc - 1);
    unsigned short ports[arsize];
    pthread_t threads[arsize];

    for(i = 0; i < arsize; i++){
        ports[i] = (unsigned short)atoi(argv[i + 1]);
    }

    for(i = 0; i < arsize ; i++){
        pthread_create(&threads[i] ,
            0 ,start_listen ,
            &(ports[i]));
    }

    for(i = 0; i < arsize ; i++){
        pthread_join(threads[i] ,0);
    }

    return 0;
}

```

The client `trgen_client` was written in Ruby:

```

#!/usr/bin/ruby
require 'socket'

```

```

class MeasureClient
def initialize(aHost, aServerPort, aFilename)
    @host = aHost
    @port = aServerPort
    @filename = aFilename
end

def sendFile
puts "Tries to open #{@host}..."
begin
socket = TCPSocket.open(@host, @port)
    file = File.open(@filename, "r")
    socket.puts(File.size(@filename).to_s)
    puts "Sending file " +
    @filename + " on port #{@port}..."
    while true
        socket.putc(file.readchar)
    end
rescue EOFError
    file.close unless file.nil?
    socket.close unless socket.nil?
end
end
end

host = (ARGV[0] || 'localhost')
serverport = (ARGV[1] || 4711).to_i
filename = (ARGV[2] || 1).to_s

client = MeasureClient.new(host, serverport, filename)
client.sendFile

```



## Appendix D - IPv6 Background

In the beginning of the project, IPv6 [1] and the flowlabel [8] were investigated. This section was written during this period. Even if IPv6 was not used in the final implementation, the section may still be of interest to the reader.

### Historical review

The protocol used for packet switched traffic on the Internet today, dates back to 1978. It is called IPv4 and is the backbone of the Internet. By making computers throughout the whole world addressable, it has made techniques like the Web, Email and file sharing possible.

Ten years ago, people around the world stated that the IP addresses would run out. The choice of 32 bits addresses may have been good in 1978, but the address space was proving to be too short. The system with class A, B and C addresses, where address spaces of 24 bits were distributed to single companies (as happened when a company booked a class A series), did not turn out to be as flexible as needed. Even if 32 bits make 4.2 billions of hosts addressable, it was not enough when a great deal of addresses were already bound to certain universities and companies.

This problem was addressed with Classless Inter Domain Routing [19]. This scheme abandons the fixed class prefixes with prefixes with dynamical length. This gives us a number of levels of network sizes instead of three fixed which makes the network addressing more flexible. This way less addresses get tied up and unused.

Even if the limited address space was a solvable problem it was not the only one. Twenty years of Internet usage had taught the people of the Internet Society what was necessary in an Internet Protocol, and what was not. The IPv4 header had fields in it which were scarcely used, as well as an option field with variable length, which made the header harder to parse. Since any optional treatment of a packet makes the routing process slower, the router designers tended to streamline their routers for non optional packets. The code dealing with the few packets with options was not trimmed at all which imposed a performance penalty. The network programmers noticed this penalty, which led them not to use the option field, which in turn made the router designers even lesser motivated to trim the option handling code. Twenty years later hardly any one used the options offered by IPv4. If any

options were to be supported, in the new protocol, these options were not to be punished or, in turn, punish standard traffic.

One feature of IPv4, that in time turned out to be less than great was the automatic fragmentation of packets. Whenever a IPv4 packet, with size  $S$ , reached a link with a maximum transmittable unit (MTU) smaller than  $S$ , the router fragmented the packet automatically. This might seem like a good idea but that is not the case. A successful send of a packet requires all fragments to be delivered. If one gets dropped, the whole packet has to be resent, which leads to poor link performance and utilization.

Another issue was the header checksum, present in the IPv4 header. The checksum was to guarantee that the IP header was correct. Since most protocols using IP (i.e TCP and UDP) have their own checksum field, the risk of delivering an undetected, erroneous packet is minimal. If something goes wrong the packet may get delivered to the wrong host. This host will drop the packets since it does not know what to do with it. The higher level protocols will detect the missing packet and deal with it just like any dropped packet.

Features like security and authentication were getting focus in the Internet debate. It would be easier if these features were implemented in the IP level instead of in higher levels. If the new Internet Protocol (which early got the acronym IPng as in “Internet Protocol Next Generation”) also supported new techniques like QoS and flow management, development and usage of Internet applications would be even easier.

All these wishes and thoughts together formed a draft for a Request For Comments (RFC) by Christian Huitama[13] in 1992. Two years of disorder followed as the proposals of how the new protocol ought to be designed were many. One thing that the people of the Internet Community thought in common was that if a transition to the new protocol were to be feasible, the two protocols must be able to coexist. The most disputes were about what address system (size and grouping) and routing strategy to be used. In 1994 the IPng directorate published their recommendation, which suggested using a proposal called SIPP (Simple IP Plus) as the basis of the new Internet protocol. IPng became an evolvement of SIPP, and the Internet Community later renamed IPng to IPv6 which is the name used today.

## The design of the IPv6

To conclude what has been stated earlier, IPv6 was designed with the following aspects and thoughts in mind:

- Preserve the simplicity that made IPv4 successful.
- Devise the mechanisms for dealing with optional behavior so they not get in the way of the non optional ones.
- Assign a constant size to the IP header in order to make parsing and packet handling easier.
- Get rid of superfluous features, like link wise fragmentation and header checksums.
- Incorporate features that would gain usability by being implemented on IP level, such as security and encryption.

In order to fulfill these requirements, the header had to be redesigned. The only field left at the same position as in IPv4 is the version field composed by the first four bits. The rest are moved elsewhere, removed or renamed.

Even if the variable length property of the options field was unwanted, the ability to describe optional treatment was not. The functionality is there but solved in a more elegant manner, with *extension headers*. This means that wanted options, and information, is appended to the packet as extra headers between the original IP header and the data payload. Every header has a field called “Next header” describing if and what type of extension header that follows the current one. This makes parsing optional headers only needed at those routers where it is appropriate.

### The header format

The header format of the IPv6 header is shown in figure 9. First there is the version field. From the beginning IPv4 and IPv6 was supposed to be run on the same wires using the same encapsulation. The intention was that the communicating program would read the first 4 bits and decide what version of IP it was running. The evolution of IPv6 networks has shown us that this is not the case. Packets from IPv4 and IPv6 are very often de multiplexed at the media level, which means that the protocol distinction is made in the Ethernet header (or whatever hardware protocol used) instead.

Version 4	Class 8	Flowlabel 20		
Payload Length 16			Next Header 8	Hop Limit 8
Source Address 128				
Destination Address 128				

Figure 9: The IPv6 header with field sizes in bits

The second field is the class field. In the first version of the IPv6 RFC, the class field was called priority, and there were only 4 bits representing 16 levels of priority. The original idea was that this priority would give the real time mechanism enough granularity to express different types of traffic. As the research and experience on real time traffic gained ground, the designers abandoned the priority concept and enlarged the field to 8 bits (taking four bits from the Flowlabel field) calling it class. The flowlabel field gives the network programmer means to distinguish different flows through the network. The routers may use the flowlabel to set up cached routing entries or other optimizations. There are specifications[8] describing how the flowlabel is supposed to be interpreted.

In IPv4 there was a field called total length, which stated the size of the packet, IP header and payload added together. In IPv6 this field has changed name to payload length, because the header length of IPv6 is constant, therefore making it pointless to include it into the length. The next header field will be described more thoroughly in the next section, but in short it describes the type of the next appended extension header if any.

The hop limit has changed name and meaning from the time-to-live (TTL) in IPv4. Hop limit simply describes the maximum number of hops the packet is allowed to do, in order to prevent short circuit routes make stale packets get stuck in loops. The time-to-live concept did not explicitly decide how many hops the packet could do but the time the packet had to live before it was dropped. This time was calculated by taking theoretical analysis of transport protocols into account. The value never got anything but an approximation which is why the designers of IPv6 choose to give the field a more exact name and value.

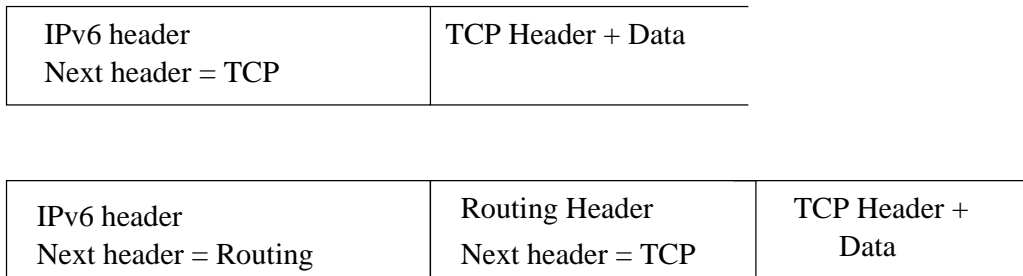


Figure 10: A chain of extension headers.

### The extension headers

The original idea with IPv6 was not only to make it a worthy successor to IPv4, but to keep it simple as well. In order to meet the demands of future techniques the designers had to think twice to match this simplicity with enhanced functionality. Experience told them that if functionality in any way punished standard traffic, it would soon be marginalized and eventually not used at all. This is why the *extension headers* were invented. Between the IP header and the payload, an arbitrary number of extension headers may be inserted. Every extension header has a type and adds a certain type of special case information to the packet. For instance, there are routing headers and fragmentation headers. In every extension header (as well as the original IP header) there is a field called “next header” which tells the router how to interpret the next header, in a linked list fashion as shown in figure 10. The major advantage of having extension headers arranged this way is that options of no interest may easily be skipped by starting reading the payload after the last header which next header field says “TCP” (or whatever protocol carrying the payload). By turning the options field into a variable number of extension headers, IPv6 provides a more structured framework for dealing with optional data.

The current IPv6 specification defines six extension headers:

- The *Hop-by-hop options header* is used to pass additional information to all routers in the path, mostly for debugging or management. Only routers wanting the information inspects the Hop-by-hop header, and the information may be anything.
- The *routing header* gives a framework for source routing a packet. The routing header carries a list of intermediate addresses through which

the packet will travel.

- The *fragment header* adds fragmentation functionality to the protocol. This fragmentation is not, as in IPv4, link wise but of source to destination type which means that the intermediate nodes do not know they are sending fragmented packets. Every fragment header carries an id, which makes reassembling of the packet at the destination possible.
- The *authentication header* makes, as the name implies, authentication at IP level feasible.
- The *encrypted security payload* is used to encrypt the payload, in order to offer safe connections between two hosts.
- The *destination options header* is a general header for passing options to the destination node. It is of the same type as the Hop-by-hop header, but with the difference that intermediate routers will not inspect it.

### The IPv6 stack in Linux

During the work with analyzing the IPv6 stack, the figure 11 was made. In this figure we see the name of the functions, the hooks used by **iptables** and where packets enter and leave the IPv6 stack. The labels starting with *ip6* are the name of the functions. The positions where there are two lines labeled with capital letters show where the **iptables** hooks are situated. At a hook an **iptables** chain may be attached. Examples of hooks are PRE\_ROUTING or FORWARD. The squares show the entrances and exits of the IPv6 stack. At the top, there is INPUT, which is where all packets coming from the net, this host is connected to, enter the stack. The two SOCKETHANDLER squares show where packets, enter and exit user space. That is, where packets that are destined for this host exit the stack or packets sent from this host enter it. The squares TRASH show where the packet gets dropped if the host decides to discard it. The OUTPUT square denotes where packets exit the stack to enter the net.

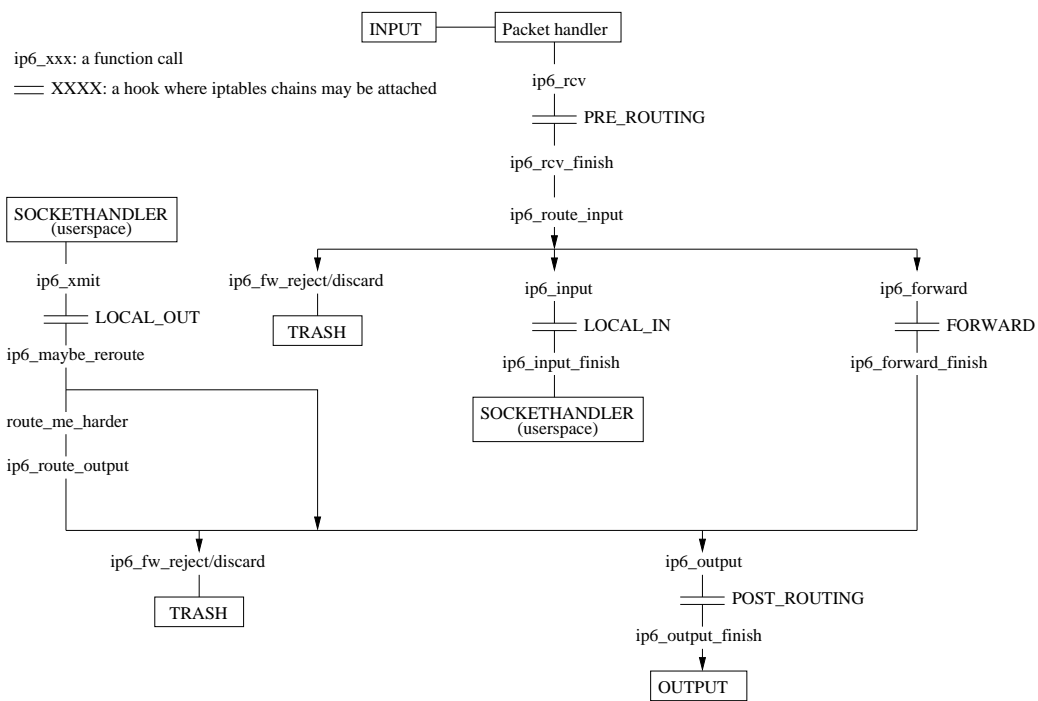


Figure 11: Overview of the IPv6 stack in the Linux kernel.

## Appendix E - Dialog on the TBF timing problems and UML

This is the mail that was sent to the networking development mailing list (netdev@oss.sgi.com):

Hello!

I use the TBF filter with good results on my computer. When I **set** a rate with the filter the traffic get shaped accordingly.

I have **set** up a couple of UML instances (user-mode-linux) as a network to test different QoS strategies. My problem is that the TBF calculates the wrong rates. An UML instance may get interrupted just like any other program. Does the timing code in TBF presume that the kernel wont be interrupted, and therefore generates the wrong rates?

Regards,

Gabriel Paues

This is the answer that Jamal Hadi (hadi@cyberus.ca) sent:

UML will never work well. The issue is related to the timer effects in UML. I believe its more of a clock skew and inaccuracy than TBF making assumptions.

cheers,  
jamal



## Appendix F - The test1.cmd file

This file was used in the capacity test as input to **DBS** to generate the traffic. For a description of the file format please refer to section 5 and [21].

```
{
#Define the sender.
sender {
hostname  = borrow; # The sending host
port      = 4711; # The sending port
mem_align = 2048;
# The send pattern, which says that we will send
# messages 1024 bytes wide every 0.01 second.
pattern {1024, 1024, 0.01, 0.0}
}
receiver {
hostname  = pets;
port      = 4711;
mem_align = 2048;
# The receiving buffer, set to very large
# to ensure that all messages will be received
pattern {10000, 4096, 0.0, 0.0}
}
file      = data/test1_udp_4711;
protocol  = UDP;
#Start sending at the time 0.0 s
start_time = 0.0;
#Stop sending at the time 30.0 s
end_time   = 30.0;
}
{
sender {
hostname  = borrow;
port      = 4712;
mem_align = 2048;
pattern {1024, 1024, 0.01, 0.0}
}
receiver {
hostname  = pets;
port      = 4712;
mem_align = 2048;
}
```

```

pattern {10000, 4096, 0.0, 0.0}
}
file          = data/test1_udp_4712;
protocol      = UDP;
start_time    = 10.0;
end_time      = 35.0;
}
{
sender {
hostname  = borrow;
port      = 4713;
mem_align = 2048;
pattern  {1024,1024,0.01,0.0}
{
hostname  = pets;
port      = 4713;
mem_align = 2048;
pattern  {10000,4096,0.0,0.0}
}
file          = data/test1_udp_4713;
protocol      = UDP;
start_time    = 20.0;
end_time      = 35.0;
}
}

```

## Appendix E - Description on how to write the flowlabel match module for iptables

These are the steps that have to be done to write a new iptables module. This example describes how to write a flowlabel match module, and other modules are made in a similar way. The things that needs to be done are:

- Write a kernel module that defines how the module will match on the flowlabel in the IPv6 header.
- Write a user-mode **iptables** library that defines how the command line should be interpreted in order to setup the module with the **iptables** user-mode tool.

To write the kernel-module an existing **iptables** module was used as starting point: `ip6t_mark`. All commands are run from the root of the Linux kernel source code tree using the BASH shell:

```
> cp net/ipv6/netfilter/ip6t_mark.c \
net/ipv6/netfilter/ip6t_flowlabel.c
> cp include/linux/netfilter_ipv6/ip6t_mark.h \
include/linux/netfilter_ipv6/ip6t_flowlabel.h
```

Then we have to alter the Makefile and the Config.in in `net/ipv6/netfilter`. Under the line

```
obj-$(CONFIG_IP6_NF_MATCH_MARK) += ip6t_mark.o
```

in `net/ipv6/netfilter/Makefile` add the line:

```
obj-$(CONFIG_IP6_NF_MATCH_FLOWLABEL) += ip6t_flowlabel.o
```

In `net/ipv6/netfilter/Config.in` search for the line:

```
dep_tristate ' netfilter MARK match \
support ' CONFIG_IP6_NF_MATCH_MARK $CONFIG_IP6_NF_IPTABLES
```

and add the line

```
dep_tristate ' netfilter FLOWLABEL \
match support ' CONFIG_IP6_NF_MATCH_FLOWLABEL \
$CONFIG_IP6_NF_IPTABLES
```

underneath. Now we have to alter the files `net/ipv6/netfilter/ip6t_flowlabel.c` and `include/linux/netfilter_ipv6/ip6t_flowlabel.h` to suit our needs. First go

to include/linux/netfilter\_ipv6/ip6t\_flowlabel.h and do a search and replace from “mark” to “flowlabel” in it. Then you go to the file ip6t\_flowlabel.c in net/ipv6/netfilter/ and do a search and replace there as well. Here we need to fill in the code that actually matches the flowlabel in the IPv6 header. The function “match” should be changed from this:

```

static int
match(const struct sk_buff *skb ,
      const struct net_device *in ,
      const struct net_device *out ,
      const void *matchinfo ,
      int offset ,
      const void *hdr ,
      u_int16_t datalen ,
      int *hotdrop)
{
    const struct ip6t_mark_info
    *info = matchinfo;

    return ((skb->nfmark & info->mask)
    == info->mark) ^ info->invert;
}

```

to this:

```

static int
match(const struct sk_buff *skb ,
      const struct net_device *in ,
      const struct net_device *out ,
      const void *matchinfo ,
      int offset ,
      const void *hdr ,
      u_int16_t datalen ,
      int *hotdrop)
{
    const struct ip6t_flowlabel_info
    *info = matchinfo;
    u32 * flowlbl;
    u32 tmp, tmp2;
    flowlbl = (u32*)(skb->nh.ipv6h);
    tmp = ntohl(*flowlbl);
    tmp2 = tmp & 0x000FFFFFF;
}

```

```

    return ((tmp2 & info->mask)
    == info->flowlabel)
    ^ info->invert;
}

```

This is what needs to be done to make the kernel module. Now we have to make a user-mode library to help **iptables** parse the command line when we setup our module. This is done by getting the source code to **iptables**. All the following commands are run in the root of the **iptables** source code tree using the BASH shell. First we copy the mark modules library file to use as template:

```

> cp extensions/libip6t_mark.c \
extensions/libip6t_flowlabel.c

```

and do a search and replace from “mark” to “flowlabel” in extensions/libip6t\_flowlabel.c. The only thing left to do is to edit the extensions/Makefile and change the line:

```
PF6_EXT_SLIB:=tcp udp icmpv6 standard MARK mark
```

to

```
PF6_EXT_SLIB:=tcp udp icmpv6 standard MARK mark flowlabel
```

Then you run:

```
> make KERNELDIR=<where_your_kernel_is>
```

to compile the match module. Then you have to copy the library extensions/libip6t\_flowlabel.so where the **iptables** module libraries are. On Red Hat this is in the directory /lib/iptables. You also have to copy the **iptables** and **ip6tables** to the correct places in your path:

```

> cp iptables 'which iptables'
> cp ip6tables 'which ip6tables'

```

Now you should be able to add a **iptables** rule with the flowlabel like this:

```

> ip6tables -t mangle -A PREROUTING \
-m flowlabel --value 2 -j LOG

```

to log all incoming packets with the flowlabel set to 2. All the files are found in the environment directory under in the iptables-1.2.5 directory as well as the linux-2.4.18-47 directory.