Testbed Evaluation of Integrating Ethernet Switches in the Differentiated Services Architecture using Virtual LANs

A Thesis
Presented to
The Academic Faculty
by

Antony Fornaro

In Partial Fulfillment
of the Requirements for the Degree
Master of Science in the
Electrical & Computer Engineering
Department

Georgia Institute of Technology
January 2003

Testbed Evaluation of
Integrating Ethernet Switches in the Differentiated Services
Architecture using Virtual LANs

Acknowledgments

I would first like to express my deep gratitude to my advisor, Dr. Henry Owen. He kindly welcomed me into his research team, trusted in my work and me and was always willing to help me. I would also like to thank Dr. Emmanouil Tentzeris and Dr. Randal Abler for being on my proposal and thesis committees. Special thanks to Dr. Tentzeris for his moral support during my time at Georgia Tech.

I would also like to thank my colleagues in the ECE program and my lab partners. They all have been inspiring and our various technical discussions have been helpful and insightful. I would like to thank especially Harris Schneiderman, Nicholas Athanasiades, and Fontas Dimitropoulos, who have all been an excellent source of strength and intellectual support.

This research work would not be possible without the support of Intracom S.A., which funded my graduate studies. I would like to thank all the people I have worked with during my employment at Intracom S.A, especially Nikiforos Korres, who greatly supported me and believed in me. Special thanks also goes to Stefanos Hadjiagapis, whose personal interest helped me to start my program and end it with this thesis.

Finally, none of this would have been possible without the love and support of my family. I owe much of who I am to them and I would like to thank them for all they have given me in the years away from home and before.

Contents

List of Tables

List of Figures

Summary

This thesis is motivated by the need for QoS in the current Internet. There are strong commercial reasons for network operators and equipment providers to offer QoS differentiation in IP networks. The main driving force so far has been the need for support of voice traffic over IP. Existing and emerging multimedia applications and mission critical programs follow. DiffServ, for some years now, has been –and will continue to be– the most promising framework for accomplishing commercial QoS because of its maturity and scalability. However, several issues arise when trying to implement the architecture in the real world. Our work involves one of these issues: the co-existence of DiffServ with Ethernet devices. We have built a Linux testbed to study the impact of an Ethernet switch inside a DiffServ domain. The results of our experiments show that Ethernet switches should implement the same per-hop behaviors as the routers of a domain and thus should be incorporated in the DiffServ model. Otherwise, the domain will fail to provide the traffic services it intends to offer. A framework is needed to describe how Ethernet switches should make use of DiffServ mechanisms. We propose a framework that is based on the use of VLANs. In our model, customers are grouped into VLANs and a traffic service is assigned to each VLAN. In this way, the layer-3 aggregative semantics of DiffServ are extended to layer 2 by the aggregative semantics of VLANs. The suggested framework seamlessly integrates Ethernet switches into the DiffServ architecture and provides a strong business model for QoS differentiation.

Chapter 1

1 Introduction

The objective of this work is to design and build an Internet domain that provides different traffic services that engineers are ready to implement, administrators are able to manage, and customers are willing to pay for**.** That is, we intend to build an Internet network that provides quality of service (QoS) for applications and users that need it, in a simple and straightforward way that is both feasible to engineer and easy to administer, and at the same time is possible to scale at the size of the Internet. In reality, if not all of these conditions hold, the evolution of QoS in the Internet will not be possible.

Our work was initiated by studying the history of QoS in the Internet. This study concludes that the differentiated services architecture (DiffServ) is now the most promising and feasible approach: the standards that define it are mature enough, there are many algorithms that can realize its most challenging components, the Internet community has conducted much research on it, and there are many public and proprietary implementations and testbeds already developed.

DiffServ was introduced as a feasible and adequate infrastructure to provide some kind of quality-of-service support in the current Internet. DiffServ enhances the Internet by adding more traffic services to the best-effort service. The best-effort service is expected to remain the dominant service, where enhanced services will take only a small fraction of the total traffic but will provide guarantees that applications and users will benefit from. The goal of DiffServ is twofold: to provide some kind of end-to-end QoS and to achieve it in a scalable way where deployment will be fast and will grow as much as the best-effort service has.

Moving from the current best-effort Internet to an Internet enhanced with more traffic services under the differentiated services architecture is a big step that is not without challenges, which is why DiffServ has not yet been commercially deployed. Several problems have restrained the deployment of DiffServ. Both engineering and administrative complexity are two main reasons. Even though the work of the last four years has led to many engineering evolutions, it is essential that a DiffServ framework be simple to administer and straightforward to commercialize. Moreover, for DiffServ to be successful, it has to be applicable to the existing networks. The Internet is comprised of many interconnected heterogeneous networks, consisting of devices that function in both layer 2 and layer 3. DiffServ does not take into account Ethernet switches or any other layer-2 devices, since it is defined for the IP layer protocol.

Intuitively, one can see that if the Ethernet switches inside a DiffServ domain do not support differentiated traffic service, there might be many scenarios where the anticipated performances do not hold. Traffic through an Ethernet switch should be conditioned, metered, and scheduled in an appropriate way for the domain to provide the guarantees it is supposed to. IEEE has published the 802.1p standard, which along with 802.1Q defines an extra field in the Ethernet frame format and several Ethernet traffic classes that can be used to differentiate traffic in layer 2. This has been done in an effort to provide QoS in the Ethernet layer, but is not by any means integrated in the DiffServ model.

We have created a DiffServ domain on a Linux testbed to study the effects of an Ethernet switch on the performance of the traffic services the domain offers. We will show how DiffServ becomes impossible when there is no QoS support in the switch and also when the switch employs IEEE 802.1p traffic conditioning. Furthermore, we will demonstrate the simplicity of integrating the switch in the differentiated services architecture by having it employ the same traffic control mechanisms as the DiffServ router. We will also show that a model for doing that is needed, and one will be proposed. The model proposed is based on the use of virtual LANs (VLANs) for grouping users / customers with similar QoS needs. It aims to make a clear and simple target market for differentiated services and to integrate seamlessly the layer-2 devices of a domain in the DiffServ framework.

At this point, a comment should be made about bridging and switching. Technically, a "bridge" is a device that interconnects networks of different physical mediums. It may or may not perform switching. In practice, though, there is no distinction between the terms "bridge" and "switch," which are used interchangeably in the literature and throughout this document. IEEE, for example, has adopted the term "transparent bridge" to describe what some people might refer to as an Ethernet switch.

The next chapter summarizes the research on the history of QoS in the Internet for the purposes of this work. It describes the major issues involved with QoS in the Internet, identifies the solutions that have been proposed over recent years, and explains the advantages and disadvantages of the differentiated services architecture. This is followed by a description of how the framework is defined and how it is designed to work. Chapter 3 discusses the testbed that has been built, provides an overview of the basic Linux traffic control components, and describes the services these components have built up in our testbed. The experiments performed on the testbed, which will analyze the impact of the Ethernet switch in the domain, are presented in Chapter 4, and Chapter 5 discusses the suggested framework for integrating layer-2 QoS in the differentiated services architecture using VLANs. Finally, Chapter 6 concludes the thesis. Appendix A provides the details on the software provisioning of our testbed, Appendix B describes how VLAN switching can be implemented in Linux, and Appendix C lists and describes the traffic control scripts used in our work. Appendix D discusses the automation procedure for performing experiments and lists the corresponding scripts.

Chapter 2

2 Background

During the last two decades, the Internet has experienced amazing growth. Millions of hosts and thousands of networks have been interconnected to form a global network where communication is cheap, fast, and simple. It has been the simplicity and scalability of the Internet protocol (IP) and other Internet-related protocols that have made this exponential growth possible. The end-to-end argument [1], which dictates that all complexity has to be pushed into the network edges and that the core network has to be kept as simple as possible, has been vastly adopted by the Internet community and has made the global deployment of the Internet a reality.

The success of the end-to-end argument does come with a price, however. This model makes it extremely difficult to deploy new network and traffic services, services that require support from the core network. The core network functionality has been designed to be fast and optimized, greatly lacking the ability for changing and upgrading. Many network services have been envisioned that would make new applications a reality, but have stumbled with the current Internet model: for a new network service to be deployed, the entire core network has to be upgraded. Things are even harder when the new service introduced is sophisticated or ambiguous. Experimental services are virtually impossible, which is also why the active networks model has been envisioned [2].

Broadcasting, multicasting, explicit congestion notification, quality of service, distributed computing, and virtual-line emulation are some of the main examples of network services that have been greatly studied, but have not yet been realized because of the need for huge core network upgrades. It is possible, however, that this upgrade will be realized for a network service that is worth deploying. This service should be well defined, engineering mature, easily administrated, and, most important, should have a clear target market that would compensate for the cost of deployment.

Quality of service has been a big chapter in the book of new Internet services. As soon as the global data network (which the Internet has become) became a reality, the telecommunications community dreamed of the integration of voice network (telephony) and data network (Internet). Voice is a real-time application that has to meet several deadlines and requires several guarantees that the best-effort packet delivery service of the current Internet cannot support. Moreover, there are several real-time applications that could be deployed in the Internet such as video conferencing, video on demand, or time-sensitive applications such as medical and military applications. All of these new applications have motivated university researchers, industry, and the Internet community to look for models that would enhance the Internet with quality-of-service support.

The Internet Engineering Task Force (IETF) has played a primary role in the effort to build a mechanism that would provide quality of service for the Internet. In the early `90s, the Integrated Services Working Group was developed, and the Integrated Services Standard was issued, which along with the Resource ReSerVation Protocol (RSVP), defined a model where an application can issue requests for specific traffic guarantees it requires from the network. The model's components ensure that the requirements will be satisfied, making the corresponding reservations along the path of the application's connection. In this way, the network provides every connection the service it requires and every application can use any kind of connection it desires, which leads to fine-grained QoS for network applications.

Integrated services and RSVP have been described as the most complicated network protocols. However, complexity is not the main disadvantage of this approach. The primary problem with integrated services is scalability. It was soon acknowledged that there are too many connections for a router to keep state for. This approach was largely contrary to the end-to-end argument that pushes complexity and state information to the edges. The need for a scalable solution to the quality-of-service problem was embraced by the IETF, which brought together the Differentiated Services Working Group [5]. The group published in 1998 the differentiated services architecture [6], which became an Internet standard along with many more Internet drafts and RFCs that supported the newly introduced model [8], [9].

This new scheme was embraced by the Internet community, even though it did not guarantee micro-flow QoS, since it promised flexibility and scalability. The Differentiated Services Working Group put all its efforts into designing a standard that would set the guidelines for QoS in the Internet, but that would still leave an open field for work and study. That is why the architecture is very open, modular, and extensible. As a matter of fact, the group has been accused of having held back the DiffServ deployment by being too liberal [27].

In the differentiated services architecture, traffic is monitored and classified in the edge routers. An edge router is the first router a packet will go through along its path to its destination. Classification results in a service indication marked in the packet in the IP header. In particular, the former TOS field of the IPv4 header and the former traffic class field of the IPv6 header are used; both have been renamed to “DS field.” Consecutive routers along the path of a packet will use this service indication to assign a prearranged hop behavior to that packet. This per-hop behavior, along with the rules and policies set in a DiffServ domain, defines the service that a particular flow will receive.

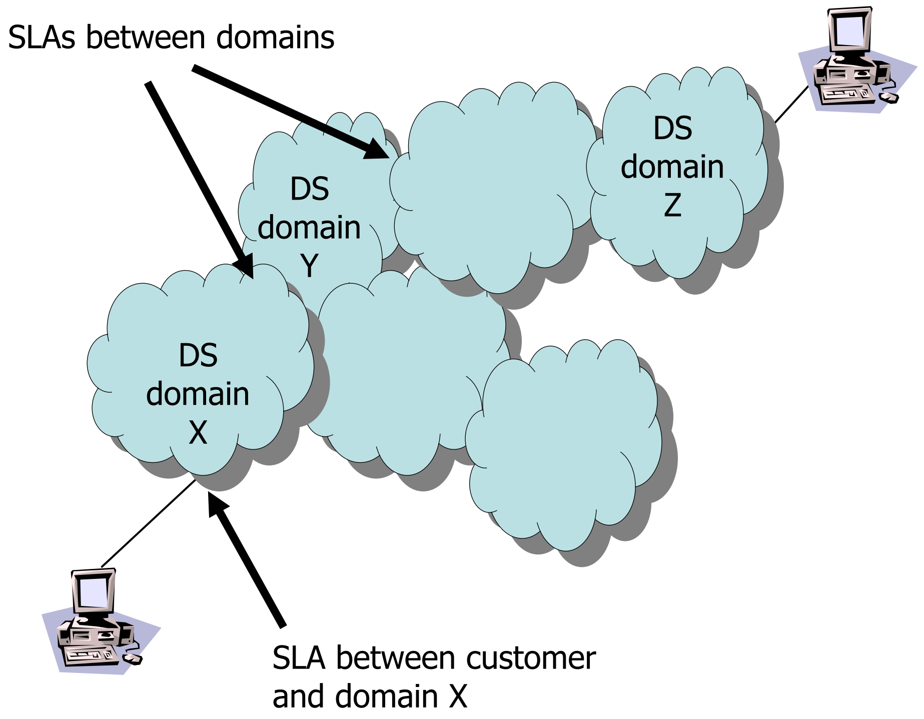

Figure 2.1. DiffServ domains and SLAs.

A contiguous DiffServ region (a network where there is support for differentiated services) is divided into DiffServ domains. A DiffServ domain is a network with a single autonomous administration. The administration is responsible for implementing and managing the differentiated services architecture inside the domain and assumes the task of providing traffic services along with setting rules and policies. A DiffServ domain authority has to make bilateral agreements, which are called service-level agreements (SLA), with the authorities of adjacent domains to extend traffic services over the domains' boundaries. The division in DiffServ domains is essential for the gradual deployment of differentiated services in the Internet and the agreements are essential for the eventual end-to-end QoS support.

The differentiated services architecture does not define any traffic services. It only describes how various components will work together to provide the services. One major component in this framework is the per-hop behaviors (PHBs). A PHB describes what a router will do with a packet. The PHBs have to be standardized and the IETF has already standardized a few of them. However, the standardization does not restrict or describe the implementation of these behaviors. There are many published algorithms that may be used in different ways to provide the same functionality and it is up to the manufacturer to decide which to use.

PHBs are used as the building blocks to create traffic services, and since IETF does not define any traffic services, academia and industry do so. Four traffic services are of interest in this work: premium, assured, olympic, and best effort. All are based on standardized PHBs and are probably the most popular DiffServ traffic services in the research community. The next chapter presents these services in detail. Finally, it is worth mentioning one more interesting and promising service, the alternative best-effort (ABE) service [13]. It is a best-effort service that requires no additional charging or usage control where applications choose between receiving a lower end-to-end delay and receiving more overall throughput.

Recapitulating, in the differentiated services architecture, administrations of DiffServ domains offer different traffic services to the users of their domains and have to make agreements with adjacent domains to extend these services. The traffic service could be assigned to a user’s host computer, to a particular connection, to a set of computers, to a LAN, or to anything else that could make sense. Differentiated services is a challenging framework to implement and deploy. Furthermore, the definition of the traffic services offered inside this framework is even more challenging because it is the value and the importance of the service with respect to the view of the end user that will assess the whole architecture. DiffServ is still a work in progress that is constantly being evaluated [16], [17], [18] and ratified [9].

Chapter 3

3 Testbed

3.1 Overview

The role of the testbed is threefold. First, it is used to illustrate the troubling issues with the introduction of Ethernet switches into a DiffServ network and at the same time to present the simplicity of the resolution given to the problem. Second, it implements and presents the suggested framework for the formal integration of the Ethernet switches in the DiffServ model, proving its feasibility. Third, its realization constitutes the primary source of understanding, motivation, and insight for our work.

The testbed consists of six identical personal computers: Pentium III at 450MHz with 256MB of physical memory and 6.8 GB removable hard drives. All network cards are Intel's eepro100 and all machines run RedHat 7.3, based on Linux-2.4.18- kernel. The kernels of the machines that act as the switch and router were recompiled to add additional features and change configuration. All links are forced at 10Mbps half duplex and all systems run an *ssh* server. The sources and the sink are provisioned with the altered *iperf* program, which we will discuss later on. The *vconfig* program is installed on both the switch and the router to control the VLAN tagging support, whereas the *bridge_utils* package is installed only on the switch and controls the bridging kernel functions. The *iproute2* package, which contains the *tc* program that controls the kernel traffic control mechanisms, is installed on both machines as well, along with the *tc* patches taken from the *tcng* package. Finally, two patches for kernel files need to be applied.

As mentioned above, the switch and the router kernels need to be recompiled. The appropriate selections in the Linux kernel configuration dialog need to be made to include traffic control and DiffServ capabilities. Details about what has been mentioned so far and of what will follow can be found in Appendix A, which also discusses the modifications made to the *iperf* program. The *iperf* program is essential to our testbed. It runs in two modes, a client and a server. A client generates a UDP flow and the server reports statistics for the received traffic on that flow. The program had to be slightly modified to accommodate an automated procedure of running experiments and collecting data.

In the next chapter, we show graphs of measurements that show throughput and packet loss percentage over values of generated traffic. These graphs are created with the integration of the measurements made for different values of generated traffic. In particular, measurements were made in steps of 0.5 Mbps of generated traffic. The number of steps in a graph may be up to nine, and every step lasts as long as 30 seconds. The measurement made in a step is the average throughput, or packet loss percentage, over a timescale of 30 seconds. Throughput and packet loss do not yield different values over time for steady generated traffic. This is expected; traffic service behavior of the domain is not supposed to change over time. The window of 30 seconds is used to accommodate errors injected by nonabsolute synchronous generation of traffic. The copious number of experiments and cases that needed to be examined in this work demanded automation of this procedure. The automation made it possible to run a single experiment in a keystroke, where the results are presented both on the screen on the fly using *xgraph* and in *excel* files for further processing. Appendix D provides more information about the automation procedure, which is based on the *expect* package –a *tcl* interpreter, along with the scripts created for our work

The next paragraph talks about the DiffServ mechanisms inside the Linux kernel and gives an overview of the most important elements. The last paragraph of this chapter describes the traffic services built by these mechanisms.

3.2 DiffServ in Linux

Linux kernels offer a variety of traffic control mechanisms that can be used in a modular way to implement service differentiation in accordance with the DiffServ model. Several traffic control elements may be used together to provide the desired functionality. Some of these elements are specific to the differentiated services architecture, but most of them can be used in many other contexts. This section gives a brief overview of the Linux traffic control structure and its elements. How these mechanisms were used to provide DiffServ functionality for our experiments is discussed in the next chapter, and the scripts that configure this functionality can be found in Appendix C. General information about traffic control in Linux can be found in [22] and [23], whereas DiffServ-specific information about Linux can be found in [24] and [25].

Figure 3.1 shows the major conceptual blocks that comprise the path a packet follows inside the Linux kernel. Incoming packets are either passed to a higher layer to be processed by the local host, or they are handed out to the “Forwarding” block. The main role of this block is to select an output network interface to transmit the packet. Upon decision, the packet is queued to the respective interface. This is where traffic control kicks off. Even though traffic control functions can and actually are sometimes implemented in the ingress interface, there are not many things one can do with traffic at the ingress interface, mainly marking and policing. On the other hand, traffic control at the egress interface may form all kinds of desired behavior for traffic. It all depends on the queuing scheme used in the output interface. When the “Forwarding” block queues a packet on an interface, the queuing scheme may decide to put the packet at the end or at the beginning of a FIFO queue, drop it, or just do anything else meaningful.

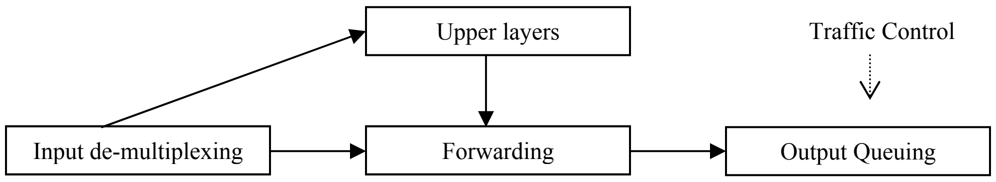

Figure 3.1. Linux kernel path for packets.

Linux provides a modular structure to use different queuing schemes that offers great flexibility and control over traffic [23]. Furthermore, many published algorithms for traffic control have been implemented for Linux and are part of recent Linux kernels. In Linux, a queuing scheme is named a *qdisc* (queuing discipline) and, along with the classifiers, the filters and the meters (for policing) provide all the necessary functionality for implementing differentiated services.

Each network interface (network device) has a *qdisc* associated with it, called the root *qdisc*. The simplest *qdisc* available is a FIFO queuing scheme where packets originating from the forwarding process enter the tail of the queue and the device transmits packets from the head of the queue as fast as it can. More sophisticated *qdiscs* have classes (classful). Classes are used for service differentiation, where any packet can be assigned to any class and packets from different classes can receive different services. The mapping of packets to classes is done by the use of filters and the service differentiation is done by assigning different queuing disciplines to classes or by assigning different priorities to classes. This is probably the most powerful feature of the traffic control in Linux kernels: a class can be associated with any available *qdisc* that in turn can consist of more classes, which may also have *qdiscs* associated with them and so on. An example of this functionality is shown in the following figure, where only the outer *qdisc* contains classes; the inner are classless qdiscs.

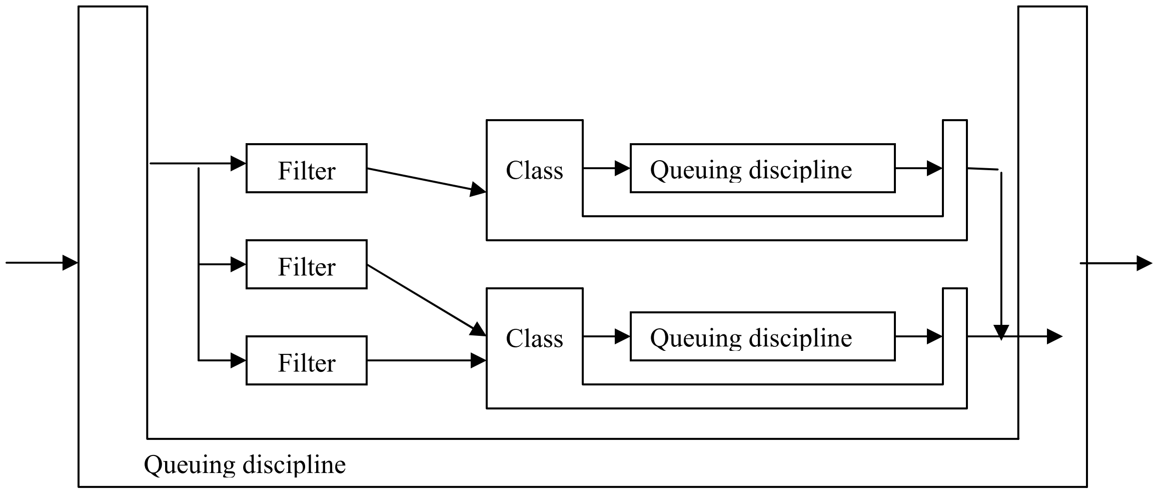

Figure 3.2. A simple queuing discipline with multiple classes.

When the forwarding process hands out a packet to an interface, it performs the *enqueue* function on it. The outer (root) *qdisc* is responsible for figuring out what class the packet belongs to (if any) and handing the packet out to the *qdisc* that will hold it.

When the device is ready to send a packet, it performs the *dequeue* operation on the outer *qdisc*. The outer *qdisc* needs to decide on which of its inner *qdiscs* it will perform the *dequeue* operation. Finally, a *qdisc* is chosen and a packet in that *qdisc* is selected and then transmitted over the link.

It is clear that using this structure and having at one's disposal several queuing disciplines, classes, and filters, it is possible to create many different traffic control schemes. DiffServ is one of them and there is indeed more than one way to combine and use the available components to implement the same desired behavior. In the following section, the major components used in our testbed are briefly described.

- *dsmark qdisc*. This is the only qdisc specific to the differentiated services architecture [25]. It is used only as a root qdisc and its role is twofold. First, it extracts the DSCP from the packet and records it in the packet's buffer descriptor in the tcindex field. This is later examined by the tcindex classifier. Second, it sets the DS field of outgoing packets. This marking is done when a packet is dequeued from the dsmark qdisc; the qdisc uses the tcindex field of the packet's buffer descriptor as an index to a table in which the outbound DSCP is stored and puts this value into the packet's DS field.

- *tcindex classifier.* A classifier, in general, is used to classify a packet to a certain class of a given *qdisc*, making use of filters. The filters can use almost any field of the packet to decide on what the classification result will be. The *tcindex* is different from regular classifiers, such as the *rsvp* and the *u32,* in that it uses the *tcindex* field of the buffer descriptor of the packet to make the classification. Also, the *tcindex* classifier may make use of a meter to police traffic.

- *CBQ qdisc.* Class-based queuing is a classful *qdisc* that implements a rich link-sharing hierarchy of classes [19]. The desired result is the sharing of available bandwidth by different classes of traffic, where each class gets a predefined proportion of the available bandwidth. This definition suggests that classes might contain more classes, where the bandwidth reserved for the parent class could be shared between its children classes. The way this is implemented in Linux is as follows: when a packet is enqueued in a CBQ *qdisc*, the filters associated with the root class of the *qdisc* is used to assign the packet to a class. Then, the packet is enqueued to that class, which may contain any available *qdisc*. If that *qdisc* is again a CBQ *qdisc*, the process starts over until the packet is enqueued to a leaf class, a class that has no children. When the device is ready to send a packet, the dequeue operation is performed on the root CBQ *qdisc*. The *qdisc* inquires its classes to give a packet for transmission. This is done in a weighted round robin fashion, where classes are assigned priorities and weights. The WRR process starts by asking the highest priority classes for packets and continues to do so until they have no more data to offer, in which case the process repeats for lower priorities. Each class, when asked by the WRR process, decides on whether it should give a packet for transmission or not. This depends on whether or not it exceeds its assigned proportion of the bandwidth. The algorithm calculates this by measuring the effective idle time of the link and in particular using an exponential weighted moving average (EWMA), which considers recent packets to be exponentially more important than past ones. When shaping a 10 Mbps link to 1 Mbps, the link will be idle 90% of the time. If it is not, it needs to be throttled so that it is idle 90% of the time. Based on that principle, each class compares its calculated idle time to the EWMA measured one and decides whether or not to send a packet.

- *sch_gred qdisc.* This *qdisc* was created to support the assured forwarding PHB for the differentiated services architecture; nevertheless its scope and use is not limited to that. It is a multi-RED implementation called generalized RED (GRED). Cisco's DWRED and RIO are special cases of GRED. It employs sixteen (16) virtual queues (VQs) and uses the last four digits of the *tcindex* field of the buffer descriptor of the packet to assign the packet to the right VQ. A distinct RED algorithm is employed for every VQ separately, and packets that have entered a VQ will either enter the physical queue or get dropped, as the RED algorithm of their VQ will decide. The GRED *qdisc* is a very flexible mechanism, since it allows the user to manually configure the RED parameters for every VQ or may even generate these values automatically by having the user define different priorities among the VQs.

- *prio qdisc.* This *qdisc* is used for strict prioritizing. It consists of classes, called bands, and a priority is assigned to every band. The *qdisc* makes sure that bands with higher priority will always be served first and bands with lower priorities will be served only when higher ones do not have any packets available.

- *ingress qdisc*. As the name implies, this *qdisc* is used for ingress interfaces. It is a very specialized *qdisc* and acts as a template for holding filters and policing mechanisms (*policers)*.

3.3 Traffic services

The mechanisms presented in the last paragraph constitute building blocks useful for creating a number of traffic services. For the purposes of our work, we use these mechanisms to provision the testbed with four traffic services: premium, assured, olympic, and best-effort service. In the rest of this chapter, the services are described. The next chapter discusses the use of the services and shows results. Appendix C presents the scripts used to set up the services in a Linux machine.

Best-effort service is the one the Internet currently provides. In this model, packets compete equally for network resources; the network provides no guarantees and thus no predictability or reliability in end-to-end packet delivery. Premium, assured, and olympic service all provide service guarantees that applications or users may benefit from. They are the services most commonly studied for the differentiated services architecture. These services are not strictly defined and may have many variations. Nevertheless, the basic building blocks that implement the services, the per-hop-behaviors (PHBs), are well defined and standardized. These PHBs constitute the core of DiffServ functionality inside the domain. The experiments in the next chapter show that only when every network node (i.e., switch and router) inside a DiffServ domain implements the same PHBs will a service based in these blocks be properly offered end to end (throughout the boundaries of the domain). There may be other services, based on different PHBs, that do not have this requirement. It will be shown, however, that for at least the services discussed in this work, which are the more mature and popular among the proposed DiffServ services, the suggested argument does hold.

Premium service was introduced by Van Jacobson in [7] and is also known as a virtual leased line. It is a peak-limited, extremely low-delay service, resembling a leased line. Traffic belonging to it is classified at the network edge, where it is shaped or policed at a preconfigured peak rate and is given its own high-priority queue in routers. Premium service is strictly not oversubscribed. Typically, mission and time-critical applications require this service. It is expected that in real networks, only a small percentage of the total network capacity would be allocated to the premium service.

The mechanism that realizes this service is the expedited forwarding (EF) PHB. Expedited forwarding provides a basic building block for low-loss, low-delay, and low-jitter services and is based on the idea that, since propagation delays are a fixed property of the topology, delay and jitter are minimized when queuing delays are minimized. The definition of the behavior is that the rate at which EF traffic is served at a given output interface should be at least the configured rate, over a suitably defined interval, independent of the offered load of non-EF traffic to that interface. EF was first defined in [10], which failed to formalize the definition in a correct mathematical manner and thus was replaced by [11]; [12] describes in further detail the reasons for that.

There is a category of services, called better-than-best-effort services. This consists of services aimed at applications that require a better reliability than the best-effort service, without having as strict requirements as those the premium service satisfies. The first such service that appeared is the assured service, which provides an assured subscribed bandwidth for its traffic. Traffic that does not exceed this rate is serviced with a high level of assurance. Traffic that exceeds this limit, however, is serviced with a lower level of assurance.

Assured service was first proposed in [13] by Clark. In this model, a service profile is defined for every user and in case of congestion, the network favors traffic that is within those service profiles. The traffic is monitored as it enters the network and the edge node tags packets as being “in” or “out” of their service profile. Then, at each router, if congestion occurs, preferentially, traffic that is tagged as “out” is dropped. This model may be implemented by a RIO (RED In Out) scheme [21]. RIO is based on the random early detect (RED) [20] differentiated dropping of packets during congestion at a router. In RIO, two sets of RED thresholds are maintained; one for packets tagged as ‘in’ and one for packets tagged as “out,” but both types of packets utilize the same queue. Two separate average occupancy calculations are performed, one for in-profile packets and one for in-profile plus out-of-profile packets. The buffer occupancy of the in-profile packets determines the possibility of dropping an in-profile packet, and the buffer occupancy of the in-profile plus out-of-profile packets determines the possibility of dropping an out-of-profile packet.

Assured service is just an example of a service that can be better than the best-effort service. It is also just an example of what the assured forwarding (AF) PHB group [14] may realize. AF is a powerful and general building block for providing different levels of forwarding assurances inside a DiffServ domain for different classes of traffic. It defines four classes of AF traffic, where each one uses reserved independent network resources. Within each class, three levels of drop precedence are defined. In case of congestion within an AF class, the drop precedence of a packet determines the relative importance of the packet within that class and thus the possibility of it being dropped.

AF, by controlling the drop preference of packets at the time of congestion, may provide a variety of better-than-best-effort services. One is the service suggested by the assured model. One other is the olympic service suggested in [14]: three classes of traffic share the same resources, the gold, the silver, and the bronze medal class. Traffic is assigned to a class and tagged in the edges and in case of congestion at any router, packets will be dropped preferentially that belong to lower medals (classes). Traffic that belongs to the gold medal is guaranteed a higher level of forwarding assurance than traffic belonging to the silver class, which in turn has higher assurances than the bronze class.

Having introduced the basic features of our testbed, the next chapter looks into the topologies studied, the experiments performed, and the results taken and motivates us to design a model that integrates Ethernet switches inside the differentiated architecture.

Chapter 4

4 Experiments

4.1 Overview

The experiments performed aim to illustrate the impact of an Ethernet switch inside a DiffServ domain. The impact is measured by the performance of the traffic services the domain provides. Our objective is obvious: we need a domain where the semantics of the traffic services it offers are met independently of the existence or nonexistence of layer-2 devices in the domain. We conclude that the effects of the switch can be significant and argue the importance of maintaining the intended QoS semantics.

The Linux testbed will be customized to different topologies and configurations of a DiffServ domain. We define and study four different cases of topologies and configurations. In the first case, the testbed is configured to implement a minimal DiffServ domain that provides four DiffServ traffic services. The performance the traffic services provide is measured and is compared to the results taken from the subsequent cases. In the second case, an Ethernet switch is introduced in the domain that provides no traffic differentiation. In the next case, the switch performs minimal QoS support and in the last case, the switch supports the same per-hop-behaviors as the router in the domain does. It will be shown that, when an Ethernet switch is present in the domain, only in the last case do the semantics of the traffic services hold. We argue that it is both important and feasible to integrate Ethernet switches in the DiffServ architecture by having them employ the same per-hop-behaviors the layer-3 devices do in a DiffServ domain. In the next chapter, we introduce a framework that realizes this integration.

In all the cases presented below, the performance the traffic services provide is measured. This performance is defined as the externally observable behavior demonstrated by a traffic flow assigned to the particular service. For the purposes of this work, this behavior is measured by the performance a UDP flow assigned to a particular traffic service experiences throughout the boundaries of the domain. The performance of the UDP flow is measured in terms of throughput and packet loss. In all the experiments described in this document, three UDP flows are generated simultaneously. UDP flows were used to avoid taking into account the TCP congestion control mechanism and to be able to measure packet loss. The flows use a constant packet size of 1000 bytes, chosen as the average packet size in the Internet. The combinations of different number, type, and profile of flows generated, along with the different provisioning schemes possible, are copious and could not all have been studied in our testbed. Besides, it is unnecessary. The examples presented in this work are adequate to illustrate that the presence of an Ethernet switch inside a DiffServ domain can significantly alter the desired traffic service semantics.

4.2 Case 1: DiffServ domain with one DiffServ-aware router

The first case to be examined is that of a minimal DiffServ domain, a domain with one DiffServ-aware router and no Ethernet switches. Four hosts are connected to the router. Three hosts act as the sources of three simultaneous traffic flows, each assigned to a different traffic service, and a fourth host is the destination of all sources, where our measurements are made. The router supports both EF and AF PHBs and thus is able to provide all four traffic services mentioned in this document: premium, assured, olympic, and best-effort. The flows have to compete for bandwidth on the bottleneck link (10 Mbps), the link between the router and the destination host.

The topology of this case as implemented in our Linux testbed is shown in Figure 4.1. In the first two experiments, premium traffic is generated from one source host destined to the receiver's port number 6001, assured traffic is generated from the second source host destined to the receiver's port number 6002, and best-effort traffic is generated from the third source host and is destined to the receiver's port number 6003.

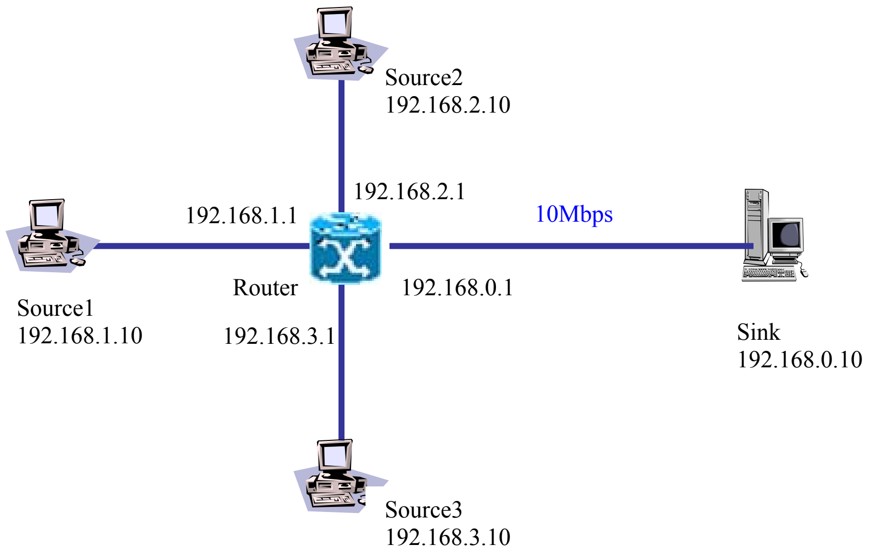

Figure 4.1. Case 1 topology.

Packets with a destination port of 6001 that do not exceed the subscribed service profile of 3 Mbps are tagged at the ingress interface of the router as belonging to the EF PHB (0x2e DSCP). If they exceed the subscribed profile, they are dropped. Assured traffic is also monitored; the ingress interface of the router marks traffic with the AF11 code point (0x0a) as long as it conforms to the service profile of 3 Mbps. The interface marks it with the AF12 code point (0x0c) when it exceeds the limit. Best-effort traffic is also tagged with the same code point, AF12. In this way, in-profile assured traffic is treated preferentially to out-of-profile assured traffic, which will experience the same service as the best-effort traffic. Table 4.1 lists these mappings.

Table 4.1. Service mappings for first and second experiments.

Source	Dest Port	Profile	Service Assigned	PHB	DSCP
Source1 192.168.1.10	6001	< 3Mbps	Premium	EF	0x2e
		> 3Mbps	dropped	-	-
Source2 192.168.2.10	6002	< 3Mbps	Assured	AF11	0x0a
		> 3Mbps	Best-effort	AF12	0x0c
Source3 192.168.3.10	6003		Best-effort	AF12	0x0c

A CBQ queuing discipline is attached to the egress interface of the router. Two classes are created within the queuing discipline, one for EF traffic and one for AF and best effort. The filters classify traffic according to the DSCP value set by the ingress interfaces. The EF class consists of one FIFO queue, is given the highest priority among the different CBQ classes, and reserves a bandwidth of 3 Mbps. The AF class consists of a generalized RED (GRED) queuing discipline (qdisc). GRED is based on the RIO framework, but provides more flexibility. Multiple levels of drop precedence can be defined for differently tagged packets inside the same queue. Particularly, one GRED qdisc can implement one AF class, where the drop precedence encoded in the DSCP of the packet defines the drop probability for the packet in the GRED algorithm. More CBQ classes may be used for implementing more AF classes, with a GRED qdisc attached to each of them. For simplicity, we are using only one AF class. The filters attached to the AF CBQ class assign packets to one of two drop preferences. AF11 packets (in-profile assured traffic) are assigned to the lowest drop precedence; AF12 packets (out-of-profile and best-effort traffic) are assigned to the highest drop precedence. In this way, assured and best-effort share the same network resources (bandwidth and buffer space), but in case of congestion, in-profile assured packets will have a greater probability of being forwarded. Finally, the egress interface of the router also marks outgoing packets with the corresponding DSCP value.

The throughput results for generated premium traffic at a constant rate of 3 Mbps, assured traffic at a constant rate of 5 Mbps, and best-effort traffic varying between 1 and 5 Mbps are shown in Figure 4.2. It is shown that the premium traffic is guaranteed a 3 Mbps throughput regardless of the traffic generated for any other service. The x-axis measures the generated throughput of the best-effort traffic. Near the point where the generated best-effort traffic reaches 1.5 Mbps, congestion starts to occur and for all values of generated best-effort traffic greater than that, congestion exists. Figure 4.2 shows for all those values how assured traffic experiences more throughput than the best-effort traffic. Characteristic are the results for the point where both assured and best-effort traffic are generated at the same rate (5 Mbps). The right-most values of the graph show 3.8 Mbps throughput for the assured traffic and 2.8 Mbps throughput for the best-effort traffic. It is clear that in case of congestion, assured traffic is treated preferentially to best-effort traffic. Figure 4.3 is more illustrative of how assured experiences better performance than best effort; it depicts the per-flow percentage of packet loss over the same set of values for generated best-effort traffic. In case of congestion, best-effort traffic experiences always greater packet loss than assured traffic, which is the effect of the RED algorithms: best-effort marked packets are dropped more aggressively than assured marked packets. Premium traffic conforms to the service profile and thus experiences no packet loss.

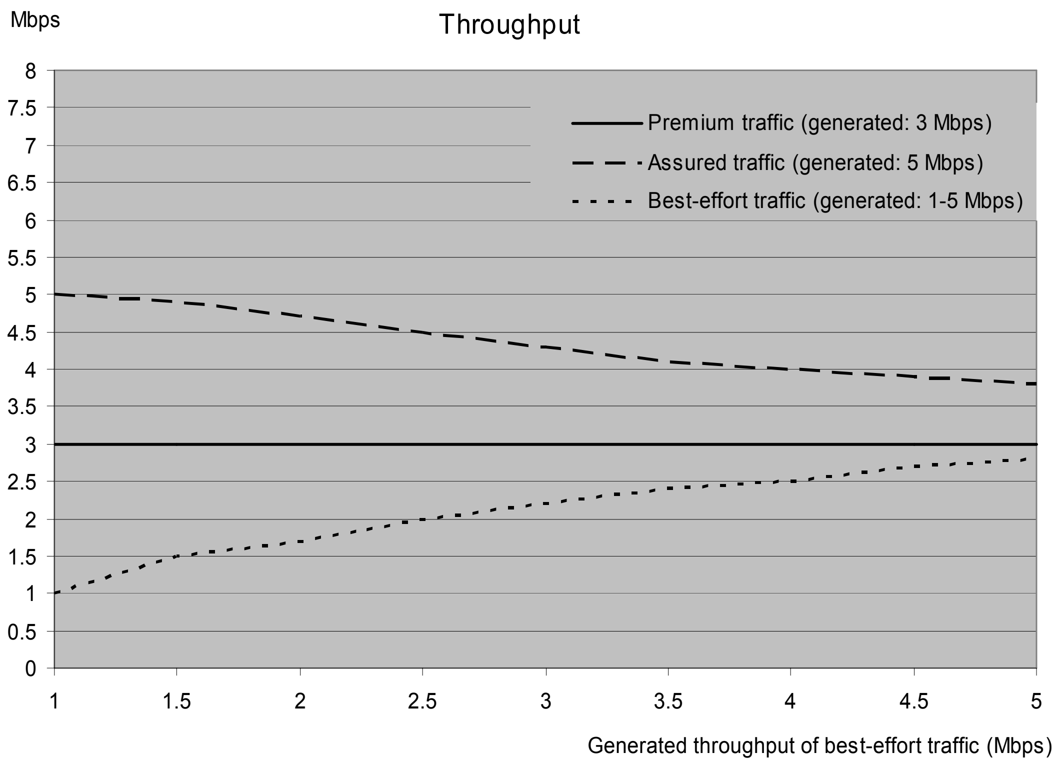

Figure 4.2. Throughput results for first experiment of case 1.

Figure 4.4 depicts the throughput results of the second experiment: premium traffic is generated at 5 Mbps, assured is generated at 3 Mbps, and best-effort varies between 1 and 5 Mbps. In this case, premium traffic does not receive the generated throughput since it is policed, and the exceeding traffic is dropped at the ingress interface of the router, though the reserved bandwidth is still provided during times of congestion, which occurs when generated best-effort traffic reaches 3.5 Mbps. From that point on, neither assured nor best-effort traffic receives its generated throughput, though the

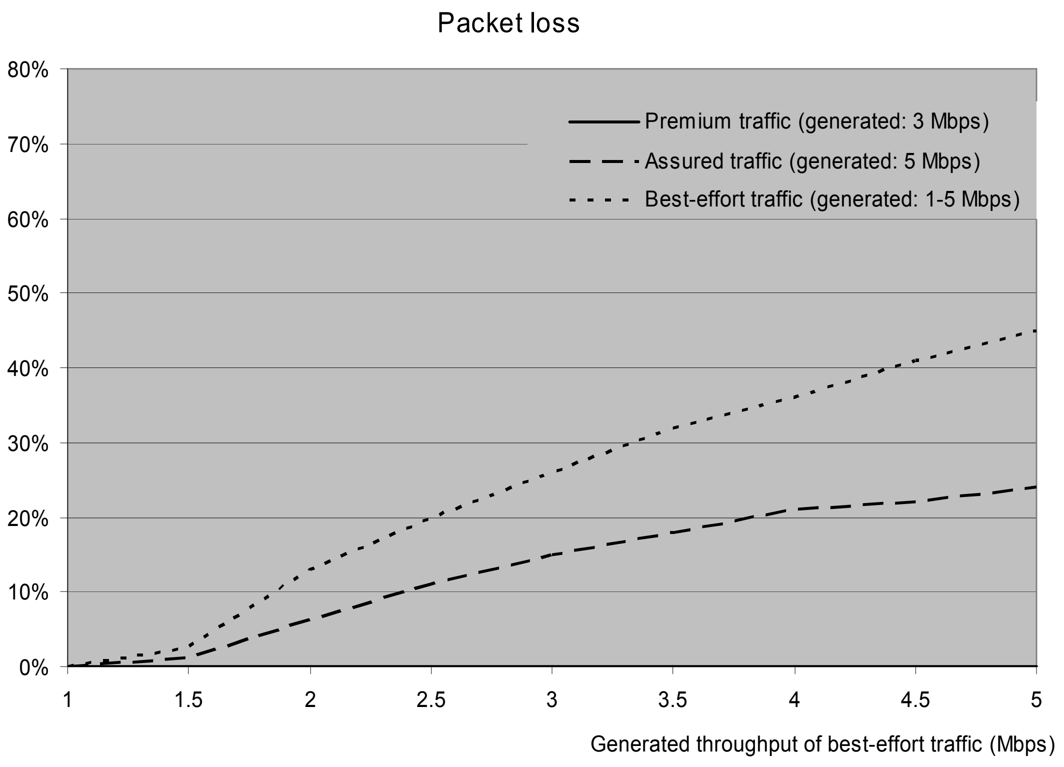

Figure 4.3. Percentage of packet loss for first experiment of case 1.

assured traffic experiences better service than best effort. Let's take for example the point where best-effort-generated traffic reaches 5 Mbps. Assured traffic is slighted affected by dropping at about 2.8 Mbps out of the generated 3 Mbps of throughput. Best-effort traffic, though, drops at 3.7 Mbps out of the generated 5 Mbps of throughput. Assured traffic experienced a 6.7% drop in throughput, whereas best-effort traffic experienced a 26% drop. The ratio between the two generated rates, 5 Mbps for the best effort and 3 Mbps for the assured, yields 1.67. The ratio for the measured performance drop, 26% for the best-effort and 6.7% for the assured, yields 3.8, which is greater than 1.67; thus, the assured service provided a better service than the best-effort one.

Figure 4.5 depicts the percentage packet loss for the same flows. Premium traffic experiences steady packet loss of 40%, which is the excess traffic dropped by the policing mechanism of the router. Assured and best-effort traffic experience packet loss only in times of congestion, which is for all values where generated best-effort throughput exceeds 3.5 Mbps, and best-effort traffic always experiences greater packet loss than assured does. It is worth comparing the curves of the assured traffic in Figures 4.3 and 4.5. The slope in the first one is greater, which means that assured traffic is dropped more aggressively. This is because, in the first case, assured exceeds the service profile, whereas in the second it does not.

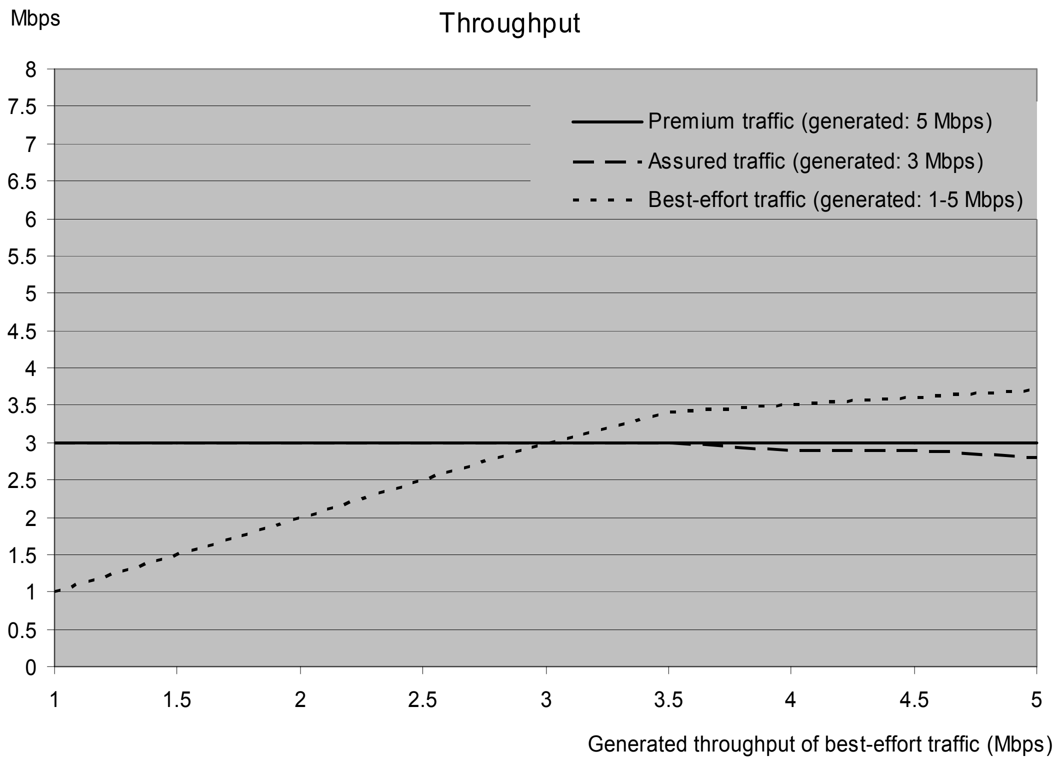

Figure 4.4. Throughput results for second experiment of case 1.

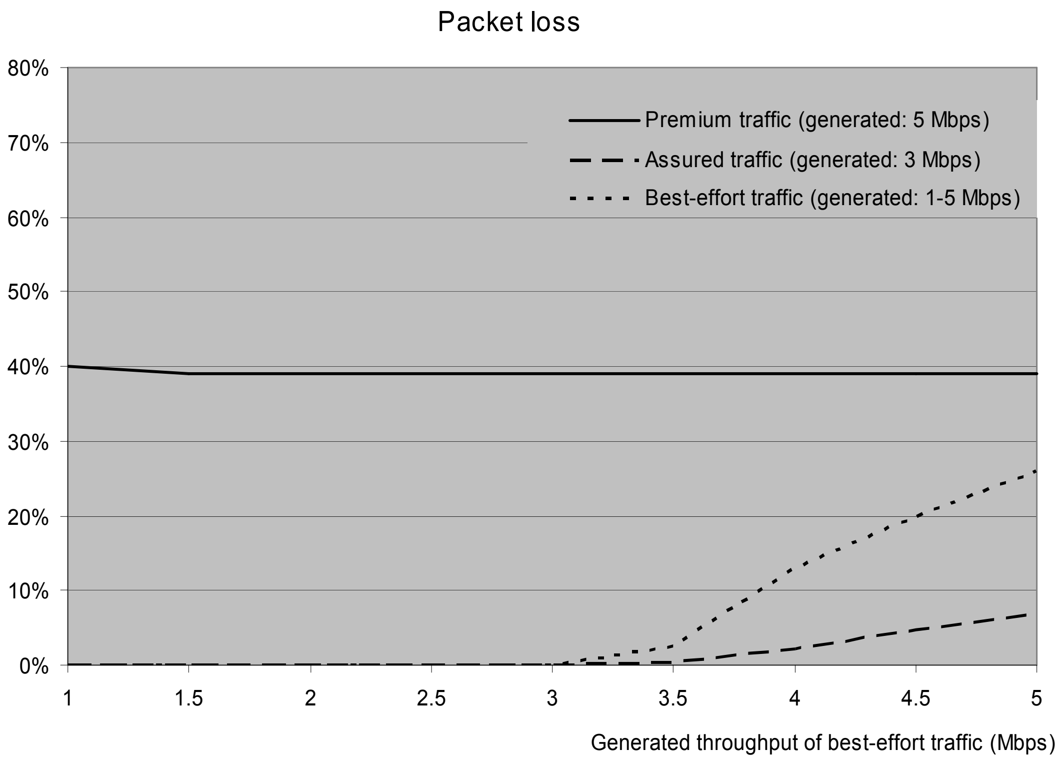

Figure 4.5. Percentage of packet loss for second experiment of case 1.

Both of these experiments demonstrate how the semantics of the premium and the assured service are met in the domain. Premium always gets its subscribed throughput. Moreover, it is expected that it experiences low delay and jitter. These metrics are not measured in our testbed because they would only be indicative of the service performance of the premium service. Throughput and packet loss are indicative for all the services studied and are considered to be adequate to describe the service results for the purposes of our work. Next, assured service is also successfully provided, because when traffic is kept within the service profile, it will lose only packets with low probability; otherwise, exceeding traffic will be forwarded with less assurance. In the results shown in Figure 4.4, low probability means that in-profile assured traffic will experience little throughput loss in case of congestion. Moreover, the packet loss will be less than the one experienced by best-effort traffic (Figures 4.3 and 4.5).

This loose definition of low probability, which determines how assured the assured service is, does not make the assured service less useful. On the contrary, this lack of state information that would need to be configured and propagated throughout the network as well as the fact that different implementations and variations can co-exist are what make assured service and most better-than-best-effort services very appealing. Such services can be much more easily and rapidly implemented, vastly deployed, and, eventually, brought to use and market.

For the third experiment, the configuration of the domain has to change to provide another better-than-best-effort service: the olympic service. A gold, a silver, and a bronze flow are generated simultaneously. Table 4.2 lists the service mappings for this experiment. All flows generate the same throughput, which varies between 1 and 4 Mbps. Figure 4.6 shows the throughput performance of the three flows in relation to the individual generated rate. Figure 4.7 shows the percentage of packet loss for every flow for the same values of generated traffic. The same conclusions are derived from both figures. The flow assigned to the gold medal service experiences better performance than the flow assigned to the silver medal service. The silver medal traffic, in turn, experiences better performance than the bronze medal traffic. Both figures show that where congestion occurs, at about 3.2 Mbps of generated traffic for every flow, all flows start experiencing less than generated throughput and packet loss. Higher medal traffic always experiences better performance than lower medal traffic: higher throughput and less packet loss. The relative performance of two flows, ratio of throughput and ratio of packet loss, is a system parameter that can be adjusted to yield the desired level of forwarding assurance for every level of traffic service.

Table 4.2. Service mappings for third experiment.

Source	Dest Port	Profile	Service Assigned	PHB	DSCP
Source1	6001	None	Gold	AF11	0x0a
Source2	6002	None	Silver	AF12	0x0c
Source3	6003	None	Bronze	AF13	0x0e

There is no reason for a domain providing only the olympic service. One AF class can be reserved for the olympic service and another for the assured service. For simplicity and because of limitations to the simultaneous different flows in our testbed, the olympic service is studied without the presence of any other traffic.

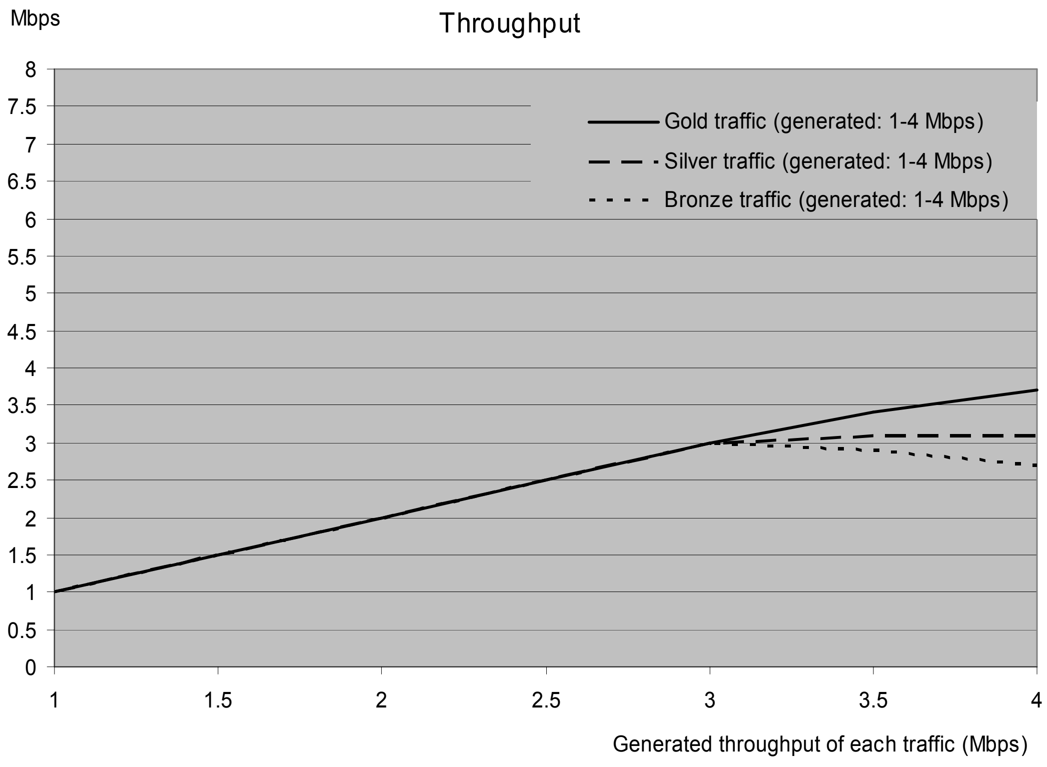

Figure 4.6. Throughput results for third experiment of case 1.

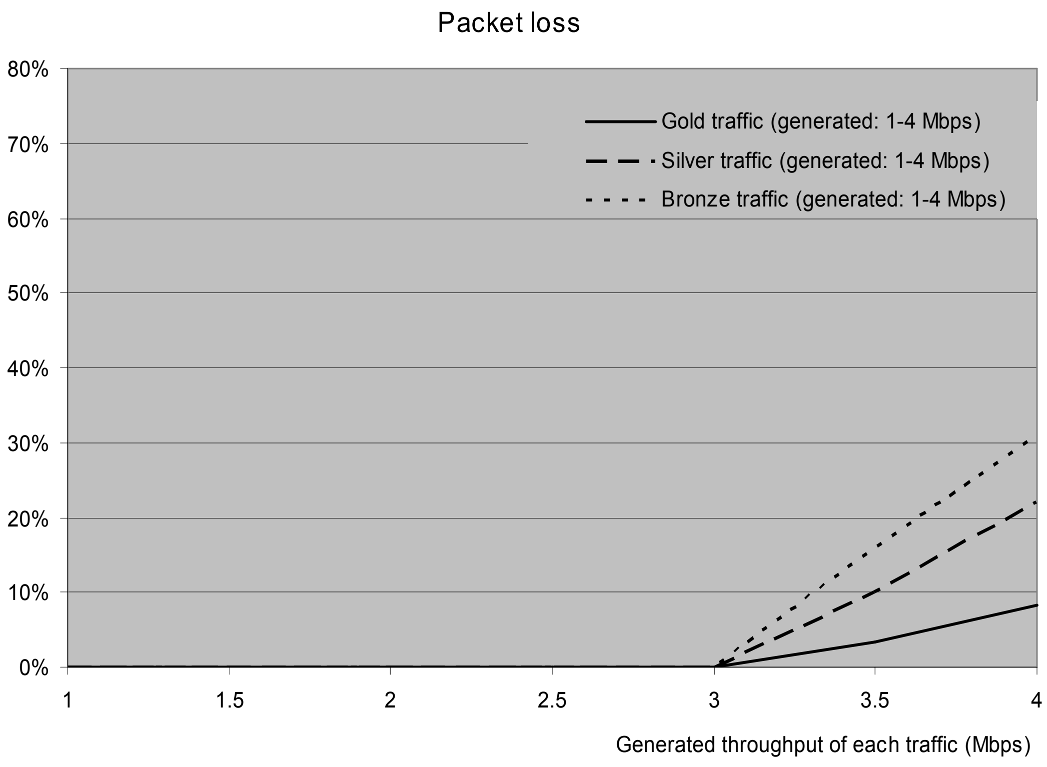

Figure 4.7. Percentage of packet loss for third experiment of case 1.

This case describes the design of a DiffServ domain and its implementation in a Linux-based network. This case is important because it helps us to understand what the services intend to offer and how they can be implemented, as well as to illustrate their use in a set of examples. Moreover, it is shown that the domain meets the semantics of the services it offers for the UDP flows that were generated. It is relatively easy and straightforward to extend to a larger topology with more DiffServ-aware routers a DiffServ domain like the one we presented. It is expected that the same conclusions would be derived: the network succeeds in delivering the traffic services it offers using the same semantics in which the services are defined.

4.3 Case 2: DiffServ domain with one DiffServ-aware router and a QoS unaware Ethernet switch

The next case studied is based on the previous case, with the addition of an Ethernet switch in the network. The switch makes no differentiation of traffic and treats all packets equally. The router has the exact same functionality as in the last case, except that it receives traffic from a single ingress interface rather than three. The topology of this case is shown in Figure 4.8. The same experiments are studied for this case as before, and the results are compared so as to research the impact of the Ethernet switch on the performance of the UDP flows (and thus on the performance of the services themselves).

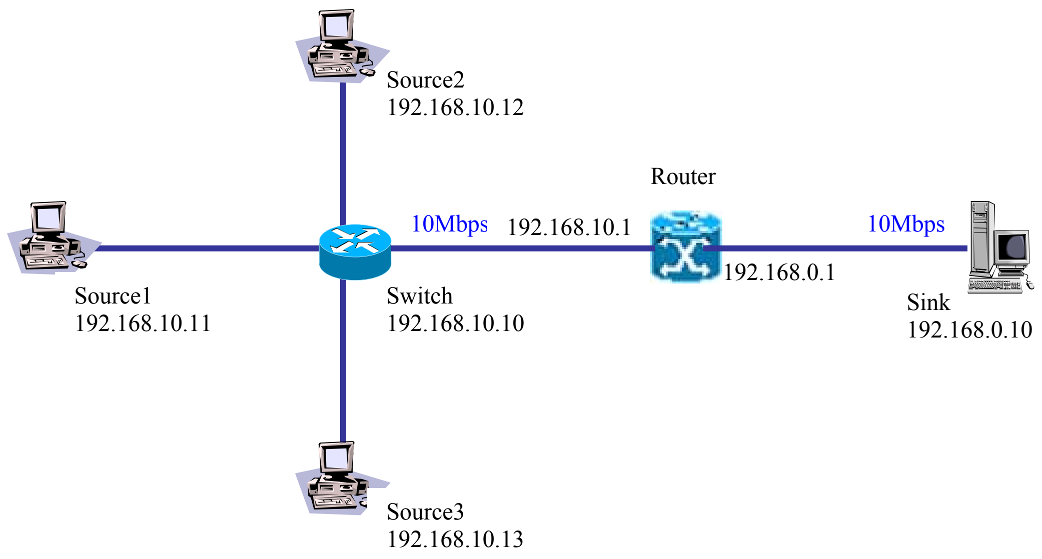

Figure 4.8. Case 2 topology.

The bottleneck link in this case is the link between the switch and the router (10 Mbps). Thus, intuitively one can see that traffic control on the egress side of the router would do little with no traffic control on the egress side of the switch. All three flows have to contend first on the link between the switch and the router, and there will be no congestion on the link between the router and the receiver host.

In the first experiment, premium traffic is generated at 3 Mbps, assured traffic is generated at 5 Mbps, and best-effort generated throughput varies between 1 and 5 Mbps. Figure 4.9 shows the throughput at the ingress interface of the router as it is transmitted by the switch. Neither premium nor assured traffic receives the desired performance so far. This is expected since there is no traffic differentiation in the switch and all traffic is experiencing best-effort service. The throughput as measured at the receiver is shown in

Figure 4.10, which is identical to Figure 4.9. This shows how the DiffServ-aware router is useless when a QoS-unaware Ethernet switch precedes it. Premium traffic receives less than the subscribed bandwidth from the moment congestion occurs (1.5 Mbps generated best-effort traffic) and keeps dropping performance to the same degree that the generated best-effort traffic increases. Also, assured traffic is not assured and does not receive any better service than best-effort traffic does. This is better illustrated in Figure 4.11, which illustrates the percentage of packet loss as measured at the receiver. The graph shows that the flows are more or less treated equally; they all experience about the same average packet loss. Overall, the switch has made DiffServ impossible.

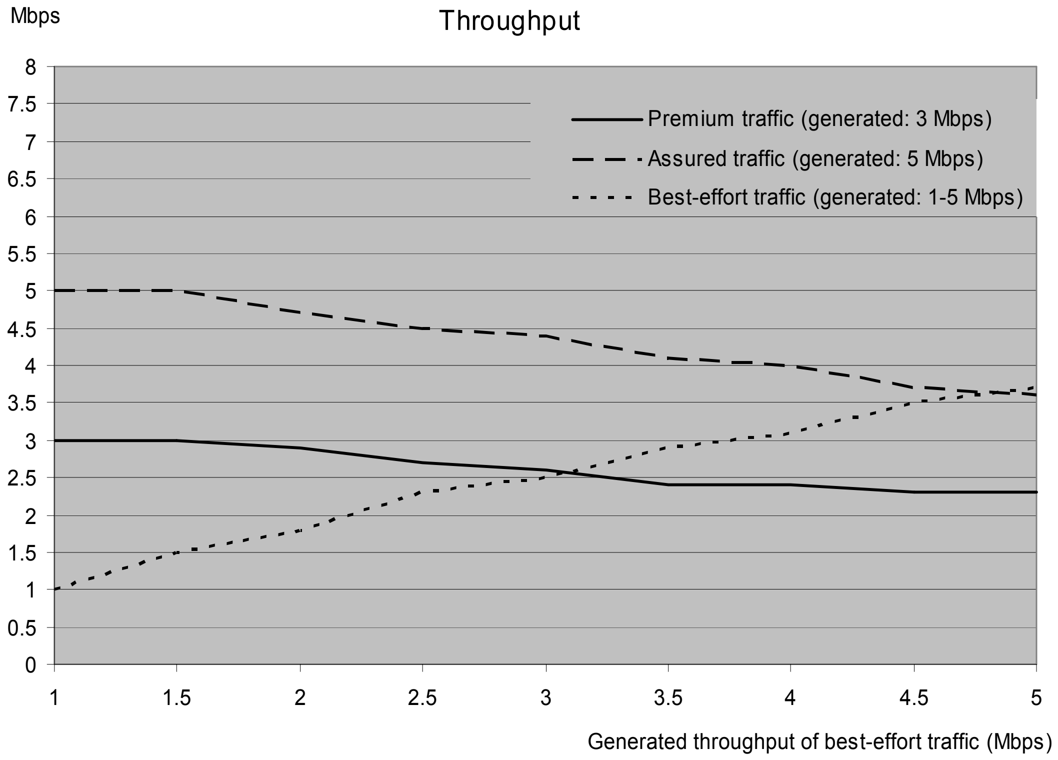

Figure 4.9. Throughput results at the ingress interface of the router for first experiment of case 2.

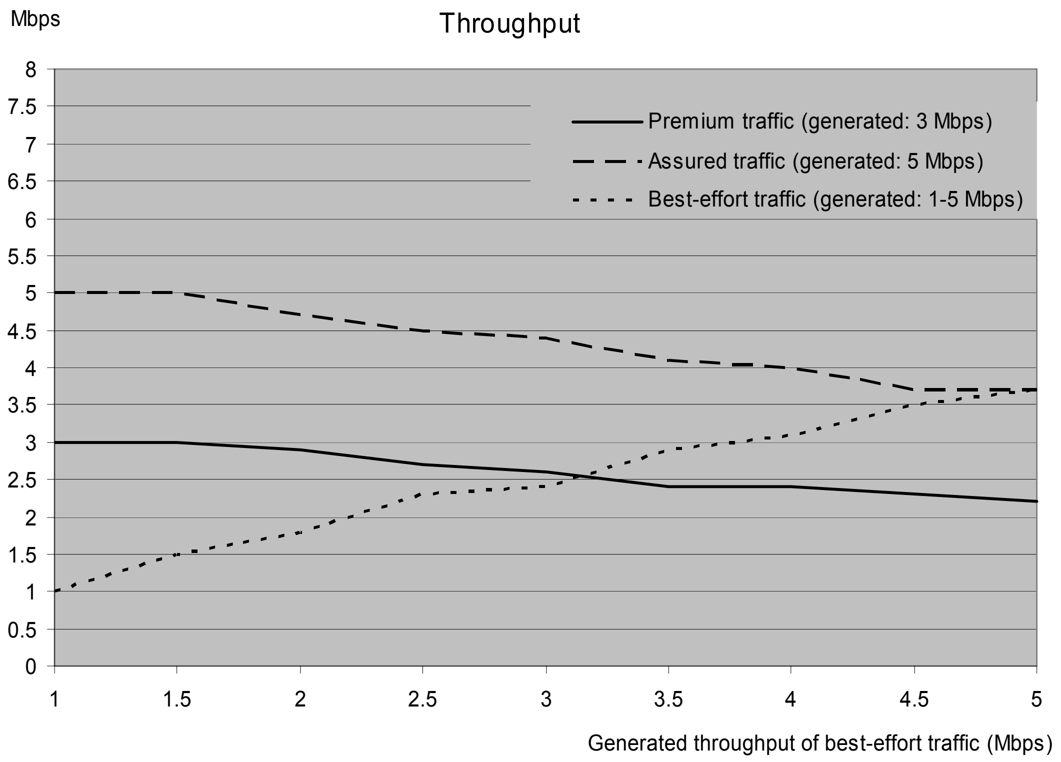

Figure 4.10. Throughput results for first experiment of case 2.

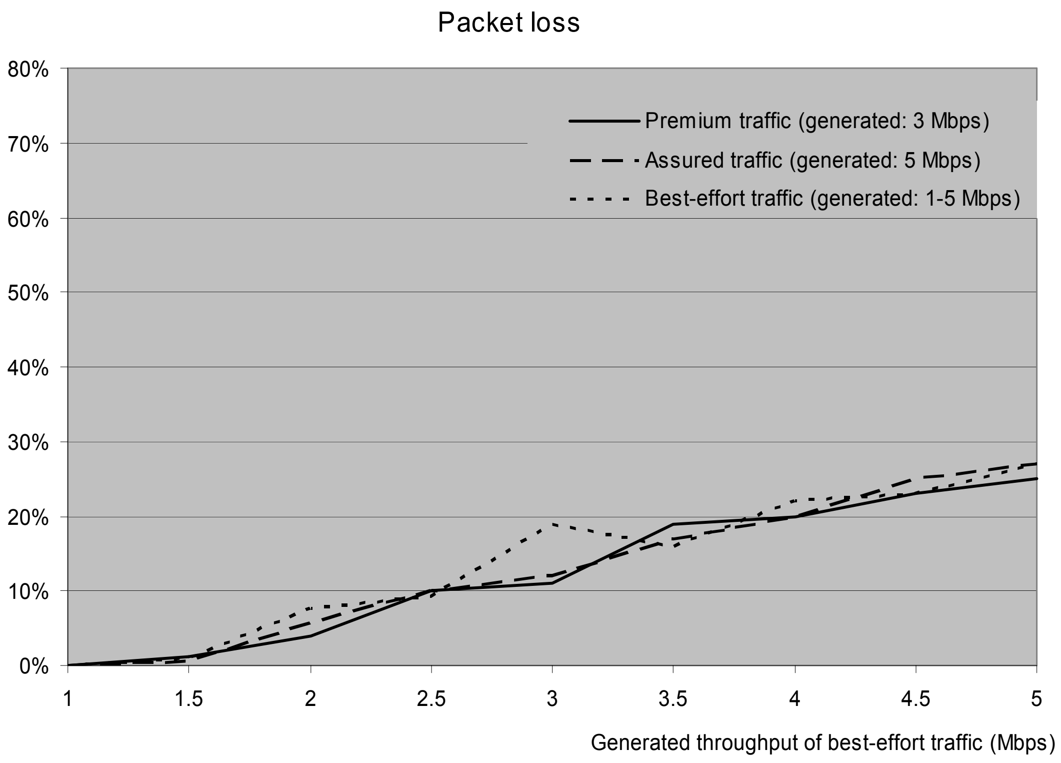

Figure 4.11. Percentage of packet loss for first experiment of case 2.

In the second experiment, 5 Mbps of premium traffic and 3 Mbps of assured traffic are generated. The best-effort-generated rate varies between 1 and 5 Mbps. Throughput results are shown in Figure 4.12. In this case, assured traffic does not receive expected throughput even for in-profile traffic. In fact, the assured traffic performance starts to fall well before congestion should occur (3.5 Mbps of best-effort generated traffic) at 2 Mbps of generated best-effort traffic. Nevertheless, premium traffic does receive the guaranteed bandwidth. Looking at the performance measured at the ingress interface of the router (Figure 4.13) helps us to understand these results. Premium traffic is not policed in the switch so all 5 Mbps of the generated traffic has to compete with the rest of the traffic in the bottleneck link. The graph in Figure 4.13 shows that in the worst case, premium traffic gets 3.5 Mbps. The router then drops exceeding traffic and ensures that 3 Mbps will be reserved for premium traffic in its egress interface, which is shown in Figure 4.12. However, in our case, where there is no cross-traffic in the router, there is no congestion at that point either. This means that exceeding premium traffic that makes it through the link between the switch and the router (but is dropped by the router) wastes useful bandwidth that could have been utilized by assured or best-effort traffic.

Figure 4.14 illustrates the packet loss the flows experience. As expected, the premium traffic experiences a steady packet loss as a result of the policing mechanisms of the router. The other two flows, on average, experience roughly equal packet loss; packet loss occurs at the switch, where traffic is serviced equally. If premium traffic were not policed, it would be expected to experience about the same packet loss as the other flows. This is illustrated in the next experiment, where the flows are not policed.

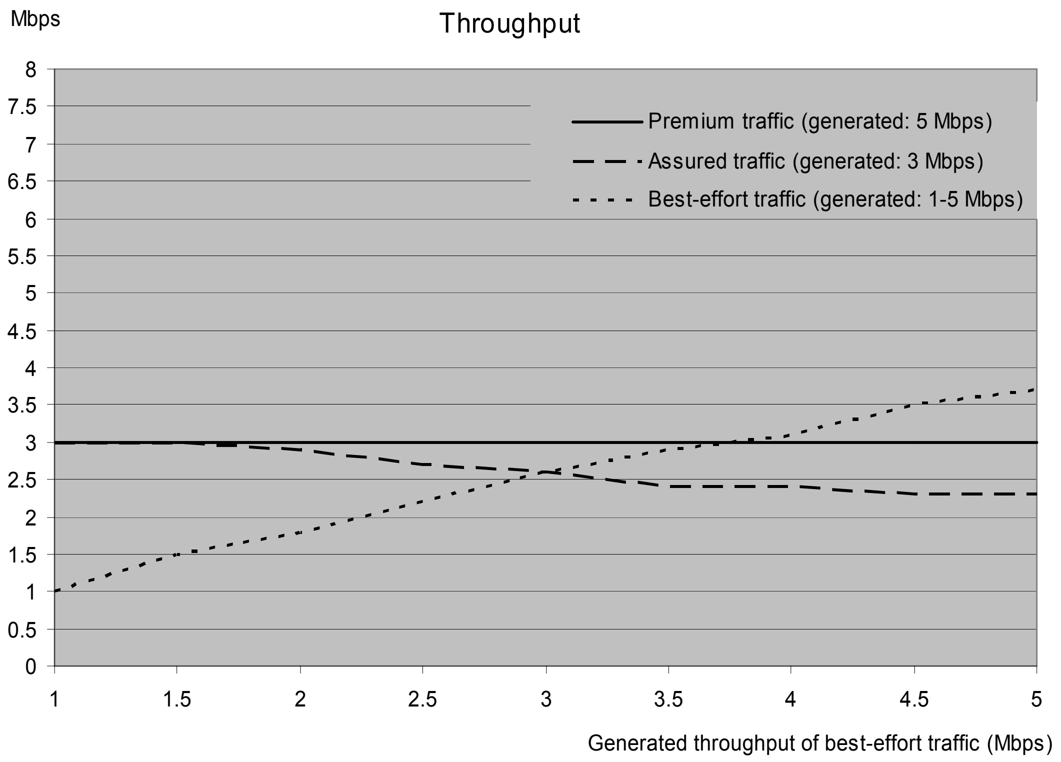

Figure 4.12. Throughput results for second experiment of case 2.

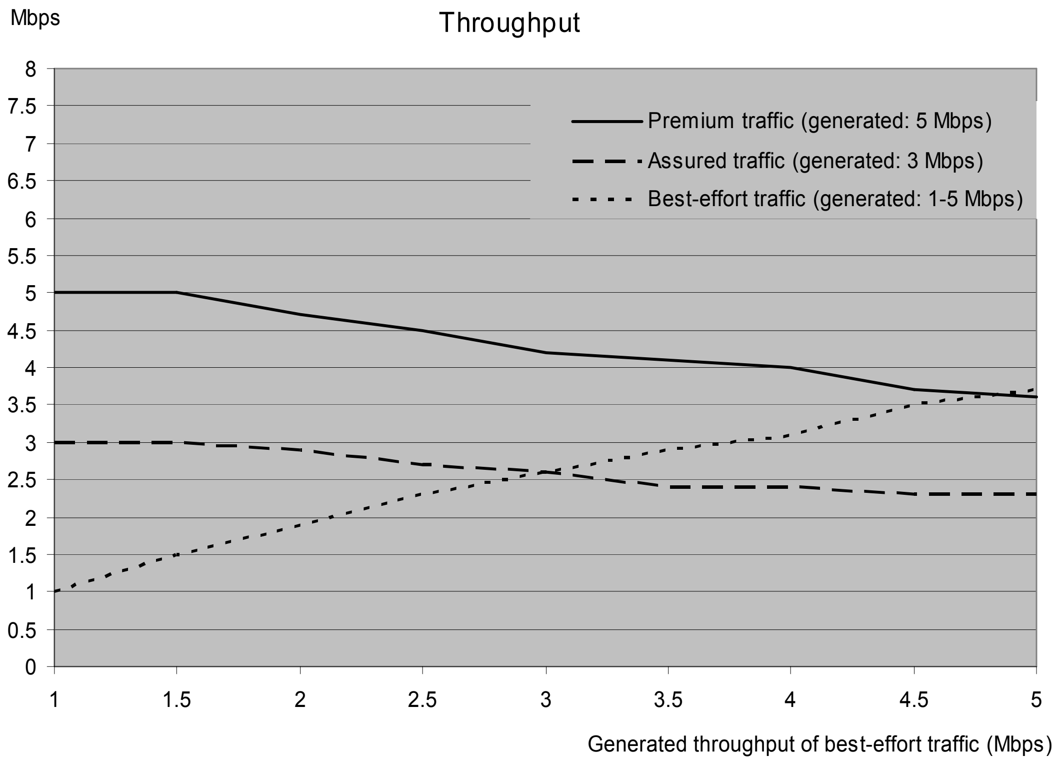

Figure 4.13. Throughput results at the ingress interface of the router for second experiment ofcase 2.

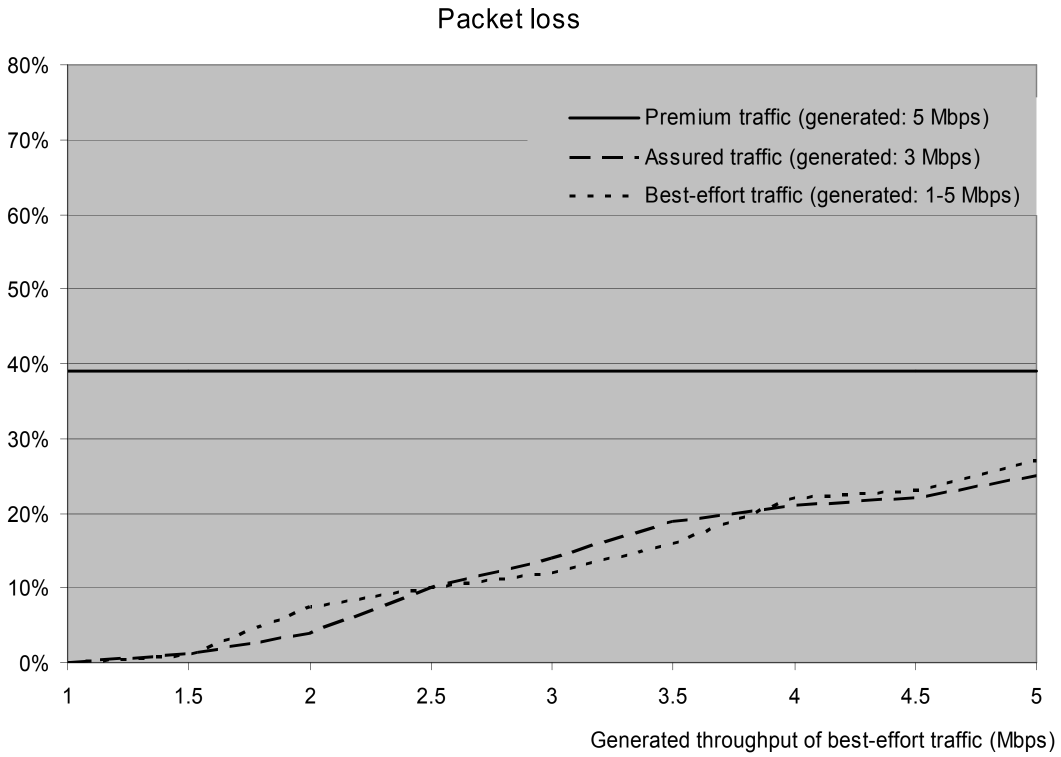

Figure 4.14. Percentage of packet loss for second experiment of case 2.

For the third experiment (olympic service), Figures 4.15 and 4.16 show the results for the throughput and the percentage of packet loss, respectively. Since the switch does not differentiate between traffic, equally generated traffic of equal flows is equally serviced in the long term. This is exactly what these figures show. On average, the three flows experience about the same throughput and packet loss ratio. At the point where congestion starts to occur, at about 3.2 Mbps, all flows receive less than the generated throughput and all flows experience packet loss. For every generated value, not all flows are affected equally, but on average it seems they all are affected the same.

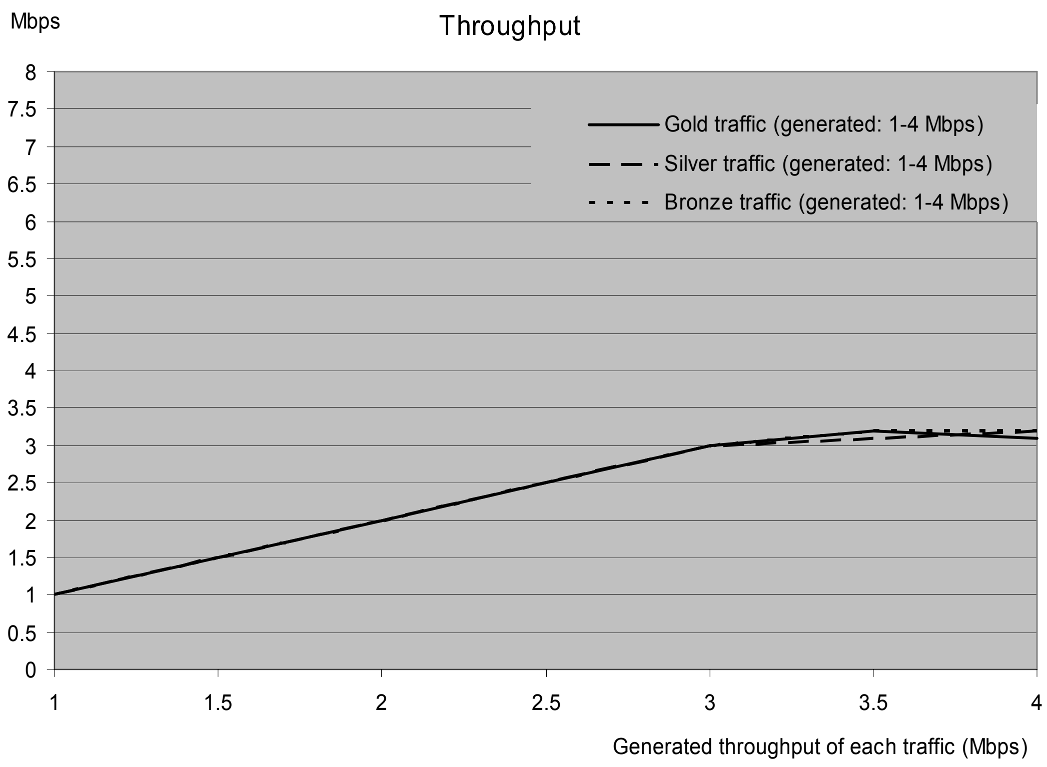

Figure 4.15. Throughput results for third experiment of case 2.

This case shows how the semantics of the traffic services offered by our testbed break apart when an Ethernet switch that makes no traffic differentiation is introduced in the DiffServ domain. The next case shows the effects of a switch that provides minimal QoS in the context of the IEEE 802.1p Ethernet traffic classes.

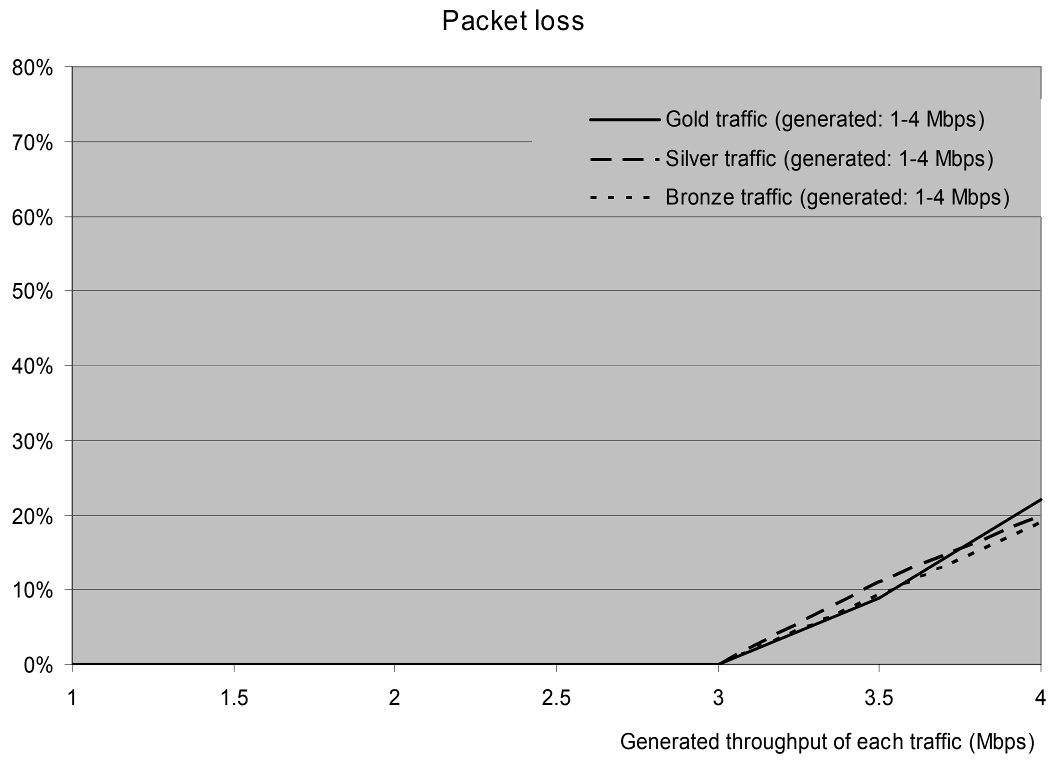

Figure 4.16. Percentage of packet loss for third experiment of case 2.

4.4 Case 3: DiffServ domain with one DiffServ-aware router and an IEEE 802.1p aware Ethernet switch (implementing strict-prioritizing)

It is well understood, and it was also shown in the last case, that a network cannot provide predictable QoS without having layer-2 devices participating in the QoS provision scheme. This is what has motivated IEEE to publish 802.1Q [30] and 802.1p [29] standards. Standard 802.1Q defines an extension to the Ethernet frame header, where a three-bit user-priority and a 12-bit VLAN ID field are added. Standard 802.1p assigns default user-priority semantics, aiming to establish a reasonable set of defaults for use in typical environments, allowing at the same time priorities, queue mappings, and queue service disciplines to be managed to best support user's goals. The standard acknowledges that the quality of service needs of an application are surely too complex to be represented by a simple number 0 through 7. Nevertheless, it claims that potential bandwidth efficiency should be traded for simplicity and that the pragmatic aim of traffic classification in Ethernet switches should be to simplify requirements radically to preserve the high-speed, low-cost handling characteristic of switches. This case shows how the default mappings and queuing disciplines in an IEEE 802.1p compliant Ethernet switch are not adequate for a DiffServ domain. The next case shows how Ethernet switches can preserve the QoS semantics of a DiffServ domain without sacrificing simplicity. On the contrary, the necessity of mapping between Ethernet and DiffServ traffic classes imposes an unnecessary burden on the network design. Furthermore, we will argue that this does not necessarily yield higher costs or lower performance of the Ethernet switches.

In this case, the same exact topology of a DiffServ domain is studied as in the last case, though the Ethernet switch does support IEEE 802.1p QoS. Seven traffic types are defined by the standard and their mappings to the user-priority field are shown in the following table. It is not mandatory to maintain a different queue for each of the user-priority values, but a mapping between them must exist. In our testbed, three queues are maintained, one for each of the following traffic types: excellent effort, background, and best effort. Queues are served in a strict prioritizing manner, which is what the standard mandates, and the priority of a queue derives from the user-priority mapping: the excellent-effort queue has the highest priority, the background queue has the second highest priority and will be served as long as there are no packets waiting in the highest priority queue, and the best-effort queue has the lowest priority and will be served only when neither of the other two queues has packets to transmit.

Table 4.3. Ethernet user-priority to traffic type mappings.

User-priority	Acronym	Traffic Type
1	BK	Background
2	-	Spare
0	BE	Best Effort
3	EE	Excellent Effort
4	CL	Controlled Load
5	VI	"Video", < 100 msecs latency and jitter
6	VO	"Voice", < 10 msecs latency and jitter
7	NC	Network Control

What is needed now is a mapping between user-priority values and PHBs. Such mappings have been suggested [31], but none has been standardized. Besides, such standardization does not seem very likely; it is more likely that autonomous domains would use their own mappings suited for their needs and the services they offer. In our testbed, our own mapping examples are used. For the first two experiments, where premium, assured, and best-effort traffic are generated, the mapping adopted is shown in Table 4.4. What this means is that incoming traffic is monitored at the switch and classified to a PHB according to the IP packet destination port. Then, this PHB maps to a segregated queue, which in turn maps to a user-priority value. Packets belonging to the same PHB utilize the same queue and are tagged with the same user-priority field. Actually, tagging the packet with the user-priority field is not needed in this topology, where there are no more switches in the domain. The user-priority field in the extended Ethernet frame is used to identify the packet so that a switch will not have to perform any classification functions. The field is usually set by an edge switch, which does perform the necessary classification functions. In a more unlikely case, trusted hosts would set the user-priority field themselves. In both cases, the rest of the switches in a network would have to extract only the user-priority field to identify the packet. In our domain, there are no more switches. Thus, there is no need to set the user-priority field.

Table 4.4. User-priorities to PHBs mappings for first and second experiments.

User-priority	Queue	Ethernet Traffic Type	PHB
0	1	Best Effort	0x0 (Best Effort)
1	2	Background	0x10-0x38 (Assured)
3	3	Excellent Effort	0x2e (Premium)

In the first experiment, premium traffic is generated at 3 Mbps, assured traffic is generated at 5 Mbps, and best-effort generated throughput varies between 1 and 5 Mbps, as in the previous cases. Figure 4.17 shows the measured throughput at the receiver host for the three traffic flows. Premium traffic, since it is assigned to the highest priority queue in the switch, is guaranteed to use as much bandwidth as necessary. The assured traffic is also given the generated throughput, since congestion occurs only because of best-effort traffic. The available bandwidth of 10 Mbps between the switch and the router is enough to accommodate both premium and assured traffic with no losses; bandwidth that is left (~1.5 Mbps) is used by the best-effort traffic. When congestion occurs (at 1.5 Mbps of generated best-effort traffic), the only traffic affected is the best-effort traffic with the lowest priority in the switch. The same is shown in Figure 4.18, where the packet loss measured in the receiver for the three flows is depicted; only the best-effort traffic experiences packet loss.

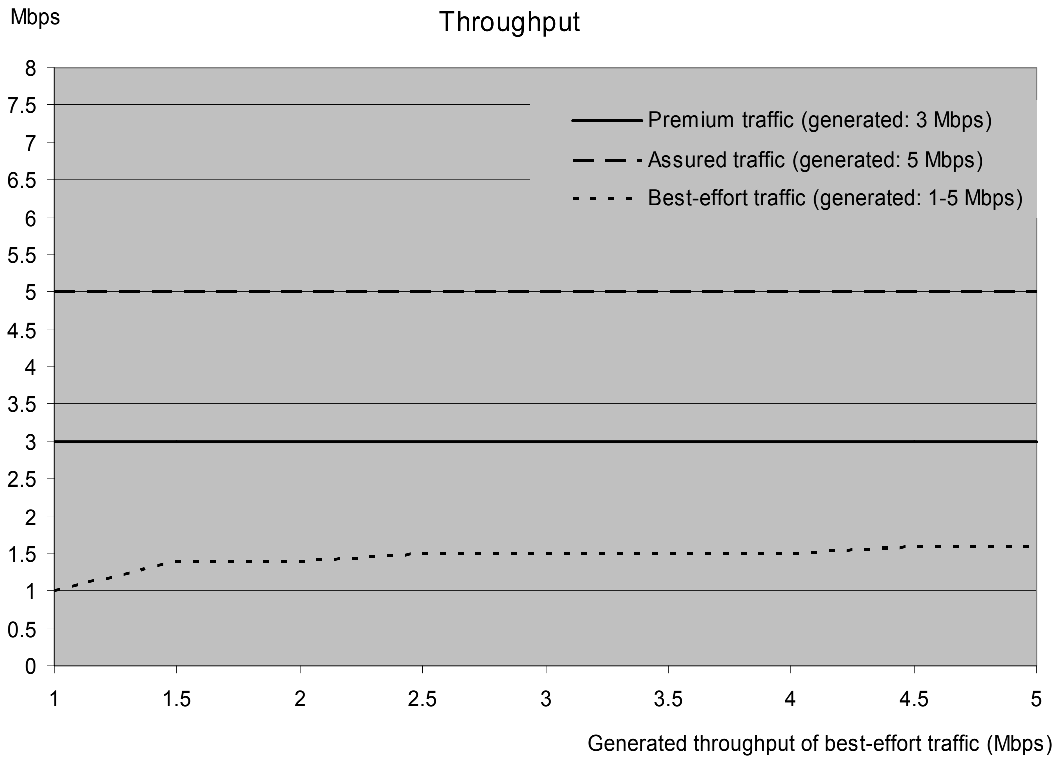

Figure 4.17. Throughput results for first experiment of case 3.

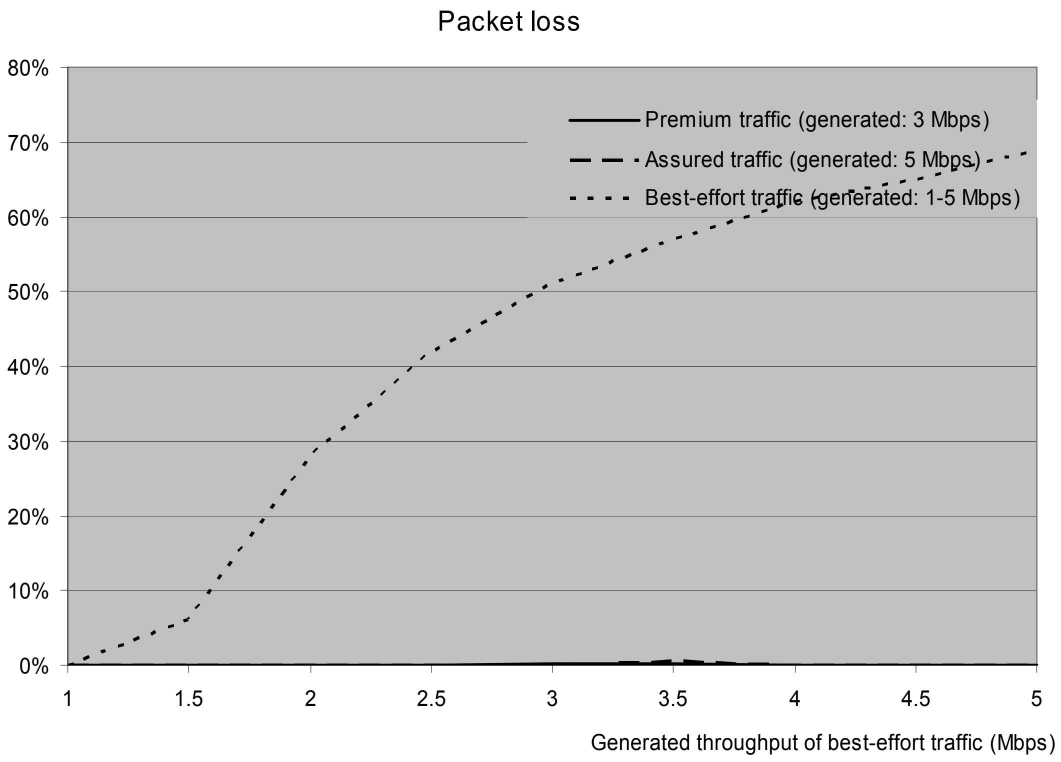

Figure 4.18. Percentage of packet loss for first experiment of case 3.

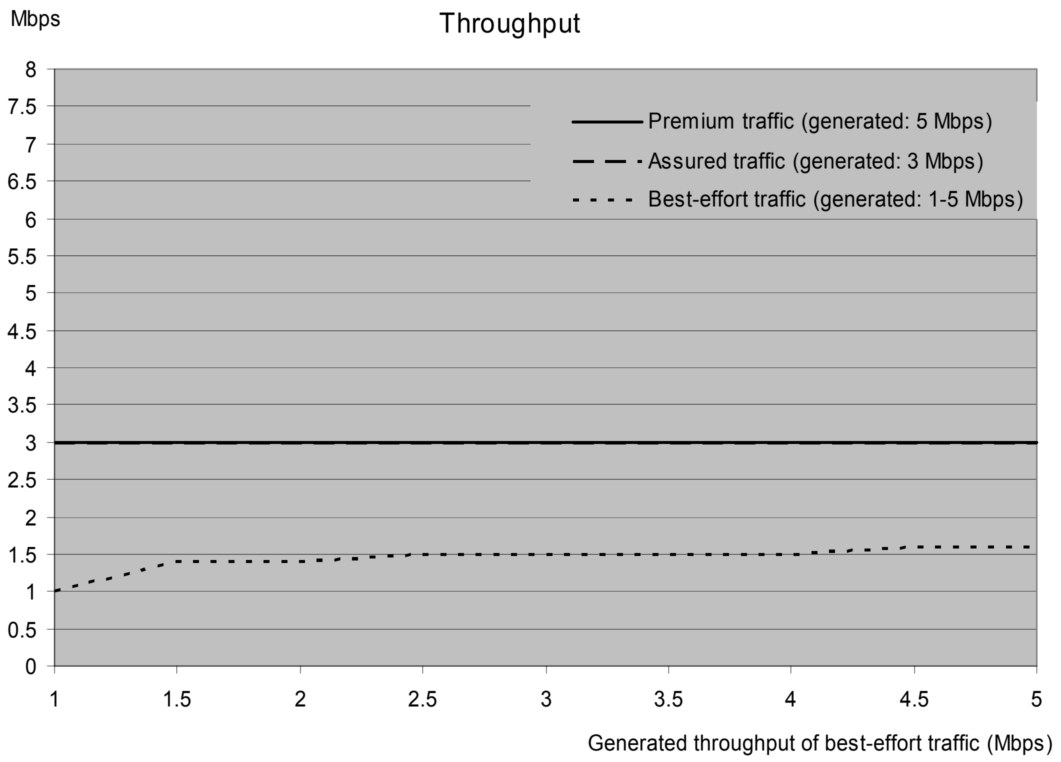

Figure 4.19. Throughput results for second experiment of case 3.

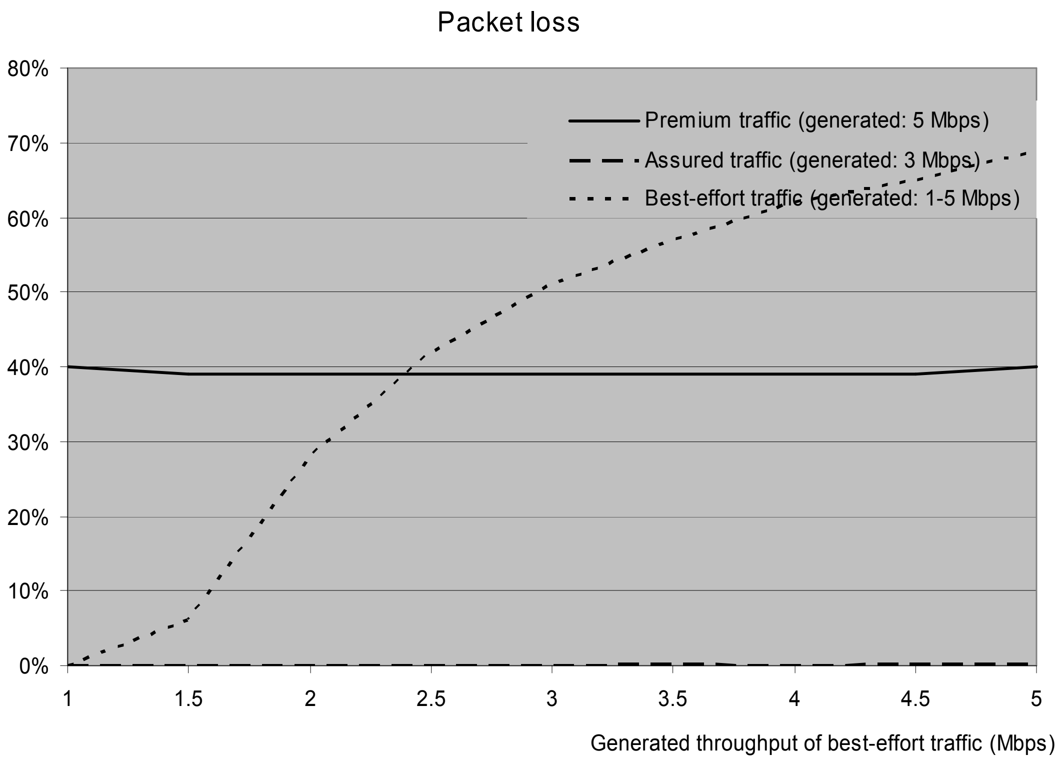

Figure 4.20. Percentage of packet loss for second experiment of case 3.

These results are different than those presented in the first case. The QoS semantics hold only for the premium traffic. Assured traffic gets much better traffic service over the best-effort traffic than it is supposed to. This is undesirable. Moreover, this case can lead to underutilization of the network. This is shown in the next experiment, where 5 Mbps of premium traffic is generated, 3 Mbps of assured service is generated, and the best-effort generated rate varies between 1 and 5 Mbps. Since premium traffic is not policed on the switch, it utilizes 5 Mbps of the link between the switch and the router. Assured traffic also utilizes 3 Mbps, and best effort is the one to suffer, receiving only 1.5 Mbps of the available bandwidth, though as the throughput results (Figure 4.19) show, the premium traffic is policed in the router and forced to 3 Mbps. In times of congestion, which starts at best-effort-generated traffic of 1.5 Mbps (and occurs only at the switch), only best-effort traffic experiences packet loss. Premium traffic eventually experiences a steady packet loss of about 40%, which is the excessive throughput, policed by the router (Figure 4.20). The total throughput on the link between the router and the sink is not more than 7.5 Mbps, albeit the abundant throughput generated for the best-effort service. This is because, on the link between the switch and the router, this bandwidth is utilized by excess premium traffic, which is dropped in the router. There are approximately 2 Mbps of available bandwidth and thus the network is underutilized.

Figures 4.21 and 4.22 show the performance results for the experiment where the three flows are assigned to the three different medals of the olympic service. Figure 4.21 shows the throughput of every flow for injected traffic that varies between 1 and 4 Mbps. The x-axis measures the generated throughput of one flow, and all the flows always generate the same throughput. Gold and silver flows always receive the generated bandwidth. When congestion occurs, which is for all values over 3.2 Mbps, bronze traffic is the only one that cuts back its bandwidth and experiences packet loss (Figure 4.22). It is certain that for values over 5 Mbps of traffic for each flow, the bronze would be led to starvation and silver would start experiencing packet loss. Table 4.5 lists the mappings between traffic types and PHBs used for this experiment.

Table 4.5. User-priorities to PHBs mappings for second scenario.

User-priority	Queue	Ethernet Traffic Type	PHB
0	1	Best Effort	AF13
1	2	Background	AF12
3	3	Excellent Effort	AF11

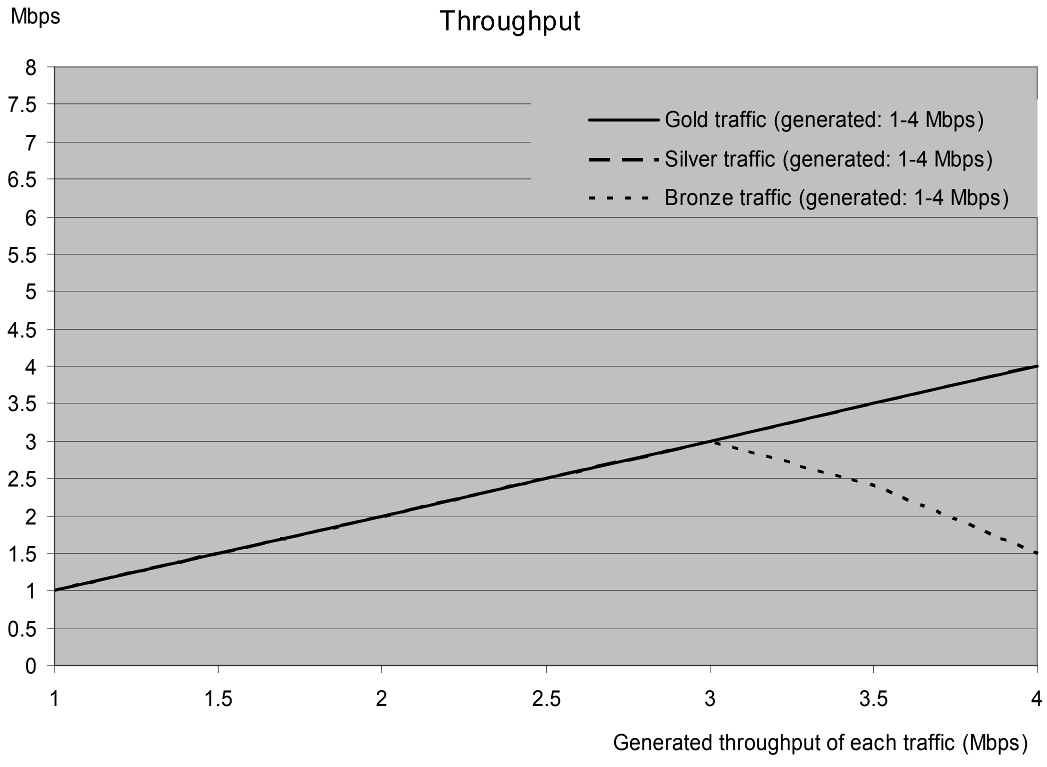

Figure 4.21. Throughput results for third experiment of case 3.

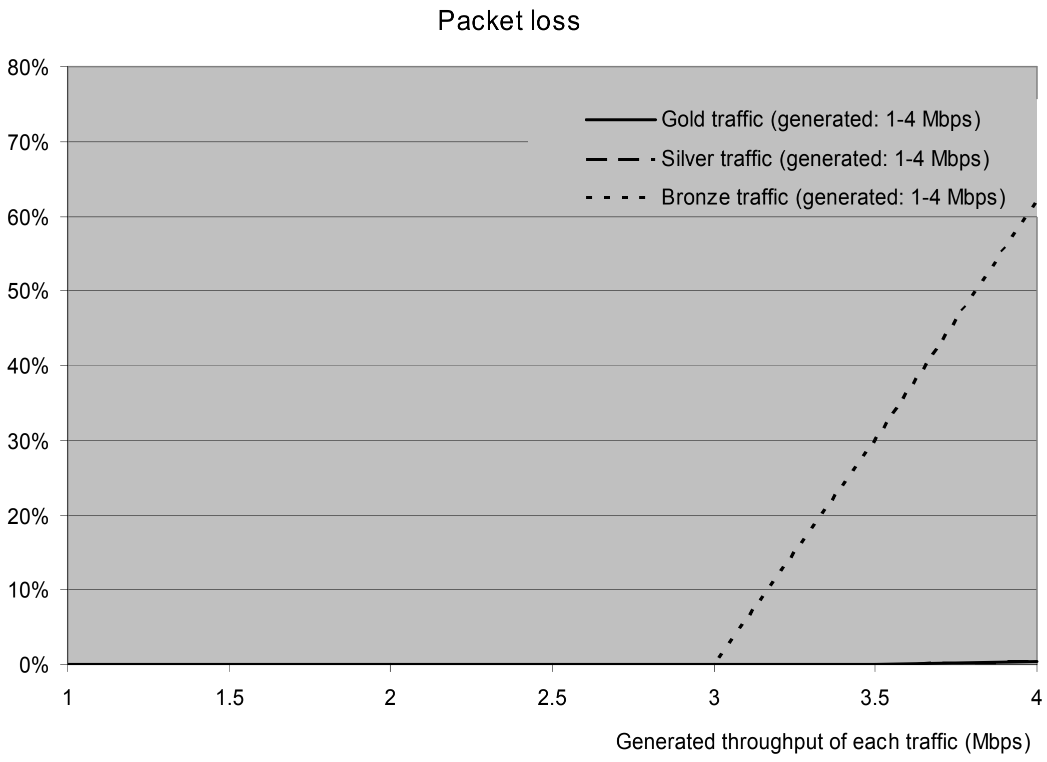

Figure 4.22. Percentage of packet loss for third experiment of case 3.

This case produces examples showing that in a DiffServ domain with no homogenous per-hop behavior, the desired performance results are not met. They can be altered significantly and, in the overall performance of a real world network, the impact would be unpredictable. Decisions about the exact allocation of resources inside a DiffServ domain constitute an important task that is expected to be performed by in-depth analysis of statistical traffic profiles to design an efficient network. The impact of the Ethernet switch performing strict prioritizing between the different traffic services alters the desired traffic performance, yielding the design for QoS support useless.

4.5 Case 4: DiffServ domain with one DiffServ-aware router and a DiffServ-aware Ethernet switch

In the last case, the Ethernet switch is configured to implement the same per-hop behaviors as the router. The results presented are expected. It was already anticipated that in a carefully designed and provisioned DiffServ domain providing the same PHBs in all the network nodes, the traffic services semantics would be preserved. This is what the DiffServ architecture is based on. There is no cross-traffic in the router and thus no congestion on the link between the switch and the router. The end-to-end service the three flows experience is mainly determined by the traffic control of the switch. This case also helps us to research the implications by integrating the Ethernet switch into the DiffServ architecture.

The Ethernet switch monitors, classifies, polices, and schedules traffic the same way the router does in the first case. If traffic were measured at the ingress interface of the router, results would yield the same plots as those presented in the first case. Traffic control in the router preserves the same service performances along the link between the router and the destination host. This is verified by the results shown in the following figures. Al plots show almost identical results as those presented in the first case.

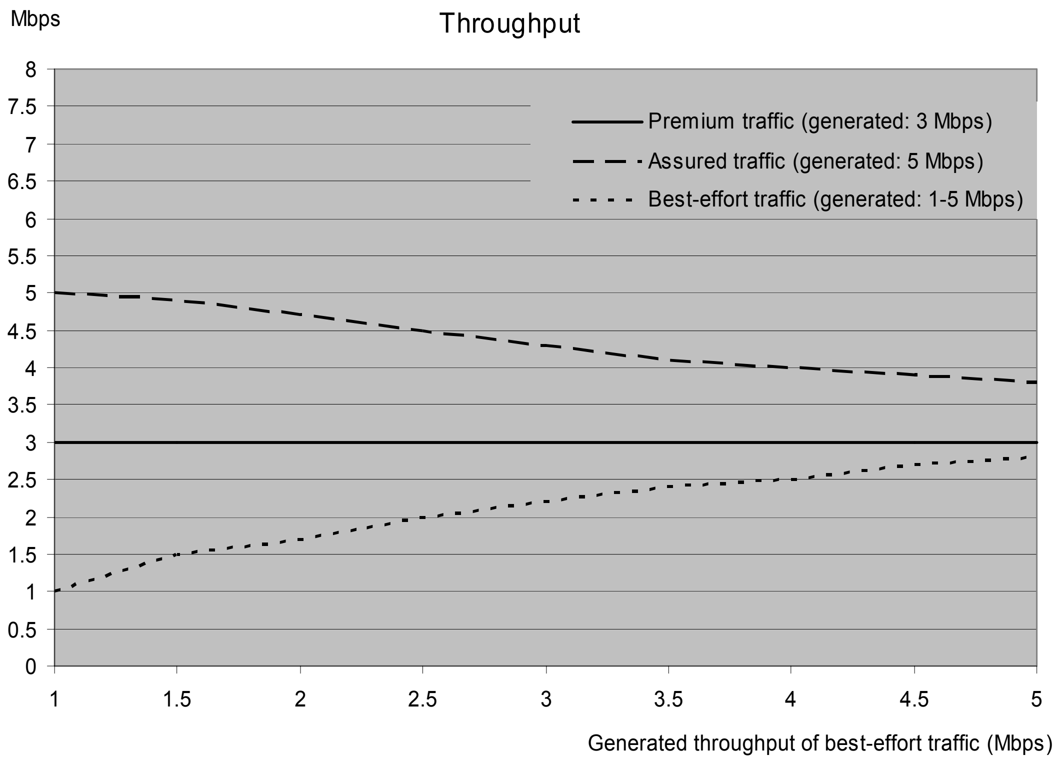

Figure 4.23. Throughput results for first experiment of case 4.

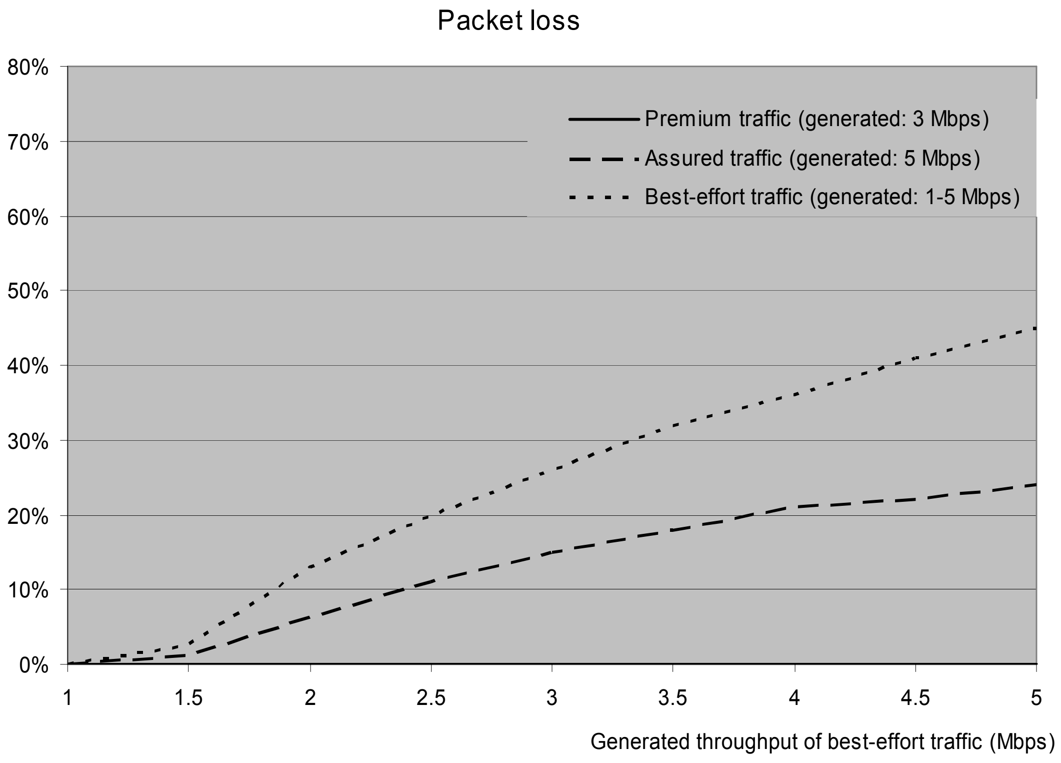

Figure 4.24. Percentage of packet loss for first experiment of case 4.

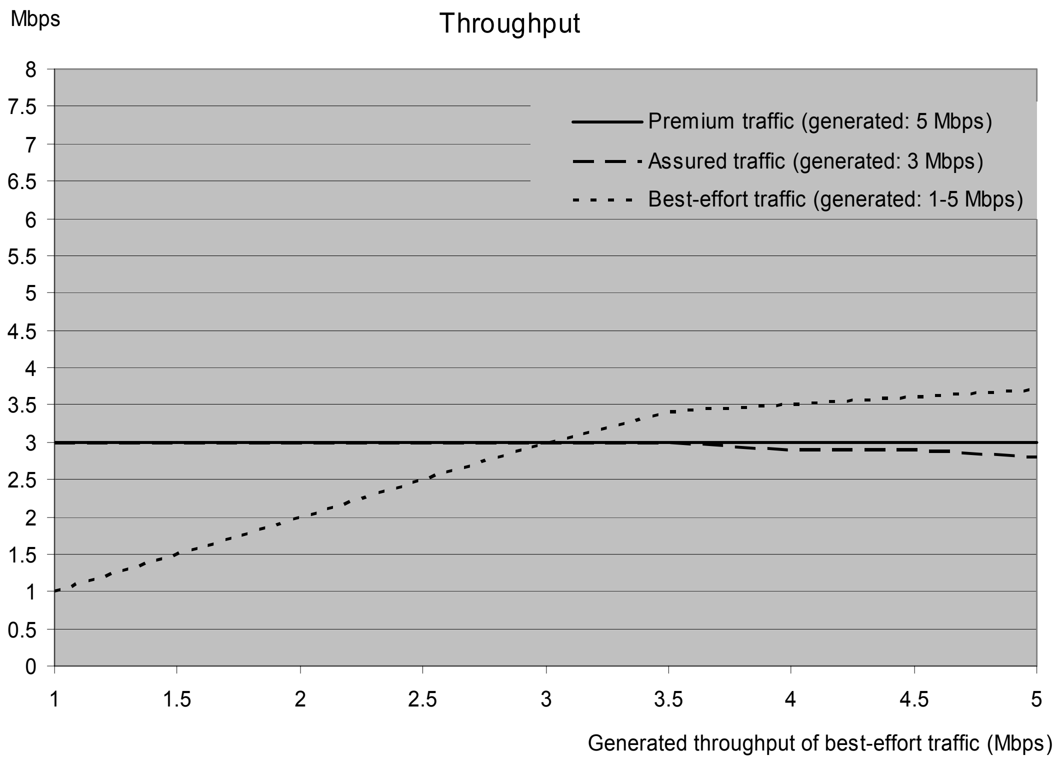

Figure 4.25. Throughput results for second experiment of case 4.

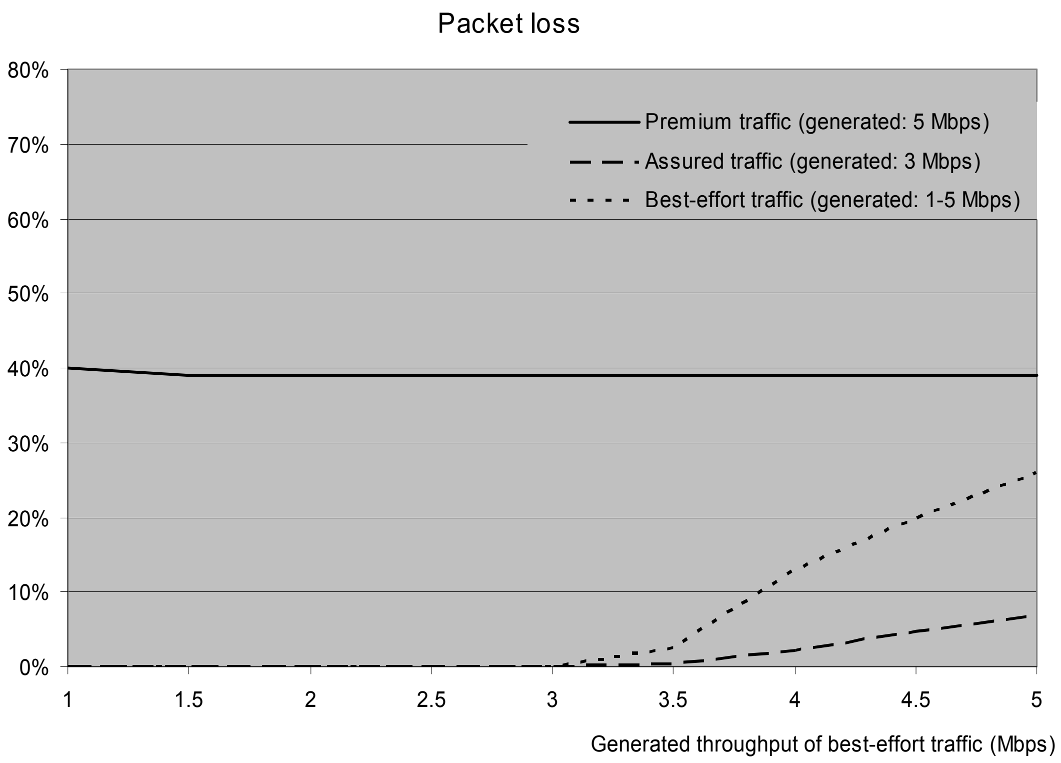

Figure 4.26. Percentage of packet loss for second experiment of case 4.

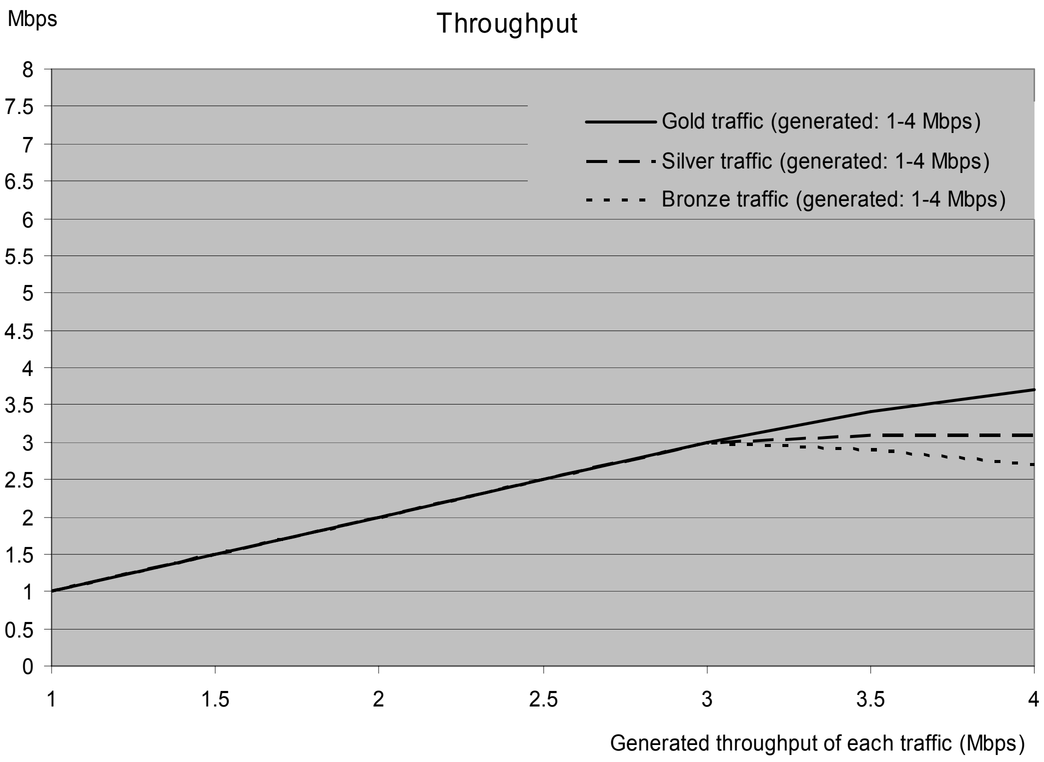

Figure 4.27. Throughput results for third experiment of case 4.

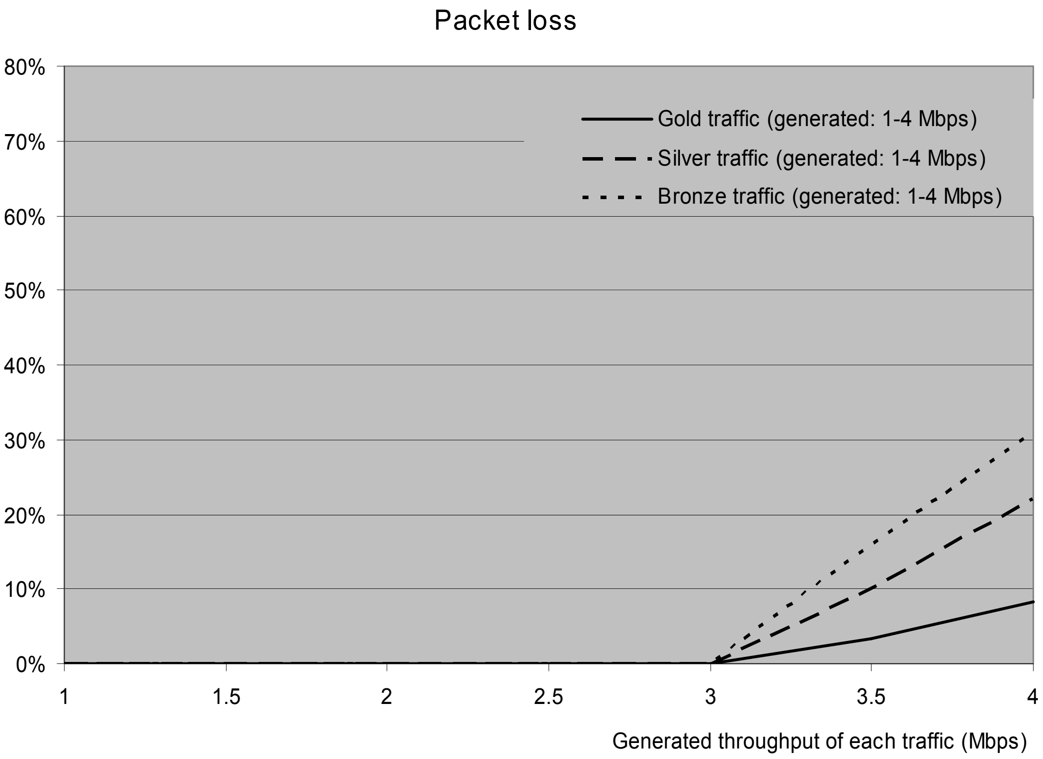

Figure 4.28. Percentage of packet loss for third experiment of case 4.

The results of the first experiment where 3 Mbps of premium, 5 Mbps of assured, and 1 – 5 Mbps of best-effort traffic are generated are shown in Figures 4.23 and 4.24. Figure 4.23 shows the throughput and Figure 4.24 shows the percentage of packet loss for the three flows. The traffic service semantics are all preserved, just as they were in the first case. Premium traffic is guaranteed its throughput of 3 Mbps even in case of congestion, and when congestion occurs, assured traffic experiences better service than best-effort traffic.

Figure 4.25 shows the throughput and Figure 4.26 shows the percentage of packet loss for the second experiment. In the second experiment, 5 Mbps of premium traffic and 3 Mbps of assured traffic are generated. Generated best-effort traffic varies between 1 and 5 Mbps. The same conclusions are derived from these plots as well. Premium is guaranteed 3 Mbps of bandwidth, premium excess traffic is policed (dropped), and assured traffic is much less affected than best-effort traffic in times of congestion.

The results of the third experiment follow. In this case, olympic service is implemented where a gold, a silver, and a bronze flow are generated with the same rate that varies between 1 and 4 Mbps. As the throughput in Figure 4.27 and the percentage of packet loss in Figure 4.28 show, the gold flow experiences the best performance and the silver experiences worse than the gold flow, but better than the bronze. The plots are again identical to the ones for the third experiment in the first case.

As already mentioned, the Ethernet switch in this case functions in the same way as the router. This means that it classifies incoming packets to traffic services according to the UDP destination port of the packet. This requires that the switch, a layer-2 device, be aware of the layer-3 protocol. This is undesired and breaks the layer separation law.

For the same reasons, the switch should not perform DiffServ marking on the outgoing IP packets, although this would be advantageous since it would not require a leaf router. In the next chapter, where a framework for integrating Ethernet switches in the DiffServ architecture is presented, we show how this problem can be tackled by incorporating the VLAN concept.

4.6 Summary

This chapter presented the experiments performed to investigate the implications of an Ethernet switch inside a DiffServ domain. It has been shown that for the services described in this document and for the cases studied, a network has to provide the same DiffServ functionality in all its network nodes, regardless of the layer in which they operate, to ensure proper service provision. Different services could have been provided by the testbed, different configurations and algorithms could have implemented the same services, different flows could have been experimented, all giving different numerical results. Nevertheless, it is believed that the conclusions would be the same: networks incorporating Ethernet switches cannot provide differentiated services without having the layer-2 devices performing service differentiation as well. Furthermore, the only available quality-of-service support for Ethernet, IEEE 802.1p, would not be adequate.

The results in the third case proved how service results may be altered by not implementing the same PHBs in all the nodes of the DiffServ domain. This alteration could very well have significant consequences to the ultimate goal of a QoS architecture in a network, which is efficiency and productivity. It is still not clear what services would be useful to applications, how these services should be implemented, or even what are the exact needs for traffic services of applications. The differentiated services architecture is really mature by now and finds itself at a critical point, we believe, where the usefulness of its adoption will be evaluated by gradually being employed in research, academic, and private networks. What is important for a network providing QoS is the service definition and the correct configuration of system parameters such as percentage of bandwidth allocation to the premium service, bandwidth reserved for assured, and so on. In a testbed like ours, it is easy to try different numbers, but in a real network, these numbers are difficult to get. They should be derived from careful study of statistical traffic profiles and the desired overall result. Alterations in the functionality of the system made by layer-2 devices, can yield undesired consequences in network behavior, leading to unpredictable results that yield the QoS infrastructure useless and a waste of resources.

By designing and building a Linux DiffServ domain, we argue that this problem can be overcome by having Ethernet switches perform the same QoS behavior as DiffServ-aware routers. In our testbed, all hosts and network nodes are comprised of Linux machines, so it was simple to configure the Linux-based switch to perform the same DiffServ functionality in the last case. The kernel supports the same functionality as in the Linux-based router and it was only a matter of reproducing the same configuration. Most equipment manufacturers produce both switches and routers, usually consisting of the same real-time operating system and other software modules. There is also the issue of managing and administrating a DiffServ domain, a task that can get much easier if Ethernet switches are integrated in the DiffServ architecture. The next chapter describes a framework for doing so.

Chapter 5

5 Suggested Framework

5.1 Overview

In the last chapter we argued the necessity for having the layer-2 devices implementing the same per-hop behaviors as the layer-3 devices inside a DiffServ domain. We even proved the feasibility and success of the argument in the modular and extensible Linux environment. However, the solution we presented earlier is not flawless. We assumed that the switch could classify traffic according to IP header values, which violates the layers' separation law. A solution to this problem is given by a model based on VLANs that we describe in this chapter. The solution proposed is proven to offer more advantages than those that originally motivated us to consider it. DiffServ and VLANs are both called aggregate mechanisms because they classify all traffic into a finite number of priority classes. When trying to bring DiffServ down to layer 2, it makes perfect sense to use an existing mechanism to apply aggregation to the Ethernet level and then integrate the two abstractions of aggregation (VLANs and DiffServ) together. First, this yields a clearer service-provisioning scheme, a network that is simple to administrate, and a straightforward business model. Second, the use of VLANs yields a model easy to describe and the model does not need any extra functionality by itself.

In this chapter, we present our model and apply it in our testbed for the same experiments we examined before. The first step we take toward this is to introduce VLANs in our testbed.

5.2 VLANs in the domain

The first step is to set up the Ethernet switch in our topology (Figure 5.1) to act as a VLAN-aware switch and assign every source host to a different VLAN. Actually, the switch does not assign hosts to VLANs, rather it assigns its Ethernet interfaces to VLANs. Thus, the interface connected to source1 host (eth1) is assigned to VLAN 1, the interface connected to source2 host (eth2) is assigned to VLAN 2, and the interface connected to source3 host (eth3) is assigned to VLAN 3 (Figure 5.2). These interfaces could have been connected to a physical LAN consisting of several hosts or even to another switch.

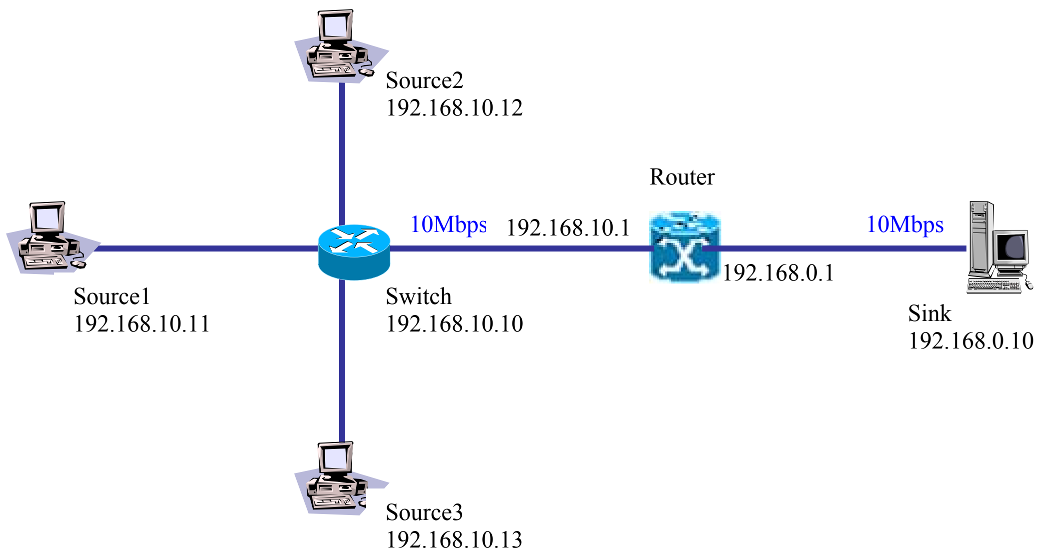

Figure 5.1. Testbed topology.

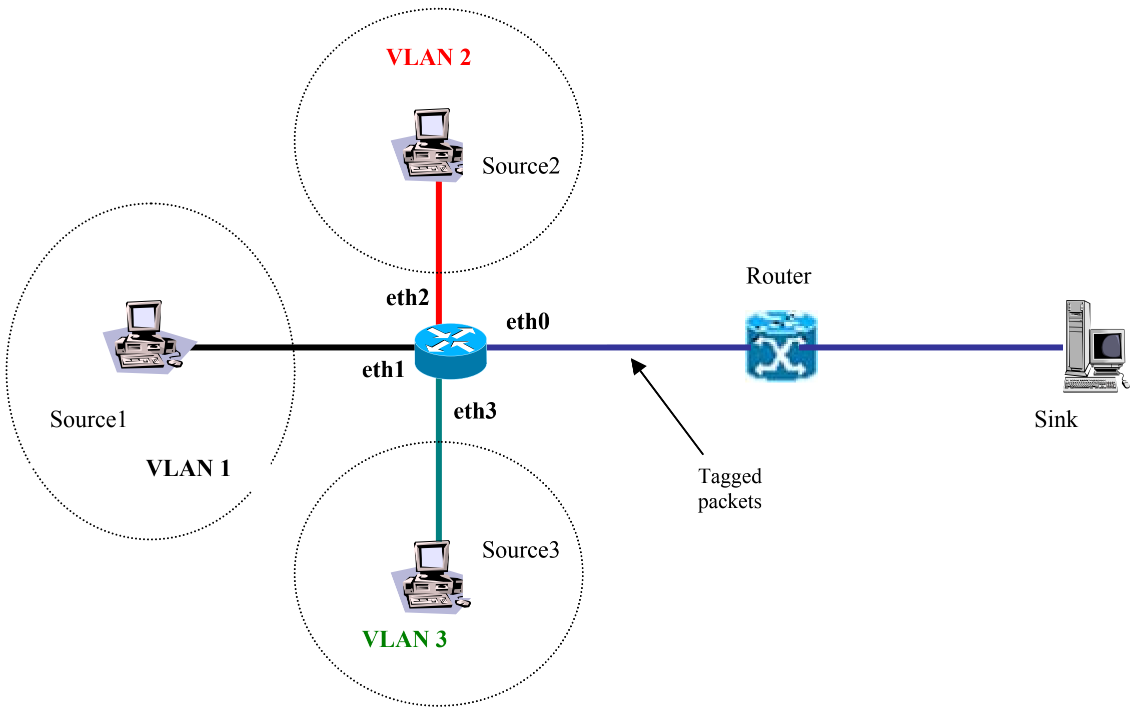

Figure 5.2. VLANs in testbed.

The Ethernet switch will not forward packets between the three mentioned interfaces. A packet received from interface eth1 will only be forwarded to the interface connecting the switch to the router (eth0) and will be tagged as belonging to VLAN 1. Respectively, packets from interface eth2 will only be forwarded to the router and tagged as belonging to VLAN 2, and packets from interface eth3 will only be forwarded through eth0 and tagged as belonging to VLAN 3. In the reverse direction, packets received by the eth0 interface will have been tagged as belonging to VLAN 1, 2, or 3. They will be forwarded only to the corresponding Ethernet interface, the one assigned to VLAN 1, 2, or 3, respectively. Source3 host will never see any packets coming from or destined to VLAN 2 and so on. To get a better understanding of the benefits of VLANs, we should look at a less simplistic topology, such as the one in Figure 5.3.

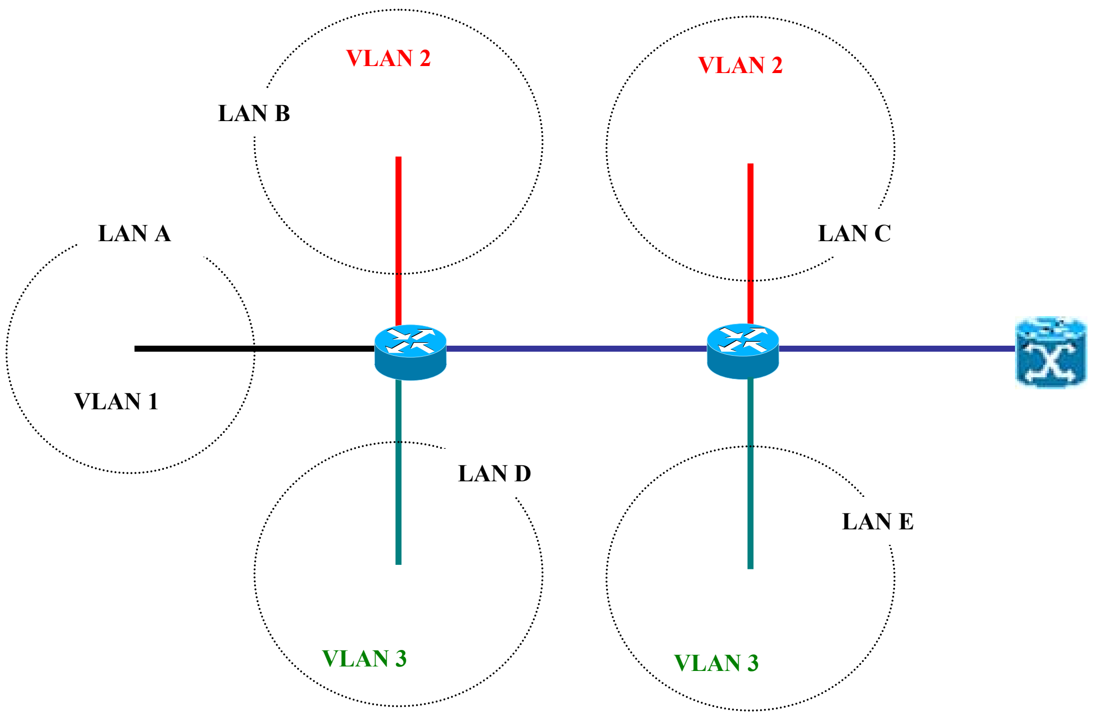

Figure 5.3. VLANs in a larger topology.

Every circle in Figure 5.3 is a LAN consisting of one or more hosts, which all belong to the same VLAN. Every Ethernet packet the switches need to forward is either coming from an access link assigned to a VLAN or contains a VLAN tag that assigns the packet to a VLAN. In both cases, the switch knows exactly to which interfaces to forward the packet: either to an access interface assigned to the same VLAN or to a trunk link configured to tag the same VLAN, or both. This results in faster switching and isolation of VLANs. For a more realistic topology, where switches will connect to a lot more than four links, this is really important. Communication between LANs belonging to the same VLAN is expected to be more frequent than communication between VLANs. The mechanism of VLANs enables communication between LANs belonging to the same VLAN without polluting the rest of the LANs. Thus, the broadcast domain of a VLAN can be much larger than the collision domain of a LAN. Furthermore, changes made in the topology of the network do not impose any serious administration burden. If, for example, LAN B swaps physical places with LAN E, the only change necessary is to assign the interfaces of the switches to the respective VLANs. There is no need for changes in the participating hosts.

In IEEE 802.1Q, which is the standard specifying VLANs, the eth1, eth2, and eth3 interfaces are called access links and the eth0 interface is called a trunk link. In our case, only packets going through the trunk link are VLAN tagged. This tag is added to the Ethernet header, as the following figure shows. It consists of a two-byte code equal to 0x8100 that indicates the existence of the VLAN tag inside the Ethernet header, a three-bit user-priority field, a one-bit field called CFI, and a 12-bit field for the VLAN identifier (ID). We have already seen the user-priority field. The VLAN ID indicates the virtual LAN the packet in question belongs to. The 12-bit field yields up to 4095 possible VLANs. A VLAN ID of zero would indicate that the Ethernet frame does not belong to any VLAN.

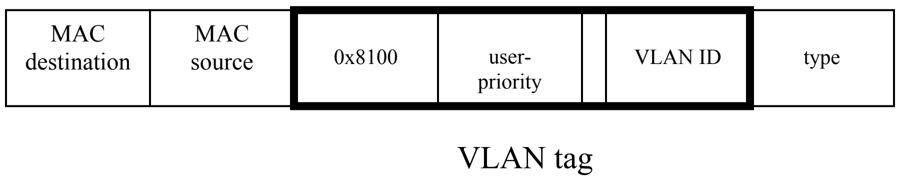

Figure 5.4. VLAN-tagged Ethernet header.

Going back to our testbed topology, there is one more important item that should be noted. The router in the network also needs to be able to understand and create

VLAN-tagged Ethernet frames. It does not need to perform VLAN switching, but it has to be able to extract the VLAN ID of an Ethernet frame and respectively insert a VLAN tag in an outgoing packet. This applies only to its interface toward the switch; the interface connecting the router to the sink host does not have to be VLAN aware.

5.3 VLANs in DiffServ

In our new VLAN-aware domain, every packet processed by either the switch or the router is associated with a VLAN ID; either the packet is VLAN tagged, containing a VLAN ID in the extended Ethernet header, or a VLAN ID is assigned to the interface the packet came from. We have already explained how the VLAN ID results in a VLAN classification for every packet used for VLAN switching. In our suggested model, described in this section, we extend the semantics of the VLAN ID: the VLAN ID of a packet tells each hop, switch, or router, not only in what VLAN it belongs , but also what traffic service should be assigned to the packet.

Looking at our VLAN-aware domain and the DiffServ provision of the fourth case of the last chapter, we realize that only one change is needed to realize the idea above. The classification rules need to change for the switch and the router. Both the switch and the router have to classify packets to traffic services according to the VLAN ID of the incoming packet. Both the switch and the router are eligible to read and set the VLAN ID of every packet.

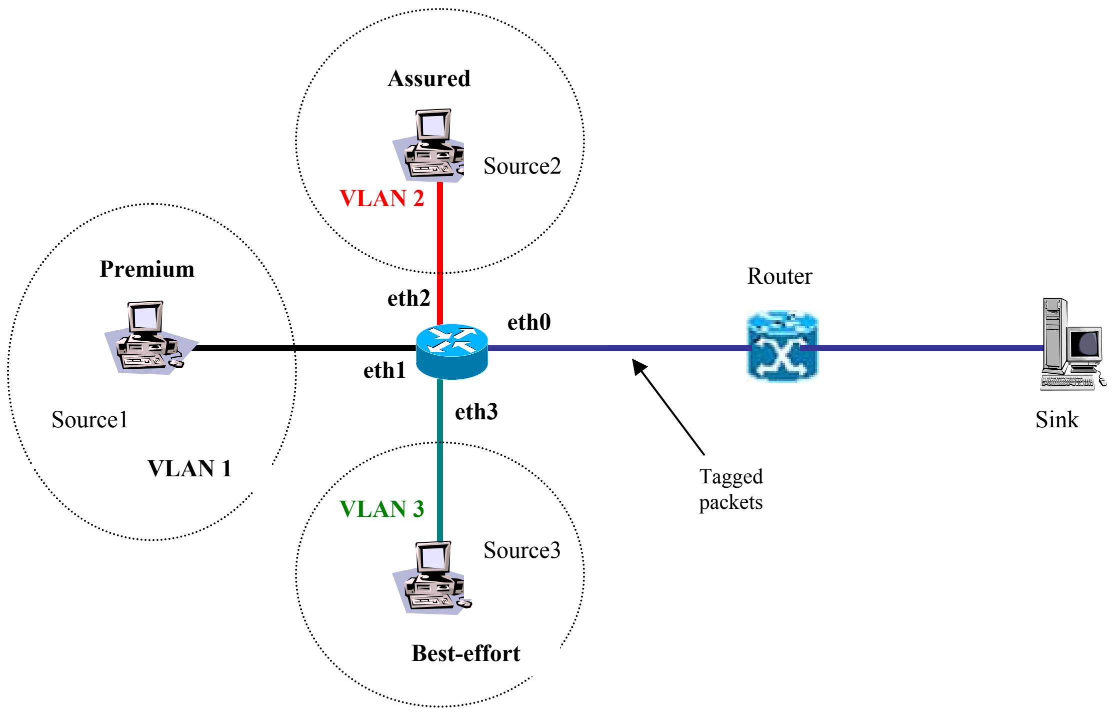

Figure 5.5. VLANs assigned to traffic services.

Table 5.1. VLAN to traffic service mappings

VLAN	Service Assigned
1	Premium
2	Assured
3	Best-effort

The model for our first and second experiments, where the DiffServ domain provides premium, assured, and best-effort traffic, is illustrated in Figure 5.5. The premium service is assigned to VLAN 1, the assured service is assigned to VLAN 2, and the best-effort service is assigned to VLAN 3 (Table 5.1). These high-level mappings are common for both the switch and the router.

The switch, on its ingress interfaces, classifies incoming packets to traffic services according to the VLAN ID of every packet. It also performs policing. These ingress rules result in an internal DSCP marking of the packets, which will be used on the egress interface. Table 5.2 lists these internal mappings. Packets from VLAN 1 that conform to the profile service are marked with a DSCP of 0x2e. Those packets that do not conform are dropped. Packets from VLAN 2 are marked with an internal DSCP of 0x0a as long as they conform to the service profile. If they do not, they are marked with the 0x0c DSCP, the same code used to mark packets from VLAN 3. The packets are then handed out to the forwarding process, which implements the VLAN switching and decides on what interface to forward the packets. The egress interface marks outgoing packets with the corresponding VLAN ID, but first it implements the per-hop behavior that corresponds to the internal DSCP of the packet. Packets internally marked with a DSCP of 0x2e experience expedited forwarding, packets marked with 0x0a experience assured forwarding with a high assurance, and packets marked with 0x0c experience assured forwarding with less assurance.

Table 5.2: Internal VLAN to DSCP mappings for both switch and router.

VLAN	Profile	Service Assigned	PHB	DSCP
1	< 3Mbps	Premium	EF	0x2e
	> 3Mbps	dropped	-	-
2	< 3Mbps	Assured	AF11	0x0a
	> 3Mbps	Best-effort	AF12	0x0c
3	-	Best-effort	AF12	0x0c

The switch in this case acts as a leaf router, but not entirely. It performs the same functionality with a leaf router in the sense that it classifies, polices, and differentiates traffic. It does not perform reclassification. A leaf router treats out-of-profile assured traffic as best effort and also marks this traffic as best effort. In this way, a core router would not need to do any policing. The switch in our model should not do so. Reclassification requires changing the VLAN ID of the packet, which is not accepted, since it would break the semantics of the VLAN ID. Thus, employing our model does not necessarily yield the switches as leaf nodes and all the routers as core routers. Even though this might be the case in some situations –for example for services that do not require reclassification such as premium or olympic service– it is not our goal to substitute leaf routers with core routers and move the mechanisms of the leaf router to the switch. Our model is based on "copying" the functionality of the leaf DiffServ router to the switch, thus ensuring that the QoS semantics are preserved in the presence of the Ethernet switch in the domain. A switch and a router are inherently different devices and thus should be treated as different. Our model does not treat Ethernet switches as leaf routers; rather it provides a means of including Ethernet switches in the DiffServ process.

Nevertheless, the fact remains that the VLAN ID is universal. Both switches and routers can base their classifications according to the same properties. Furthermore, VLANs are assigned by the switch and these assignments are preserved throughout the domain.

The router will employ the same functionality as the one described for the switch. The difference here lies in the fact that the forwarding process of the router will not base its decisions on the VLAN ID of the packet, but on the destination IP address of the packet. Packets will be policed and classified on the ingress interface, an internal DSCP will be assigned to them, and the traffic control on the egress interface will provide the corresponding service to the internal DSCP per-hop-behavior.

What has been described so far can easily be extended to a larger topology or for different services provisioned by a DiffServ domain. The VLAN-to-traffic service mappings for the olympic service are shown in Table 5.3. In a different or larger topology, the same services could be provided with no further modifications to our model.

Table 5.3. VLAN to traffic service mappings for olympic service.

VLAN	Service Assigned
1	Gold medal
2	Silver medal
3	Bronze medal

So far, our model has been described in the context of our testbed and the Linux machines it consists of. We will generalize the model by putting forth four requirements of the model and a formal description. These requirements must be met by any domain that wants to extend the DiffServ mechanisms into Ethernet switches using the suggested model. All four requirements are met by our Linux-based DiffServ domain.

1) The switch must provide traffic control that may implement any of the PHBs the DiffServ domain offers.
2) The switch and the router must support VLAN tagging.
3) The switch must be able to classify incoming packets to traffic services with respect to their VLAN ID.

4) The router must be able to classify incoming packets to traffic services with respect to their VLAN ID.

Our argument in the last chapter confirms the first requirement. We have already seen that a domain where the switch does not provide the same traffic control as the router will fail to provide the promised traffic services. The second requirement is self-explanatory. Both the switch and the router need to support VLAN tagging to understand and create VLAN-tagged Ethernet frames. Even though a switch that supports VLAN tagging would most probably support VLAN switching as well, the latter is not essential for our model to work. The third and the fourth requirements are the quintessence of our model. The classification in both the switch and the router must be based on the VLAN ID of the packet.

The router, of course, must also provide traffic control that may implement any of the PHBs the DiffServ domain offers, but this is not a requirement of our model; it is a requirement of the DiffServ model, upon which our model relies. Also, it is worth saying that a system that intends to act either as a switch or as a router should provide traffic control independently of the forwarding process. In Chapter 3, where we described the main principles of the path of a packet inside the Linux kernel, we saw that traffic control is independent of the forwarding process. The system needs to be able to bind traffic control to any of its interfaces, but traffic needs to be explicitly assigned for traffic control to be affected by those mechanisms.

Given the above requirements, and recapitulating what has been discussed so far, our model can be described as follows.

Inside a DiffServ domain, users –who in our case are hosts, but could in practice be anything else– are assigned to VLANs. VLANs are common for both switches and routers inside the domain, and each VLAN is assigned to a traffic service. A leaf router classifies packets to traffic classes with respect to their VLAN ID and employs traffic control on these classes according to the PHBs the domain provides and under the differentiated services architecture. The switch also classifies packets to traffic classes with respect to their VLAN ID and employs the same traffic control mechanisms as the leaf routers of the domain, with the exception that it does not set the DSCP of outgoing packets.

Any other requirements of our model that are not included in the above description are already included in the standard DiffServ model.

The experimental results from the VLAN case are not repeated here because they are exactly the same as the fourth case of Chapter 4. This was expected and verifies that the configuration is correct. The changes we have made in our testbed throughout this chapter do not affect the behavior of the domain. These changes illustrate the use of a scheme that integrates Ethernet switches into the DiffServ model in a proper manner (using existing protocols and without breaking any rules) and that is realistic and feasible since it was actually implemented in our Linux testbed.

5.4 Summary

The suggested model integrates Ethernet switches in the DiffServ architecture through the use of VLANs. Primarily, it ensures the coherency of the semantics of the traffic services provided by the domain. Furthermore, it does so in a seamless way, without defining any new protocols. The model is based on existing protocols and standards, making use of existing mechanisms to implement its own function.

This integration of DiffServ with VLANs actually has even more advantages than those that initially motivated us to consider it. It provides a straightforward and clear business model. This model is ready to bring differentiated services for different groups of users, VLANs, inside a private network. Network administrators that already use VLANs in their networks would acclaim the idea of providing different traffic services to each VLAN. Administrators who wish to provide differentiated services for the users of their networks could benefit from the idea of grouping users to VLANs and provision their networks with different traffic services for different VLANs.

Moreover, the fact that traffic control is independent of network control (switching and routing) inside the Linux kernel makes it possible to use the same exact code for traffic control in both cases (Linux as a switch, Linux as a router). This should be the case for product devices too. The same manufacturers usually produce both switches and routers and would usually use the same operating system for both lines of products and would also use much of the same software. Moreover, many devices by now may have real-time Linux kernels. The industry implementing DiffServ support for its products would probably use the example of Linux kernels and ns2 of providing large modularity and extensibility to their software code.

It seems that recently, many network designers and administrators have resorted to having Linux machines undertaking the role of an Ethernet switch or a router in their networks. The flexibility and the richness of the Linux kernel allow them to provide efficient and very cheap solutions. At a time when some small companies take a while to substitute a broken Ethernet switch, the network administrator can substitute the broken switch with an old and unused personal computer running Linux. They can even use the more sophisticated mechanisms of the latest Linux kernels to provide services for multiple customers, employing VLAN, NAT, and more. They also can use the advanced security mechanisms of the Linux kernels and build powerful firewalls. There are some who are already doing all that, and these administrators are the first who could profit from using the integration of the Ethernet switches in the DiffServ architecture. Actually, in a way, they might already been doing so. Network administrators who base their networks on Linux machines sometimes also provide traffic control, such as load balancing or prioritizing. This is not necessarily done under the DiffServ scheme, but always, we believe, the network designer treats switches and routers as equivalent hops.

In theory, though, this integration is not yet formulized. The DiffServ standards and any of the Ethernet-related specifications do not take into account such integration. We argue that this integration is extremely simple in concept and it needs only to be accepted. In practice, this integration will come one way or another. We believe that our work gives a clear picture of the problem and its solution, and we argue that our model formalizes this integration, which can expedite the use of DiffServ in intranets.

Chapter 6

6 Conclusions

The need for QoS in the Internet is well known and proven; there are many applications and services that can benefit from QoS and, along with the ever increasing bandwidth, there will be virtually no limit to what the Internet can incorporate. However, the actual deployment of a global QoS framework in the Internet is far from simple. The design of the Internet has made its revolutionary growth possible but holds back deployment of new network services such as QoS. All the proposed work for QoS in the Internet has met with difficulties, but it is a fact that QoS will be part of the Internet in the future –it is only a question of how. The differentiated services architecture stands as the most promising solution for QoS in the Internet for the near future. It is not as fine grained as might be desired, but it can surely provide quality of service in a feasible, scalable way.

DiffServ does not come easy either. Despite the fact that DiffServ has been around for quite some time, it still has not been deployed commercially. The services offered by a DiffServ network still need to be ratified, the mechanisms and the algorithms that will implement the framework still need to get more mature, and the administration of such a network definitely needs to become simpler for the operator and more straightforward for the end user.

But these are not DiffServ's only drawbacks. An additional problem is its lack of integration with layer-2 devices. It does not take into account Ethernet switches or any other layer-2 devices. Instead, it refers only to layer-3, IP, devices. However, Ethernet switches are an integral part of most networks today. Traffic services defined for DiffServ assume an underlying model where per-hop behaviors are common throughout all the nodes of the network. Our work has shown how harmful the effects of an oblivious-to-traffic differentiation Ethernet switch can be on the performance of the services provided by a DiffServ domain. It has also been shown that the only current formal means of extending QoS to layer 2, which is IEEE's 802.1p, is not adequate for a DiffServ domain.

We have argued this is a very important problem. Ethernet switches are everywhere, part of almost any network. Deploying DiffServ over networks with no adequate layer-2 QoS support could be disastrous. At a time when traffic services need to convince users and network administrators about their value, thus enabling DiffServ deployment, these services would be distorted and could result in a less effective network than a best-effort network. This could further discourage the use of DiffServ.

There is though a very simple –with respect to the extent of the problem– solution. This thesis has suggested that Ethernet switches, and layer-2 devices in general, should employ the same traffic control mechanisms as layer-3 devices, i.e., routers. Thus, per-hop behaviors would be common for all network nodes inside a DiffServ domain. There could be two antitheses to this. First is that functionality in Ethernet switches should be kept as simple as possible and second is that DiffServ is too tightly attached to the IP protocol.

We object to the first argument. We believe there is no reason for an Ethernet switch not to have all the functionality it needs. In practice, switches and routers are made of much the same hardware and software. Most of the time, a manufacturer would produce both type of devices, sharing parts of the same software. Thus, software developed to support DiffServ in a router could be easily re-used in an Ethernet switch. Actually, some manufacturers are already enhancing Ethernet switches with traffic control and DiffServ capabilities. As far as the second argument goes, this thesis has proposed a framework where Ethernet switches are seamlessly integrated in the DiffServ architecture.

The suggested model is based on the concept of VLANs. It integrates switches in DiffServ without breaking the layers' separation law and uses only existing protocols and mechanisms to do so. The VLAN ID is used as the universal (common to both switches and routers) identifier for traffic classification. All nodes classify traffic according to the VLAN ID and implement a common set of PHBs. Network administrators need to create only a mapping between a VLAN ID and a traffic service, which would be propagated among all the nodes of their domain.

This scheme yields a straightforward business model. The users of a network domain with similar QoS needs are grouped in a single VLAN, and every VLAN is assigned a specific traffic service. For example, a group of users receives best-effort service, whereas another group receives assured. Another group, which could consist of only one user, could be assigned the premium service. The domain could be a corporation that anticipates better productivity with the use of different traffic services or a commercial network that makes a profit by selling better services.

Furthermore, VLAN classification does not have to be done by Ethernet ports. This is the IEEE standard prerequisite, but a more sophisticated means of classification could be used. For example, a VLAN could consist of all the VoIP connections, where a connection is identified by the tuple: source address, source port, destination address, and destination port. Moreover, a distributed system in a private network could be assigned to the better-than-best-effort service. This means that using the VLAN IDs as the user identifier does not restrict at all who/what the user will be.

This thesis has made the following contributions:

- The design of a DiffServ domain has been presented. Four traffic services have been described in the context of this DiffServ domain.
- The DiffServ domain was implemented in a Linux testbed. This thesis provides all the necessary information for such an implementation, as well all the Linux-related configuration work required for this.
- The effects of a QoS-unaware Ethernet switch inside the domain have been studied. In particular, how the traffic services offered by the domain are affected is investigated. The results show that the Ethernet switch has made DiffServ impossible.
- The provisioning of the switch with IEEE 802.1p traffic control and how this can help has been researched. Again, the results show that the intended traffic service semantics break apart.
- The fact that the traffic service semantics can be preserved by having the switch incorporate the same traffic control mechanisms as the router is illustrated in the testbed.

- A model that seamlessly integrates Ethernet switches in the differentiated services architecture is suggested. The model is based on the concept of VLANs.
- The strong business model that the use of VLANs with DiffServ constitutes has been discussed.

Also,

- A modified version of the iperf program, suitable for a script process, has been created.
- The configuration for a Linux machine to act as a VLAN switch was presented.
- An automation procedure for conducting performance measurements in the domain has been developed, which can be repeated or modified for future use.

APPENDIX A – Testbed Software Provisioning

This section describes in detail the software provisioning of the Linux machines that comprise the testbed.

RedHat 7.3 Linux distribution, which is based on the 2.4.18-3 Linux kernel, is installed in all six machines. This version of the Linux kernel contains all the kernel-level software components for traffic control and DiffServ implementation that are necessary in our study. The kernel, however, needs to be reconfigured and rebuilt so the desired functionality is included in the runtime image of the kernel.

Rebuilding the kernel, the following kernel configuration options have to be enabled in the section "Networking options":

- Kernel/User netlink socket (CONFIG_NETLINK)
- Network packet filtering (CONFIG_NETFILTER)
- QoS and/or fair queuing (CONFIG_NET_SCHED)
- 802.1d Ethernet Bridging (CONFIG_BRIDGE)
- 802.1Q VLAN Support (CONFIG_VLAN_8021Q)

In the section "Networking options, QoS and/or fair queuing," all the configuration options should be enabled.

In the "Processor type and features" section, the "Symmetric multi-processing support" (CONFIG_SMP) option must be disabled when single processor machines are used. It has come to our attention that when this option is enabled for single processor machines, unpredictable problems can occur.

In the “Network device support, Ethernet (10 or 100Mbit) ” section, the “EtherExpress Pro/100 support” option (CONFIG_EEPRO100) must be enabled as a loadable module. Furthermore, the “/etc/modules.conf” file needs to be edited to force 10 Mbps half duplex speed for all Ethernet links.

Also, before rebuilding the kernel, the “ing-stats.patch” needs to be applied to the “sch_ingress.c” kernel file. This fixes a couple of bugs related to the ingress queuing discipline. Finally, the “vlan-eepro100.patch” needs to be applied to the “eepro100.c” kernel file. This patch enables the Ethernet driver to cope with frames with a maximum size of 1518 bytes, instead of 1514. The VLAN tagging support adds an extra field of four bytes in the Ethernet header; thus, for a full-size IP packet of 1500 bytes, the total maximum frame size, including Ethernet header and the VLAN tag, will be 1518 bytes. Neither of those patches, to the best of our knowledge, resides in any permanent Internet location.

Some of the configurations described above are necessary only for the Linux machine acting as a switch and the Linux machine acting as a router (e.g., traffic control capabilities). Other configurations are necessary for all the machines (e.g., eepro100 module). Nevertheless, in building a testbed like ours it is most often best to configure and build one kernel and then replicate it in all the machines.

The same goes for the configuration of the user-space programs of the Linux machines. The programs described next do not have to be installed in all machines, but creating a Linux image once and then replicating it for all machines makes the testbed setup easier. Furthermore, it is straightforward in terms of which machines need to have installed which particular programs.

The first user-space program we discuss is the *tc* (traffic control) program. This will probably be part of the original Linux installation, but it has to be updated to include DiffServ support. The program comes as part of the *iproute2* package, which can be found in [24]. The same reference indicates how the package should be built. Then, the newly created *tc* program should replace the original one, usually in the "/usr/sbin" directory. However, before the building process, a number of patches should be applied to the *tc* source code. These patches come as a part of the *tcng* (traffic control next-generation) package, which can be found in [26]. The package constitutes a new means of controlling the traffic control mechanisms of the Linux kernel. However, it also includes four patches for the old *tc* source code.

The bridging functionality of the Linux kernel needs to be configured and controlled by a user-space program. The program is called *brctl* and is included in the *bridge_utils* package, which can be found in [32]. The package comes in an *rpm* format and its installation is fairly simple. The *brctl* program is used to create instances of bridging entities inside the kernel, to add and remove interfaces to and from the bridges, and to report statistics and status information.

The user-space program that controls the VLAN tagging support in the Linux kernel is called *vconfig* and is part of the *vlan* package, which can be found in [33]. The same web site provides help for building the program and using it.

The programs described so far constitute the means for controlling Linux- kernel capabilities. Next, application-level programs used in our testbed are discussed. First, an *ssh* server needs to be installed and enabled in all machines. This is required for the automatic process of performing experiments and collecting data. The server is included in the RedHat distribution. Also, the *xgraph* program needs to be built and installed. The *xgraph* program is used to display experimental results on the fly.

The major tool for performing automatic experiments and collection of data is the *expect* program. This is a *tcl* program that enables scripting for interactive procedures. It comes in an *rpm* format and its installation is fairly simple. The rationale behind using this program is discussed in Chapter 3, and the scripts are presented in Appendix D.

Finally, the *iperf* program is installed in our testbed, as discussed in Chapter 3. The program and its documentation can be found in [28], though, as already mentioned, the program had to be modified to accommodate necessary changes for the automation of our experiments. The changes made have to do only with the server side of the program. By default, the program prints out on the console throughput and packet loss statistics. Since this is not adequate when the program runs as part of a script, it was modified to write statistics in files. The name of the files is passed as a command line argument to the program. Actually, one name is given to the program, which then creates two files with the same name, but with a different suffix. The letters "_bw" are appended to the given name for the file that keeps bandwidth statistics and the letters "_ls" are appended to the given name for the file that keeps packet loss statistics. The format of the output files is closely related to the *expect* scripts that process them to produce *xgraph* and *excel* input files. The modified source code, along with the detailed documentation of the changes, can be found in the electronic material that accompanies this thesis.

APPENDIX B – VLAN switching in Linux

The Linux kernel does not provide explicit VLAN support. It does provide, however, VLAN tagging support. There is a way to implement VLAN bridging in Linux by exploiting the features of the bridging module, which can run multiple instances of a bridging entity. To see how this is done, we first need to examine the VLAN tagging support of Linux.

The *vconfig* program creates logical interfaces on top of existing physical Ethernet interfaces. A VLAN ID (an integer between 1 and 4095) and a reference to an existing interface are given to the program and it creates a logical interface assigned to the given VLAN ID. The interfaces created by the *vconfig* program will be listed by the *ifconfig* program, along with the rest of the (physical) interfaces. The new interface takes its name from the name of the physical interface on which it was created and the VLAN ID, e.g., “eth0.1”. A physical interface accepts tagged packets only for those VLAN IDs for which a logical interface has been created. The received packets are then handed out to the kernel at the point where the logical interface is attached. Outgoing packets will be tagged with the VLAN ID associated with the logical interface they are leaving. These interfaces can be attached to a bridge or can be given an IP address. In any case, they look no different to the upper layers than the rest of the interfaces of the system.

IEEE 802.1D bridging support (Ethernet switching) can be controlled in Linux using the *brctl* program. The program allows the creation of multiple bridging instances, assumes the task of adding or removing interfaces to each bridging instance, and can control or monitor any other task related to Ethernet switching.

Three different bridging instances are created by the *brctl* program, named VLAN1, VLAN2, and VLAN3. Interface eth1 is attached to VLAN1, eth2 is attached to VLAN2, and eth3 is attached to VLAN3. On interface eth0, three logical interfaces are created: eth0.1, eth0.2, and eth0.3. The first one is attached to VLAN1, the second one to VLAN2, and the third one to VLAN3. This is depicted in the following figure. This results in the exact desired behavior: packets originating from source1 (connected to the switch with eth1) will be forwarded by the switch only on the trunk link, tagged with VLAN ID of 1. In the reverse direction, a packet tagged with a VLAN ID of 1 will be forwarded only to source1. The same goes for source2 and source3.

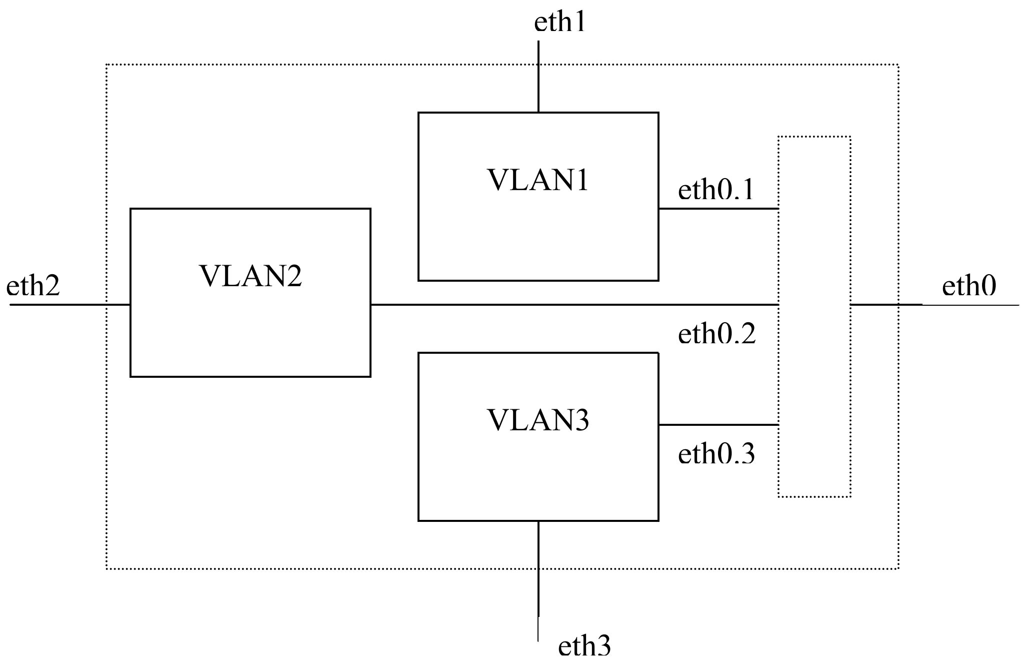

Figure B.1. Block diagram of bridging entities in Linux kernel.

Three logical interfaces for the same VLANs (eth1.1, eth1.2, and eth1.3) also have to be created on top of the ingress interface of the router, where tagged packets are received. Each of the interfaces is attached to the same IP address. For its forwarding process, the router does not care about the interface the packet came from. Nevertheless, in the reverse direction, the router needs to know the VLAN ID to mark every outgoing packet with and thus the logical interface to go out from. A set of routing rules must be given to the router. Hosts belonging to the same VLAN are expected to belong to the same IP subnetwork as well, creating a broadcast domain. Thus, the rules in the routers are expected to assign subnetworks to VLANs. In our case, we need to map only unique IP addresses to VLANs; packets destined to source1 are routed through the eth1.1 interface, packets destined to source2 are routed through the eth1.2 interface, and packets destined to source3 are routed through the eth1.3 interface.

APPENDIX C – Traffic control scripts

A user-space program, called *tc*, configures the traffic control mechanisms of the Linux kernel. Using this program requires in-depth knowledge of the traffic control structure and the elements involved. The general descriptions of the major mechanisms used in our work were described in Chapter 3. In Chapter 4, we explained how these mechanisms were used to drive the desired behavior. This section presents the shell scripts that repeatedly call the *tc* program to set the traffic control mechanisms for each case. There are different scripts for the router and the switch and there are different scripts for different services, but there are also some that are common.

C.1 Scripts for the first case

Only one script needs to be run to implement DiffServ functionality in the domain, though there is a different script for the experiments where the premium, the assured, and the best-effort services are offered and for the experiment where the olympic service is provided. For the former case the script is called "set-router1-all" and for the latter "set-router1-af". The number "1' in the filename refers to the fact that the router is the only node in the domain. The suffix "all" refers to the fact that almost all services are provided and the suffix "af" refers to the fact that the olympic service is provided by the AF PHB. These two scripts are quite similar. Both attach a filter on every ingress interface and mark the incoming traffic according to predefined patterns. The first script assigns UDP 6001 port traffic to premium service, UDP 6002 port traffic to assured, and UDP 6003 port to best-effort traffic. The second script assigns UDP 6001 port traffic to the gold medal, UDP 6002 port traffic to the silver medal, and UDP 6003 port traffic to the bronze medal. The other difference is that the first script installs meters (policers), whereas the second does not. In every case, the mechanisms mentioned so far are installed on the ingress interfaces, where traffic is filtered and marked internally with the corresponding DSCP value, as discussed in Chapter 4.

At the end, both scripts run the “set-egress” script, which configures the traffic control at the egress interface of the router. In every case and for every experiment, the same script configures the egress interface of the router. The same script also configures traffic control at the egress interface of the switch when DiffServ functionality is enabled on the switch. The configured functionality in the egress interface contains all the necessary mechanisms to implement any combinations of services studied in this work. The internal DSCP value of every packet, set by the ingress interface, indicates which mechanisms will be used in every case. For different cases of topologies and for different cases of services, the ingress rules will change, but the egress functionality remains the same.

The script “set-router1-all” follows:

```
#! /bin/sh
#
# use flag -x for echoeing commands
#
# file: set-router1-all
#

IN_DEV1=eth1
IN_DEV2=eth2
IN_DEV3=eth3


echo
echo Setting diffserv for router1
echo
```

```
# All packets are marked with a DSCP in the tcindex field that is
# going to be used on the egress side.
# The value set is the whole TOS. DSCP is TOS shifted by two
# EF  : 0xb8 (TOS) = 0x2e (DSCP)
# AF11: 0x28 (TOS) = 0x0a (DSCP)
# AF12: 0x30 (TOS) = 0x0c (DSCP)

##############  Ingress Side 1 (EF)  ##############

# Install the ingress qdisc on the ingress interface
tc qdisc add dev $IN_DEV1 handle ffff: ingress

# Add u32 filter
tc filter add dev $IN_DEV1 parent ffff: protocol ip prio 4 \
        handle 1: u32 divisor 1

# EF traffic (port 6001) is policed and is marked
tc filter add dev $IN_DEV1 parent ffff: protocol ip prio 4 u32 \
        match ip dport 6001 0xffff \
        police rate 3000kbit burst 90k drop \
      flowid 0xb8

##############  Ingress Side 2 (AF)  ##############

# Install the ingress qdisc on the ingress interface
tc qdisc add dev $IN_DEV2 handle ffff: ingress

# Add u32 filter
tc filter add dev $IN_DEV2 parent ffff: protocol ip prio 4 \
        handle 1: u32 divisor 1

# AF in-profile traffic (port 6002) is marked as assured
tc filter add dev $IN_DEV2 parent ffff: protocol ip prio 4 u32 \
        match ip dport 6002 0xffff \
        police rate 2500kbit burst 90K continue \
      flowid 0x28

# AF out-of-profile traffic (port 6002) is marked as best-effort
tc filter add dev $IN_DEV2 parent ffff: protocol ip prio 5 u32 \
      match ip dport 6002 0xffff \
      flowid 0x30

##############  Ingress Side 3 (BE)  ##############

# Install the ingress qdisc on the ingress interface
tc qdisc add dev $IN_DEV3 handle ffff: ingress

# Add u32 filter
tc filter add dev $IN_DEV3 parent ffff: protocol ip prio 4 \
        handle 1: u32 divisor 1

# BE traffic is marked
tc filter add dev $IN_DEV3 parent ffff: protocol ip prio 4 u32 \
        match ip dport 6003 0xffff \
        flowid 0x30
```

```
./set-egress
```

The script “set-router1-af” follows:

```
#! /bin/sh
#
# use flag -x for echoeing commands
#
# file: set-router1-af
#

IN_DEV1=eth1
IN_DEV2=eth2
IN_DEV3=eth3


echo
echo Setting diffserv for router1
echo


# All packets are marked with a DSCP in the tcindex field that is
# going to be used on the egress side.
# The value set is the whole TOS. DSCP is TOS shifted by two
# AF11: 0x28 (TOS) = 0x0a (DSCP)
# AF12: 0x30 (TOS) = 0x0c (DSCP)
# AF13: 0x38 (TOS) = 0x0e (DSCP)

##############  Ingress Side 1 (AF11)  ##############

# Install the ingress qdisc on the ingress interface
tc qdisc add dev $IN_DEV1 handle ffff: ingress

# Add u32 filter
tc filter add dev $IN_DEV1 parent ffff: protocol ip prio 4 \
        handle 1: u32 divisor 1

# AF11 traffic (port 6001) is marked
tc filter add dev $IN_DEV1 parent ffff: protocol ip prio 4 u32 \
        match ip dport 6001 0xffff \
      flowid 0x28

##############  Ingress Side 2 (AF12)  ##############

# Install the ingress qdisc on the ingress interface
tc qdisc add dev $IN_DEV2 handle ffff: ingress

# Add u32 filter
tc filter add dev $IN_DEV2 parent ffff: protocol ip prio 4 \
        handle 1: u32 divisor 1

# AF12 traffic (port 6002) is marked
tc filter add dev $IN_DEV2 parent ffff: protocol ip prio 4 u32 \
        match ip dport 6002 0xffff \
        flowid 0x30

##############  Ingress Side 3 (AF13)  ##############

# Install the ingress qdisc on the ingress interface
tc qdisc add dev $IN_DEV3 handle ffff: ingress

# Add u32 filter
tc filter add dev $IN_DEV3 parent ffff: protocol ip prio 4 \
        handle 1: u32 divisor 1

# AF13 traffic is marked
tc filter add dev $IN_DEV3 parent ffff: protocol ip prio 4 u32 \
        match ip dport 6003 0xffff \
        flowid 0x38


./set-egress
```

The script "set-egress" follows:

```
#! /bin/sh
#
# use flag -x for echoeing commands
#
# file: set-egress
#

OUT_DEV=eth0

AVGPKT=1000


##############  Egress Side  ##############

#### Main

# Attach a dsmarker on the egress interface
tc qdisc add dev $OUT_DEV handle 1:0 root dsmark indices 64

# Shift out the two last bits of the TOS field to
# get the DSCP field
tc filter add dev $OUT_DEV parent 1:0 protocol ip prio 1 \
      tcindex mask 0xfc shift 2 pass_on

# Add root CBQ qdisc on top of DSMARK
# All packets go to this qdisc
tc qdisc add dev $OUT_DEV parent 1:0 handle 2:0 cbq \
        bandwidth 10Mbit allot 1514 cell 8 avpkt $AVGPKT mpu 64

#### EF Traffic

# Add CBQ class for EF traffic
```

```
tc class add dev $OUT_DEV parent 2:0 classid 2:1 cbq isolated borrow\
        bandwidth 10Mbit \
        rate 3Mbit avpkt $AVGPKT prio 1 \
        allot 1514 weight 1 maxburst 20

# Classify packets to EF class
tc filter add dev $OUT_DEV parent 2:0 protocol ip prio 1 \
      handle 0xb8 tcindex classid 2:1 pass_on

#### AF Traffic

# Shift DSCP to get number of AF class
tc filter add dev $OUT_DEV parent 2:0 protocol ip prio 2 \
      tcindex mask 0xf0 shift 4 pass_on

# Add CBQ class for AF1 traffic
tc class add dev $OUT_DEV parent 2:0 classid 2:2 cbq isolated borrow\
      bandwidth 10Mbit \
      rate 3Mbit avpkt $AVGPKT prio 2 \
      allot 1514 weight 0.3 maxburst 20

# Classify packets to AF1 class
tc filter add dev $OUT_DEV parent 2:0 protocol ip prio 2 \
      handle 1 tcindex classid 2:2 pass_on

# Create GRED qdisc for AF1 class
tc qdisc add dev $OUT_DEV parent 2:2 gred setup DPs 3 default 2

# Classify packets to AF11 drop precedence
tc filter add dev $OUT_DEV parent 1:0 protocol ip prio 1 \
      handle 10 tcindex classid 1:111

# Configure DP 1 of GRED
tc qdisc change dev $OUT_DEV parent 2:2 gred limit 60KB min 15KB \
      max 45KB burst 20 avpkt 1000 bandwidth 10Mbit DP 1 \
      probability 0.1

# Classify packets to AF12 drop precedence
tc filter add dev $OUT_DEV parent 1:0 protocol ip prio 1 \
      handle 12 tcindex classid 1:112

# Configure DP 2 of GRED
tc qdisc change dev $OUT_DEV parent 2:2 gred limit 60KB min 15KB \
      max 45KB burst 20 avpkt 1000 bandwidth 10Mbit DP 2 \
      probability 0.3 \

# Classify packets to AF13 drop precedence
tc filter add dev $OUT_DEV parent 1:0 protocol ip prio 1 \
      handle 14 tcindex classid 1:113

# Configure DP 3 of GRED
tc qdisc change dev $OUT_DEV parent 2:2 gred limit 60KB min 15KB \
      max 45KB burst 20 avpkt 1000 bandwidth 10Mbit DP 3 \
      probability 0.6 \

#### BE Traffic
```

```
# Add CBQ class for BE traffic
tc class add dev $OUT_DEV parent 2:0 classid 2:3 cbq bounded \
      bandwidth 10Mbit \
      rate 3Mbit avpkt $AVGPKT prio 2 \
      allot 1514 weight 0.3 maxburst 20

# Classify packets to BE class
tc filter add dev $OUT_DEV parent 2:0 protocol ip prio 1 \
      handle 0 tcindex classid 2:3 pass_on
```

C.2 Scripts for the second case

For this case, the scripts “set-router2-all” and “set-router2-af” need to run in the router for the corresponding cases of services. These scripts are different from the respective “set-router1-all” and “set-router1-af” in that the filters and policers are installed on the single ingress interface of the router.

The script “set-router2-all” follows:

```
#! /bin/sh
#
# use flag -x for echoeing commands
#
# file: set-router2-all
#

IN_DEV=eth1

echo
echo Setting diffserv for router2
echo


# All packets are marked with a DSCP in the tcindex field that is
# going to be used on the egress side.
# The value set is the whole TOS. DSCP is TOS shifted by two
# EF  : 0xb8 (TOS) = 0x2e (DSCP)
# AF11: 0x28 (TOS) = 0x0a (DSCP)
# AF12: 0x30 (TOS) = 0x0c (DSCP)


# Install the ingress qdisc on the ingress interface
tc qdisc add dev $IN_DEV handle ffff: ingress
```

```
# Add u32 filter
tc filter add dev $IN_DEV parent ffff: protocol ip prio 4 \
        handle 1: u32 divisor 1

# EF traffic (port 6001) is policed and is marked
tc filter add dev $IN_DEV parent ffff: protocol ip prio 4 u32 \
        match ip dport 6001 0xffff \
        police rate 3000kbit burst 90k \
        drop flowid 0xb8

# AF in-profile traffic (port 6002) is marked as assured
tc filter add dev $IN_DEV parent ffff: protocol ip prio 4 u32 \
        match ip dport 6002 0xffff \
        police rate 2500kbit burst 90K continue \
      flowid 0x28

# AF out-of-profile traffic (port 6002) is marked as best-effort
tc filter add dev $IN_DEV parent ffff: protocol ip prio 5 u32 \
      match ip dport 6002 0xffff \
      flowid 0x30

# BE traffic is marked
tc filter add dev $IN_DEV parent ffff: protocol ip prio 4 u32 \
        match ip dport 6003 0xffff \
        flowid 0x30

./set-egress
```

The script “set-router2-af” follows:

```
#! /bin/sh
#
# use flag -x for echoeing commands
#
# file: set-router2-af
#

IN_DEV=eth1

echo
echo Setting diffserv for router2
echo


# All packets are marked with a DSCP in the tcindex field that is
# going to be used on the egress side.
# The value set is the whole TOS. DSCP is TOS shifted by two
# AF11: 0x28 (TOS) = 0x0a (DSCP)
# AF12: 0x30 (TOS) = 0x0c (DSCP)
# AF13: 0x38 (TOS) = 0x0e (DSCP)
# Install the ingress qdisc on the ingress interface
tc qdisc add dev $IN_DEV handle ffff: ingress

# Add u32 filter
tc filter add dev $IN_DEV parent ffff: protocol ip prio 4 \
        handle 1: u32 divisor 1

# AF11 traffic (port 6001) is marked
tc filter add dev $IN_DEV parent ffff: protocol ip prio 4 u32 \
        match ip dport 6001 0xffff \
        flowid 0x28

# AF12 traffic (port 6002) is marked
tc filter add dev $IN_DEV parent ffff: protocol ip prio 4 u32 \
        match ip dport 6002 0xffff \
      flowid 0x30

# AF13 traffic (port 6003) is marked
tc filter add dev $IN_DEV parent ffff: protocol ip prio 4 u32 \
        match ip dport 6003 0xffff \
        flowid 0x38

./set-egress
```

C.3 Scripts for the third case

In this case, besides setting DiffServ functionality for the router as in the last case, traffic control is also configured for the switch. This is done by the “set-switch-prio” script, which configures strict prioritizing in the switch. In particular, it assigns the highest priority to traffic destined to port 6001, next priority for traffic destined to port 6002, and finally, least priority for traffic destined to port 6003.

The script “set-switch-prio” follows:

```
#! /bin/sh
#
# use flag -x for echoeing commands
#
# file: set-switch-prio
#

OUT_DEV=eth0
```

```
echo
echo Setting IEEE 802.1p switch case
echo


tc qdisc add dev $OUT_DEV root handle 1: prio

tc qdisc add dev $OUT_DEV parent 1:1 handle 10: pfifo limit 40
tc qdisc add dev $OUT_DEV parent 1:2 handle 20: pfifo limit 40
tc qdisc add dev $OUT_DEV parent 1:3 handle 30: pfifo limit 40

tc filter add dev $OUT_DEV protocol ip parent 1: prio 1 u32 \
        match ip dport 6001 0xffff \
        flowid 1:1

tc filter add dev $OUT_DEV protocol ip parent 1: prio 1 u32 \
        match ip dport 6002 0xffff \
        flowid 1:2

tc filter add dev $OUT_DEV protocol ip parent 1: prio 1 u32 \
        match ip dport 6003 0xffff \
        flowid 1:3
```

C.4 Scripts for the fourth case

In this case, DiffServ functionality is set in the switch. Thus, two scripts are needed, one that configures the switch to provide premium, assured, and best-effort service ("set-switch-all") and one that configures the switch to provide the olympic service ("set-switch-af"). These scripts are different from those written for the router in that they attach the filters to the interface that represents the bridging entity, rather than to the physical interfaces themselves.

The script "set-switch-all" follows:

```
#! /bin/sh
#
# use flag -x for echoeing commands
#
# file: set-switch-all
#

IN_DEV=bridge
```

```
OUT_DEV=eth0

echo
echo Setting diffserv for switch
echo


# All packets are marked with a DSCP in the tcindex field that is
# going to be used on the egress side.
# The value set is the whole TOS. DSCP is TOS shifted by two
# EF  : 0xb8 (TOS) = 0x2e (DSCP)
# AF11: 0x28 (TOS) = 0x0a (DSCP)
# AF12: 0x30 (TOS) = 0x0c (DSCP)


# Install the ingress qdisc on the ingress interface
tc qdisc add dev $IN_DEV handle ffff: ingress

# Add u32 filter
tc filter add dev $IN_DEV parent ffff: protocol ip prio 4 \
        handle 1: u32 divisor 1

# EF traffic (port 6001) is policed and is marked
tc filter add dev $IN_DEV parent ffff: protocol ip prio 4 u32 \
        match ip dport 6001 0xffff \
        police rate 3000kbit burst 90k \
        drop flowid 0xb8

# AF in-profile traffic (port 6002) is marked as assured
tc filter add dev $IN_DEV parent ffff: protocol ip prio 4 u32 \
        match ip dport 6002 0xffff \
        police rate 2500kbit burst 90K continue \
        flowid 0x28

# AF out-of-profile traffic (port 6002) is marked as best-effort
tc filter add dev $IN_DEV parent ffff: protocol ip prio 5 u32 \
      match ip dport 6002 0xffff \
      flowid 0x30

# BE traffic is marked
tc filter add dev $IN_DEV parent ffff: protocol ip prio 4 u32 \
        match ip dport 6003 0xffff \
        flowid 0x30

./set-egress
```

The script “set-switch-af” follows:

```
#! /bin/sh
#
# use flag -x for echoeing commands
```

```
#
# file: set-switch-af
#

IN_DEV=bridge
OUT_DEV=eth0

echo
echo Setting diffserv for switch
echo


# All packets are marked with a DSCP in the tcindex field that is
# going to be used on the egress side.
# The value set is the whole TOS. DSCP is TOS shifted by two
# AF11: 0x28 (TOS) = 0x0a (DSCP)
# AF12: 0x30 (TOS) = 0x0c (DSCP)
# AF13: 0x38 (TOS) = 0x0e (DSCP)


# Install the ingress qdisc on the ingress interface
tc qdisc add dev $IN_DEV handle ffff: ingress

# Add u32 filter
tc filter add dev $IN_DEV parent ffff: protocol ip prio 4 \
        handle 1: u32 divisor 1

# AF11 traffic (port 6001) is marked
tc filter add dev $IN_DEV parent ffff: protocol ip prio 4 u32 \
        match ip dport 6001 0xffff \
        police rate 3000kbit burst 90k \
        drop flowid 0x28

# AF12 traffic (port 6002) is marked
tc filter add dev $IN_DEV parent ffff: protocol ip prio 4 u32 \
        match ip dport 6002 0xffff \
      flowid 0x30

# AF13 traffic (port 6003) is marked
tc filter add dev $IN_DEV parent ffff: protocol ip prio 4 u32 \
        match ip dport 6003 0xffff \
        flowid 0x38

./set-egress
```

C.5 Scripts for the VLAN case

Having configured VLANs in the switch and the router of our domain, the scripts that set the DiffServ functionality in the switch and the router need to be adjusted. The following scripts do not filter traffic according to UDP destination port number; rather, they classify traffic based on the VLAN ID of a packet. The scripts that configure premium, assured, and best-effort services are named “set-vlanswitch-all” for the switch and “set-vlanrouter2-all” for the router. The scripts that configure the olympic service are named “set-vlanswitch-af” for the switch and “set-vlanrouter2-af” for the router.

The script “set-vlanswitch-all” follows:

```
#! /bin/sh
#
# use flag -x for echoeing commands
#
# file: set-vlanswitch-all
#

IN_DEV1=bridge11
IN_DEV2=bridge12
IN_DEV3=bridge13


echo
echo Setting vlan diffserv for switch
echo


# All packets are marked with a DSCP in the tcindex field that is
# going to be used on the egress side.
# The value set is the whole TOS. DSCP is TOS shifted by two
# EF  : 0xb8 (TOS) = 0x2e (DSCP)
# AF11: 0x28 (TOS) = 0x0a (DSCP)
# AF12: 0x30 (TOS) = 0x0c (DSCP)

##############  VLAN 11 (EF)  ##############

# Install the ingress qdisc on the ingress interface
tc qdisc add dev $IN_DEV1 handle ffff: ingress

# Add u32 filter
tc filter add dev $IN_DEV1 parent ffff: protocol ip prio 4 \
        handle 1: u32 divisor 1

# EF traffic (VLAN 11) is policed and is marked
tc filter add dev $IN_DEV1 parent ffff: protocol ip prio 4 u32 \
      match ip dport 0xffff 0x0000 \
      police rate 3000kbit burst 90k drop \
      flowid 0xb8

##############  VLAN 12 (AF)  ##############

# Install the ingress qdisc on the ingress interface
```

```
tc qdisc add dev $IN_DEV2 handle ffff: ingress

# Add u32 filter
tc filter add dev $IN_DEV2 parent ffff: protocol ip prio 4 \
        handle 1: u32 divisor 1

# AF in-profile traffic (VLAN 12) is marked as assured
tc filter add dev $IN_DEV2 parent ffff: protocol ip prio 4 u32\
      match ip dport 0xffff 0x0000 \
      police rate 2500kbit burst 90K continue \
      flowid 0x28

# AF out-of-profile traffic (VLAN 12) is marked as best-effort
tc filter add dev $IN_DEV2 parent ffff: protocol ip prio 5 u32 \
      match ip dport 0xffff 0x0000 \
      flowid 0x30


##############  VLAN 13 (BE)  ##############

# Install the ingress qdisc on the ingress interface
tc qdisc add dev $IN_DEV3 handle ffff: ingress

# Add u32 filter
tc filter add dev $IN_DEV3 parent ffff: protocol ip prio 4 \
        handle 1: u32 divisor 1

# BE traffic (VLAN 13) is marked
tc filter add dev $IN_DEV3 parent ffff: protocol ip prio 4 u32\
        match ip dport 0xffff 0x0000 \
        flowid 0x30

./set-egress
```

The script “set-vlanrouter2-all” follows:

```
#! /bin/sh
#
# use flag -x for echoeing commands
#
# file: set-vlanrouter2-all
#

IN_DEV1=eth1.11
IN_DEV2=eth1.12
IN_DEV3=eth1.13


echo
echo Setting vlan diffserv for router2
echo
```

```
# All packets are marked with a DSCP in the tcindex field that is
# going to be used on the egress side.
# The value set is the whole TOS. DSCP is TOS shifted by two
# EF  : 0xb8 (TOS) = 0x2e (DSCP)
# AF11: 0x28 (TOS) = 0x0a (DSCP)
# AF12: 0x30 (TOS) = 0x0c (DSCP)

##############  VLAN 11 (EF)  ##############

# Install the ingress qdisc on the ingress interface
tc qdisc add dev $IN_DEV1 handle ffff: ingress

# Add u32 filter
tc filter add dev $IN_DEV1 parent ffff: protocol ip prio 4 \
        handle 1: u32 divisor 1

# EF traffic (VLAN 11) is policed and is marked
tc filter add dev $IN_DEV1 parent ffff: protocol ip prio 4 u32 \
        match ip dport 0xffff 0x0000 \
        police rate 3000kbit burst 90k drop \
        flowid 0xb8

##############  VLAN 12 (AF)  ##############

# Install the ingress qdisc on the ingress interface
tc qdisc add dev $IN_DEV2 handle ffff: ingress

# Add u32 filter
tc filter add dev $IN_DEV2 parent ffff: protocol ip prio 4 \
        handle 1: u32 divisor 1

# AF in-profile traffic (VLAN 12) is marked as assured
tc filter add dev $IN_DEV2 parent ffff: protocol ip prio 4 u32 \
        match ip dport 0xffff 0x0000 \
        police rate 2500kbit burst 90K continue \
        flowid 0x28

# AF out-of-profile traffic (VLAN 12) is marked as best-effort
tc filter add dev $IN_DEV2 parent ffff: protocol ip prio 5 u32 \
        match ip dport 0xffff 0x0000 \
        flowid 0x30

##############  VLAN 13 (BE)  ##############

# Install the ingress qdisc on the ingress interface
tc qdisc add dev $IN_DEV3 handle ffff: ingress

# Add u32 filter
tc filter add dev $IN_DEV3 parent ffff: protocol ip prio 4 \
        handle 1: u32 divisor 1

# BE traffic (VLAN 13) is marked
tc filter add dev $IN_DEV3 parent ffff: protocol ip prio 4 u32 \
        match ip dport 0xffff 0x0000 \
        flowid 0x30

./set-egress
```

The script “set-vlanswitch-af” follows:

```
#! /bin/sh
#
# use flag -x for echoeing commands
#
# file: set-vlanswitch-af
#

IN_DEV1=bridge11
IN_DEV2=bridge12
IN_DEV3=bridge13


echo
echo Setting diffserv for vlanswitch
echo


# All packets are marked with a DSCP in the tcindex filed that is
# going to be used on the egress side.
# The value set is the whole TOS. DSCP is TOS shifted by two
# AF11: 0x28 (TOS) = 0x0a (DSCP)
# AF12: 0x30 (TOS) = 0x0c (DSCP)
# AF13: 0x38 (TOS) = 0x0e (DSCP)

##############  VLAN 11 (AF11)  ##############

# Install the ingress qdisc on the ingress interface
tc qdisc add dev $IN_DEV1 handle ffff: ingress

# Add u32 filter
tc filter add dev $IN_DEV1 parent ffff: protocol ip prio 4 \
        handle 1: u32 divisor 1

# AF11 traffic (VLAN 11) is marked
tc filter add dev $IN_DEV1 parent ffff: protocol ip prio 4 u32 \
      match ip dport 0xffff 0x0000 \
      flowid 0x28

##############  VLAN 12 (AF12)  ##############

# Install the ingress qdisc on the ingress interface
tc qdisc add dev $IN_DEV2 handle ffff: ingress

# Add u32 filter
tc filter add dev $IN_DEV2 parent ffff: protocol ip prio 4 \
        handle 1: u32 divisor 1

# AF12 traffic (VLAN 12) is marked
tc filter add dev $IN_DEV2 parent ffff: protocol ip prio 4 u32\
      match ip dport 0xffff 0x0000 \
      flowid 0x30
```

```
##############  VLAN 13 (AF13)  ##############

# Install the ingress qdisc on the ingress interface
tc qdisc add dev $IN_DEV3 handle ffff: ingress

# Add u32 filter
tc filter add dev $IN_DEV3 parent ffff: protocol ip prio 4 \
        handle 1: u32 divisor 1

# AF13 traffic (VLAN 13) is marked
tc filter add dev $IN_DEV3 parent ffff: protocol ip prio 4 u32\
        match ip dport 0xffff 0x0000 \
        flowid 0x38


./set-egress
```

The script “set-vlanrouter2-af” follows:

```
#! /bin/sh
#
# use flag -x for echoeing commands
#
# file: set-vlanrouter2-af
#

IN_DEV1=eth1.11
IN_DEV2=eth1.12
IN_DEV3=eth1.13


echo
echo Setting vlan diffserv for router2
echo


# All packets are marked with a DSCP in the tcindex filed that is
# going to be used on the egress side.
# The value set is the whole TOS. DSCP is TOS shifted by two
# AF11: 0x28 (TOS) = 0x0a (DSCP)
# AF12: 0x30 (TOS) = 0x0c (DSCP)
# AF13: 0x38 (TOS) = 0x0e (DSCP)

##############  VLAN 11 (AF11)  ##############

# Install the ingress qdisc on the ingress interface
tc qdisc add dev $IN_DEV1 handle ffff: ingress

# Add u32 filter
tc filter add dev $IN_DEV1 parent ffff: protocol ip prio 4 \
        handle 1: u32 divisor 1
```

```
# AF11 traffic (VLAN 11) is marked
tc filter add dev $IN_DEV1 parent ffff: protocol ip prio 4 u32 \
        match ip dport 0xffff 0x0000 \
      flowid 0x28

##############  VLAN 12 (AF12)  ##############

# Install the ingress qdisc on the ingress interface
tc qdisc add dev $IN_DEV2 handle ffff: ingress

# Add u32 filter
tc filter add dev $IN_DEV2 parent ffff: protocol ip prio 4 \
        handle 1: u32 divisor 1

# AF12 traffic (VLAN 12) is marked
tc filter add dev $IN_DEV2 parent ffff: protocol ip prio 4 u32 \
        match ip dport 0xffff 0x0000 \
      flowid 0x30

##############  VLAN 13 (AF13)  ##############

# Install the ingress qdisc on the ingress interface
tc qdisc add dev $IN_DEV3 handle ffff: ingress

# Add u32 filter
tc filter add dev $IN_DEV3 parent ffff: protocol ip prio 4 \
        handle 1: u32 divisor 1

# AF13 traffic (VLAN 13) is marked
tc filter add dev $IN_DEV3 parent ffff: protocol ip prio 4 u32 \
        match ip dport 0xffff 0x0000 \
        flowid 0x38


./set-egress
```

APPENDIX D – Automation scripts

The experiments presented in Chapter 4 were conducted with the help of an automated procedure. The *expect* program, a *tcl* interpreter, was used to generate traffic flows, collect data, and present results. This program interprets *tcl* scripts that run shell programs that are inherent interactive. It interacts with the programs in a predefined manner, as described in the script it executes. Moreover, it can perform any other task that can be performed by a *tcl* program. The experiments that make use of the premium, assured, and best-effort services utilize the script "run-all," whereas the experiments that make use of the olympic service utilize the script "run-af."

The process in both cases is essentially the same. The script first logs into the sink host using an *ssh client*. It cleans up of any old files and launches in the background three instances of the server side of the *iperf* program that will collect statistics for the three UDP flows. Then, it logs into each source host, using *ssh* again, and generates a UDP flow from each, launching the client side of the *iperf* program. The flows last for typically 30 seconds and when they are over, the script logs back into the server and collects statistics for this step. It then repeats the procedure from the beginning, performing the second step, which will typically differ from the last one in the generated flow rates. When all the steps are over, the script collects all data, makes all the necessary computations, and produces two sets of files. The first set is input for the *xgraph* program, which will be automatically executed by the script and display the plots with the results. The second set of files is in an *excel* format for further processing.

The script “run-all” follows:

```
#!/usr/bin/expect -f
#
# file: run-all

# runs an EF, an AF and a BE flow of X_RATE and for TIME (Mbps and
secs)
# for STEPS times, for TIME each.

#puts $argc
#set l [ llength $argv ]

set EF_RATE [ lindex $argv 0 ]
set AF_RATE [ lindex $argv 1 ]
set BE_RATE [ lindex $argv 2 ]
set STEPS   [ lindex $argv 3 ]

set STEP    0.5
set TIME    30

# pause between every round
set INTERVAL 4000

set BE_RATE_BEGIN $BE_RATE
set BE_RATE_END [expr $BE_RATE_BEGIN + ($STEPS*$STEP-$STEP) ]

# script assumes you are logged as root

#set seconds for expect to wait
set timeout 120

set PASS palamakia

set CLIENT1_IP 192.168.10.11
set CLIENT2_IP 192.168.10.12
set CLIENT3_IP 192.168.10.13
set SERVER_IP 192.168.0.10

set EF_PORT 6001
set AF_PORT 6002
set BE_PORT 6003

set P_SIZE 1000

######################### Connect to Server and cleanup
spawn ssh $SERVER_IP
expect "password:"
send "$PASS\r"

expect "root]#"
send "rm -f ef_bw\r"
expect "root]#"
send "rm -f ef_ls\r"
expect "root]#"
send "rm -f af_bw\r"
```

```
expect "root]#"
send "rm -f af_ls\r"
expect "root]#"
send "rm -f be_bw\r"
expect "root]#"
send "rm -f be_ls\r"
expect "root]#"
send "logout\r"
#expect "root]#"

################################### Run Sessions

for {set x 0} {$x<$STEPS} {incr x} {

######################### Connect to server
spawn ssh $SERVER_IP
#send "ssh $SERVER_IP\r"
expect "password:"
send "$PASS\r"

expect "root]#"
send "echo -n $BE_RATE >> ef_bw\r"
expect "root]#"
send "echo -n $BE_RATE >> ef_ls\r"
expect "root]#"
send "echo -n $BE_RATE >> af_bw\r"
expect "root]#"
send "echo -n $BE_RATE >> af_ls\r"
expect "root]#"
send "echo -n $BE_RATE >> be_bw\r"
expect "root]#"
send "echo -n $BE_RATE >> be_ls\r"
expect "root]#"

send "iperf -s -f m -p $EF_PORT -u -o 2 -r ef -l 1000 &\r"
expect "root]#"
send "iperf -s -f m -p $AF_PORT -u -o 2 -r af -l 1000 &\r"
expect "root]#"
send "iperf -s -f m -p $BE_PORT -u -o 2 -r be -l 1000 &\r"

expect "root]#"
send "logout\r"
#expect "root]#"

######################### Connect to Source1 - EF
spawn ssh $CLIENT1_IP
#send "ssh $CLIENT1_IP\r"
expect "password:"
send "$PASS\r"

expect "root]#"
send "iperf -c $SERVER_IP -b $EF_RATE\m -p $EF_PORT -l $P_SIZE -t $TIME
&\r"
expect "root]#"
send "logout\r"
#expect "root]#"
```

```
########################## Connect to Source2 - AF
spawn ssh $CLIENT2_IP
#send "ssh $CLIENT2_IP\r"
expect "password:"
send "$PASS\r"

expect "root]#"
send "iperf -c $SERVER_IP -b $AF_RATE\m -p $AF_PORT -l $P_SIZE -t $TIME
&\r"
expect "root]#"
send "logout\r"
#expect "root]#"

########################## Connect to Source3 - BE
spawn ssh $CLIENT3_IP
#send "ssh $CLIENT3_IP\r"
expect "password:"
send "$PASS\r"

expect "root]#"
send "iperf -c $SERVER_IP -b $BE_RATE\m -p $BE_PORT -l $P_SIZE -t
$TIME\r"
expect "root]#"
after $INTERVAL
send "logout\r"

set BE_RATE [expr $BE_RATE + $STEP]

}

########################## Connect to Server & get results

spawn ssh -X $SERVER_IP
expect "password:"
send "$PASS\r"

#### Create the excel files

expect "root]#"
after 8000

send "join ef_bw af_bw > temp.xls\r"
expect "root]#"
send "join temp.xls be_bw > bw.xls\r"
expect "root]#"
send "join ef_ls af_ls > temp.xls\r"
expect "root]#"
send "join temp.xls be_ls > ls.xls\r"

### Show throughput
expect "root]#"
send "rm -f graphfile1\r"
send "cat ef_bw >> graphfile1\r"
expect "root]#"
send "echo  >> graphfile1\r"
expect "root]#"
send "cat af_bw >> graphfile1\r"
```

```
expect "root]#"
send "echo  >> graphfile1\r"
expect "root]#"
send "cat be_bw >> graphfile1\r"
expect "root]#"
send "xgraph -ly 0,10 -0 \"EF traffic $EF_RATE Mbps\" \
        -1 \"AF traffic $AF_RATE Mbps\" -2 \"BE traffic $BE_RATE_BEGIN
- $BE_RATE_END Mbps\" \
      -x Mbps -y Mbps \
        -t \"EF, AF, and BE traffic classes throughput\" graphfile1
&\r"

expect "root]#"

after 2000

send "rm -f graphfile2\r"

### Show packet loss
send "cat ef_ls >> graphfile2\r"
expect "root]#"
send "echo  >> graphfile2\r"
expect "root]#"
send "cat af_ls >> graphfile2\r"
expect "root]#"
send "echo  >> graphfile2\r"
expect "root]#"
send "cat be_ls >> graphfile2\r"
expect "root]#"
send "xgraph -ly 0,100 -0 \"EF traffic $EF_RATE Mbps\" \
        -1 \"AF traffic $AF_RATE Mbps\" -2 \"BE traffic $BE_RATE_BEGIN
- $BE_RATE_END Mbps\" \
      -x Mbps -y %\
        -t \"EF, AF and BE packet loss\" graphfile2 &\r"

expect "root]#"

after 2000

send "logout\r"

interact
```

The script “run-af” follows:

```
#!/usr/bin/expect -f
#
# file: run-af

# runs an AF11, an AF12 and an AF13 flow of AFX_RATE and for TIME (Mbps
and secs)
# for STEPS times, for TIME each.
```

```
#puts $argc
#set l [ llength $argv ]

set AF1_RATE [ lindex $argv 0 ]
set AF2_RATE [ lindex $argv 1 ]
set AF3_RATE [ lindex $argv 2 ]
set STEPS   [ lindex $argv 3 ]

set STEP    0.5
set TIME    30

# pause between every round
set INTERVAL 4000

set AF1_RATE_BEGIN $AF1_RATE
set AF2_RATE_BEGIN $AF2_RATE
set AF3_RATE_BEGIN $AF3_RATE

set AF1_RATE_END [expr $AF1_RATE_BEGIN + ($STEPS*$STEP-$STEP) ]
set AF2_RATE_END [expr $AF2_RATE_BEGIN + ($STEPS*$STEP-$STEP) ]
set AF3_RATE_END [expr $AF3_RATE_BEGIN + ($STEPS*$STEP-$STEP) ]

# script assumes you are logged as root

#set seconds for expect to wait
set timeout 120

set PASS palamakia

set CLIENT1_IP 192.168.10.11
set CLIENT2_IP 192.168.10.12
set CLIENT3_IP 192.168.10.13
set SERVER_IP 192.168.0.10

set AF1_PORT 6001
set AF2_PORT 6002
set AF3_PORT 6003

set P_SIZE 1000

########################## Connect to Server and cleanup
spawn ssh $SERVER_IP
expect "password:"
send "$PASS\r"

expect "root]#"
send "rm -f af1_bw\r"
expect "root]#"
send "rm -f af1_ls\r"
expect "root]#"
send "rm -f af2_bw\r"
expect "root]#"
send "rm -f af2_ls\r"
expect "root]#"
send "rm -f af3_bw\r"
expect "root]#"
send "rm -f af3_ls\r"
```

```
expect "root]#"
send "logout\r"

#################################### Run Sessions


for {set x 0} {$x<$STEPS} {incr x} {


########################## Connect to server
spawn ssh $SERVER_IP
expect "password:"
send "$PASS\r"

expect "root]#"
send "echo -n $AF1_RATE >> af1_bw\r"
expect "root]#"
send "echo -n $AF1_RATE >> af1_ls\r"
expect "root]#"
send "echo -n $AF2_RATE >> af2_bw\r"
expect "root]#"
send "echo -n $AF2_RATE >> af2_ls\r"
expect "root]#"
send "echo -n $AF3_RATE >> af3_bw\r"
expect "root]#"
send "echo -n $AF3_RATE >> af3_ls\r"
expect "root]#"

send "iperf -s -f m -p $AF1_PORT -u -i 1 -o 2 -r af1 &\r"
expect "root]#"
send "iperf -s -f m -p $AF2_PORT -u -i 1 -o 2 -r af2 &\r"
expect "root]#"
send "iperf -s -f m -p $AF3_PORT -u -i 1 -o 2 -r af3 &\r"

expect "root]#"
send "logout\r"

########################## Connect to Source1 - EF
spawn ssh $CLIENT1_IP
expect "password:"
send "$PASS\r"

expect "root]#"
send "iperf -c $SERVER_IP -b $AF1_RATE\m -p $AF1_PORT -l $P_SIZE -t
$TIME &\r"
expect "root]#"
send "logout\r"

########################## Connect to Source2 - AF
spawn ssh $CLIENT2_IP
expect "password:"
send "$PASS\r"

expect "root]#"
send "iperf -c $SERVER_IP -b $AF2_RATE\m -p $AF2_PORT -l $P_SIZE -t
$TIME &\r"
```

```
expect "root]#"
send "logout\r"

########################## Connect to Source3 - BE
spawn ssh $CLIENT3_IP
expect "password:"
send "$PASS\r"

expect "root]#"
send "iperf -c $SERVER_IP -b $AF3_RATE\m -p $AF3_PORT -l $P_SIZE -t
$TIME\r"
expect "root]#"
send "logout\r"

set AF1_RATE [expr $AF1_RATE + $STEP]
set AF2_RATE [expr $AF2_RATE + $STEP]
set AF3_RATE [expr $AF3_RATE + $STEP]

after $INTERVAL

}

########################## Connect to Server & get results

spawn ssh -X $SERVER_IP
expect "password:"
send "$PASS\r"

#### Create the excel files

expect "root]#"
after 8000

send "join af1_bw af2_bw > temp.xls\r"
expect "root]#"
send "join temp.xls af3_bw > bw.xls\r"
expect "root]#"
send "join af1_ls af2_ls > temp.xls\r"
expect "root]#"
send "join temp.xls af3_ls > ls.xls\r"

### Show throughput
expect "root]#"
send "rm -f graphfile1\r"
send "cat af1_bw >> graphfile1\r"
expect "root]#"
send "echo  >> graphfile1\r"
expect "root]#"
send "cat af2_bw >> graphfile1\r"
expect "root]#"
send "echo  >> graphfile1\r"
expect "root]#"
send "cat af3_bw >> graphfile1\r"
expect "root]#"
send "xgraph -ly 0,10 -0 \"AF11 traffic $AF1_RATE_BEGIN - $AF1_RATE_END
Mbps\" \
        -1 \"AF12 traffic $AF2_RATE_BEGIN - $AF2_RATE_END Mbps\" -2
\"AF13 traffic $AF3_RATE_BEGIN - $AF3_RATE_END Mbps\" \
      -x Mbps -y Mbps \
        -t \"AF11, AF12 and AF13 throughput\" graphfile1 &\r"

expect "root]#"

#after 8000

send "rm -f graphfile2\r"

### Show packet loss
send "cat af1_ls >> graphfile2\r"
expect "root]#"
send "echo  >> graphfile2\r"
expect "root]#"
send "cat af2_ls >> graphfile2\r"
expect "root]#"
send "echo  >> graphfile2\r"
expect "root]#"
send "cat af3_ls >> graphfile2\r"
expect "root]#"
send "xgraph -ly 0,100 -0 \"AF11 traffic $AF1_RATE_BEGIN -
$AF1_RATE_END Mbps\" \
        -1 \"AF12 traffic $AF2_RATE_BEGIN - $AF2_RATE_END Mbps\" -2
\"AF13 traffic $AF3_RATE_BEGIN - $AF3_RATE_END Mbps\" \
      -x Mbps -y %\
        -t \"AF11, AF12 and AF13 packet loss\" graphfile2 &\r"

expect "root]#"

after 2000

send "logout\r"

interact
```

REFERENCES

[1] H. Saltzer, D. P. Reed, and D. Clark, “End-to-end arguments in system design”, *ACM Transactions on Computing Systems*, vol. 2, no. 4, 1984.

[2] S. Bhattacharjee, K. L. Calvert, and E. W. Zegura, “Active Networking and the End-to-End Argument”, *Proceedings of ICNP’97*, Atlanta, GA, October 1997.

[3] R. Braden, L. Zhang, and S. Shenker, “Integrated Services in the Internet Architecture: an Overview”, RFC 1633, June, 1994.

[4] B. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin, “Resource ReSerVation Protocol (RSVP), Version 1 Functional Specification”, RFC 2205, September, 1997.

[5] IETF Differentiated Services Working Group, http://www.ietf.org/html.charters/diffserv-charter.html.

[6] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang and W. Weiss, “An Architecture for Differentiated Services”, RFC 2475, December, 1998.

[7] K. Nichols, V. Jacobson, and L. Zhang, "A Two-bit Differentiated Services Architecture for the Internet", ftp://ftp.ee.lbl.gov/papers/dsarch.pdf, November 1997.

[8] K. Nichols, S. Blake, F. Baker, and D. Black, “Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers”, RFC 2474, December 1998.

[9] D. Grossman, “New Terminology and Clarifications for Diffserv”, RFC 3260, April 2002.

[10] V. Jacobson, K. Nichols, and K. Poduri, “An Expedited Forwarding PHB (Per-Hop Behavior)”, RFC 2598, June 1999.

[11] B. Davie, A. Charny, J.C.R. Bennett, K. Benson, J.Y. Le Boudec, W. Courtney, S. Davari, V. Firoiu, and D. Stiliadis, “An Expedited Forwarding PHB (Per-Hop Behavior)”, RFC 3246, March 2002

[12] A. Charny, J.C.R. Bennett, K. Benson, J.Y. Le Boudec, A. Chiu, W. Courtney, S. Davari, V. Firoiu, C. Kalmanek, and K.K. Ramakrishnan, “Supplemental Information for the New Definition of the EF PHB (Expedited Forwarding Per-Hop Behavior)”, RFC 3247, March 2002.

[13] D. Clark, J. Wroclawski, “An Approach to Service Allocation in the Internet”, Internet Draft, July 1997.

[14] J. Heinanen, F. Baker, W. Weiss, and J. Wroclawski, “Assured Forwarding PHB Group”, RFC 2597, June 1999.

[15] P. Hurley, Mourad Kara, J. Y. Le Boudec, and P.Thiran "ABE: Providing a Low-Delay Service within Best Effort", *IEEE Network Magazine*, vol. 15 no. 3, May 2001.

[16] QBone Architecture Design Team, http://qbone.internet2.edu/arch-dt.shtml.

[17] TF-NGN Task Force, http://www.terena.nl/tech/task-forces/tf-ngn.

[18] ITTC – IP/QoS Research, http://qos.ittc.ukans.edu.

[19] S. Floyd and V. Jacobson, "Link-sharing and Resource Management Models for Packet Networks", *IEEE/ACM Transactions on Networking*, pp 365-386, August 1995.

[20] S. Floyd and V. Jacobson, “Random Early Detection Gateways for Congestion Avoidance”, *IEEE/ACM Transactions on Networking*, August 1993

[21] D. D. Clark and W. Fang, “Explicit Allocation of Best-Effort Packet Delivery Service”, *IEEE/ACM Transactions on Networking*, vol. 6, no. 4, August 1998.

[22] Linux Advanced Routing & Traffic Control, http://lartc.org.

[23] W. Almesberger, “Linux Network Traffic Control - Implementation Overview”, April 1999.

[24] Differentiated Services on Linux, http://diffserv.sourceforge.net.

[25] W. Almesberger, J. H. Salim, and A. Kuznetsov, “Differentiated Services on Linux”, June 1999.

[26] Traffic Control Next Generation, http://tcng.sourceforge.net/index.html.

[27] Comments and Contributions Invited: "Why Premium IP Service Has Not Deployed (and Probably Never Will)", http://mail.internet2.edu:8080/guest/archives/qbone-arch-dt/log200205/msg00000.html.

[28] Iperf, http://dast.nlanr.net/Projects/Iperf.

[29] IEEE P 802.1D/D15 (Incorporating IEEE P802.1p), November 1997.

[30] IEEE P 802.1Q/D9, February 1998.

[31] D. McDysan, and L. Yao, “Differentiated Services Over 802.3 Networks Framework”, July 2001.

[32] Linux Ethernet Bridging, http://bridge.sourceforge.net.

[33] 802.1Q VLAN Implementation for Linux,
http://www.candelatech.com/~greear/vlan.html.