

Scalable Emulation of IP Networks through Virtualization

by

Amit P Kucheria

B.E. (Computer Science and Engineering), SSGMCE, India, 1998

Submitted to the Department of Electrical Engineering and Computer Science and the
Faculty of the Graduate School of the University of Kansas in partial fulfillment of the
requirements for the degree of Master of Science

Professor in Charge

Committee Members

Date Thesis Accepted

© Copyright 2003 by Amit P Kucheria
All Rights Reserved

Acknowledgments

I would like to extend my sincere and utmost gratitude to Dr. Douglas Niehaus, my advisor and committee chairman, for his invaluable guidance, advice and inspiration throughout this research and my stay at KU. I would like to thank Dr. Victor Frost for serving on my committee and for his direction. Thanks are due to Dr. Jeremiah James for serving on my committee, for his insight in the classroom and for providing a constant source of amusement during the project meetings. I am also thankful to Sprint Corporation for their support for this research.

Members of room 245E at ITTC made it an exceptional working environment and provided valuable assistance and friendship. I would like to thank the network administration staff at ITTC, especially Michael Hulet, for all their help. A special thanks is due to my project mates, Vijey and Karthik, who were great pals to work with. My friends at KU deserve a special mention for providing much-needed distractions as well as assistance.

My parents and sister Sheetal have always been a profound source of support, understanding and inspiration. I would like to thank them for just being there for me.

Abstract

Improving the behavior of the IP protocol suite for new applications with real-time network requirements is an area of considerable interest for researchers in the networking research community. With an eye towards ubiquitous IP networks, researchers have been involved with evaluating the application and performance of various Quality of Service protocols. One of the key requirements to improving these protocols is testing them in large-scale computer networks. Most of the relevant studies have been conducted using small-scale networks or discrete-event simulation due to the expense of setting up large-scale networks. This research presents an application of virtualization of IP network elements to emulate large-scale networks. In this work, Virtual Network extensions to Linux were used to create large-scale virtual networks on top of a smaller set of physical machines. Various approaches to virtualization of generic IP networks are considered with different scaling abilities and emulation results from Diffserv and Intserv based network testing are compared with physical network test results.

Contents

1	Introduction	1
1.1	Motivation	3
1.2	Virtual Network Elements	5
2	Related Work	7
2.1	NIST Network Emulation Tool	7
2.2	IP Infusion	8
2.3	Proportional Time Emulation and Simulation	8
3	Virtual Network Elements	11
3.1	Overview	11
3.2	Packets inside Linux	14
3.3	The Linux Network bottom-half	15
3.4	Overview of a Linux network device driver	16
3.5	Design of a Virtual Ethernet Device Driver	21
3.6	The Routing Table	24
3.7	Virtual Host and Virtual Router	32
3.8	Limitations of Virtual Network elements	36
4	Quality of Service	39
4.1	Quality of Service Models	39
4.1.1	Integrated Services	40
4.1.1.1	Intserv model	41
4.1.1.2	Reference Implementation Framework	44

4.1.1.3	RSVP signaling	45
4.1.1.4	Disadvantages of Intserv	46
4.1.2	Differentiated Services	47
4.1.2.1	Diffserv model	47
4.1.2.2	Expedited Forwarding and Assured Forwarding	50
4.1.2.3	Disadvantages of Diffserv	51
4.1.3	Hybrid Qos Architectures	52
4.1.3.1	Microsoft Model	54
4.1.3.2	Tunneled Aggregate Model	55
4.2	Quality of Service support in Linux	56
4.2.1	Linux Traffic control	56
4.2.2	Intserv and RSVP	58
4.2.3	Diffserv	61
4.3	Problems with testing QoS networks	66
5	Implementation	68
5.1	Modifications to QoS software to work with VNET	68
5.1.1	Enhancements to VNET code	69
5.1.2	Modifications to <code>iproute2</code> package	69
5.1.3	Modifications to Diffserv code	70
5.1.4	Modifications to RSVP daemon	71
5.2	Netspec	72
5.2.1	Variable-phase Netspec	73
5.2.2	VNET Netspec daemon	74
5.2.3	System command daemon	75
6	Evaluation	79
6.1	Faithfully emulated virtual networks	80
6.1.1	Test network topology	80
6.1.2	Virtual network topologies	82
6.1.3	Evaluation and Results	86

6.2	Emulation of 9-element Diffserv network	87
6.2.1	Architecture of Physical 9-element Diffserv network	87
6.2.2	Conclusions from 9-element Physical Diffserv network testing . .	91
6.2.3	Architecture of 9-element Virtual Diffserv network	91
6.2.4	Conclusions from 9-element Virtual Diffserv network test	95
6.2.5	Comparison of physical and virtual 9-element Diffserv results . .	97
6.3	Emulation of 17-element Diffserv network	97
6.3.1	Architecture of 17-element Virtual Diffserv network	99
6.3.2	Conclusions from 17-element Virtual Diffserv network test	100
6.4	Emulation of Intserv network	103
7	Conclusions and Future Work	107
7.1	Conclusions	107
7.2	Future Work	108
7.3	Virtual Network Elements and ProTEuS	111
A	Important Linux functions	112
B	Patches and Scripts	114
B.1	Patches	114
B.2	Scripts	116

List of Tables

3.1	Virtual routing table for virtual host A	31
3.2	Virtual routing table for virtual router	31
3.3	Subnet Map table for virtual router in Figure 3.10(a)	32
3.4	Subnet Map table for virtual router in Figure 3.10(b)	32
3.5	List of <code>ioctl</code> calls defined for Virtual Network Elements	35
4.1	TSpec parameters	43
4.2	DSCP values for Assured forwarding	51
4.3	Classification of traffic in Intserv nodes	59
4.4	Relation between DSCP, <code>tcindex</code> , AF classes and drop priority	63
5.1	Slot-based execution in Variable-phase Netspec	74
6.1	Pros and Cons of various virtual network topologies	85
6.2	Traffic generated in Physical Diffserv network	87
6.3	Traffic characterization - Token bucket parameters (Physical)	89
6.4	Results - 9-element Physical Diffserv network test	90
6.5	Traffic generated in Virtual Diffserv network	91
6.6	Traffic characterization - Token bucket parameters (Virtual)	94
6.7	Results - 9-element Virtual Diffserv network test	96
6.8	DSCP values assigned to traffic in 17-element Diffserv network	98
6.9	Results - 17-element Virtual Diffserv network test	100
A.1	Important functions in the Linux network stack	113
B.1	Patches applied to Linux kernel	114

B.2	Miscellaneous Patches	115
B.3	Packet sizes for background BE traffic (period = 10ms)	116
B.4	Period for background RT traffic (Packet size = 1472 bytes)	116
B.5	Period for background RT traffic (Packet size = 1456 bytes)	117

List of Figures

3.1	A simple virtual network	12
3.2	Family of Virtual devices	13
3.3	The <code>sk_buff</code> structure	14
3.4	Working of RX and TX in a network driver	18
3.5	Packet flow down the Linux protocol stack	19
3.6	Packet flow up the Linux protocol stack	20
3.7	The Virtual Network layer in the Protocol Stack	22
3.8	Functioning of the Virtual Network Layer	25
3.9	Goals of virtualization	29
3.10	Virtualization of the sample network	30
3.11	Functioning of the Virtual Network layer (modified)	33
3.12	Trivial physical network	37
3.13	More complex physical network	37
4.1	A sample Intserv network	42
4.2	A sample Diffserv network	48
4.3	A Sample Hybrid network model	53
4.4	Packet processing inside the kernel	56
4.5	Example of qdiscs, classes and filters	57
4.6	Queue structure in Intserv nodes	60
4.7	Queue structure in Diffserv edge router	63
4.8	Queue structure in Diffserv core router	64
6.1	Network Topology to test Emulation Framework	81

6.2	Emulation Testbed	83
6.3	Various virtual network topologies	84
6.4	Physical Network Architecture for 9-element Diffserv network test . . .	88
6.5	Throughput in 9-element Diffserv network	92
6.6	Delay in 9-element Diffserv network	93
6.7	Virtual Network Architecture for 9-element Diffserv network test	93
6.8	Physical Network Architecture for 17-element Diffserv network test . . .	98
6.9	Virtual Network Architecture for 17-element Diffserv network test . . .	99
6.10	Throughput in 17-element Diffserv network	102

List of Programs

3.1	Important members of <code>struct device</code>	17
5.1	Netspec script to setup a 2-host, 1-router virtual network	76
5.2	Netspec script using <code>nssyscmd</code> to setup queues on routers	77
6.1	RSVP commands	104
B.1	<code>iproute</code> VNET patch	115
B.2	Traffic control <code>UTIME</code> patch	115
B.3	Netspec script for generating Background BE traffic - Physical Network	117
B.4	Netspec script for generating Background RT traffic - Physical Network	118
B.5	Netspec script for generating Test traffic - Physical Network	118
B.6	Netspec script for generating Background BE traffic - Virtual Network .	119
B.7	Netspec script for generating Background RT traffic - Virtual Network .	119
B.8	Netspec script for generating Test traffic - Virtual Network	120
B.9	TC script for marking background RT traffic - Physical Network	120
B.10	TC script for marking test traffic - Physical Network	121
B.11	TC script for marking background RT traffic - Virtual Network	122
B.12	TC script for marking test traffic - Virtual Network	123
B.13	TC script for core Diffserv routers - Physical Network	124
B.14	TC script for core Diffserv routers - Virtual Network	125

Chapter 1

Introduction

The ubiquity of IP networks has led to new applications being developed which use the IP protocol stack to support new services. These new services include various streaming audio and video services used for interactive messaging, videoconferencing and IP telephony. Legacy application protocols like ftp, telnet and http now coexist with newer protocols like RTP (Real Time transport protocol), RTSP (Real Time Streaming Protocol), H.323 * (for videoconferencing) and various other telephony protocols, which have different bandwidth, latency and jitter requirements.

The legacy IP-based networks provide *best-effort* data delivery which ensures that the complexity stays in the end-hosts and the core network stays relatively simple (edge-to-edge). As more hosts are connected, demands on the network eventually exceed capacity, but service is not denied; it degrades gracefully. But the resulting packet loss and variability in delay (jitter), though not a problem for legacy applications such as email, file transfer and web applications, causes havoc for the newer real-time applications, the most demanding of which are interactive two-way videoconferencing and IP telephony. Increasing the bandwidth will improve the situation to a certain extent but it soon becomes economically inviable for the network provider because the spare network capacity lies unused most of the time. And even on a relatively unloaded IP network, there can be enough jitter to trouble real-time applications [?]. What's needed is a way to provide predictability and control beyond the current best-effort service. This

*RTSP and H.323 use RTP.

requires extending the IP protocol to distinguish real-traffic from legacy traffic that can tolerate delay and jitter. This is what *Quality of Service (QoS)* protocols are designed to do.

Quality of service is a broad term used to describe the performance attributes of an end-to-end network connection. These attributes which affect the predictability of a service include bandwidth (throughput), latency (delay) and error rate (packet loss). QoS is used to specify the traffic parameters which guarantee certain service levels to the application in terms of available bandwidth, maximum delay, allowed jitter etc. One of the important roles of QoS is to improve handling of sensitive streams such as video and voice traffic during periods of network congestion or at network bottlenecks such as routers. Most QoS techniques involve prioritizing of sensitive or premium network traffic.

The evaluation of QoS architectures is subject to a lot of research among the academia as well as network equipment-vendor communities. A typical network is comprised of three major elements: *hosts*, *routers* and *links*. There are many questions regarding predictability of bounds on network characteristics such as delay, latency and jitter when customer traffic flows pass through the core routers in the providers' network. Network providers need to know if other traffic flowing over the network can impact certain privileged flows, and if so, by how much. There is also a need to study the complexity involved in implementing and managing these QoS networks. Measurement of this complexity is the topic of our current research [34, 42] and improving the scalability of the experiments is the problem that this work attempts to solve.

Various studies have been carried out to study these networks but they have been limited either by the scale of experiments [34, 42, 41, 13] or suffer from oversimplification of the simulations [24]. Besides, computer simulations of QoS networks do not address an important factor in QoS deployment: *Management Complexity*. Hosts and Routers are expensive hardware and require significant amounts of space. They also need a non-trivial amount of configuration. These drawbacks limit their numbers in an experimental setup in academic and research organizations. This in turn limits the size of networks that can be built when experimenting with new network technologies using physical

models.

We present a new approach which attacks both these problems using existing operating system protocol stacks (thereby preventing oversimplification) and increasing the scale of experiments without a proportional increase in the number of machines. This gives us a measure of the management complexity of the control plane of such networks without compromising on the data plane results. Our approach has created **Virtual Network Elements** to address this problem.

1.1 Motivation

The motivation for this work comes from observing current network protocol evaluation strategies which can be characterized [44] into three broad classes:

- **Simulation:** This involves constructing a simulation model of the network protocol using software tools like *ns* and Opnet [24, 1] and constructing a network model to test the protocol.
- **Live testing:** This involves writing code to implement the network protocol and configuring a test network of hosts and routers to use and test the protocol.
- **Emulation:** This approach lies somewhere in between simulation and live testing. It involves running real code on test machines, but recreates certain network characteristics in software. These characteristics involve those that are difficult or expensive to test with real hardware. e.g. delaying every packet by 400 milliseconds to simulate a satellite link.

Simulations have the following drawbacks:

1. **Re-implementation of software:** Software has to be adapted or re-implemented to become part of the simulator. e.g. *Ns-2* requires (re)implementation of the new protocol code in C++ and the models to be written in Object Tcl.
2. **Oblivious to OS interactions:** Simulations do not take into account the complex interactions inside a real operating system that could change the nature of

the real network. e.g. Consider the simulation of a streaming video protocol which is deployed on server. When a client connects to this server, this server reads the video file from the hard-drive and starts streaming it to the client after chopping it into packets of Δ bytes each. Although simulations can successfully implement the part where packets of size Δ bytes are sent out according to a certain frequency distribution, they most often do not take into account the subtle but complex interactions with the OS in reading the file from the disk.

3. **Poor representation of real environment:** The simulation environment may be considerably different from or poorly represent the real-life environment.
4. **Change of focus:** The simulation environment changes the focus from debugging and evaluating the protocol to first debugging and evaluating the accuracy of the model.
5. **No metrics for Management Complexity:** Simulations do not address configuration management complexity. e.g. setting up a network, configuring the routing tables, setting up QoS on routers and automation of these tasks. Very often this information is invaluable to networking companies in taking business decisions about supporting a new protocol.

Despite these drawbacks, simulations are important in situations where the protocol software does not yet exist or it is infeasible to carry out live testing or to do cheap, quick and reproducible *proof-of-concept* tests that need not be very accurate.

Live testing on the other hand, has the following drawbacks:

1. **Equipment costs:** This could very well be the number one drawback of live testing. Very often, the cost of hosts and routers limits their numbers in live testing. Due to budget constraints, it becomes infeasible to perform large-scale live testing.
2. **Infrastructure costs:** Adding to the cost of the equipment is the infrastructure cost accrued in terms of storage space for additional hosts and routers and operational facilities like electricity and air-conditioning.

3. **Extrapolation of results of small tests:** Due to the cost drawbacks, very often results of smaller-scale tests are wrongly (erroneously) extrapolated to large-scale scenarios.

Our approach is a kind of emulation where we emulate the basic network elements in software to emulate a network. However, whereas emulation artificially recreates certain network characteristics in software (e.g. packet loss, delay, reordering) to emulate a real network, we *do not* introduce any artificial conditions. Instead, we create multiple (software) instances of the basic network elements (host and router) which transmit and receive packets in exactly the same way as physical hosts and routers.

1.2 Virtual Network Elements

Virtual Network Element A virtual network element (VNE) is a software object that emulates the functions of network elements such as hosts and routers. It can emulate the common tasks performed by network elements such as transmitting packets generated by application programs as well as receiving and processing the packets sent over the network by other application programs.

We want to test real networks, but it is hard and expensive to do so as seen in Section 1.1. Virtual Network Elements offer an attractive alternative since they offer a common software framework which can be used to test a variety of network software while still remaining cost-effective. We are interested in emulating the network element only so much so as to be able to faithfully reproduce its interactions with the network which in its most basic form involves exchanging packets with other network elements. These interactions on a typical host could be familiar interactions such as file transfer through FTP or HTTP or newer ones such as watching a streaming video clip or IP telephony. On servers, these interactions could be serving hundreds of file transfer requests simultaneously or acting as a proxy for all the HTTP traffic of an organization or acting as a mail server and exchanging emails with other mail servers. On a router, these interactions can be commonplace routing of all incoming traffic or in case of newer routers classifying incoming traffic according to certain preconfigured management rules

and according due priority to certain forms of traffic. We do not care about emulating non-network interactions such as for example, reading a file from a disk or executing a program, etc. In the case of VNEs, these non-network interactions take place automatically as a side effect of the emulated network interactions. In short we want to emulate a host and a router in software and be able to configure these *virtual hosts* and *virtual routers* to form a *virtual network*. We are then interested in using these virtual networks to test new network technologies without having to setup large networks of physical machines. We are also interested in making multiple VNEs coexist in a single box to improve hardware utilization. Multiple virtual hosts can emulate a subnet while multiple virtual routers can effectively represent a core network.

Chapter 2 discusses some of the previous work that uses the concept of virtualization or virtual devices as well as the level of virtualization achieved and it's effectiveness. Chapter 3 describes the design and implementation of Virtual Network Elements. Chapter 4 is an overview of standard and hybrid QoS models which is the subject of this research and their implementation in Linux. Chapter 5 discusses the adaptations made to the QoS software to enable it to run in a virtualized environment along with the tools that were developed and improved to perform our experiments. Chapter 6 discusses the results of the virtualization experiments in contrast to live physical experiments. Finally, Chapter 7 lists the conclusions of this research and possible future extensions or improvements to the system.

Chapter 2

Related Work

2.1 NIST Network Emulation Tool

NIST Network Emulation Tool (NIST Net)[44] is a network emulation package that runs on Linux. NIST Net allows a single Linux PC set up as a router to emulate a wide variety of network conditions. The tool is designed to allow control of experiments involving network performance sensitive/adaptive applications protocols in a simple laboratory setting. NIST Net tries to emulate the critical end-to-end performance characteristics imposed by various wide area network situations such as packet delays, congestion loss, bandwidth limitation, and packet reordering or duplication.

NIST Net is a nifty emulation tool for emulation of certain conditions in a laboratory setting by inserting the NIST Net kernel module into the Linux kernel. However, it is typically intended to be used as a router that besides forwarding packets also subjects the packets flowing through it through a pre-configured series of conditions. Therefore, it only aims to emulate certain conditions in the network, not the network itself. Also, because of the way it is implemented, it only affects traffic coming into the machine, not the traffic leaving the machine. This is not helpful in cases where we need to study the output queuing properties of a router, for instance, in the case of Linux Traffic control.

2.2 IP Infusion

IP Infusion[33] is a provider of intelligent network software for enhanced IP services. Their specialty lies in building PC-based routers with advanced control plane software running on Linux, FreeBSD and Solaris. They have implemented a feature known as *virtual routing* that is defined to be an emulation of a physical router at the software levels. It has exactly the same mechanisms as a physical router, and therefore inherits all existing mechanisms and tools for configuration, deployment and operation.

Their implementation of virtual routing runs multiple instances of router and protocol code on a single box, each operating independently from one another and for a different purpose. Each instance maintains its own routing information base (RIB) and forwarding information base (FIB), thus isolating the actions of one virtual router from another. IP Infusion sells virtual routing solutions to service providers who can then compartmentalize their different customer networks onto different virtual routers thereby reducing deployment costs and improving management ability.

IP Infusion's implementation of Virtual routing is extremely interesting and some of their features would be a nice addition to our VNET implementation. However, their goals are very different from ours: whereas they merely want to isolate different customer networks being served by a single router, we aim to be able to emulate any physical IP network for evaluation purposes. We would eventually like to be able to implement multiple FIB and RIBs corresponding to each virtual router emulated on a machine. This would allow our virtual routers to directly interact with most routing software available.

2.3 Proportional Time Emulation and Simulation

Proportional Time Emulation and Simulation (ProTEuS) is a network evaluation framework developed at The University of Kansas[28, 29] for simulation of ATM networks using a concept of *proportional time*. It was designed to simulate complex ATM networks by virtualizing the time line in which the simulation is performed.

ProTEuS uses virtual ATM devices to emulate the ATM sources and sinks and

virtual switches device to emulate the ATM cell switch along with real-time extensions to Linux, KU Real Time (KURT), which allows it finer-grained control over network events and simulation execution. It is one of the projects besides the RDRN project[20], to utilize the concept of virtual devices successfully. ProTEuS uses the concept of an *epoch* to virtualize time. An epoch of execution in ProTEuS is the real time it takes to simulate a virtual time interval.

Since ProTEuS simulates ATM networks, the virtual ATM devices it uses are simpler than virtual Ethernet devices. The ProTEuS framework too has certain drawbacks some of which are listed below.

1. ATM networks have reached a dead-end in terms of technology and popularity. Most research is currently being carried out on IP networks with a dream to make them ubiquitous.
2. Virtual ATM devices have to handle fixed-length 53-byte cells; Virtual Ethernet devices have to handle packets of variable lengths between 46* and 1500 bytes.
3. Virtual ATM devices like their physical counterparts are essentially point-to-point devices. Thus, they are aware of their own address and the address of the peer device they are communicating with. These peer relationships are established as part of the initial virtual circuit (VC) setup. Virtual Ethernet devices aim to help emulate IP networks. Therefore, similar to a IP host, they only know about their *next-hop* destination for sending packets to different networks. IP routing takes over from there to eventually forward the packet toward it's destination through intermediate routers.
4. By design, ATM networks were designed to be switching networks, with high speed switches rapidly switching 53-byte cells. Hence the virtual switch used by ProTEuS simply has to keep a table of the VC connections established a priori, thus keeping the code simple. Switching is simply a matter of receiving a cell on one port, looking up the outgoing port in the table and sending it out via that port.

*The minimum size of an Ethernet frame is 46 bytes

Virtual routers on the other hand, only have a idea of their *next-door* neighbors. Thus, they have to run routing protocols to discover the network topology. Virtual routers also have to perform expensive *longest-prefix match* on the destination IP address in the packet to determine the best neighbor to use from the forwarding table to forward the packet toward it's ultimate destination.

5. The virtual switch implementation in ProTEuS uses independent implementations of various output scheduling algorithms such as Weighted Round Robin(WRR). These implementations are not very configurable and lack in performance. Virtual routers use the standard scheduling algorithms that are part of the Linux kernel, collectively called as Linux Traffic Control.

Despite these drawbacks, ProTEuS provides an excellent framework to simulate large and complex networks. Ultimately, we wish to integrate the virtual time properties of ProTEuS with our Virtual Ethernet devices (hosts and routers) to be able to simulate arbitrarily large and complex IP networks. We now look at the design and implementation of Virtual Network Elements over Ethernet in the next chapter.

Chapter 3

Virtual Network Elements

3.1 Overview

A typical network is comprised of three major elements: *hosts*, *routers* and *links*. Hosts and Routers are expensive hardware and require significant amounts of space. These drawbacks limit their numbers in a realistic experimental setup in academic and research organizations. This in turn limits the size of networks that can be emulated while experimenting with new network technologies.

We defined a Virtual Network Element (VNE) as a software object which emulates the functions of network elements such as hosts and routers. We have also hinted at the need for two kinds of virtual elements, a virtual host and a virtual router, to accomplish our goals of emulating larger networks. The concept of being able to put multiple virtual hosts and virtual routers on a single machine is a powerful one. It enables us to set up a comparatively larger and more complex *virtual networks* over a set of physical machines in a simple switched network. Combining many such virtual elements onto a single physical machine has the obvious advantage of reducing the number of machines required for an experiment, thereby reducing the cost of conducting an experiment.

Figure 3.1 shows a simple virtual network consisting of two subnets $10.1.0.0/16$ and $10.2.0.0/16$. Notice how multiple virtual hosts coexist on a single machine to form a subnet in the emulated network. Each of these virtual hosts have a unique *virtual* IP address so that they can communicate with each other unambiguously. This network is

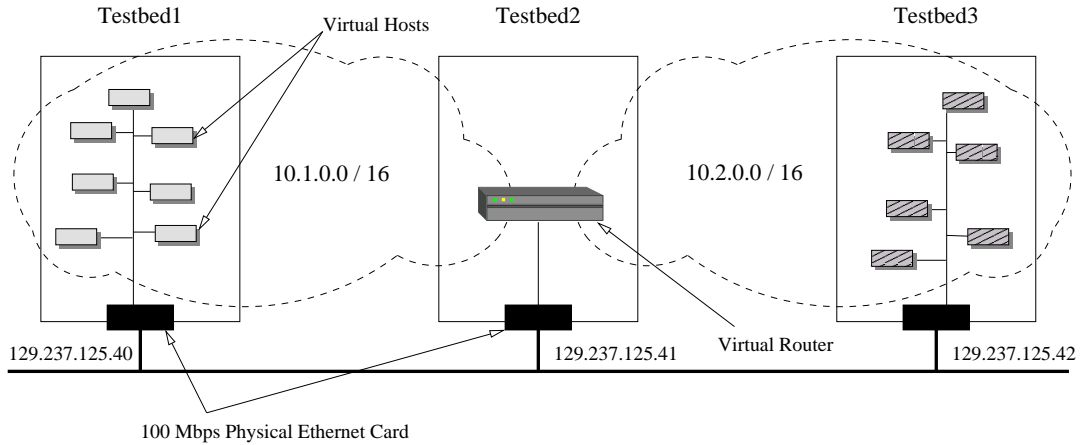


Figure 3.1: A simple virtual network

emulated over three machines: `testbed1`, `testbed2` and `testbed3` and all the virtual network traffic is multiplexed over the physical ethernet interfaces of the machines. A reader might question this design in which a single physical interface is carrying the traffic of multiple virtual interfaces. But we have to bear in mind, that the problem of QoS occurs only when a high-speed network (100 Mbps Ethernet intranet) has to send traffic over a low-speed link (e.g. 1.5 Mbps T1 link) to the core network. Thus, the virtual hosts represent the high-speed intranet which are vying for the capacity of the throttled, low-speed physical ethernet link.

Hosts and Routers communicate with the rest of the physical network through physical network interfaces; also called *ports* in routers. The task of a physical network interface is to send and receive packets over the network. These interfaces can run any link-layer technology (such as *Ethernet*, *ATM*, *Token ring*) as long as the operating system has a module to support the link-layer protocol. To be able to emulate a physical network element such as a host or a router, we need to be able to emulate in software their most basic component: the Network Interface. This software, the *Virtual Network Interface*, can run any link-layer protocol and multiplexes all its traffic over the underlying physical interface in the machine. Figure 3.2 shows the family of virtual network interfaces classified on the basis of the underlying physical interface and the

virtual interface software layer above. Thus we get four different combinations with the Ethernet and ATM link-layers: VETH over Ethernet, VATM over Ethernet, VETH over ATM and VATM over ATM. Various projects at the University of Kansas have utilized virtual devices in various forms such as ProTEuS [28] and Rapidly Deployable Radio Network [20]. Our choice of Ethernet as the underlying physical link-layer medium is motivated by its widespread, almost omnipresent deployment and lower costs as compared to technologies such as ATM. Since it runs Ethernet at the virtual layer too, our virtual network interfaces are VETH over Ethernet; simply known as the *Virtual Ethernet device* (VETH device) for the sake of brevity.

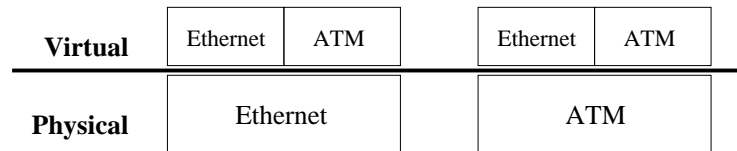


Figure 3.2: Family of Virtual devices

A VETH device has the same interface and data structures as the driver of a typical physical network interface whose only task is to send and receive packets over the network. Hence if each virtual network element has a VETH device associated with it, it provides a way for these virtual network elements to communicate. In other words, once we can create VETH devices which can send and receive packets, then combined with some private data structures and some book-keeping in the virtual network layer code, we can emulate virtual hosts and virtual routers. These data structures would be those that facilitate the multiplexing and de-multiplexing of packets for different VNEs onto the physical interfaces in the machine and those that store routing information for the virtual network.

We now look at the working of a network device driver to understand its internals. In particular, we are interested in the way in which the device driver receives and transmits packets, since that is the primary task of a VETH device.

3.2 Packets inside Linux

Before understanding how network device drivers manipulate packets, we present a quick overview of how packets are stored in Linux. Each packet handled by the Linux kernel is contained in a socket buffer structure, `struct sk_buff`. The structure gets its name from the Unix abstraction used to represent a network connection, the *socket*. The same `sk_buff` structure is used to hold network data throughout the Linux network subsystems, but it is a *packet* as far as the interface is concerned.

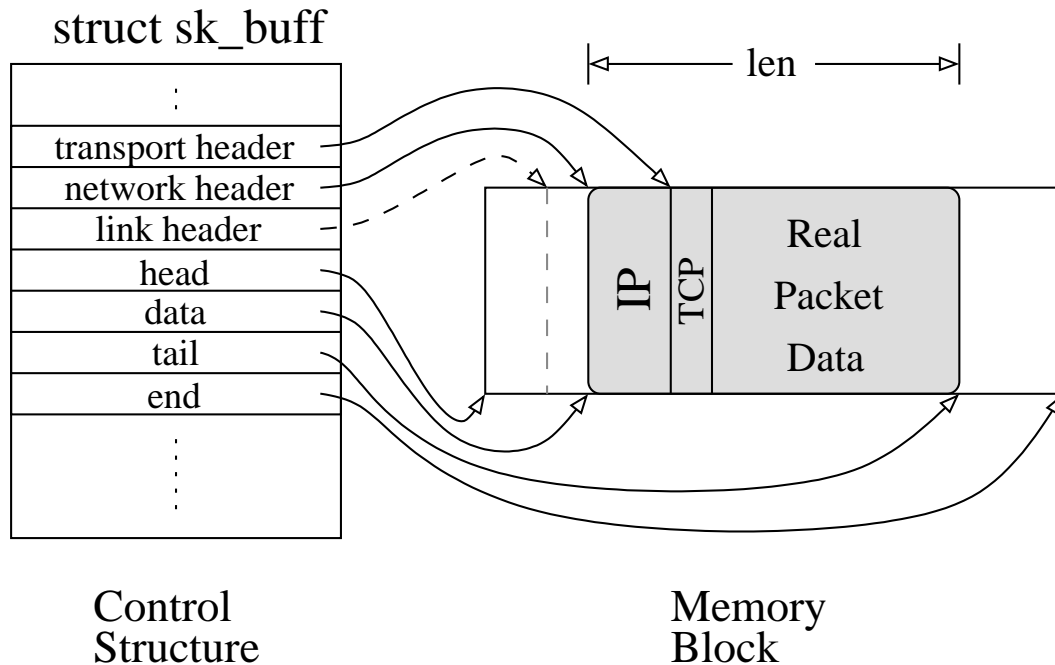


Figure 3.3: The `sk_buff` structure

Thus, when a socket application sends some data over the socket, a `sk_buff` buffer is allocated enough space to store the data as well as corresponding protocol headers that are added as the packet travels down the protocol stack to the network device. A `sk_buff` provides a control structure with a block of memory attached. The control structure includes pointers which point to various parts of the attached memory block to denote the start of various protocol headers and data. As shown in Figure 3.3, the beginning

and ending of the memory block is denoted by the `head` and `end` pointers respectively, while the beginning and ending of the real packet data is denoted by the `data` and `tail` pointers in `struct sk_buff`. The `data` and `tail` pointers are modified as the headers and trailers are appended to the packet as it travels inside the protocol stack. Other control structures point to the network layer header (TCP, UDP or ICMP), transport layer header (IP, IPv6 or ARP) and link layer header (RAW or Ethernet) *inside* the packet data. Thus, a packet just about to be transmitted (hence at the link layer), will have the `data` pointer pointing to the same location as the link layer header pointer. Similarly, in the transport layer, the `data` pointer will point to the same location in the packet as the transport layer header, say TCP header.

3.3 The Linux Network bottom-half

In subsequent sections, we will discuss the movement of packets in the network protocol stack of Linux. The concept of bottom-halves is very important to understand packet arrival in Linux. Packet arrival takes place in two stages: at the network interface level (hardware) and at the kernel level (software).

When a packet arrives at the network interface, the interface triggers a hardware interrupt which causes the CPU to temporarily suspend its work to handle the interrupt. In order to do so, the CPU disables all other interrupts*, and calls the interrupt handler, which is a routine in the device driver of the network interface that transfers the packet from the network interface's hardware queue to a *software* queue inside the network stack for further processing. Since the interrupts are disabled, all interrupts of the same or lower priority are ignored. This could mean that the kernel might miss other incoming packets. In order to minimize the chances of missing a packet, the interrupt handler has to be an extremely fast and efficient routine.

Instead of further analyzing the packet to send it to the correct protocol handler, the interrupt handler simply puts the packet onto the *backlog* queue by calling `netif_rx`, marks the network bottom-half flag and exits. Thus the packet is held for further pro-

*unless they have a higher-priority than the network interrupt

cessing and the interrupts are re-enabled. Now, Linux periodically checks the bottom-half flags of various sub-systems to check if any of them need processing. When it finds the network bottom-half flag set, it calls `net_bh` which then analyzes the packet and passes it off to the correct protocol handler.

This technique avoids extended interrupt disabling and offers an overall optimum processing for incoming packets.

3.4 Overview of a Linux network device driver

A network interface communicates with the operating system through software modules known as *device drivers**. The device driver hides the specifics of a particular network interface's hardware from the operating system. Due to the variety of network interfaces available in the market with different hardware architectures, the Linux kernel uses an object-oriented methodology to provide a common interface (`struct device`) to the device drivers to communicate with it. Thus `struct device` (Program 3.1) is an abstraction through which the kernel communicates with the network interface. The device driver implements the *methods* defined in the data structure (as function pointers) and fills in data pertaining to the capabilities of the device. Every network interface (hardware) thus has an instance of `struct device` (software) associated with it in the Linux kernel.

The device driver defines a *receive* (RX) routine which responds to interrupts from the hardware. It also defines a *transmit* (TX) routine which is called by the IP layer when it has a packet to transmit through that device. Figure 3.4 shows a schematic of the working of RX and TX in the driver. It does not show the queues associated with each device and the packet scheduling routines used to service those queues since they are not relevant until later (discussed in Section 3.7).

Once the network device driver has filled in all the fields in Program 3.1 besides a few others pertaining to memory addresses, I/O addresses and IRQ numbers to use, it registers itself with the kernel by making a call to `register_netdevice()`. This

*Refer to Rubini's excellent book *Linux Device Drivers* [48] for details on device drivers

Program 3.1 Important members of struct device

```
struct device
{
    char            *name;
    unsigned long   tbusy;                /* transmitter busy */

    /* Device initialization function */
    int             (*init)(struct device *dev);

    /* Interface index. Unique device identifier */
    int             ifindex;
    int             iflink;

    struct net_device_stats* (*get_stats)(struct device *dev);

    unsigned long   trans_start;          /* Time of last Tx */
    unsigned long   last_rx;              /* Time of last Rx */

    unsigned short  flags;                /* interface flags */
    unsigned        mtu;                  /* interface MTU */
    unsigned short  type;                 /* hardware type */
    unsigned short  hard_header_len;
    void            *priv;                /* pointer to private data */

    /* Interface address info */
    unsigned char   broadcast[MAX_ADDR_LEN]; /* hw bcast add */
    unsigned char   dev_addr[MAX_ADDR_LEN]; /* hw address */
    unsigned char   addr_len;             /* hw addr len */

    struct Qdisc    *qdisc;
    struct Qdisc    *qdisc_sleeping;
    struct Qdisc    *qdisc_list;
    unsigned long   tx_queue_len;

    /* Pointers to interface service routines */
    int (*open)(struct device *dev);
    int (*stop)(struct device *dev);
    int (*hard_start_xmit) (struct sk_buff *skb, struct device *dev);
    int (*hard_header) (struct sk_buff *skb, struct device *dev,
                        unsigned short type, void *daddr,
                        void *saddr, unsigned len);
    int (*rebuild_header)(struct sk_buff *skb);
    int (*do_ioctl)(struct device *dev, struct ifreq *ifr, int cmd);
    int (*neigh_setup)(struct device *dev, struct neigh_parms *);
};
```

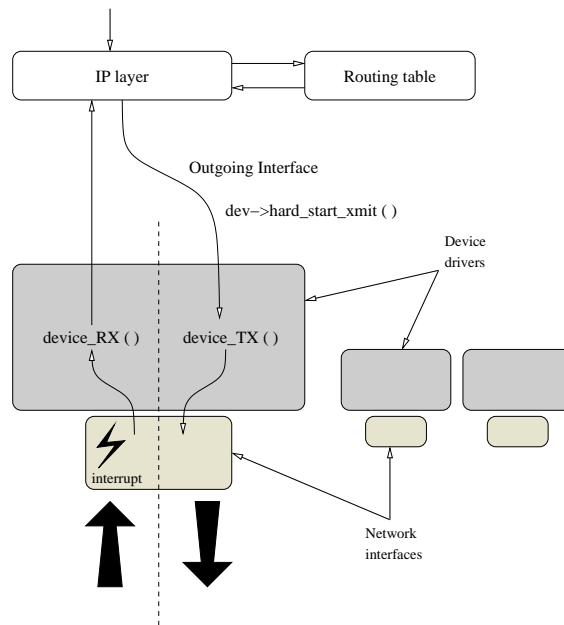


Figure 3.4: Working of RX and TX in a network driver

completes the initialization of the device and it can now be configured using a user-level program such as `ifconfig`. Socket applications should now be able to use this device, provided that the routes corresponding to the device are present in the kernel's routing table (see Section 3.6).

Figure 3.5 shows the path of packet on its way from the application layer to the data-link layer. Depending on the type of socket created in the application layer, the packet takes different paths to the IP layer. At the IP layer, the following operations are performed in order:

1. The link-layer-specific header is prepended to the packet by `dev->hard_header()`.
2. `ip_route_output()` uses the destination address of the packet to find its route (outgoing interface).
3. The packet is scheduled to be transmitted through the device returned by the route lookup using `dev_queue_xmit()`.
4. The device queue is polled in `qdisc_restart()` and depending on the scheduling

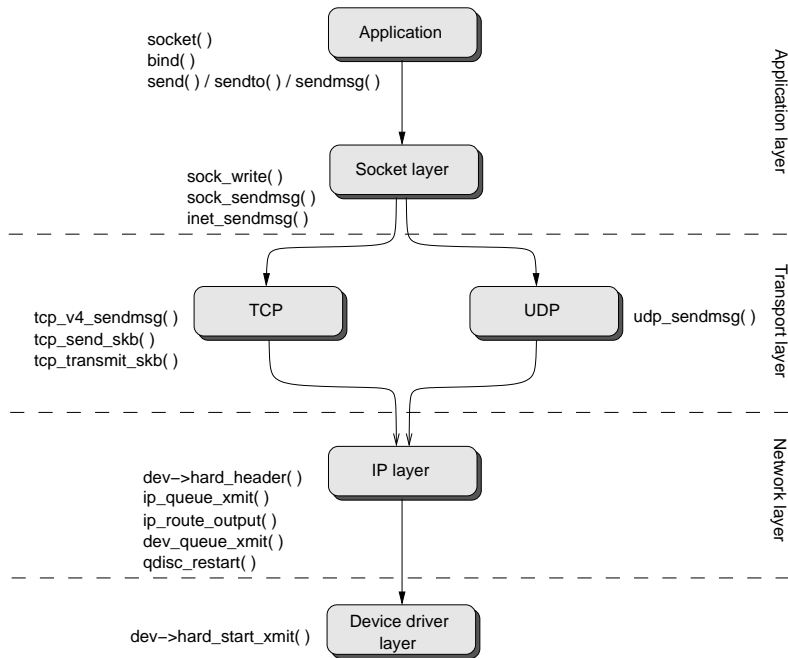


Figure 3.5: Packet flow down the Linux protocol stack

discipline associated with the device, the packet is kicked out of the queue onto the link by calling the device-driver-specific routine, `dev->hard_start_xmit()`.

The code in the device driver corresponding to `dev->hard_start_xmit()` takes care of the details of transmitting a packet through the network interface onto the physical medium. Thus it will be hardware-dependent and change with different network interfaces.

Figure 3.6 shows the path of the packet to the application layer after it is received by the network interface. The device driver receives an *interrupt* when a packet arrives on the wire. It then does the following:

1. The driver allocates a socket buffer for the packet through `dev_alloc_skb()` and queues the packet in the `backlog` queue and marks the network bottom half for later processing in `netif_rx()`.
2. The network bottom-half, `net_bh()`, on being scheduled by the Linux scheduler,

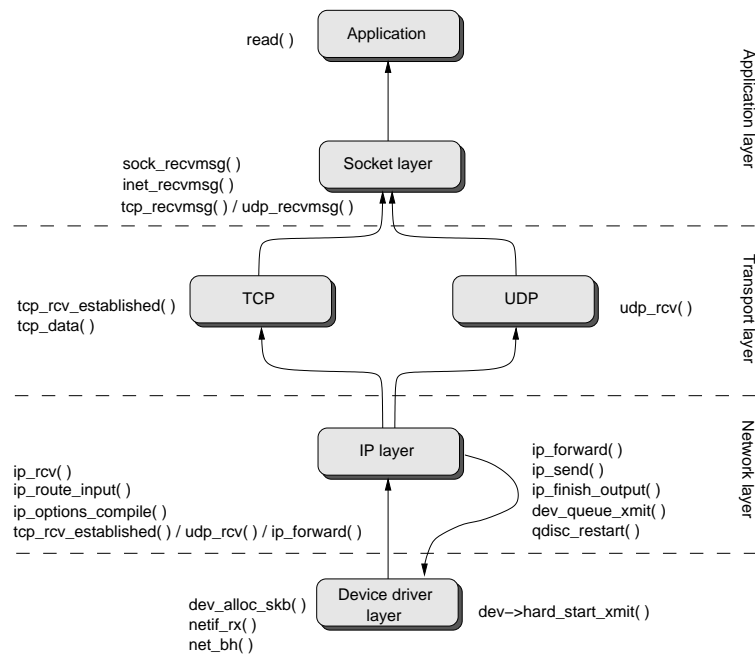


Figure 3.6: Packet flow up the Linux protocol stack

determines the protocol of the packet and passes the packet to the appropriate protocol handler e.g. `ip_rcv()` for IP packets.

3. The `ip_rcv()` routine determines whether the packet is to be locally delivered or forwarded (in case of routers) by calling `ip_route_input()`.
 - (a) If the packet is to be forwarded, it is processed (TTL decrement, fragmentation etc.) and eventually sent out via `dev->hard_start_xmit()`.
 - (b) In case of local delivery, depending on the transport protocol, either the TCP or UDP **receive** function is called and the packet added to the socket queue. A `read()` call by the application then reads data corresponding to the socket.

Armed with knowledge of the working of a network device driver and flow of packets through the Linux protocol stack, we are now ready to look at the design of a Virtual Network Interface which can then be used to realize Virtual Hosts and Virtual Routers.

3.5 Design of a Virtual Ethernet Device Driver

We start by identifying the features and goals of the VETH device:

1. *Creation/Deletion*: Ability to add and destroy VETH devices at will from user-space programs.
2. *Configurability*: Ability to configure the properties of the VETH device.
3. *Socket layer compatibility*: Ability to use existing socket-based applications with a VETH device with little or no modifications.

As alluded to previously, a *correctly* initialized `struct device` represents a *real* network interface to the protocol stack. The VETH device driver uses this property of the kernel to *instantiate* and register multiple instances of `struct device` thus leading the kernel, and therefore the applications, to believe that there are more network interfaces available on the machine than there really are. In addition to instantiating `struct device`, it adds some book-keeping code which we call the *Virtual Network Layer* (VNET). The VNET layer is inserted between the IP and device-driver layers. This is by far the most convenient and flexible place in terms of modifications to the existing kernel source code: it requires none! All the VNET layer code is self-contained and provides hooks to both the IP layer and device-driver layer, forming a bridge between them.

If we had tried to move the VNET layer above the IP layer, we would have to make it behave like a transport protocol layer and make modifications to IP to understand the new protocol. If we had moved the VNET layer to the device-driver layer, we would have been constrained to use only a particular network device driver without the ability to use different cards.

The power of virtual devices lies in the fact that from the protocol stack entry-point and up (`qdisc_restart()` in Figure 3.5), kernel and user-level entities are oblivious to the fact that the devices are virtual; *they do not know or care*. Similarly, from the physical device driver entry-point and down, the physical devices have *no idea* that the virtual devices are not actually protocols. In short, the IP protocol stack and socket

layer think of the VNET layer as a real physical interface and the device drivers for the physical interfaces think of it as a protocol.

The ability to create, delete and configure VETH devices from user-space requires us to use `ioctl()` calls from user-level programs to exchange data with predefined routines in the kernel-space. To achieve this, we first create a permanent virtual device, `vnet`, in `drivers/net/Space.c` which is initialized at the time of kernel bootup. It gives us a *master device* which initializes the VNET layer through a call to `vnet_layer_init()` in `drivers/net/vnet/vnet.c`. The initialization provides `vnet_ioctl()` as the routine to handle all subsequent `ioctl()` calls from user-space. This bootstrapping process now allows us to pass information back and forth between user-level programs and the `vnet` device. We can thus write `ioctl()` calls to add, delete and configure VETH devices or other virtual elements. Actual details about each `ioctl` is documented elsewhere [30]. Figure 3.7 shows how the virtual network layer fits into the protocol stack.

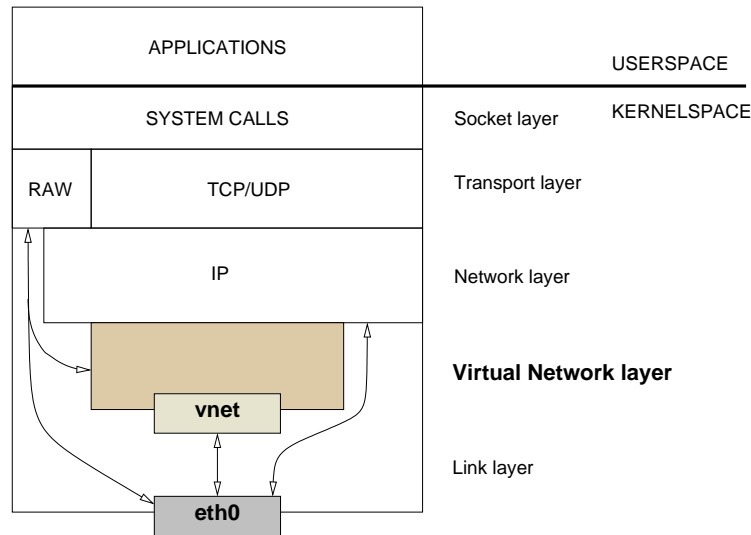


Figure 3.7: The Virtual Network layer in the Protocol Stack

Once the devices are created, a user-level program such as `ifconfig` can be used to configure their IP addresses, netmask and gateway addresses. These VETH devices in this form will behave like ordinary interfaces, in that the socket applications can use the `bind()`, `connect()`, `accept()`, etc. system calls on them. But they cannot be

employed in useful applications due to the following problems:

1. There is no way to transmit packets over the network to other VETH devices on other machines over the physical medium.
2. Since this is a software-only implementation, it cannot receive interrupts from the hardware on packet arrival.
3. There is no way to distinguish packets meant for the virtual network from other packets arriving at the physical interfaces.

Since the VETH device driver is not associated with any real hardware, in order for it to be able to transmit packets, it will have to use the physical interfaces present on the machine. In essence, what this means is that when a packet is scheduled to be transmitted through the VETH device* the `hard_start_xmit()` code of the VETH device driver should call the corresponding transmit routine belonging to the underlying physical interface's device driver. In short, the VETH devices will use the underlying physical interfaces to carry their packets.

Once we gain the ability to transmit a packet over VETH devices by multiplexing them over the physical interface, we need the ability to receive these packets on other machines and be able to distinguish these packets as belonging to the virtual network. Our solution is to define a new Ethernet packet type, `ETH_P_KUVNET`, defined in `include/linux/if_ether.h`, which denotes a packet belonging to the virtual network. Linux uses a table of predefined Ethernet packet types and their corresponding packet-handling routines (`receive` routines of the respective protocols) to hand-off the various incoming packets to the correct protocol handlers. Hence the advantages of adding a new packet type are twofold:

1. It allows unambiguous identification of packets meant for the virtual network.
2. Adding a new packet type allows us to specify its corresponding protocol handler.

Thus the network bottom half which processes the queue of incoming packets di-

*How the IP layer chooses the VETH device for transmission is discussed in Section 3.6.

rectly hands-off these packets to our VETH device driver's `receive` routine thinking of the VNET layer as another layer 3 protocol like IP, ARP, etc.

Having acquired the ability to send and receive packets over VETH devices, we introduce `vnet_xmit()` and `vnet_rcv()` as the `transmit` and `receive` routines of the virtual network layer respectively. The new packet type is registered using `dev_add_pack()` and associated with the `vnet_rcv()` routine which is therefore designed to receive all packets belonging to the virtual network. Similarly, the transmit routine, `vnet_xmit()`, is added to accept all packets originating from the socket layer with a destination IP address in the virtual network. All packets meant for a node of the virtual network are multiplexed and de-multiplexed at the `vnet` device as shown in Figure 3.8. Thus if two VETH devices `veth1` and `veth2` are both receiving packets, then all their packets would first arrive at the `vnet_rcv()` routine where they would be de-multiplexed by destination address and sent to the respective VETH devices.

To summarize the solutions to the aforementioned problems in using VETH devices:

1. Definition of a new packet type, `ETH_P_KUVNET`, for the virtual network, which allows us to unambiguously distinguish incoming packets meant for the virtual network.
2. Multiplexing all the packets for the VETH devices onto the physical interfaces of the machine.

3.6 The Routing Table

The Linux kernel maintains a routing table which lists all the hosts and networks that can be reached via the physical interfaces connected to the machine; either directly or through a gateway. It may also contain generic entries for *default gateways* which are routers to which it can send all packets which cannot be routed using the more specific entries in the local routing table. When the IP layer needs to transmit a packet, it looks in the routing table to find the best path available through which to send the packet. The `ip_route_output()` routine provides information about the outgoing interface over which the packet should be transmitted to reach its destination.

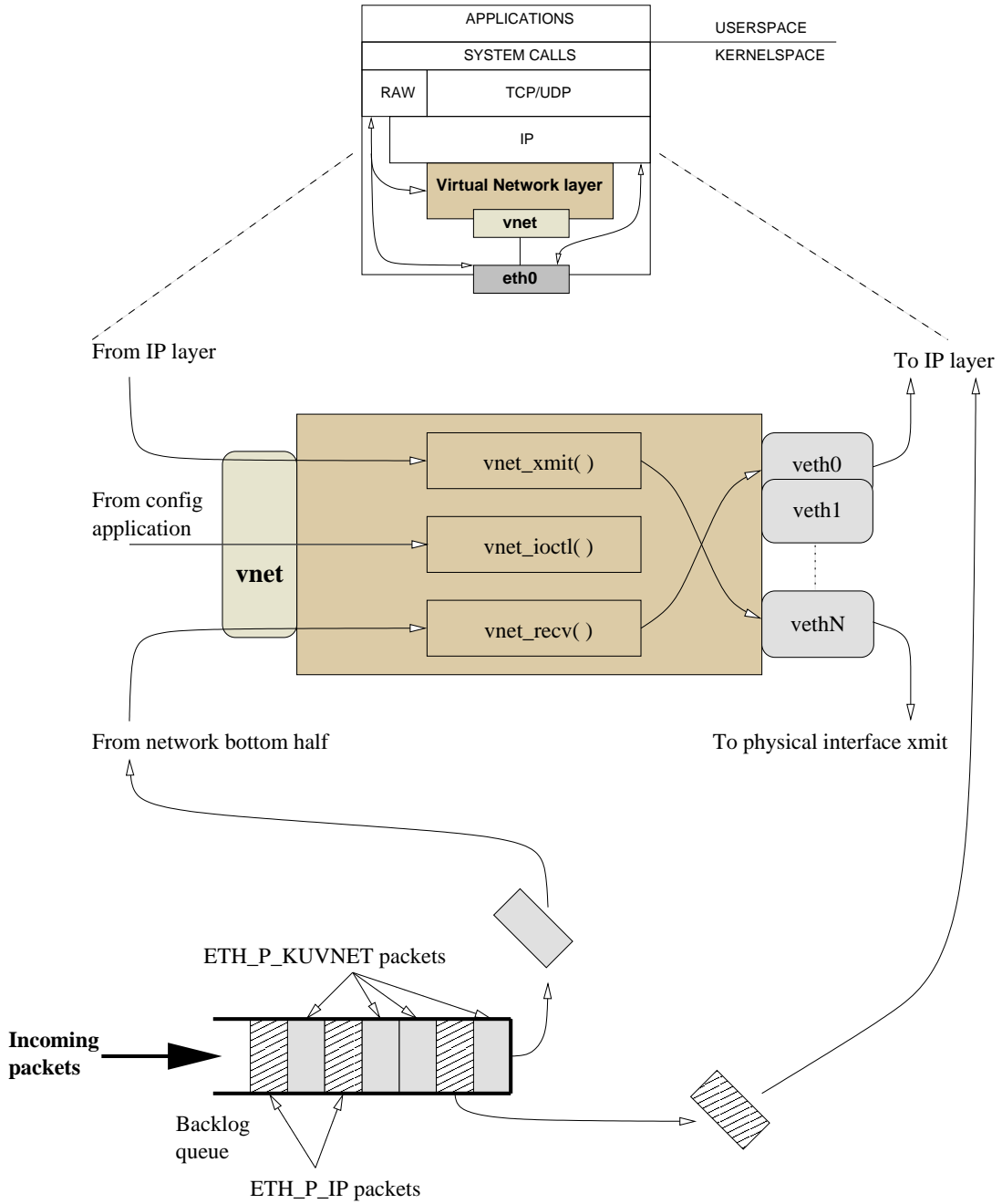


Figure 3.8: Functioning of the Virtual Network Layer

When a VETH device is activated using the `ifconfig` command such as

```
ifconfig veth0 10.1.1.1 up
```

it adds to the routing table a host entry corresponding to the IP address of the interface and a network entry to the 10.0.0.0 network via `veth0`. Now if we initialize another VETH device `veth2` (10.2.1.1), we get a host entry corresponding to 10.2.1.1, but we also get another network entry to 10.0.0.0, this time via `veth2`. This method of configuring the routing table for the virtual network has the following drawbacks:

1. We *pollute* the system routing table with entries for the virtual network.
2. There is no way to have distinct sets of routing entries for different sets of interfaces on the machine.
3. As a corollary to the second drawback, we cannot emulate a real host having its own routing table. e.g. If we have two VETH devices each acting as an interface of a different virtual host, then, for the two virtual hosts to realistically emulate two physical hosts, a routing table should be associated with each of the VETH devices.

Thus, creating multiple virtual devices is only part of the problem; making them communicate and behave like hosts (*virtual hosts*) is a new problem. This problem is amplified because we want to avoid manipulating the IP layer of Linux to achieve our objectives and avoid using the IP layer route lookup to find the route for a packet in the virtual network.

To alleviate these drawbacks we introduce the concept of *virtual routing tables*. A virtual routing table has the following features:

1. It can be *associated* with a VETH device.
2. It reflects only the virtual network atop the physical network.
3. It can be shared between multiple VETH devices. This feature is useful in emulating a multi-homed host or a router since they have a single routing table with multiple interfaces listed in it.

An entry in the virtual routing table is of the form `<Destination, Gateway, Netmask, Flag, Interface>`, similar to a kernel routing table. We have written routines to search these virtual routing tables and come up with a best-prefix match. With the introduction of virtual routing tables, we now needed a mechanism to effectively bypass the IP layer's route lookup i.e. we want to find the routes in the virtual network using the virtual routing table associated with the VETH device from where the packet originated*. We achieve this by using the following mechanism:

- First select an IP address range which we want to use in virtual network for our experiments. e.g. `10.0.0.0/8`.
- Define a single entry in the kernel routing table for this network with `vnet` as the outgoing interface. e.g. `route add -net 10.0.0.0/8 dev vnet`

Whenever a packet with a destination address in the `10.0.0.0/8` network has to be transmitted, the IP layer route lookup selects `vnet` as the outgoing interface and calls its corresponding `hard_start_xmit()` routine, which happens to be `vnet_xmit()`. Thus, as shown in Figure 3.8 on Page 25, all packets transmitted via VETH devices enter the virtual network book-keeping code through a single routine, `vnet_xmit()`. Once the packet enters `vnet_xmit()`, we can then find out the correct source[†] address of the packet, use it to query the correct virtual routing table and find the egress virtual interface for the packet. Thus we succeed in bypassing the IP level route lookup by forcing the IP layer send all packets in the `10.0.0.0/8` range to the VNET book-keeping code. Once we get control, we can process the packet according to the virtual network book-keeping rules. We are thus in a position to make a VETH device behave like an interface of a virtual host. A note of caution is in order here; If the IP layer finds the destination or next-hop IP address in the packet to be associated with an interface on the same machine, it *short-circuits* the normal path taken by the packet to the link layer and directly transfers the packet from the sender's queue into the receiver's queue. This has important consequences for designing a virtual network since we now have

*or from where it entered the virtual network

†or entry-point into the virtual network

the following limitations while designing the mapping of the virtual network to a set of physical machines. Two important limitations of the existing implementation are:

Same host limitation A source and destination host in a virtual network *cannot* be hosted on the same machine. If a source and destination host are on the same machine, the IP layer would transfer the packet from the source queue to the destination queue without bothering to check for the next-hop.

Connected-elements limitation Two directly connected network elements *may not* be hosted on the same machine if we want the packets to pass through the traffic control code for implementing QoS. In case of two connected routers hosted on the same machine, packets between the connected ports will not go down the protocol stack to the traffic control layer.

Due to the routing capabilities added due to virtual routing tables, we added two more goals to the original set of goals for the VETH device:

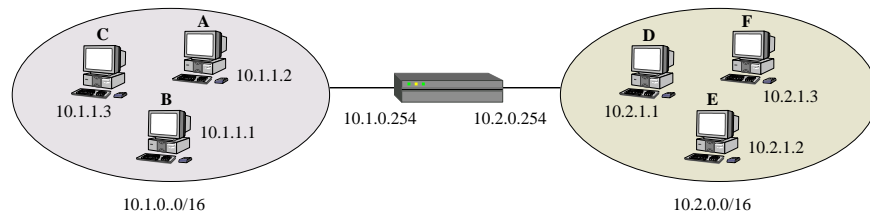
4. *Communication*: The VETH devices should be able to communicate with each other, even across physical machine boundaries
5. *Network emulation*: Ability to emulate arbitrary, non-trivial network topologies on fewer physical machines

Just the virtual routing tables are not sufficient to be able to communicate with VETH devices across physical machine boundaries. This is due to the fact that VETH device *do not* support the Address Resolution Protocol (ARP)*. Without ARP, it is impossible for the transmitting side to know which machine is hosting the destination VETH devices. Since packets for the virtual network flow via the physical interfaces, we need to know the IP addresses and link-layer addresses of all the machines which host the virtual network. We bypass the need for ARP by using a simple table containing IP address and link-layer address[†] pairs of all the machines hosting the virtual network. This table is loaded into the memory of each of these machines. Now, although we

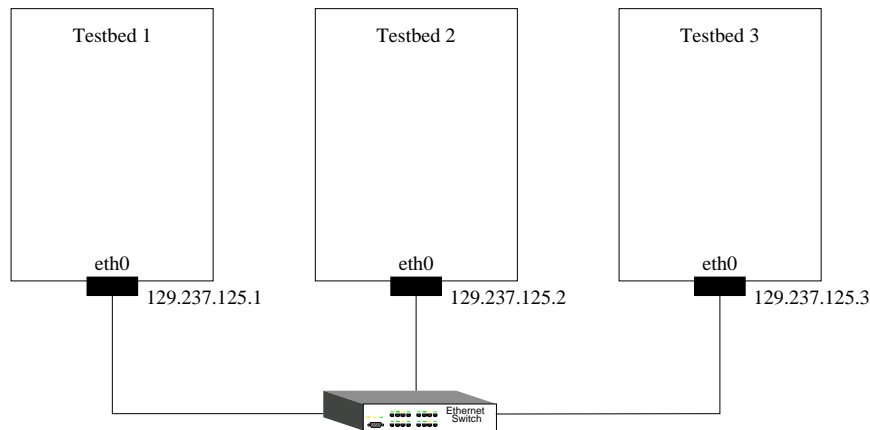
*See [?] for notes on why ARP is not supported

[†]MAC address in case of Ethernet

know the identities of all the machines participating in hosting the virtual network, we still cannot answer the query of a transmitting VETH device: *Which machine hosts a VETH device with the IP address 10.2.1.1?* This is because the virtual network has a different structure atop the physical network. What we need is a *map* of the virtual network as emulated on the physical network. Consider Figures 3.9 and 3.10 to understand this problem. Figure 3.9 depicts a emulation target network and the physical network hosting the virtual network while Figure 3.10 depicts two ways to *virtualize* the target network. We take into account the limitations in designing virtual networks alluded to previously.



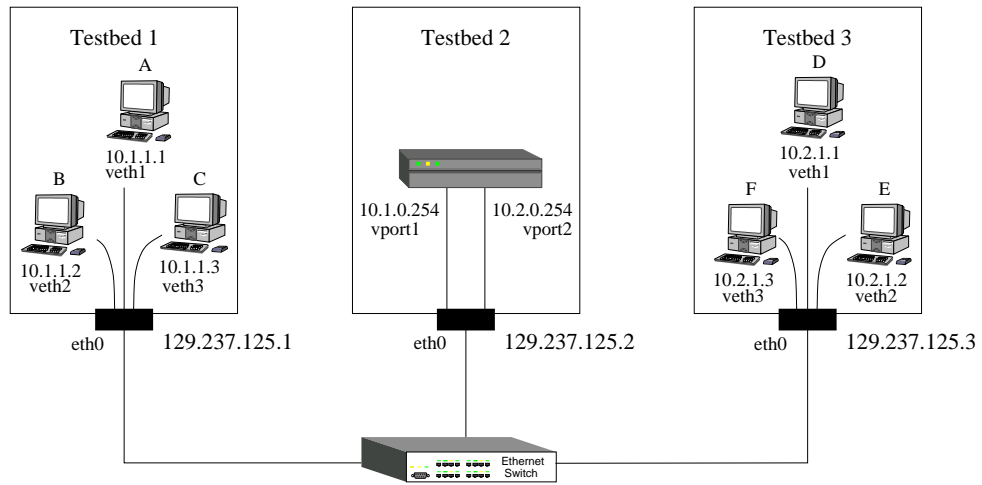
(a) Emulation Target Network



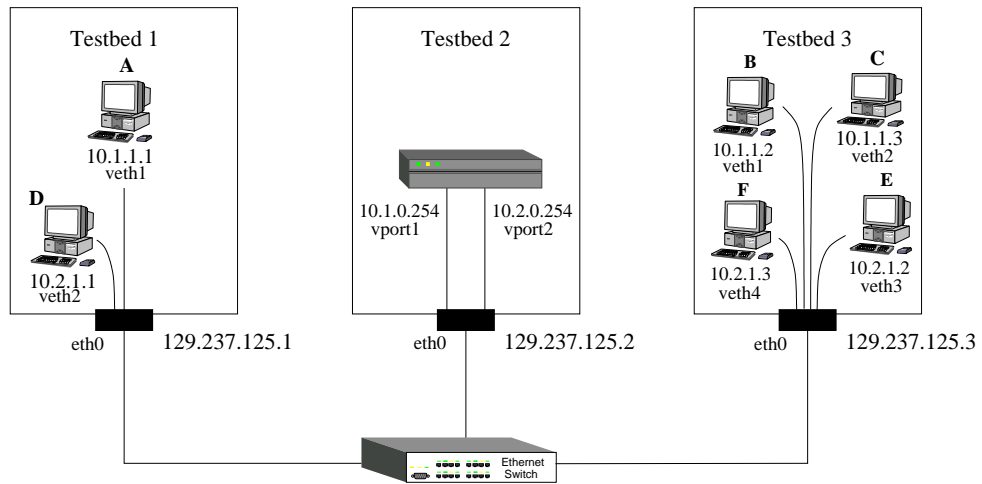
(b) Physical network hosting the virtual network

Figure 3.9: Goals of virtualization

Figure 3.9(a) shows the target network to be emulated which consists of two sub-



(a) Simplest virtual network mapping



(b) A function-based virtual network mapping

Figure 3.10: Virtualization of the sample network

Destination	Gateway	Netmask	Flag	Interface
10.1.1.1	0.0.0.0	255.255.255.255	H	veth1
10.1.0.0	0.0.0.0	255.255.255.255	N	veth1
0.0.0.0	10.1.0.254	0.0.0.0	G	veth1

Table 3.1: Virtual routing table for virtual host A

nets `10.1.0.0/16` and `10.2.0.0/16` connected through a router. Let us assume hosts A and D to be servers for clients pairs [F, E] and [B, C] respectively. Also assume that servers A and D need to do CPU intensive tasks, so that they benefit by using faster machines. Figure 3.9(b) shows the physical network comprising of three machines `testbed1`, `testbed2` and `testbed3`. Each of these machines has a single Ethernet interface (`eth0`) and all of them are plugged into a single Ethernet switch. Assume that `testbed1` is a high speed machine. Figure 3.10(a) shows the most obvious mapping of the sample network over the physical network. Each subnet is mapped onto one of the machines while the router is hosted on the remaining machine. In this case, each of the virtual hosts will have a virtual routing table similar to that shown in Table 3.1. The first entry is the *host entry* used to describe the virtual interface itself, the second is the *network entry* used to describe a directly connected network. In this case, since `10.1.1.1` is on the `10.1.0.0/16` network, it is directly connected. The third entry is a *gateway entry* specifying routes to networks that are not directly connected. In this case, anything other than `10.1.0.0/16` is not directly connected and should be redirected to the router on port `10.1.0.254`. Table 3.2 shows a similar virtual routing table for the router, the only difference being that this routing table is *shared* by the two ports of the virtual router similar to what happens on a real router. Hence we

Destination	Gateway	Netmask	Flag	Interface
10.1.0.254	0.0.0.0	255.255.255.255	H	vport1
10.2.0.254	0.0.0.0	255.255.255.255	H	vport2
10.1.0.0	0.0.0.0	255.255.255.255	N	vport1
10.2.0.0	0.0.0.0	255.255.255.255	N	vport2

Table 3.2: Virtual routing table for virtual router

see entries for both interfaces `vport1` and `vport2` showing up in the routing table. We could also configure the virtual network as shown in figure 3.10(b) with both the servers on `testbed1` to be able to take advantage of its faster CPU.

Destination	Gateway	Netmask	Flag	Interface
10.1.0.0	129.237.125.1	255.255.0.0	P	vport1
10.2.0.0	129.237.125.3	255.255.0.0	P	vport2

Table 3.3: Subnet Map table for virtual router in Figure 3.10(a)

Destination	Gateway	Netmask	Flag	Interface
10.1.1.1	129.237.125.1	0.0.0.0	P	vport1
10.2.1.1	129.237.125.1	0.0.0.0	P	vport1
10.1.0.0	129.237.125.3	255.255.0.0	P	vport2
10.2.0.0	129.237.125.3	255.255.0.0	P	vport2

Table 3.4: Subnet Map table for virtual router in Figure 3.10(b)

These different configurations for the virtual network show that we need to supplement the virtual routing tables with more information about the topology of the virtual network to be able to identify the physical machine hosting a given VETH device. This information is known as the *subnet map*. A subnet map is very similar to the virtual routing table in structure except that there is only one *type* of entry: the *physical entry* denoted by 'P'. There is one subnet map for each routing table and it contains information about *where* certain subnets or hosts are located. Tables 3.3 and 3.4 show the subnet map for the virtual router for virtual network architectures in figures 3.10(a) and 3.10(b) respectively.

3.7 Virtual Host and Virtual Router

We now have all the information we need associated with a VETH device. Using this information, a VETH device in the form of a virtual host or virtual router can emulate a physical host or a physical router. But the internals of the virtual network layer code need to change to accommodate the added complexity of supporting these virtual network

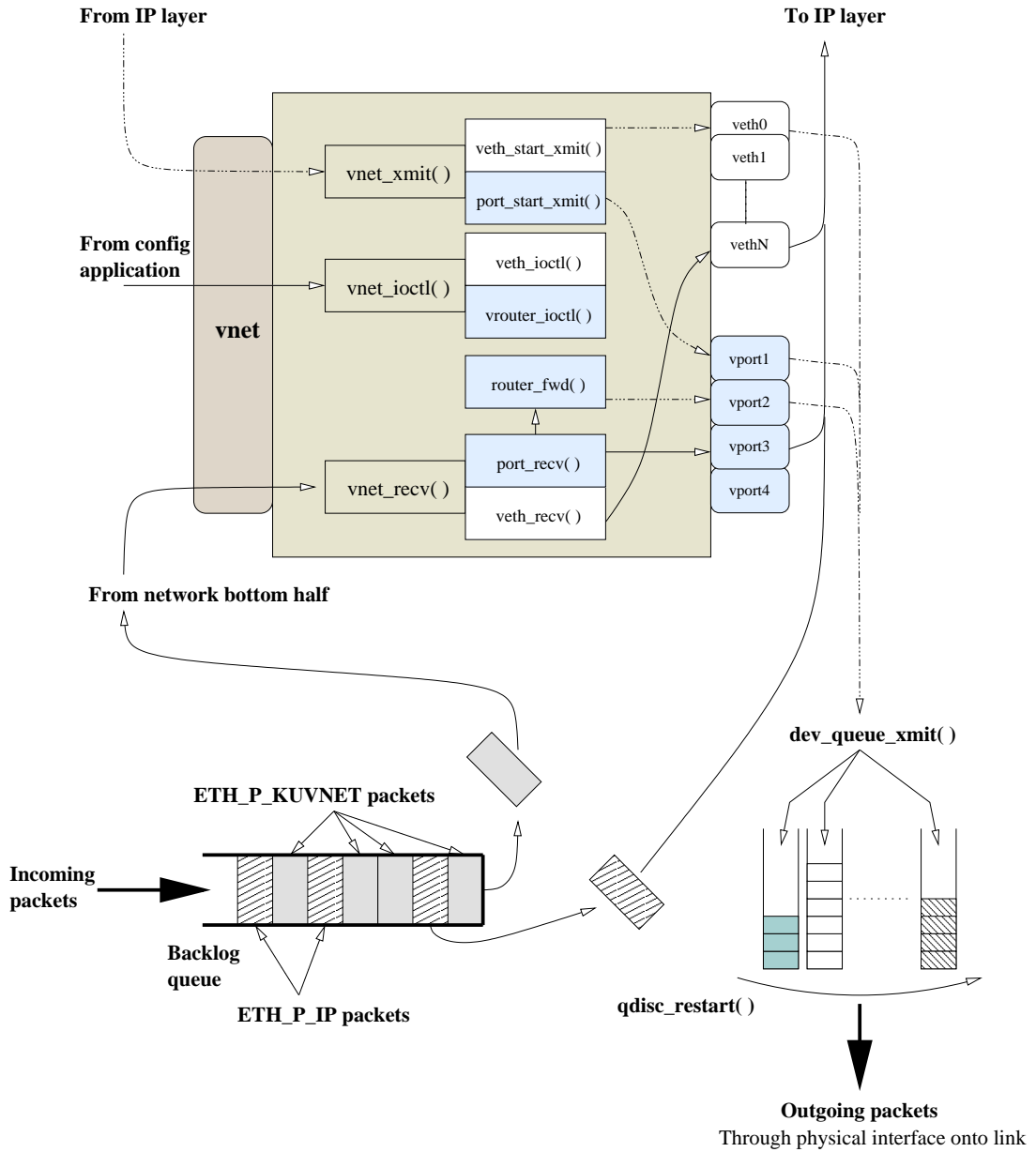


Figure 3.11: Functioning of the Virtual Network layer (modified)

elements. Thus the `vnet_recv()` and `vnet_xmit()` routine have to be intelligent enough to determine whether the packet should be associated with a virtual host or a port of the virtual router, since they can coexist on the same machine. The modified virtual network layer is shown in Figure 3.11. The dotted lines show packets being transmitted from the machine and solid lines show packets being received by the machine. A packet received by a machine enters the virtual network layer of a machine through `vnet_recv()`^{*} and is checked for its *next-hop* IP address. If the next-hop address belongs to a VETH device (interface of a virtual host), the packet has reached its final destination and is passed to `veth_recv()` which executes the book-keeping code for the particular VETH device and passes the packet to the IP layer. If the next-hop address belongs to a VPORT device (port of a virtual router), it needs to be forwarded in the virtual network toward its ultimate destination. Thus, it is passed to `port_recv()` and then onto `router_fwd()` which sends it on its way out according to the information in the virtual routing table. A packet originating from a machine and destined to the address range designated for the virtual network[†] makes its way to `vnet_xmit()` from the IP layer. In `vnet_xmit()`, depending on whether the source of the packet was a VETH device or a VPORT device, the packet is passed to `veth_start_xmit()` or `port_start_xmit()` which calls `dev_queue_xmit()` to schedule the packet[‡] to be transmitted over the physical interfaces associated with the virtual elements.

Table 3.5 lists some of the `ioctl()` calls used to setup and configure virtual hosts and virtual routers. These include calls to create a virtual host (`SIOCCREATEVETH`, `SIOCCREATEVRT` and `SIOCCREATEMAP`), create a virtual router (`SIOCCREATEVRROUTER`, `SIOCADDPORT` and `SIOCCREATEROUTERRT`) and calls to query information from user-space (`SIOCGETROUTE` and `SIOCGETROUTERPORT`). We have created a Netspec daemon `nsvethd` that parses a virtual network configuration script and calls the correct `ioctls` to configure the virtual network. It is discussed in greater detail in Section 5.2.2. Section 6.1 describes a physical network topology and the process of the setting up an equivalent virtual network topology. It also describes the pros and cons of various equivalent

^{*}because it is of type `ETHLP_KUVNET`

[†]As described in Section 3.6

[‡]According to the scheduling disciplines attached to the devices

VNET layer ioctls	
SIOCCREATEARPTAB	Create an ARP table for use by virtual network elements
SIOCGETROUTE	Find outgoing interface for a packet given it's final destination
Virtual Host ioctls	
SIOCCREATEVETH	Create Virtual Ethernet Device
SIOCCREATEVRT	Create a Routing Table associated with a particular device
SIOCCREATEMAP	Create a table depicting Virtual to Physical mapping.
SIOCLINKRT	Allow virtual routing table to be shared by multiple similar VNEs.(to create multi-homed hosts or routers)
SIOCGETVETHPARMS	Return VNE configuration to a user-space program
SIOCDELVETH	Delete a VETH device
Virtual Router ioctls	
SIOCCREATEVRROUTER	Create a Virtual router structure
SIOCADDPORT	Add a new port to the virtual router
SIOCCREATEROUTERRT	Create the router's routing table
SIOCGETROUTERPORT	Return ports associated with a particular router
SIOCDELVRROUTER	Delete a router and its associated ports
SIOCGETVRROUTERPARMS	Returns the configuration information of a router

Table 3.5: List of `ioctl` calls defined for Virtual Network Elements

virtual network topologies that exist for a given physical network topology. The next section considers some limitations of virtual network elements.

3.8 Limitations of Virtual Network elements

The current implementation of the VNET software is limited by an inherent property of the Linux IP protocol stack. When a packet originates on a machine and is destined to an interface on the same machine, typically it would be sent out the source interface from where it will be picked up by the destination interface on the same machine. Linux does not bother to send the packet down to the outgoing interface, instead, it directly puts the packet into the receive queue of the destination interface, thereby short-circuiting the path of the packet through the protocol stack.

Since Linux treats the virtual interfaces similarly, if source and destination virtual hosts are emulated on the same machine, the packet will be directly transferred to the destination virtual host without reaching the VNET layer where the packet might have been routed to a gateway in the virtual network, possibly on another physical machine. This limitation puts some restrictions on the topology of the virtual networks. These restrictions can be kept in mind by following a few rules in designing virtual networks as follows:

1. The source and destination virtual hosts for a traffic flow cannot be emulated on the same machine.
2. If a packet needs to flow through two virtual network elements in sequence, and it needs to be treated by the layers below the VNET layer (e.g. Traffic control layer in Linux), then those elements cannot be emulated on the same physical machine. There are two possible combinations: (virtual host \leftrightarrow virtual router) and (virtual router \leftrightarrow virtual router). This is currently a limitation of the VNET software logic. Eventually, we can make the VNET layer send the packet down the protocol stack through the first virtual element and get it back again through the second virtual element as a normal virtual network packet..



Figure 3.12: Trivial physical network

For trivial experiments such as that shown in Figure 3.12, vNET offers few advantages, since all the components have to be on different physical machines. The only advantage is that one can emulate virtual subnets of clients on one machine communicating through a virtual router on the second machine to a virtual subnet of servers on the third machine.

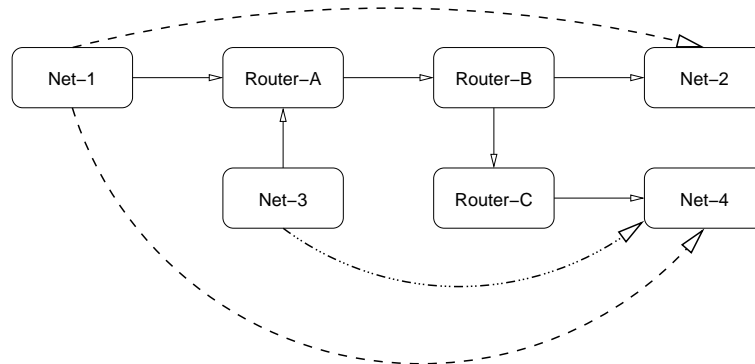


Figure 3.13: More complex physical network

But as the networks to be emulated are scaled up, such as the one shown in Figure 3.13, depending on the traffic flow through the network, one can have different virtual network topologies emulating the target network. Once again, this depends on the nature of traffic flow through the network. Figure 3.13 shows that **Net-1** communicates only with **Net-2** and **Net-4**. Similarly, **Net-3** communicates only with **Net-2**. The following is a list of some of the possible *virtualization* topologies based on these traffic patterns:

1. **Net-3** can be emulated on the same machine as **Net-1**.
2. **Net-1** can be emulated on the same machine as **Router-B** or **Router-C** but not

both since `Router-B` cannot be on the same machine as `Router-C`.

3. Similarly, `routerA` and `routerC` can be on the same machine, since they do not communicate directly. The user might choose to emulate `Net-2` on the same machine too, if he/she so desires.

Thus, emulation of a physical network using virtual network elements is currently dependent on the traffic flow in the network to be emulated. Thus, it is quite possible that sometimes there can be only one virtualized network topology: a one-to-one physical-to-virtual mapping which would require the same number of machines in the emulation as in the real network. This is a limitation which can be fixed in subsequent versions of the VNET software by modifying the IP routing code to remove the code that short-circuits the path of a packet if its destination is on the same machine as its source.

Chapter 4

Quality of Service

4.1 Quality of Service Models

Based on the level of QoS stringency required by the applications, there are three ways [4, 3] to tackle the QoS problem:

- **Overprovisioning:** This is the obvious solution to handle the high-demand periods, since the surplus network capacity can handle the peak data rates. But it quickly becomes economically infeasible for network operators because the spare network capacity lies unused for long periods of time.
- **Explicit Reservations:** This solution has its roots in the telecommunications industry where an explicit path is established during call-setup before the call is allowed into the network. Network resources are apportioned according to an application's QoS request, and subject to a bandwidth management policy.
- **Prioritization:** This solution marks different classes of traffic with different priorities and gives high priority to flows that are apportioned better QoS according to bandwidth management policy, also called as service level agreement (SLA).

QoS models can be characterized on the basis of how they handle individual application flows [25, 3] as follows:

Fine-grained QoS architectures: Reservations are guaranteed to *individual* flows and resources are established through a signaling protocol. These architectures

have problems with scalability as the number of flows goes up since the core network has to manage resources and keep state information for each flow.

Coarse-grained QoS architectures: Multiple streams are grouped into classes and the *flow aggregates* are provided service guarantees instead of individual flows. Since the resources are shared by streams within a class, no deterministic guarantees can be provided.

Two contrasting efforts by the Internet Engineering Task Force (IETF) which address QoS mechanisms and architectures are *Integrated Services*(Intserv) [15] and *Differentiated Services*(Diffserv) [12]. While Intserv is an example of a fine-grained QoS architecture involving explicit reservations for each flow, Diffserv is an example of a coarse-grained QoS architecture using prioritization of flow aggregates. More recently, there have been efforts to merge these two architectures and create a hybrid architecture which uses Intserv at the edge of networks and Diffserv in the core [11, 10]. In the following sections, we describe each of these QoS architectures in some detail. For more details, the reader is urged to consult [31, 12, 43, 27, 18, 8, 13] for the Diffserv architecture, its implementations and evaluation, [32, 15, 14, 2, 53, 51, 52, 50, 37] for the Intserv architecture, RSVP and implementations and finally [11, 10, 34, 42, 7, 45] for the hybrid architectures and their implementations.

So many references look ugly. Put in only the important ones? Or is this sentence redundant?

4.1.1 Integrated Services

The Integrated Services (Intserv) architecture [15], developed by the IETF Integrated Services Working Group [32] is based on a fine-grained QoS approach for real-time applications where explicit reservation is made for each real-time flow as is done in the telephony world. The service model is concerned almost exclusively with the time-of-delivery of packets. Thus, per-packet delay is the central quantity about which the network makes quality of service commitments. This involves *a priori* traffic characterization through significant control plane signaling which triggers admission control, classification and resource reservation mechanisms before the actual data can be transmitted. Integrated Services recommends the use of Resource Reservation Protocol

(RSVP), as the signaling protocol. The working group produced a service specification and a reference implementation framework which we discuss briefly.

4.1.1.1 Intserv model

Intserv characterizes two types of applications: **elastic** and **real-time**. Elastic applications which include Email, FTP and DNS do not care too much about *when* packets arrive; arrived packets are processed immediately. TCP-based applications are almost always elastic since TCP does not lend itself to real-time service due to its dependence on handshaking, ACKing and round-trip times. In contrast, real-time applications such as IP telephony and streaming multimedia require that packets must reach the destination within some bounded period of time after the previous packet, otherwise the packet is useless. Jitter* in packet arrival is handled by using play-out buffers.

Real-time applications are further classified into **tolerant** and **intolerant** real-time applications. Tolerant real-time applications can handle infrequent emptying of play-out buffers due to excessive jitter by fabricating *filler* packets. They can also calculate typical end-to-end delay values to get away with smaller buffers and lower latency at the cost of infrequent degradation of application performance. Intolerant real-time applications on the other hand require the calculation of a strict *worst-case* end-to-end latency to ensure adequate buffering since they cannot tolerate degradation of application performance. An example of intolerant real time application is circuit emulation, that is, setting up a telephone circuit over a data network.

Based on the above characterization of traffic, Intserv defines two types of services:

- **Controlled Load Service** [51]: Under this service, applications would receive service atleast equivalent to *best-effort* even under peak load conditions. This service was developed for **tolerant** real-time applications and elastic applications which can adapt to changes in latency and jitter. e.g. streaming audio and video applications.

*Change in the temporal characteristics of a sequence of packets

- **Guaranteed Load Service** [52]: Under this service, a flow conforming to its negotiated traffic specifications will receive service with bounded delay and no datagram loss. This service was intended for **intolerant** real-time applications which needed a rigid bound on end-to-end latency. e.g. Interactive videoconferencing, IP telephony.

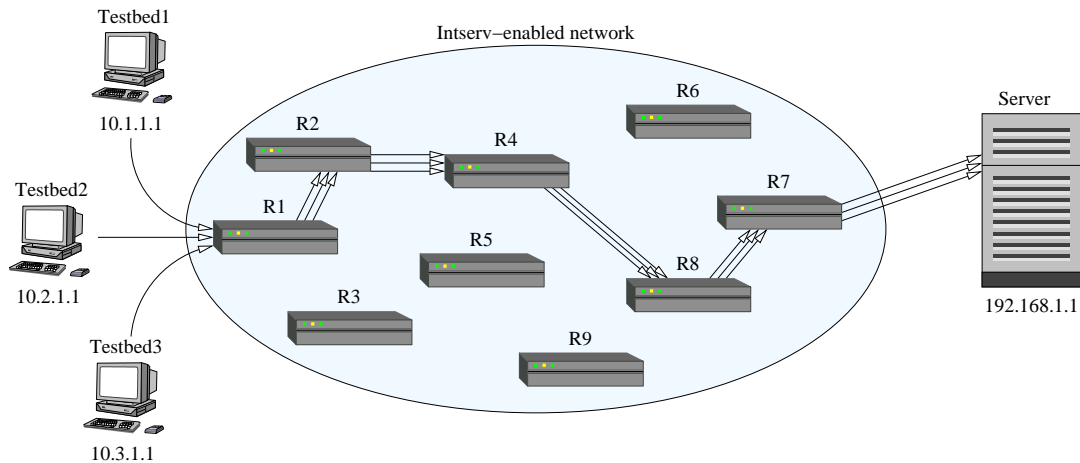


Figure 4.1: A sample Intserv network

Supporting these two services requires all network elements along the end-to-end path to implement Intserv mechanisms. Figure 4.1 shows a typical Intserv network architecture. The cloud of routers constitute one or more service provider networks and all routers perform all the tasks required by the Intserv specification - there is no distinction in the roles of the routers. The end-to-end path from the client to the server is denoted by the arrows. All network elements along this end-to-end path are required to implement Intserv mechanisms as mentioned previously. Also, the end hosts need to implement the signaling mechanism to request either CL or GL service from the Intserv network. The number of flows that the network needs to keep track of increases linearly as the number requests to the network.

Whereas Controlled Load (CL) service lends itself easily to statistical multiplexing of resources allocated for CL service by allowing for infrequent degradation of service, Guaranteed Load (GL) service isn't so flexible. More formally, the CL service defines

the end-to-end behavior provided by the series of network elements to the application’s data packets as follows:

- The transit delay experienced by a high percentage of packets will not greatly exceed the minimum end-to-end delay of a successfully delivered packet which in turn is always within the bounds experienced in best-effort conditions.
- A high percentage of packets will be successfully delivered, approximately equal to the percentage of packets successfully delivered under lightly-loaded conditions.

r	Token Bucket Rate (bytes/s)
b	Token Bucket Size (bytes)
p	Peak Data Rate (bytes/s)
m	Minimum policed Unit (bytes)
M	Maximum datagram size (bytes)

Table 4.1: TSpec parameters

To request a CL service, the application provides the network elements with an estimate of the traffic it will generate, the *traffic specification*(**TSpec**). The parameters of a TSpec are shown in Table 4.1. These parameters are used to specify the maximum *burst size* and *transmission rate* required by the application’s data flow. Flows under CL service experience little or no average packet queuing delay over all timescales significantly larger than the *burst time*. Burst time is defined as the time required for the flow’s maximum size data burst to be transmitted at the flow’s requested transmission rate [51]. If the flow characteristics exceed the values of the parameters specified in the TSpec, it exhibits the characteristics of overload including a large number of delayed packets. Hence, CL service is only specified for flows conforming to their traffic specification.

The GL service on the other hand, is designed to provide firm guarantees on the bandwidth available and end-to-end delay experienced by all its flows. If the flow’s traffic stays within its specified traffic parameters then GL service guarantees that datagrams will arrive within the guaranteed delivery time and will not be discarded due to queue overflows. This guarantee is provided to the end-to-end delay, not the jitter or minimum

delay. To request a GL service, the application, in addition to the TSpec provides the network elements with the *desired service specification*, RSpec. RSpec consists of a rate term R and slack term S. Each network element receives a service request of the form (TSpec, RSpec), where RSpec is (R_{in}, S_{in}) . The network element processes the request and either rejects it or accepts it and returns a new RSpec of the form (R_{out}, S_{out}) .

4.1.1.2 Reference Implementation Framework

The reference implementation framework for Integrated Services consists of the packet scheduler, the admission control routine, the classifier, and the reservation setup protocol. The first three are usually part of the **traffic control layer** in a router which is the software function which creates different classes of services in an otherwise egalitarian IP network. We briefly consider each of the traffic control components and their roles.

Packet Scheduler: The packet scheduler manages the forwarding of different packet streams using a set of queues and mechanisms such as timers. The packet scheduler must be implemented at the point where packets are queued; this is usually just before the packet is transferred onto the link by the network device driver in a typical operating system (before `hard_start_xmit` in Figure 3.5).

Classifier: For the purpose of traffic control (and accounting), each incoming packet must be mapped into some class; all packets in the same class get the same treatment from the packet scheduler. This mapping is performed by the classifier. Choice of a class may be based upon the contents of the existing packet header(s) and/or some additional classification number added to each packet.

Admission Control: Admission control implements the decision algorithm that a router or host uses to determine whether a new flow can be granted the requested QoS without impacting earlier guarantees. At the time a host requests a real-time service along a path through the Internet, the Admission control is invoked at each and every node along the path to make a local accept/reject decision.

The final component of the implementation framework is the signaling protocol. The Intserv Working Group has recommended the use of the Resource Reservation Protocol

(RSVP) as the signaling protocol for resource reservation in an Intserv network[14, 53]. We consider RSVP in greater detail in the next (sub?)section.

4.1.1.3 RSVP signaling

The Resource Reservation Protocol (RSVP) is a signaling protocol designed for Intserv networks to request QoS for simplex flows from the underlying network.[14]. It supports dynamic multicast groups and various routing protocols. RSVP protocol mechanisms facilitate the creation and maintenance of a dynamic and distributed reservation state across a mesh of multicast or unicast delivery paths. To account for changing network topologies and the resulting changes to routing topologies and multicast trees, RSVP is designed to be a *soft state* protocol; it requires periodic refresh messages in the absence of which the reservation state automatically times out and is deleted. Another feature is that RSVP is *receiver-oriented*, in that, it is the job of the receiver of a data flow to initiate and maintain the resource reservations used for that flow.

RSVP makes resource reservations for a *session*, which is defined to be a data flow with a particular destination address and transport-layer protocol. Thus an RSVP session can be defined by a triple: (DestinationAddress, ProtocolId [, DestPort]), where DestPort is optional. A reservation request for a session consists of a *flowspec* and a *filterspec*. The flowspec, which includes a service class along with the TSpec and RSpec, defines the desired QoS. It is used by RSVP to configure the packet scheduler in the traffic control layer at each network element if the resource reservation request succeeds. The filterspec together with the session specification defines the *flow* that is to receive the QoS. It is used to configure the packet classifier in each node.

RSVP defines 7 signaling messages, two of which are *PATH* and *RESV* (reservation) messages. The others are management and notification messages to signal errors and tear down reservations along a path. These messages carry the traffic control and policy control parameters as opaque objects; they are only interpreted and modified by the admission control and traffic control mechanisms at each network element, and not by the RSVP process. All the messages are send as raw IP datagrams*. The PATH message

*The RSVP daemon opens a raw socket with protocol id 46 to send and receive these messages.

which originates from the traffic source is composed of the TSpec and ADSpec objects, TSpec being used to inform the receiver and the intermediate network elements of the traffic characteristics required and ADSpec being updated at each network element to describe the QoS parameters it can guarantee. When the PATH message reaches the receiver, the RSVP process at the receiver decides on the reservation parameters based on the ADSpec object and sends back a RESV message along the path taken by the PATH message. The RESV message is composed of the flowspec and filterspec. The network element passes the flowspec information to the local admission control to calculate if it can support the required QoS. If successful, the packet classifier is configured using the filterspec to identify the flow and the flowspec parameters are passed to the traffic control layer to obtain necessary QoS support. If unsuccessful, an error message is sent to the receiver. Thus, if the RESV message reaches the traffic source, it is guaranteed that each network element along the path has reserved some bandwidth to provide the flow with the desired QoS.

4.1.1.4 Disadvantages of Intserv

Intserv requires significant changes to the Internet infrastructure. It is a highly *intrusive* mechanism to provide Quality of service and thereby entails a significant investment of the part on the whole Internet community for successful deployment. The Intserv model has not yet been widely deployed due to the following limitations:

1. Since Intserv requires tracking of the commitments made to each flow through the network as part of its state information, it puts storage and processing burden on the core routers. This directly affects the Internet service providers' costs of running the network.
2. Both Controlled Load and Guaranteed Services require all network elements (routers and end hosts) along the path to implement Intserv mechanisms, otherwise they cannot provide the expected delay bounds.

4.1.2 Differentiated Services

The Differentiated Services (Diffserv) architecture[12], developed by the IETF Differentiated Services Working Group[31], tries to solve the QoS problem using a contrasting approach to the Intserv architecture. One of the major drawbacks of Intserv is its lack of scalability due to the amount of state information that needs to be maintained for each flow passing through the core network. Diffserv attacks this problem by *aggregating* flows at the edge of the provider's network, thus keeping complexity at bay in the core network. The aggregated flows are mapped to traffic classes at the edges and this information, in the form of 6-bit Diffserv code point(DSCP), is carried in the IP header's Type of Service (TOS) field which is redefined for use by Diffserv [43]. DSCP occupies the first 6 bits of the TOS byte; bits 7 and 8 are set to zero.

4.1.2.1 Diffserv model

The Diffserv model reduces complexity at the core network by *aggregating* multiple traffic flows into *behavior aggregates* (BA) by marking each packet with one in a finite set of DSCPs. Thus depending on the service level agreement (SLA) with the customer, certain types of flows get better priority through the provider network than others. Each DSCP value signifies a BA which gets a pre-defined treatment at the interior routers in the network. This pre-defined treatment is known as *per-hop behavior* (PHB). Of course, the service providers have their own control mechanisms in place to check if the users' traffic adheres to the traffic conditioning agreement (TCA)* before entering their network which is a *Diffserv domain* (DS domain).

Figure 4.2 depicts a typical Diffserv network architecture in which the customer networks on either end are connected through one or more DS domains. These DS domains could belong to a single or multiple service providers, though typically each provider would have a single DS domain. The figure also shows the details of one of the DS domains. It shows many traffic flows entering the ingress router of the domain. For the purposes of this example, they are classified by the ingress router into

*The TCA is agreed upon by the customer as part of the SLA

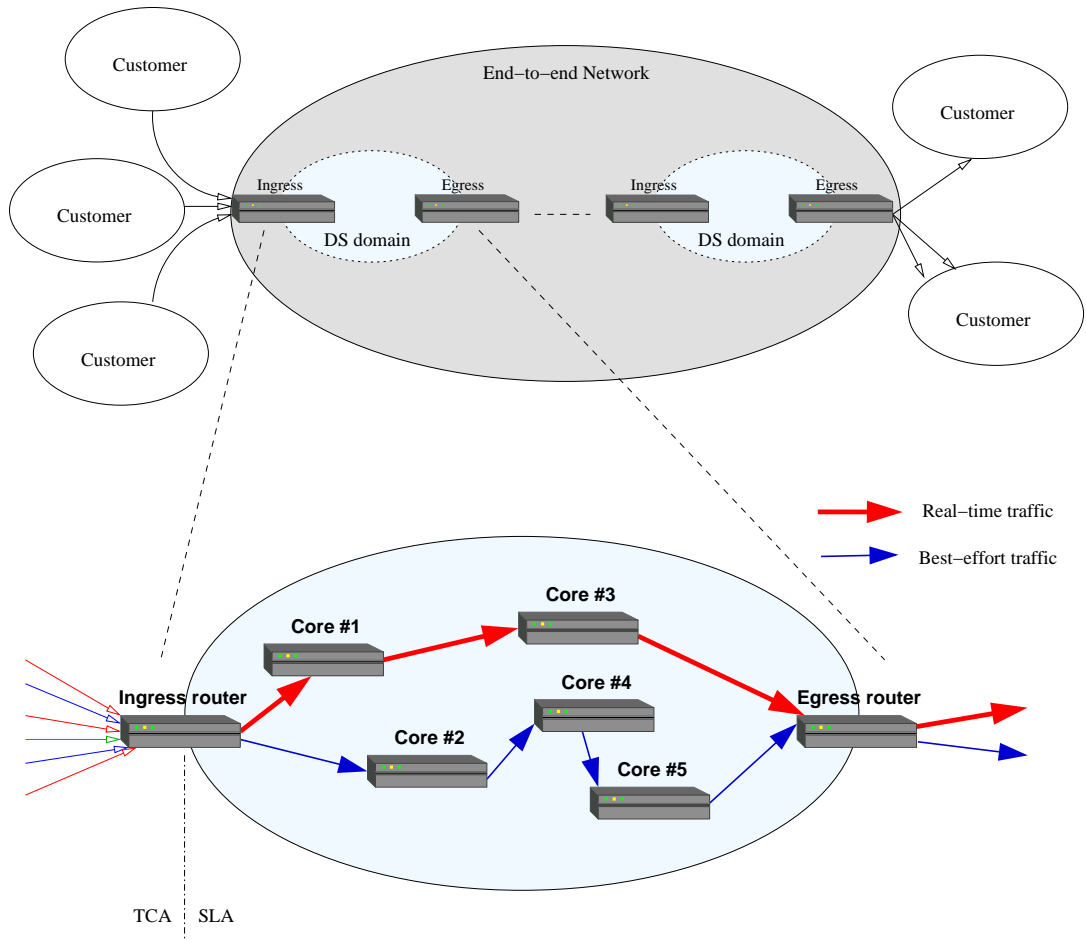


Figure 4.2: A sample Diffserv network

two distinct behavior aggregates: real-time and best-effort, shown by the red and blue arrows respectively. Each of these BAs, which are now marked with different DSCP values, are serviced by different PHBs which dictate how each interior router (Core) processes them. In this case they take a different path through the network, with the real-time BA having to pass through only two routers before reaching the egress router. Thus, in order to offer QoS, the Diffserv model defines the following functions:

Classifier: The classifier function reads the classification key in the packets on the input stream to assign the packets to various output flows. The classification key is different depending on where the router is in the Diffserv network. Ingress routers (at the entry to a provider network) have a *multi-field classifier* which use the source and destination IP address and port pairs along with the protocol id to classify a packet. Interior routers on the other hand, have a *behavior aggregate classifier* which classifies packets based on the Diffserv code point (DSCP) in the IP header's TOS field which is marked by the ingress routers.

Conditioner: The conditioner function is charged with task of ensuring that the *classified* traffic adheres to the traffic conditioning agreement (TCA). This ensures that on an average each behavior aggregate gets the service promised to it by the SLA, while also ensuring that it does not starve other BAs though the core network. The conditioner uses *meters, markers, policers, shapers* and *droppers* to enforce the TCA. The *meter* compares the actual traffic against the traffic profile expected according to the TCA. All conforming traffic is called as *in-profile* and everything else is marked as *out-of-profile*. The out-of-profile packets are queued until they are marked as in-profile by the *policer/shaper* or dropped by the *dropper*. All in-profile packets may be allowed to enter the DS domain as-is, or they may be remarked with new DSCP values by the *marker*.

Forwarding Behavior: Diffserv allows for different PHBs for various behavior aggregates. A PHB defines the forwarding behavior adopted by the interior routers for a particular BA. Currently, the IETF Diffserv group has standardized on two PHBs, the Expedited Forwarding PHB (EF) and the Assured Forwarding PHB

(AF) which are discussed in the next section.

4.1.2.2 Expedited Forwarding and Assured Forwarding

The Expedited Forwarding (EF) PHB[18] provides the building blocks for a service which minimizes packet loss, delay and jitter. To minimize delay and jitter, the traffic under EF PHB should encounter small or empty queues. To ensure small queue lengths, the EF queues are serviced at a rate greater than the EF class' arrival rate. This in turn requires that the BA expecting EF is strictly policed at the ingress router. Packet losses can be minimized by keeping the buffer length much larger than the expected queue length.

The conditioner for EF at the ingress router aggressively shapes or drops traffic to ensure that EF class does not get congested in the interior routers. EF PHB can be implemented in routers in the following ways:

- Priority Queuing: EF flows get a higher priority than non-EF flows. This can cause starvation for non-EF flows.
- Weighted-Fair Queuing (WFQ): Using WFQ between EF and non-EF flows with appropriate weight to the EF flow resolves the problem of starvation of non-EF flows while ensuring an adequate service for EF flows.

The Assured Forwarding (AF) PHB[27] provides a flexible framework to service providers to address different forwarding needs of their customers. It provides four classes for traffic classification, and each class in turn has three drop precedences. These drop precedences are the probabilities of dropping of dropping the packet in a congested network. The DSCP for the four classes and their three drop precedences (DP) are shown in Table 4.2.

At the ingress router, a flow needing AF is mapped with a DSCP value for one of the four AF classes (001, 010, 011 or 100). It is then metered according to the TCA to mark their drop precedence (010, 100 or 110). Traffic that is in-profile will be marked with the lowest DP - DP1, while out-of-profile traffic is marked with a higher DP - DP2 or DP3. This way, when there is congestion in the interior routers, packets with

	DP1	DP2	DP3
AF1	001010	001100	001110
AF2	010010	010100	010110
AF3	011010	011100	011110
AF4	100010	100100	100110

Table 4.2: DSCP values for Assured forwarding

a higher DP are more likely to be dropped. On the other hand, when the network is lightly loaded, even packets with a high DP will successfully pass through it allowing for statistical multiplexing for optimal utilization of resources.

The AF PHB is most often implemented in routers using the Random Early Detection (RED) queuing discipline. RED provides two important thresholds for the average queue length in the router - MIN_{in} and MIN_{out} with respect to the corresponding dropping probability of in-profile and out-of-profile packets where $MIN_{out} < MIN_{in}$. Thus, as the queue length increases, out-of-profile packets have a higher probability of being dropped due to a lower queue length threshold than that of in-profile packets.

4.1.2.3 Disadvantages of Diffserv

Despite easing the burden on the core routers through traffic aggregation, Diffserv introduces new problems in the quest to find a perfect QoS mechanism. Some of the problems associated with the Diffserv architecture can be summarized as follows:

1. While Intserv was too fine-grained leading to high resource requirements, Diffserv is too coarse-grained in its attempt to provide QoS through traffic aggregation. e.g. If a BA consists of x flows, and one of them misbehaves by violating the SLA, then the other $x - 1$ flows are needlessly penalized by the interior routers just for being a part of the misbehaving BA.
2. Diffserv networks by their very nature are statically provisioned through manual configuration that reflect the SLAs. They lack dynamic admission control or policy-based admission for a user or application. Thus they cannot respond

intelligently to the changing conditions inside the network for optimal resource utilization.

3. Diffserv cannot easily support specialized flows such as IP-tunneled flows or others which encrypt or modify their headers since Diffserv requires access to the IP header to mark, remark and read the DSCP values in the TOS byte of the IP header
4. Dropping packets in the interior nodes has a more adverse effect on connection-oriented flows such as TCP flows than on connection-less flows such as UDP. This is due to the flow (congestion?) control mechanisms of TCP that respond to packet losses in the network by drastically reducing their sending rate. Thus, if an interior router experience congestion and drops a TCP packet in a BA containing both TCP and UDP flows, the TCP flow will receive unfair treatment.

4.1.3 Hybrid Qos Architectures

With the individual drawbacks of the Diffserv and Intserv models as discussed in previous sections, a body of researchers were drawn to the obvious question - *Was it possible to take the salient features of both the models and integrate them into a hybrid model?*

This question led to two the development of two hybrid models. From a abstract point of view, these hybrid models do not attempt to design a new protocol, per se. They both use the Diffserv and Intserv models in their original form and only differ in the way that the Intserv-Diffserv service mapping is done and the role of the border elements. Both the models suggest the use of the Intserv model in the edge networks of the customer for fine-grained QoS control and Diffserv model in the core networks of the service provider for a coarser QoS control in terms of behavior aggregates. This approach allows enhanced control inside customer networks while keeping out that complexity from the provider network.

The first hybrid model, which is widely known as *Microsoft model*[11], defines an Intserv-provided edge network (customer) that has static service mappings to the Diffserv-provided core network (provider). The second hybrid model, which is widely

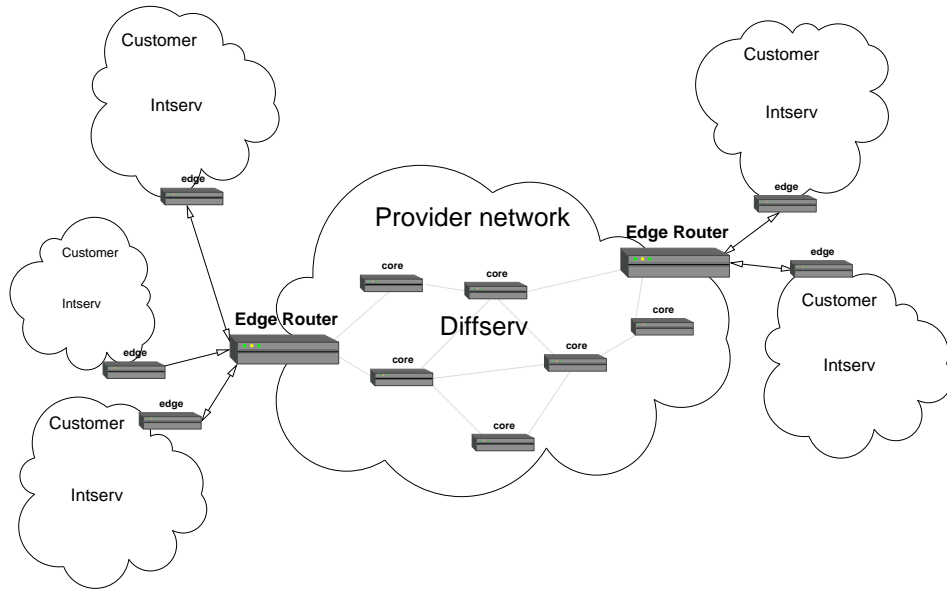


Figure 4.3: A Sample Hybrid network model

know as *Tunneled Aggregate model*[10], defines an Intserv-provided edge network with dynamically assigned service mappings to the Diffserv-provided core network. Figure 4.3 depicts a network to deploy hybrid QoS models. The two models only differ in the way the Diffserv core network responds to the resource reservation requests flowing between two Intserv networks on opposite ends.

The main task in hybrid models is the mapping of the Intserv flowspec parameters onto Diffserv PHBs and vice versa. This task is assigned to the *mapping* node, which could be the edge router of customer's Intserv network or the edge router of the provider's Diffserv network. That tasks that such a router would have to perform include the following:

Traffic conditioning: The router would have to classify, police and shape the traffic and mark it with an appropriate DSCP.

Admission control: The router would have to implement a policy-based admission control or a rigid SLA-based one. This admission control would in effect be responsible for the service mapping by selecting the appropriate PHB in the Diffserv

network.

PHB selection: Once an Intserv flow is admitted, the router would have to map it to an appropriate Diffserv PHB depending of whether the flow requires CL or GL service. GL service due to its very nature maps naturally to the EF PHB, while the CL service could map to either AF or EF PHBs.

The following sections consider the differences in the hybrid models in some detail.

4.1.3.1 Microsoft Model

The Microsoft model describes the simplest possible integration of Intserv and Diffserv possible. The edge routers in the Intserv network are statically provisioned with a pre-defined SLA. These routers are responsible for traffic conditioning, admission control and shaping to meet the Diffserv region requirements. The core routers in Diffserv network are pure Diffserv routers which receive aggregated traffic from the edge routers in the Intserv networks.

Figure 4.3 depicts a typical network deploying the Microsoft model. The various customer networks deploy Intserv and the edge router of the Intserv network or the edge router of the Diffserv provider network is responsible for mapping the Intserv service available to the flows to the corresponding Diffserv classes in the provider network. This edge router understands RSVP signaling to deal with Intserv flows and maps them to the corresponding DSCP to form BAs which are then injected into the Diffserv core network. On the other side, the edge router does a reverse mapping from a Diffserv DSCP to an Intserv flow corresponding to the parameters that were used to make resource reservations during admission control initially. At the beginning when the resource reservations are established in the source and destination networks, the RSVP signaling messages are passed through the Diffserv as-is. Thus the Diffserv core is unaware of the service commitments made by the Intserv networks and is thus said to be *RSVP-unaware* in the Microsoft model.

The Microsoft model has a key drawback, in that, it does not make efficient use of the resources in the core network. Since the Diffserv core is statically provisioned for

the Intserv mapping, any changes in the Intserv resource reservations cause undesired behavior. If the Intserv resources exceed the static Diffserv provisioning for Intserv traffic, then packets are dropped at the conditioner even if there is spare capacity in the core. If the static Diffserv provisioning for Intserv traffic exceeds the real Intserv traffic, that capacity lies unutilized even when the core is overloaded. There exist a couple of practical implementations of the Microsoft model that can be referred for details of the mapping mechanisms[34, 47].

4.1.3.2 Tunneled Aggregate Model

The Tunneled Aggregate model is a more sophisticated hybrid QoS model that tries to solve the fundamental drawback of the Microsoft model: inefficient use of the core network capacity. It does this by using an *intelligent* core network which takes part in the resource reservation process between two Intserv networks.

In the Tunneled Aggregate model, the Diffserv routers in the core also understand RSVP signaling. Hence, they can use RSVP signaling to inform the Intserv edge networks about the condition of the core network leading to better network utilization. But there are two kinds of RSVP signals in a Tunneled Aggregate model:

End-to-end signaling: This RSVP signaling is the same as in the Microsoft model between the two Intserv nodes on either side of the Diffserv core network. These messages are passed through the core unmodified, albeit with a different protocol id (134).

Aggregated signaling: This signaling takes place *inside* the Diffserv network. The edge routers of the Diffserv region which handle the end-to-end signaling are known as the *aggregator* and *de-aggregator*. They map individual Intserv flows to corresponding Diffserv classes and only pass the aggregate reservation messages to the core routers while protecting them from each individual resource reservation message coming from the Intserv networks.

There exists a practical implementations of the Tunneled Aggregate model that can be referred for details of the mapping mechanisms[42].

4.2 Quality of Service support in Linux

4.2.1 Linux Traffic control

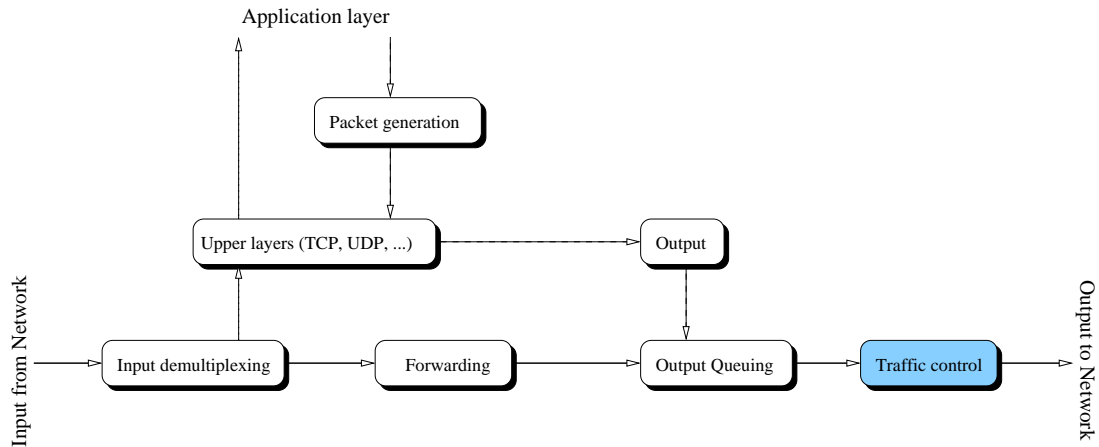


Figure 4.4: Packet processing inside the kernel

Starting from the 2.2.x kernels, Linux offers a rich set of traffic control (`tc`) functions that can be used to implement QoS mechanisms [5]. The software consists of a kernel-based traffic control layer and user-space programs to control and configure these layers. The traffic control layer supports the link-sharing mechanisms described in [26] and those required to support the architectures developed by the IETF Intserv and Diffserv groups [12, 15].

Figure 4.4 shows how network packets are processed inside the kernel. Packets entering a machine are checked for their IP destination address. If it matches one of the interfaces of the machine, the packet is sent to the upper layers which send it to the application layer. Otherwise the packet needs to be *forwarded* to the network through another interface (in case of a router). Applications on the machine might also generate packets which need to be transmitted onto the network. These packets are queued at the outgoing interface. Traffic control decides how to transmit these packets queued at all the interfaces. It can choose to do any of the following:

- **Drop packets** (for queue exceeding rate limit or length limit)

- **Reorder packets** (to give priority to certain flows)
- **Delay packets** (to rate limit the outbound traffic)
- **Mark/Modify packets** (to signal certain behavior to downstream routers)

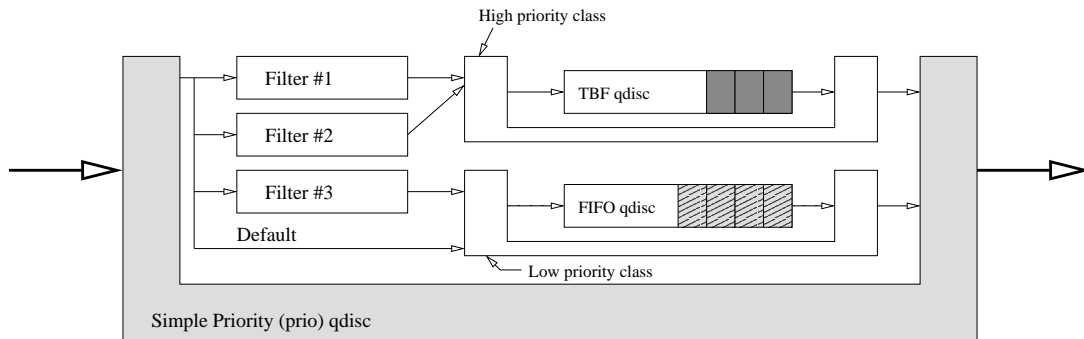


Figure 4.5: Example of qdiscs, classes and filters

The `tc` suite consists for four types of components: *queuing disciplines* (qdiscs), *classes*, *filters* and *policers*. A qdisc is a software module inserted into the Linux IP stack to alter queuing behavior. Every network interface has a qdisc associated with it which controls how packets queued at the device are treated. Some *qdiscs* may use *filters* to distinguish among different types of traffic and classify it into different *classes*. Classes use qdiscs to store packets in them. The qdisc can process each class in a specific way, e.g. by prioritizing one class over the other. Thus there exists a parent-child relationship between qdiscs and classes and this nesting of qdiscs and classes can be arbitrarily deep. Some of the common qdiscs used in Linux are *Class-based queuing*(CBQ), *Token bucket filter*(TBF), *Simple Priority scheduler*(PRIO), *First-in First-out*(FIFO), *Hierarchical Token bucket*(HTB) and *Diffserv marker*(DSMARK). Figure 4.5 shows a simple configuration consisting of a PRIO qdisc with three filters, two of them classifying to the *high priority* class being serviced by a TBF qdisc and the remaining one redirecting its traffic to a *low priority* class being serviced by a best-effort FIFO qdisc. Additional information about the design of the `tc` suite can be found in [5, 8, 19].

The kernel-space Linux Traffic control (`tc`) is configured using a user-space program

`tc` provided in the `iproute2` package[39]. `tc` allows a user to create qdiscs, associate filters and classes with these qdiscs and view statistics of all components. The `iproute2` package contains documentation on usage of these features and more information about specific examples can be found on the Linux Advanced Routing and Traffic control website[19].

When a qdisc is attached to a device, it initializes the device data structure* with a pointer to the qdisc code. When a packet is enqueued on an interface by a call to `dev_queue_xmit()` the `enqueue` function of the root qdisc of the device is invoked. After queuing the packet, the queues are activated by calling `qdisc_wakeup()` and all the device queues are polled for any packets to send by calling `qdisc_restart()`. If `qdisc_restart()` obtains a packet from the device's qdisc, it calls the `hard_start_xmit()` routine of the device to actually transmit the packet onto the link. Sections 3.4 and 3.5 discuss additional details of packet flow inside the Linux protocol stack.

In the following sections we take a look at the features provided by Linux Traffic Control for implementation of Intserv and Diffserv.

4.2.2 Intserv and RSVP

An Intserv implementation needs RSVP for signaling and a native traffic control mechanism to implement admission control, filtering and policing of traffic. The University of Southern California's Information Sciences Institute provides an implementation of RSVP on Linux[2]. This code has been integrated with the Linux kernel's traffic control layer to offer a full-fledged RSVP/Intserv implementation on Linux[40].

The RSVP daemon runs in user-space and provides the following three interfaces:

Application Programming Interface: This interface is used to communicate with applications desirous of using Intserv. It includes functions to request resource reservations, tearing down of existing reservations and other management tasks.

Traffic Control Interface: This interface is used to establish reservations by creating qdiscs, classes and filters which are later used to implement admission control,

*See Section 3.4 for details of a device data structure

classification and scheduling of flows.

Routing Interface: This interface communicates with the routing protocols to get notifications of route updates made by protocols such as OSPF, RIP or BGP. Any updates result in establishment of new reservations paths and tear down of existing ones.

Class Id	Traffic Type	Characteristics	Defmap (TOS)	Priority
1:2	Best effort	Unclassified	0x3d	6
1:3	Interactive burst	High priority	0xc0	2
1:4	Asynchronous Bulk	Low priority, high reliability	0x02	8
1:7FFE	Intserv managed	Reserves bandwidth	RSVP filters	Various

Table 4.3: Classification of traffic in Intserv nodes

The RSVP daemon classifies its traffic into four classes as shown in Table 4.3. The queue structure in Intserv nodes to service these classes is shown in Figure 4.6. The non-intserv traffic is classified on the basis of its TOS field in the IP header, also known as *defmap*. Intserv traffic is classified using the RSVP filters that are dynamically established to classify each flow that has been admitted. Thus the class 1:7FFE, henceforth known as the *Intserv class*, handles all the Intserv traffic. Intserv utilizes the CBQ qdisc and class to implement resource reservation. The Intserv class contains other classes to differentiate between guaranteed service flows and controlled load flows. Each GL flow is associated with a RSVP filter and assigned its own class inside the Intserv class. All CL flows are mapped by their RSVP filters to one of two CL classes. If the packets are in-profile, they are directed to the adaptive first CL class (1:x), else they are redirected to fixed second CL class (1:7FFF). The bandwidth allocated to the first CL class is changed dynamically as resource reservations for CL flows are set up and torn down. The fixed, second CL class has the same priority as best-effort traffic to ensure that excessive out-of-profile packets will not affect the best-effort traffic in class 1:2.

The RSVP package provides a real-time application program, RTAP, which uses the RSVP API to communicate with the RSVP daemon process. RTAP provides a command-line interface for interactively exchanging Intserv messages (e.g. PATH, RESV, etc.)

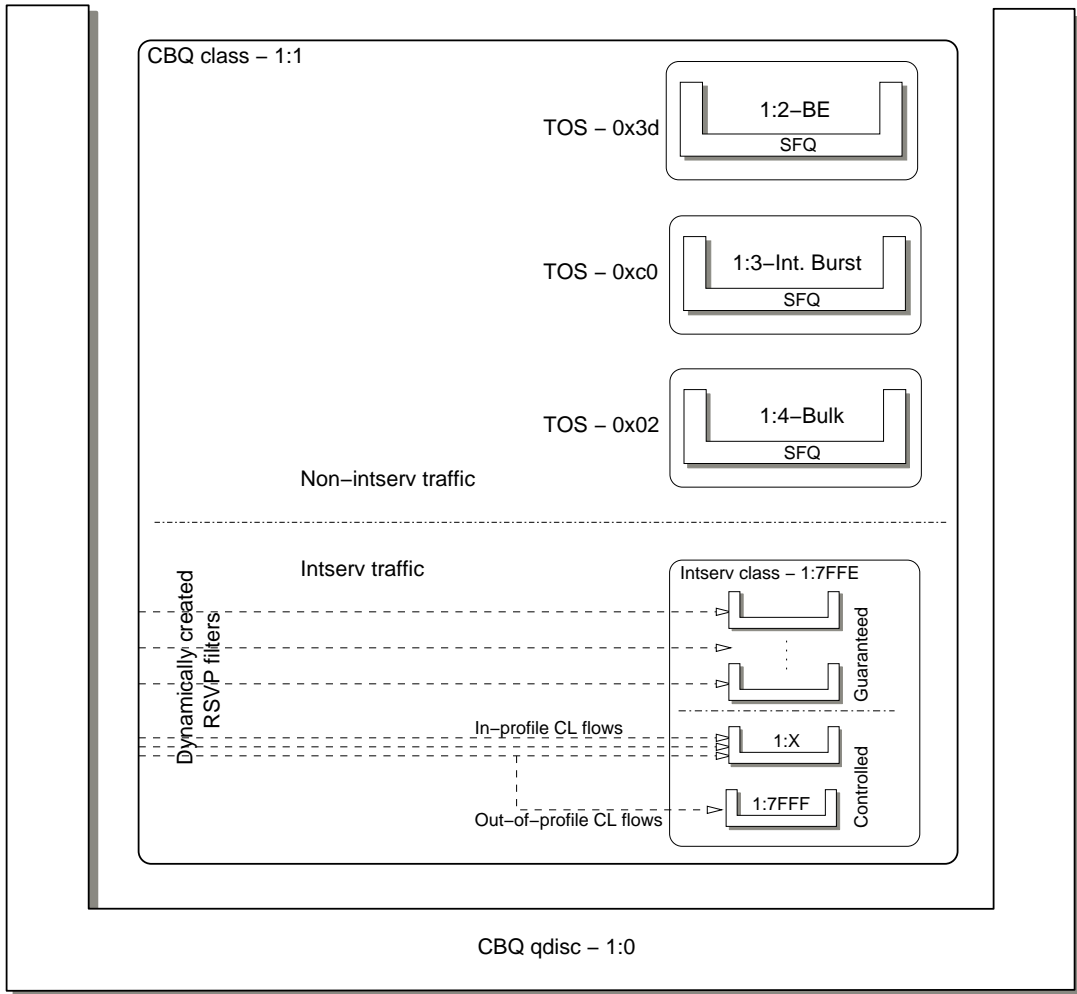


Figure 4.6: Queue structure in Intserv nodes

with other Intserv-capable hosts. A typical sequence of actions to configure an Intserv capable network would be as follows:

1. Configure an RSVP daemon on each host that needs to participate in the Intserv network.
2. Set up routes so that traffic from the source host, **SRC**, can reach the destination host, **DEST**. Routes can be statically configured or dynamically determined using various routing protocols.
3. Calculate the traffic characteristics of the application that generates the traffic
4. Start **RTAP** on **SRC** and send a **PATH** message to **DEST** which contains the traffic characteristics of the application requesting the resources.
5. When the request is successful, the RSVP daemon on all routers on the path between **SRC** and **DEST** will create the filters required to classify this application flow and allocate resources for it. If unsuccessful, **RTAP** will receive an error message.
6. Once the reservation is established, start the application on **SRC** to send its traffic.

4.2.3 Diffserv

Linux provides a robust Diffserv implementation by extending the existing traffic control implementation to add the *Diffserv Marking*(**DSMARK**) and *Generalized Random Early Detection*(**GRED**) qdiscs and the **tcindex** classifier[8, 23]. This code is kernel-based with user-space configuration provided through **tc** in the **iproute2** package.

The Diffserv specification assigns different roles to the edge router and the core router to provide the required end-to-end QoS required by the application as shown in Figure 4.2. Typically, the edge router is assigned the complex task of classifying various flows from customer networks, metering them for compliance with the traffic conditioning agreement (TCA) and then marking (or re-marking) them with a Diffserv Code Point(**DSCP**) into a fixed number of behavior aggregates (**BA**). This involves the use of a classifier, a meter, a marker, a scheduler and possibly a shaper. The core router on the other hand simply reads the **DSCP** in the marked packets and schedules them

according to the per-hop behavior (PHB) expected by a BA with that DSCP. This involves using of a simpler classifier to read DSCP and a scheduler to implement the PHB.

The three additions to the Linux traffic control implementation for implementing Diffserv have the following functions:

DSMARK: The DSMARK qdisc is used to retrieve and manipulate the DSCP in the IP header. It is the root queuing discipline on a Diffserv router. It has the following specific functions:

1. Retrieval of DSCP from IP header and saving it in the `skb->tc_index` field of `struct sk_buff`. This saves the expensive DSCP retrieval task in future processing of the packet.
2. Mapping a DSCP to its corresponding class in the queuing structure in the router.
3. Re-marking a DSCP in a packet if required.

GRED: The GRED qdisc is used to implement the three drop probabilities required for each AF class in the AF PHB.

tcindex: The `tcindex` classifier is used for simpler, single-field classification of packets using `skb->tc_index` field of `struct sk_buff`. This classifier depends upon a parent DSMARK qdisc to fill-in the `skb->tc_index` field.

We are primarily interested in the Assured Forwarding (AF) PHB of Diffserv. Figure 4.7 shows an example queue structure in a edge router providing the AF PHB. The figure shows a DSMARK root qdisc 1:0 which contains three other DSMARK qdiscs. There are three classifiers associated with the root qdisc: in-profile, out-of-profile and best-effort. The classifiers can be either the `rsvp*` or `u32` classifiers from the `tc` suite. This example classifies the traffic into only three BAs: DSCP `0x28` for in-profile packets, `0x38` for

*This `rsvp` classifier is simply used to classify packets based on a certain destination, port pair and the traffic profile. There is no need to run the RSVP daemon for it to work.

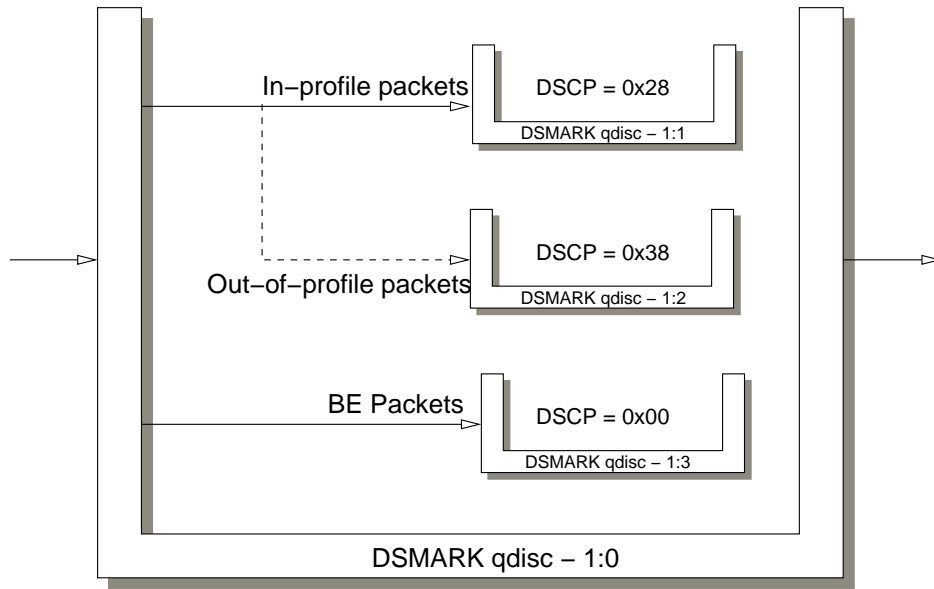


Figure 4.7: Queue structure in Diffserv edge router

Name	DSCP	TOS Field	tc_index	Virtual class	AF class	Drop Priority
NA	0x00	00000000	0	NA	0	0
AF11	0x28	00101000	10	111	1	1
AF12	0x30	00110000	12	112	1	2
AF13	0x38	00111000	14	113	1	3
AF21	0x48	01001000	18	121	2	1
AF22	0x50	01010000	20	122	2	2
AF23	0x58	01011000	22	123	2	3
AF31	0x68	01101000	26	131	3	1
AF41	0x88	10001000	34	141	4	1

where,

$$\text{tc_index} = (\text{TOS} \& 0\text{xFC}) \gg 2$$

$$\text{AF class} = (\text{virtualclass} \& 0\text{xF0}) \gg 4$$

$$\text{Drop priority} = (\text{virtualclass} \& 0\text{x0F})$$

Table 4.4: Relation between DSCP, tcindex, AF classes and drop priority

out-of-profile packets and 0x00 for best-effort packets. More complex networks could define up to 14 BAs: 12 for AF flows, 1 for EF flows and 1 for BE flows.

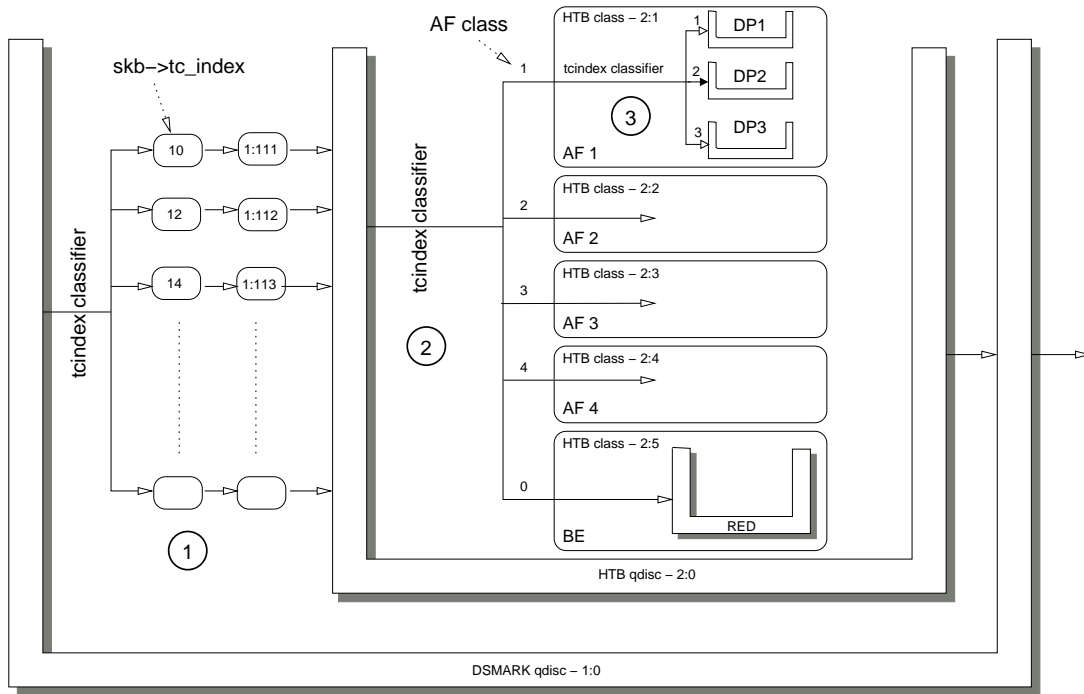


Figure 4.8: Queue structure in Diffserv core router

Figure 4.8 shows an example queue structure in a core router providing AF PHB. There is a DSMARK root qdisc to retrieve the DSCP from the packet and put it in the `skb->tc_index` field. The root qdisc contains a child qdisc which implements link sharing principles. Linux provides a choice between *Hierarchical Token Bucket* (HTB) and CBQ qdiscs to implement link sharing as outlined in [26]. Link sharing allows traffic classes to isolate or borrow bandwidth. Isolation is useful to provide services such as EF PHB which require a dedicated bandwidth, whereas borrowing allows a class to temporarily borrow *unused* bandwidth from its parent class. The HTB qdisc 2:0 used in Diffserv can contain up to four AF classes and a BE class. The AF class and the drop probability inside the AF class is selected based on the value of `skb->tc_index`.

Table 4.4 shows the relation between the DSCP, TOS, Virtual Diffserv classes, AF class and drop priority of a BA. In step 1 in Figure 4.8, the `tcindex` classifier retrieves

the DSCP from the packet and copies it to `skb->tc_index`. For instance, a DSCP of `0x28` corresponds to a TOS byte of `00101000` in binary and `tc_index` of `10` as shown below.

$$\begin{aligned}
 \text{tc_index} &= (\text{TOS} \& 0\text{xFC}) \gg 2 \\
 &= (00101000 \& 0\text{xFC}) \gg 2 \\
 &= (00101000 \& 11111100) \gg 2 \\
 &= (00101000) \gg 2 \\
 &= 001010 \\
 \text{tc_index} &= 10
 \end{aligned} \tag{4.1}$$

Once the `tc_index` handle is obtained, it is mapped to a virtual Diffserv class by the `tcindex` classifier. Hence `tc_index` handle `10` corresponds to classid `1:111`, `12` \rightarrow `1:112`, `18` \rightarrow `1:121`, `22` \rightarrow `1:123`, and so and so forth. So, `tc_index` is set to the minor number of the virtual class. Hence, from Equation 4.1,

$$\text{tc_index} = 0\text{x0111} \tag{4.2}$$

This mapping is not arbitrary. It is designed so that further processing on the `tc_index` field returns the AF class and corresponding drop priority (DP) inside the class. Steps 2 and 3 in Figure 4.8 involves mapping the packet to a particular AF class or BE class and its DP.

$$\begin{aligned}
 \text{AFclass} &= (\text{tc_index} \& 0\text{xF0}) \gg 4 \\
 &= (0000000100010001 \& 11110000) \gg 4 \\
 &= (00010000) \gg 4 \\
 &= 0001
 \end{aligned}$$

$$\text{AFclass} = 1 \tag{4.3}$$

$$\begin{aligned} \text{DropPriority} &= (\text{tc_index} \& 0x0F) \\ &= (0000000100010001 \& 00001111) \\ &= (00000001) \\ \text{DropPriority} &= 1 \end{aligned} \tag{4.4}$$

Thus, the Diffserv core router only has to perform three inexpensive *ANDing* and *right-shift* operations to classify a previously marked packet into its AF class and DP, in this case AF11.

4.3 Problems with testing QoS networks

Testing of QoS networks has been going on for sometime now. In particular, a lot of testing has been conducted on Linux [13, 41, 4, 7, 45, 47, 34, 42] because of its open source nature which allows researchers to modify its robust network protocol stack to add new functionality. But all of these evaluations suffer from one or more the following drawbacks:

Cost of infrastructure: Testing protocols on real networks is an expensive proposition due to the number of machines and supporting infrastructure required. Most service providers have networks consisting of millions of dollars worth of equipment, something that most research labs will not have.

Small scale of experiments: As a direct result of the previous drawback, most experimental setups consist of a small number of machines, ranging from 3 machines to more elaborate experiments with 7-8 machines that *we* have performed[34, 42]. But this is too small a scale to test protocols which we want to become an Internet standard!

Unavailability of equipment: The hybrid networks which we designed and tested require uncommon functionality from routers. e.g. Tunneler Aggregated RSVP

requires routers in the Diffserv core to take part in the RSVP signaling too. This kind of equipment has not hit the market because these architectures still require extensive testing.

Lack of testing standards: Currently, there exist no testing standards for evaluating data plane and control plane in QoS networks. Lack of these standards hampers widespread testing.

Lack of Control Plane complexity evaluation: Although a lot of research has been done in evaluating the performance of a new protocol at the data plane, very little has been done to measure the impact of control plane. Though a lot of data plane results can be obtained through network simulations on sophisticated simulators such as NS-2[24] and OPNET[1], these simulations cannot predict impact of complexity of a protocol in terms of different configuration commands, signaling messages through the network and the training required for operators to utilize the protocol effectively.

We hope to address these problems using virtual network elements(VNE). VNEs allow us to increase the size of the experiments without a linear increase in the number of machines required. Also, since we are using public domain software whose source code is available, we can modify the source to implement new protocols or variants for testing without waiting for supporting equipment to be available in the market. Since VNEs utilize the real protocol stack and also require the same configuration as a real network, the same commands that would be used to configure a physical network element are used to configure the VNE which gives us the opportunity to measure the control plane complexity of the protocol.

Chapter 5

Implementation

Performing reproducible experiments required the creation of a simulation infrastructure, Virtual Network Elements (VNE), modification of existing tools and creation of several new tools. This chapter talks about the software development undertaken to reach our ultimate goal: *Emulation of non-trivial IP networks using virtual network elements*. Effective emulation of Intserv and Diffserv networks proves the efficacy of our approach. We discuss the modifications to Diffserv and RSVP code to work with Virtual network elements and changes to Netspec to automate the experiments.

5.1 Modifications to QoS software to work with VNET

To be able to use the power of Virtual network elements (VNE) to emulate Intserv or Diffserv network required us to be able to use the VNEs to emulate Intserv network elements and Diffserv routers respectively. The original VNE code was generic enough to be used without major enhancements with the QoS software in Linux. The fact that very minor changes were required to the Diffserv and Intserv code to work with VNEs proves the effectiveness of utilizing our methods for general purpose IP network emulation.

5.1.1 Enhancements to VNET code

In the original design, the virtual network (VNET) layer was unobtrusive to the IP layer. A packet belonging to the virtual network (of type `ETH_P_KUVNET`), when received by a machine, was sent to the VNET layer that stripped off the VNET header before sending it up to the IP layer. Hence, the IP layer did not know of the existence of the virtual network and the VNET layer did not manipulate any IP layer data. While this approach was effective to virtualize most applications, it did not work for some applications that used the *IP options* field in the IP header. We need to be able process IP options like the *IP Router Alert* option in the VNET layer for special handling of certain packets in the virtual network.

Typically, routers process packets at the IP layer and make a forwarding decision if the final destination in the IP header is not one of the interfaces of the router. The *IP Router Alert* option is a mechanism used by routers to pass on packets to layers above the IP layer even when it is not the final destination of the packet. This packet would typically be passed to a user-space application listening for a particular protocol by establishing a raw socket. This mechanism is used by the RSVP daemon to ensure that the RSVP signaling messages reach each Intserv router (or more correctly the RSVP daemon on each Intserv router) on the end-to-end path. The daemon can then make modifications to the signaling messages (e.g. modifying the `ADSPEC` in a `PATH` message) and forward the packet along its way to the final destination.

Thus, the VNET code was modified to check if the IP header was longer than 20 bytes*, in which case, the IP options were used to populate the `skb->cb` member of `struct sk_buff`. Then, if a virtual router received a packet with the *IP Router Alert* option set, it was passed up the stack instead of being forwarded.

5.1.2 Modifications to iproute2 package

The Linux Traffic Control system consists of two parts: kernel-space packet queuing data structures and user-space tools. The kernel-space code is manipulated using the

*Standard IP header length without IP options

user-space tools provided by the `iproute2` package[39].

The `iproute2` package consists of the `tc` tool that is used to create, delete and modify traffic control classes, queuing disciplines and filters. `tc` needs to specify the packet type or protocol that these queues, classes and filters will act on. Hence, it is obvious that they will not work with the new packet type (`ETH_P_KUVNET`) created for the VNET packets.

Fortunately, getting `tc` to work with `ETH_P_KUVNET` packets requires trivial code changes in two locations. It requires the addition of `ETH_P_KUVNET` in list of packet types supported by `tc` in `lib/ll_proto.c` in the `iproute2` package as shown in Program B.1. This packet type is referenced by the name `vnet` in the `tc` commands. Also, the system headers provided by the `glibc` for user-space programs must contain a reference to the packet type in `include/linux/if_ether.h`.

These changes allow the creation of classes, qdiscs and filters that act on the VNET packets. This in turn allows the creation of virtual networks that use traffic control mechanisms.

5.1.3 Modifications to Diffserv code

The Diffserv code is part of the Linux Traffic Control system in the kernel. It defines a `DSMARK` qdisc that is used for DSCP marking. The code for `DSMARK` qdisc checks for `skb->protocol` to be set to `ETH_P_IP` or `ETH_P_IPV6` to get or set the `skb->tc_index` field i.e. it processes only IPv4 and IPv6 packets. We added another condition to do the same for packets of type `ETH_P_KUVNET`. Now, when we create a `DSMARK` qdisc using `tc`, we can specify the `vnet` packet type for correct treatment by the qdisc.

The original software for Diffserv[8] used `CBQ` qdiscs and classes to queue Diffserv behavior aggregates. The `CBQ` implementation in Linux is not very precise and has been known to give inaccurate results during link sharing. When we were evaluating Diffserv with `CBQ`, we came across a new qdisc, `HTB`, which stands for Hierarchical Token Bucket that implements the same concept of link sharing the `CBQ` attempts to implement. We made some measurements using `HTB` and found it to be very accurate and easier to setup than `CBQ`. Hence we started using `HTB` for our experiments and over time `HTB` grew in

popularity in the Linux community and was incorporated into the Linux kernel.

5.1.4 Modifications to RSVP daemon

The Intserv implementation on Linux comprises of the user-space RSVP daemon that uses the CBQ qdisc and classes in the kernel to serve Intserv flows. CBQ supports VNET packets by specifying `vnet` when creating the qdiscs and classes. To truly emulate a Intserv network on fewer nodes we need to be able to run multiple RSVP daemons on the multiple virtual routers on a physical host. This was perhaps the most complex change required to enable virtualization. The following is the list of modifications made to the RSVP daemon to enable multiple instances of the daemon to run on a single machine, each instance bound to a different virtual router.

Addition of `ioctl(SIOCGETROUTE)`: The original VNET code did not provide any mechanism to find the next-hop for a packet in the virtual network from the user-space. This is required since the RSVP daemon fills the packet headers in user-space before forwarding it. This was fixed by adding `ioctl(SIOCGETROUTE)` in the VNET code to find the next-hop from a given virtual interface for a given destination. The next-hop is found by consulting the virtual routing table of the router or host to which that virtual interface belongs and returning that address to the user-space.

Use of `ioctl(SIOCGETROUTE)`: The `ioctl` is used in all the places where the RSVP daemon tries to query the kernel routing tables. e.g. `unicast_route()`.

Enabling command-line parameters: By default, the RSVP daemon associates all the sockets to `INADDR_ANY` thereby using *any* interface address on a router. This needs to be changed since we should be able to specify which virtual router should be associated with each instance of the RSVP daemon. Hence, we changed all socket `bind()` calls to attach to the device specified on the command-line of `rsvp` daemon, instead of `INADDR_ANY`. Similarly, the `api port`, `encapsulation port`, `lock port` and `status port` are specified on the command-line instead of being fixed so that no two RSVP daemons on a machine have these ports in common.

Processing of IP options in vnet layer: As described in Section 5.1.1, the VNET code now processes the IP options to see if the `IP Router Alert` option is set. If set, the packet is sent up the stack to the RSVP daemon attached to the router.

More intelligent packet receiving: Since multiple RSVP daemons can be running on a single physical box, a packet with a protocol id of 46 was sent to all the RSVP daemons. Since some RSVP daemons were not expecting it due to the topology of the virtual network, they could not handle the packet and crash. So changes were made so that the daemon recognizes if the raw packet is meant for it. Now, the `receive()` routine checks if the incoming interface is port of the router to which the RSVP daemon is bound, or the interface of the end host. If it belongs to the router, the packet continues its normal path into RSVP else we return zero so that the packet is not processed. This was necessary because the `bind()` system call does not bind to a particular interface in the presence of the `IP Router Alert` option*.

Dynamic filenames: Since multiple RSVP instances can exist on a machine to support a virtual network, fixed filenames cannot be used. Hence a facility was created to generate dynamic filenames for storing process ids, unix socket, pipe names, etc.

5.2 Netspec

Netspec[35, 36] was developed at the University of Kansas as a tool for network experimentation and measurement. It's strength lies in it's ability to perform a distributed experiment involving multiple machines running various processes, co-ordinating the processes, generating results and displaying them to the user. All of these activities can be specified in a Netspec *script* that is then passed to the central Netspec *controller*. The controller breaks up this script into chunks meant for each process on each machine in the experiment and distributes the tasks. These processes executing on the various

*This fact is mentioned in the manual pages - `man 7 ip`

machines are the Netspec *daemons* each of which has a specific task in the experiment. These daemons run in *phases* controlled by the centralized controller daemon allowing for co-ordination of the various phases of various Netspec daemons across various machines through a string of *phase* commands from the controller and corresponding acknowledgments from the various daemons on completion of the phase.

Various daemons written to use with Netspec include the following:

Test daemon(nstestd): This versatile daemon can be a traffic source and sink. It can generate TCP or UDP traffic, emulated traffic such as FTP, Telnet, Voice, and traffic with special properties e.g. TOS of 0x28

Report daemon(reportd): This daemon generates reports at each machine and then sends them to the controller daemon.

Corba daemon(nscorbad): This daemon is used to setup and execute CORBA* based programs.

We decided to use the Netspec framework to automate our experiments to make them repeatable and infinitely less tedious to setup. But very soon, we found some of the features of Netspec inadequate for our purposes. This included the fixed number of phases in a daemon and no support for serialized execution. These are discussed in the next section.

5.2.1 Variable-phase Netspec

Though the original Netspec architecture supported execution in phases, there was no concept of *execution order*; phase X in all daemons on all machines was executed together and all daemons had to implement phase X. There was no way to specify that Phase X of daemon D1 *should* be executed before Phase X of daemon D2. Also, the number of phases were hard-coded into Netspec, so we had to build our experiment into those fixed number of phases regardless of our requirements. Very soon these became major hindrances that needed to be fixed without breaking existing daemons.

*Component Object Request Broker Architecture

Slot	D1 phases	D2 phases	D3 phases
1	D1-P1	D2-P1	-
2	-	-	D3-P1
3	D1-P2	-	D3-P2
4	D1-P3	D2-P2	-
5	D1-P4	D2-P1	D3-P3
6	-	D2-P2	-

Table 5.1: Slot-based execution in Variable-phase Netspec

As part of a semester project we re-engineered the Netspec controller daemon to accept an arbitrary number of phases in the *variable-phase mode*. We also introduced the concept of *slots* so that a user could specify the *exact* order of execution in terms of slot and phase names. This allowed the user to design experiments with arbitrary execution ordering similar to that shown in Table 5.1. Thus, the user was freed from the requirement of having the same number of phases for each daemon and executing only a certain phase at a time. As shown in Table 5.1, daemon D1 has 4 phases and daemons D2 and D3 have 2 phases each. Also, the user has specified that the first phase of D3 (D3-P1) *should* be executed *after* the first phase of daemons D1 and D2 (D1-P1, D2-P1).

These changes improved the usability of Netspec tremendously. All that was now necessary was to write new daemons to automate our tasks. Three new daemons were written: RSVP daemon(`nsrsvpd`)[42], Diffserv daemon(`nsdiffd`)[34] and VNET daemon(`nsvethd`). The VNET daemon is described in greater detail in the next section.

5.2.2 VNET Netspec daemon

Initially, the virtual network elements were created using user-space programs that passed the characteristics of the network element such as its name, IP address and routing table on the command-line. This approach to setting up a virtual network was tedious, time consuming and prone to errors since it required the commands to be executed several times each, on multiple machines with different command-line parameters. So we developed the VNET Netspec daemon called `nsvethd`. All the parameters

of a virtual network configuration are written in a single script that is then passed by the Netspec controller to the various `nsvethd` daemons on various machines. These daemons then create the required virtual hosts and virtual routers.

Program 5.1 shows a Netspec script used to setup a virtual network on three physical machines: `testbed40`, `testbed41` and `testbed42`. The string `nsveth testbed40` denotes that the `nsvethd` daemon on `testbed40` is instructed to execute the commands in between the braces. This script creates a virtual host on `testbed40` and `testbed42` and a virtual router on `testbed41`. The virtual network uses the `10.0.0.0/16` address space as shown in the script. Each virtual network element has a routing table and a routing subnet map associated with it that tells it where to route packets in the virtual network and also informs it of the mapping between the virtual and physical network. The `arptable` entries are used to send a packet to a particular machine participating in the experiment without calling the ARP routines.

5.2.3 System command daemon

There are many occasions when we need to execute a sequence of commands over a large number of machines. This typically requires logging in to the machine, typing the commands and logging out. This gets cumbersome very quickly, especially if a large number of machines are involved. We encountered this requirement numerous times during the course of our experiments. The result is the Netspec System Command daemon, `nssystemcmd`.

Program 5.2 shows a Netspec script using `nssystemcmd` to setup queues on routers. The first `cmd` statement deletes all the existing queues, and the second command establishes the queues by running a shell script. Thus any number of commands can be executed on each testbed by listing each command in a `cmd=" "` directive. This Netspec daemon allowed us to completely automate experiments by automating the setup of data collection points of various machines, copying the data to a central location, processing the data and outputting the results.

In the next chapter, we look at the various experiments carried out to show the efficacy of our approach in emulating Diffserv and Intserv networks using virtual network

Program 5.1 Netspec script to setup a 2-host, 1-router virtual network

```
cluster
{
  nsveth testbed40
  {
    vdev = create(vdeviface="veth0", phydevname="eth0", vipaddr=10.40.1.1,
                 netmask="255.255.0.0", vmacaddr="12:34:43:21:12:34");

    routing = create(dev="veth0",
                    entry=new(dest=10.40.1.1, gw=0.0.0.0,i
                              netmask=255.255.255.255, flag="H", dev="veth0"),

                    entry=new(dest=10.40.0.0, gw=0.0.0.0,
                              netmask=255.255.255.255, flag="N", dev="veth0"),

                    entry=new(dest=0.0.0.0, gw=10.40.0.254,
                              netmask=255.255.0.0, flag="G", dev="veth0"));

    routing = subnet_map(dev="veth0",
                        entry=new(dest=10.40.0.254, gw=129.237.127.241,
                                  netmask=255.255.0.0, flag="P", dev="veth0"));

    arptable = create(entry=new(ip=129.237.127.241, mac="00:E0:81:01:31:A8"));
  }

  nsveth testbed42
  {
    vdev = create(vdeviface="veth1", phydevname="eth0", vipaddr=10.42.1.1,
                 netmask="255.255.0.0", vmacaddr="12:34:43:21:12:34");

    routing = create(dev="veth1",
                    entry=new(dest=10.42.1.1, gw=0.0.0.0,
                              netmask=255.255.255.255, flag="H", dev="veth1"),

                    entry=new(dest=10.42.0.0, gw=0.0.0.0,
                              netmask=255.255.255.255, flag="N", dev="veth1"),

                    entry=new(dest=0.0.0.0, gw=10.42.0.254,
                              netmask=255.255.0.0, flag="G", dev="veth1"));

    routing = subnet_map(dev="veth1",
                        entry=new(dest=10.42.0.254, gw=129.237.127.241,
                                  netmask=255.255.0.0, flag="P", dev="veth1"));

    arptable = create(entry=new(ip=129.237.127.241, mac="00:E0:81:01:31:A8"));
  }

  nsveth testbed41
  {
    router = create(routername="vrouter0", routermac="22:22:22:22:34",
                   routerip=10.100.100.254,
                   vdev = create(vdeviface="vport0", phydevname="eth0", vipaddr=10.40.0.254,
                                 netmask="255.255.0.0", vmacaddr="12:34:56:78:12:34"),

                   vdev=create(vdeviface="vport1", phydevname="eth0", vipaddr=10.42.0.254,
                                netmask="255.255.0.0", vmacaddr="22:24:56:78:12:34"),

                   routing = create(dev="vport0",
                                   entry=new(dest=10.40.0.254, gw=0.0.0.0,
                                             netmask=255.255.255.255, flag="H", dev="vport0"),

                                   entry=new(dest=10.42.0.254, gw=0.0.0.0,
                                             netmask=255.255.255.255, flag="H", dev="vport1"),

                                   entry=new(dest=10.40.0.0, gw=0.0.0.0,
                                             netmask=255.255.0.0, flag="N", dev="vport0"),

                                   entry=new(dest=10.42.0.0, gw=0.0.0.0,
                                             netmask=255.255.0.0, flag="N", dev="vport1"),

                                   entry=new(dest=0.0.0.0, gw=10.40.0.254,
                                             netmask=255.255.0.0, flag="G", dev="vport0")),

                   routing = subnet_map(dev="vport0",
                                       entry=new(dest=10.40.0.0, gw=129.237.127.240,
                                                 netmask=255.255.0.0, flag="P", dev="vport0"),

                                       entry=new(dest=10.42.0.0, gw=129.237.127.242,
                                                 netmask=255.255.0.0, flag="P", dev="vport1")));

    arptable = create(entry=new(ip=129.237.127.240, mac="00:E0:81:01:31:61"),
                     entry=new(ip=129.237.127.241, mac="00:E0:81:01:31:A8"),
                     entry=new(ip=129.237.127.242, mac="00:E0:81:01:BC:29"));
  }
}
```

Program 5.2 Netspec script using `nssyscmd` to setup queues on routers

```
cluster {  
  
  nssyscmd testbed40 {  
    cmd="/usr/bin/tc qdisc del dev veth2 root";  
    cmd="/users/amitk/expt/vnet-testing/data-plane/diffserv/mark-RT.sh";  
  }  
  
  nssyscmd testbed42 {  
    cmd="/usr/bin/tc qdisc del dev veth1 root";  
    cmd="/users/amitk/expt/vnet-testing/data-plane/diffserv/mark-test.sh";  
  }  
  
  nssyscmd testbed43 {  
    cmd="/usr/bin/tc qdisc del dev r1p4 root";  
    cmd="/users/amitk/expt/vnet-testing/data-plane/diffserv/diff-core-r1p4.sh";  
  }  
  
  nssyscmd testbed44 {  
    cmd="/usr/bin/tc qdisc del dev r2p2 root";  
    cmd="/users/amitk/expt/vnet-testing/data-plane/diffserv/diff-core-r2p2.sh";  
  }  
  
  nssyscmd testbed45 {  
    cmd="/usr/bin/tc qdisc del dev r3p2 root";  
    cmd="/users/amitk/expt/vnet-testing/data-plane/diffserv/diff-core-r3p2.sh";  
  }  
}
```

elements.

Chapter 6

Evaluation

The network architectures we are interested in emulating were developed for experiments on physical machines as part of research on Hybrid QoS architectures[34, 42]. The research studied the management and complexity of Hybrid Intserv-Diffserv QoS architectures and focused on control-plane complexity of these architecture i.e. number of signaling messages, number of lines of configuration on routers, CPU and memory usage in routers, etc. In emulating the Diffserv and Intserv architectures using virtual network elements, we are merely concerned about the data-plane results since the control plane complexity remains the same in the emulated model. If we can verify that the emulated network has the same data-plane characteristics (throughput, latency) as the real network, we would have opened up a way to emulate larger networks without the requirement of corresponding physical resources.

Before we started emulating Diffserv and Intserv networks using virtual network elements (VNEs), we had to iron out all the wrinkles in setting up a simple *no-frills* virtual IP network. Section 6.1 discusses some of the technical performance issues and configuration issues encountered while setting up a simple virtual network. Section 6.2 and 6.4 discuss the experimental results of emulating Diffserv and Intserv network respectively.

6.1 Faithfully emulated virtual networks

The first task in trying to emulate any phenomenon is to confirm the verity of the emulation. The emulation should be checked against the real phenomenon in behavior and characteristics. Though minor difference in results can be attributed to experimental error there has to be one-to-one correspondence between both sets of results.

When the task is to emulate networks, the following characteristics would be expected from (of?) the emulation:

1. All packets inserted into the network by a host should be accounted for in the virtual network as they are in the physical network.
2. All conditions remaining the same, a packet traveling through a virtual network should experience the same latencies as that experienced by a packet traveling through a physical network.
3. The virtual network should use the same application logic as the physical network it emulates; the semantics of the applications should not change, otherwise the emulated network might behave differently.
4. The results from a virtual network experiment must be reproducible when the same parameters are re-applied to the emulation.

Although these rules might seem self-evident, they need to be vigorously applied to test any emulation framework to prove its veracity. We now discuss our efforts to test our emulation framework for conformity to these rules. We also discuss few of the issues we encountered along the way and solutions for them.

6.1.1 Test network topology

To test our emulation framework we need a network topology that is not trivial, emulates a realistic network, is complex enough to stress all parts of the system code but not too complex to debug for performance issues. Figure 6.1 shows the network topology we used for our evaluation of our emulation framework based on virtual network software.

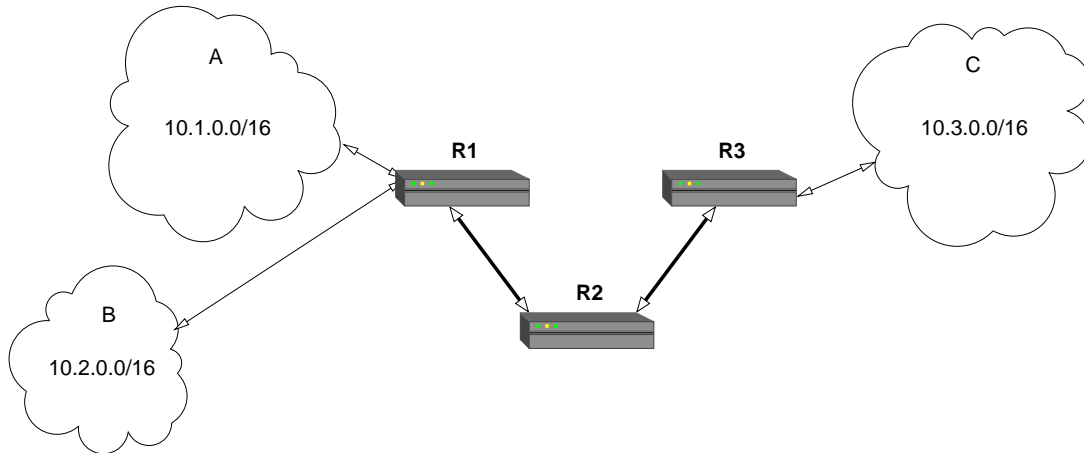


Figure 6.1: Network Topology to test Emulation Framework

It shows subnets A and B connected to router R1 and subnet C connected to router R3. R1 and R2 are connected through R3.

The network topology depicted in Figure 6.1 can stress test the following parts of the VNET software:

Multiple subnets on a physical host: The VNET software can emulate multiple IP subnets on a single physical host. In this case, virtual hosts of subnets A and B will be emulated on a single host.

Multiple routers on a physical host: Like virtual hosts, virtual routers can be emulated on a single physical host. In this case, virtual routers R1 and R3 are candidates for emulation on a single host. See Section 3.6 (Page 24 for an explanation on why (R1, R2) or (R2, R3) cannot be emulated on the same physical host.

Subnet splitting: The VNET software also allows the user to split up a single IP subnet to emulate parts of it on different physical machines. This feature is convenient in situations where there are many hosts to be emulated in the subnet and utilizing multiple CPUs would speed up the experiment.

Host-to-router communication: The proposed network topology will test host-to-

router and router-to-host communication and well as allows us to see results of multiple virtual hosts sending packets to a single router.

Router-to-router communication: Since R1 can communicate with R3 only through R2, it allows us to test the forwarding code of the virtual routers.

Our experiments involved sending traffic from subnets A and B to subnet C and vice versa. The next section discusses some of the virtual network topologies that can emulate the network topology shown in Figure 6.1.

6.1.2 Virtual network topologies

The VNET code offers a lot of flexibility to the user in designing the topology of the virtual network that emulates the real one. Hence, a real network can be emulated in more than one ways, depending on various factors such as number of nodes and routers to emulate, patterns in the real network, CPU usage by emulated hosts and applications, etc.

Figure 6.2 shows the physical emulation testbed used to conduct the experiments. The *ITTC network* denotes the student workstations from where the experiments on testbeds (TB) are controlled. TB40, TB41, TB42 and TB46 are used as traffic sources and sinks while TB43, TB44, TB45 are used for emulating routers. All the testbeds used are 1GHZ Intel PIII machines with at least 256Mb RAM. The `eth0` interfaces of all machines are used only for connectivity to the ITTC network. The experimental traffic flows on the `eth1` and `eth2` interfaces of the machines. Hence, the traffic flowing inside the emulation testbed does not affect the ITTC network and vice versa i.e. the emulation testbed is isolated from external traffic. The `eth1` interfaces of the sources and sinks (TB40, TB41, TB42 and TB46) are connected to a 100Mbps Ethernet switch. TB43, TB44 and TB45 are connected to a separate switch. TB43 and TB45 are the border routers which connects to both the edge network on `eth1` as well as the core network on `eth2`. TB44 is a core router. Traffic enters and leaves the sources and sinks on their `eth1` interfaces. This traffic then enters TB43 on it's `eth1` interface and then continues onto TB44 before leaving TB45 on it's `eth1` interface. This model ensures that the traffic on the core network is isolated from the edge network.

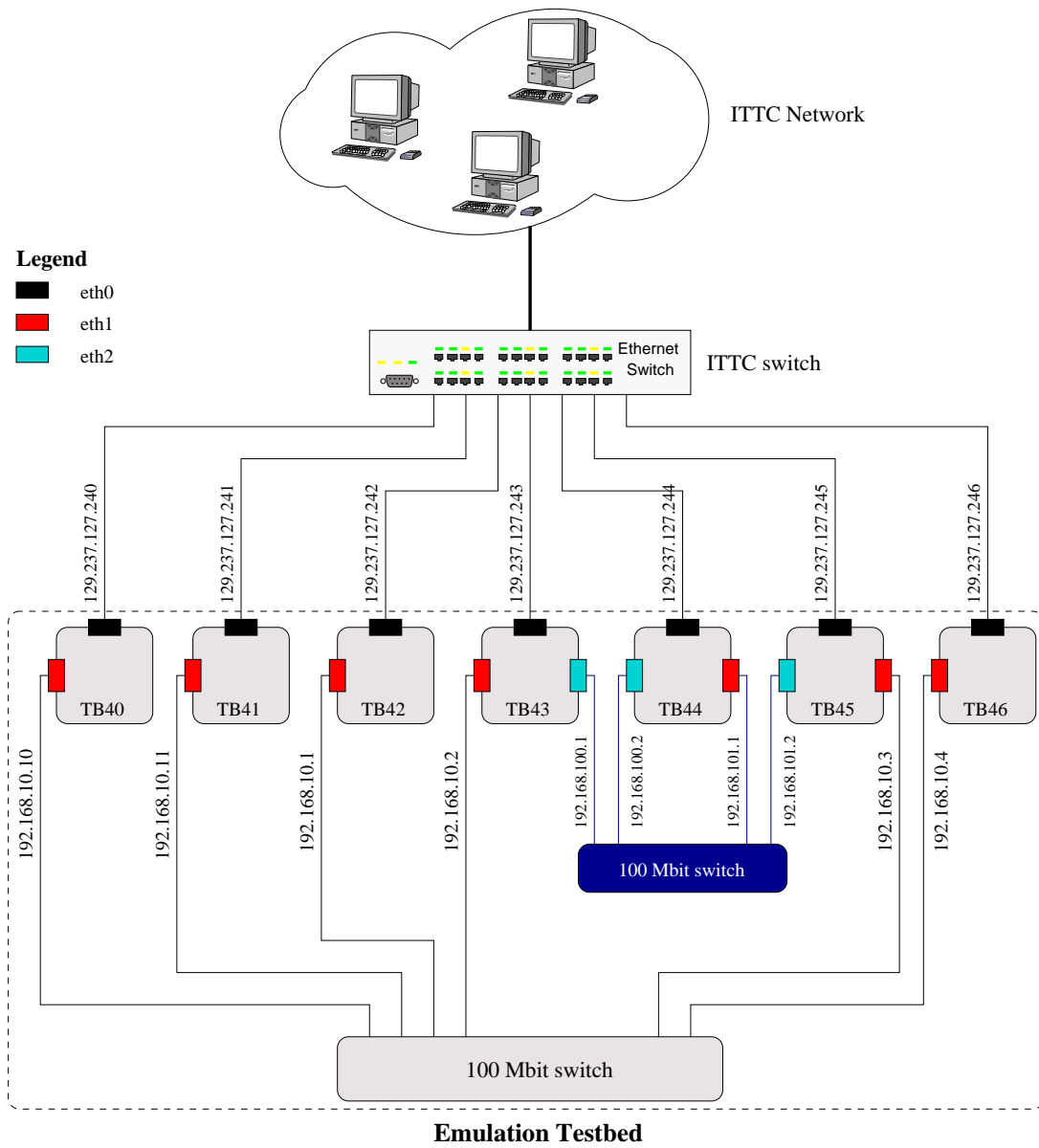
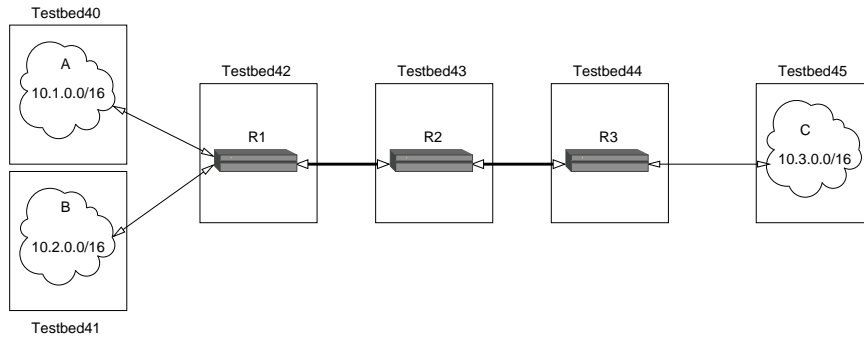
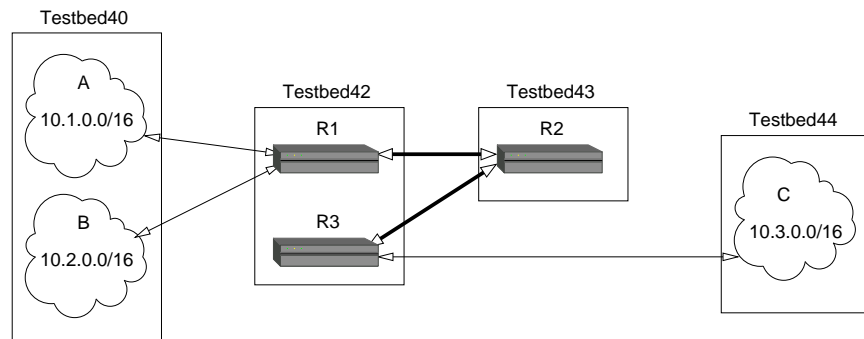


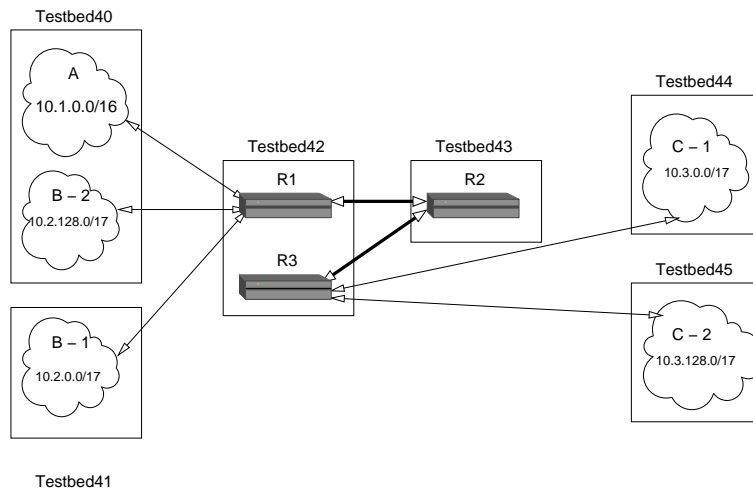
Figure 6.2: Emulation Testbed



(a) Virtual Network Topology - One-to-one mapping



(b) Virtual Network Topology - Many-to-one mapping



(c) Virtual Network Topology - Split subnet mapping

Figure 6.3: Various virtual network topologies

Figure 6.3 shows three different virtual network topologies to emulate the test network shown in Figure 6.1. Figure 6.3(a) shows an emulated virtual network that is a one-to-one mapping to the physical network in that subnets A, B and C are emulated on separate physical boxes and so are the routers R1, R2 and R3. A point to be noted though is that there is some amount of virtualization since each subnet, which can contain many hosts, is being emulated on a single physical machine using virtual hosts. Figure 6.3(b) shows a more *aggressively* virtualized network, in that subnet A and B are simulated on the same physical machine and so are routers R1 and R3. Note that we cannot send traffic from subnet A to B in this case since the source and destination hosts are on the same physical machine as mentioned in Section 3.6 (Page 24). Figure 6.3(c) shows a virtualized network in which the emulation of subnets B and C are split up and emulated on two physical machines. If necessary, they could be split up into smaller subnets to be emulated on more machines. These different virtual network topologies allow the users to choose the best topology to suit the physical network to be emulated. There are pros and cons in selecting each topology as shown in Table 6.1. Typically, if possible, one should start virtualizing a network by virtualizing a subset of the network with a one-to-one mapping. Once all the problems have been ironed out, one can move to a many-to-one or split-subnet topology.

VNET Topology	Pros	Cons
One-to-one	Simple to visualize, maintain and configure	Requires more machines
	Easier to debug for potential problems due to virtualization	
Many-to-one	Requires fewer machines	More difficult to setup correctly; complex subnet mapping
		More difficult to observe obscure problems
Split-subnet	Allows load balancing by utilization of multiple CPUs	More difficult to setup correctly; complex subnet mapping

Table 6.1: Pros and Cons of various virtual network topologies

The next section discusses the results of performing experiments using virtual network elements and some of the difficulties encountered along the way.

6.1.3 Evaluation and Results

We performed extensive testing on the three virtual network topologies depicted in Figure 6.3 to study the effectiveness of network emulation using our virtual network element framework. We were interested in understanding the difficulties in setting up virtual networks, performance characteristics of these networks and any unexpected behavior resulting from virtualization. For our experiments we ensured that each physical machine (testbed) was running a Network Time Protocol (NTP) daemon so that all the machines were fairly* synchronized.

The traffic on the virtual network was generated through the Netspec `nstestd` daemon. Initially, the traffic was uni-directional, flowing from subnet A and B to subnet C and care was taken to ensure that the aggregate traffic sent from subnet A and B did not exceed 100Mbps[†]. The virtual network was configured using the `nsvethd` daemon and scripts similar to that shown in Program 5.1. Each of the subnets had variable number of virtual hosts in them (2-10). The aggregate traffic flowing through the virtual network was varied from 10Mbps to 100Mbps with each virtual host generating different amounts of traffic.

As we cranked up the data-rates, we noticed that we did not receive the expected throughput when packets were sent through the virtual devices. When the same data-rates were send on a physical network, we received the expected throughput. This led to a belief that the virtual network elements were somehow dropping packets. After spending an extensive amount of time trying to debug the VNET code, it was determined that the NetGear switch was acting as a hub and broadcasting the traffic to all ports. More investigation showed that the reason for this was the inability of the switch to determine the MAC addresses of the machines connected to it because of a lack of ARP packets on the network. Apparently, the switch learns the MAC address of the hosts connected to it's ports by reading the ARP request and reply packets. In the case of the virtual network, all the MAC addresses are available to the virtual network elements through configuration. Therefore, the machines do not send out ARP requests. We fixed

*NTP can be accurate to within a few milliseconds

[†]Virtual network traffic is multiplexed over 100Mbps physical interfaces.

the problem by running a small script on each testbed that sends out ARP requests to each testbed participating in the experiment. This ensured that the switch recognized which ports those machines were connected to before the start of the experiment.

Once the ARP problem was solved, the virtual network behaved as expected and we received the expected throughput over the virtual network. We found no packet losses and the effective end-to-end latency was almost the same as that observed over the physical network. With the successful testing of a simple case of network emulation using virtual network elements, we turned our attention to emulating QoS networks.

6.2 Emulation of 9-element Diffserv network

The Diffserv network architecture we are interested in emulating was developed for experiments on physical machines as part of research on Hybrid QoS architectures[34, 42]. The research focused on control-plane complexity of this architecture while in emulation we are merely concerned about the data-plane results. Our goal is to verify that the emulated network has the same data-plane characteristics for the real-time behavior aggregates (BA) passing through the Diffserv core network.

Emulation of a Diffserv network over a virtual network requires minor changes to the traffic control code inside the kernel and `tc` utility in user-space as discussed in Section 5.1.3 and Section 5.1.2 respectively. These changes allow the Linux traffic control framework to recognize packets of the virtual network.

6.2.1 Architecture of Physical 9-element Diffserv network

Traffic type	Source	Sink	DSCP
Best effort	TB11	TB17	0x00
Real time	TB40	TB41	0x38
Test	TB42	TB46	0x28

Table 6.2: Traffic generated in Physical Diffserv network

The Diffserv network architecture used for physical testing in [34, 42] is shown in Figure 6.4. It consists of nine network elements: 6 hosts and 3 routers. TB43, TB44

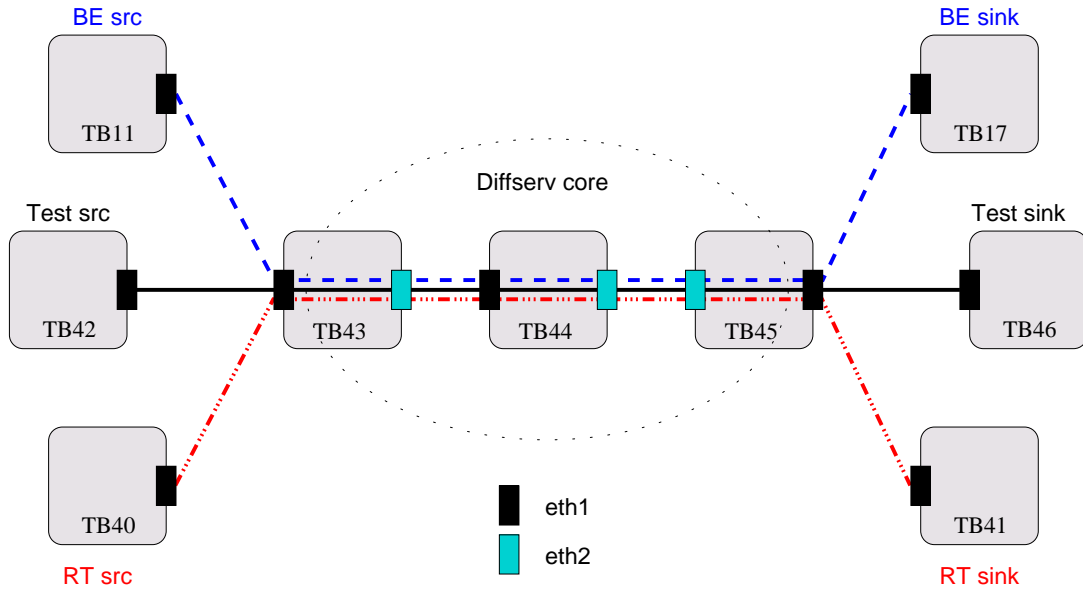


Figure 6.4: Physical Network Architecture for 9-element Diffserv network test

and TB45 form the Diffserv core routers. The others machines are used to generate background best-effort(BE) load, background real-time(RT) load and the *test* load as shown in Table 6.2. The background BE and RT flows are used to simulate network traffic that is competing with the *test* traffic for link bandwidth in the Diffserv core. Also, the outgoing links on all core routers are capped at 10Mbps using a HTB class: 6Mbps for RT and 4 Mbps for BE traffic. This ensures that when an aggregate traffic greater than 10Mbps enters the Diffserv core network, it gets congested causing loss or re-prioritizing of packets.

The *test* load is fixed at 4Mbps (Program B.5 in Appendix B) while the BE and RT loads are each varied from 0-10Mbps as shown in Programs B.3 and B.4 and Tables B.3 and B.4. Since *test* traffic is fixed at 4Mbps, that leaves about 2Mbps of background RT traffic to go through without causing congestion in the RT queues of the routers. Background RT traffic greater than that causes congestion in the RT queues.

The *test* and background RT traffic streams are marked with a Diffserv code point (DSCP) at their sources (Programs B.10 and B.9) depending on whether they are in-profile (DSCP=0x28) or out-of-profile (DSCP=0x38) at a given instant as shown in

Figure 4.7. This DSCP marking is achieved by initially characterizing the test and background RT traffic using token bucket parameters and the result is then passed to the filter in the `tc` scripts. The traffic characterization parameters are shown in Table 6.3.

Traffic type	Rate (bytes/s)	Burst (bytes)
Background RT	202355	1514
Test	542078	1514

Table 6.3: Traffic characterization - Token bucket parameters (Physical)

The Diffserv core routers are configured (Program B.13) with a queue structure similar to the one shown in Figure 4.8, so that the RT and BE traffic are serviced by different classes thus isolating them. The steps involved in automated testing of a Diffserv network are as follows:

1. Setup routes so that,
 - TB43 is the next-hop gateway for TB11, TB40 and TB42,
 - TB45 is the next-hop gateway for TB17, TB41 and TB46,
 - TB44 is next-hop gateway for TB43 and TB45.
2. Configure Linux traffic control to mark packets on TB40 and TB42 and apply Diffserv PHBs on core routers TB43, TB44 and TB45.
3. Setup `tcpdump` to capture transmitted packets on TB42 and received packets on TB46.
4. Start the background BE and RT traffic sources and let them fill up the queues in the network.
5. Start the *test* traffic source and measure the throughput received by the *test* traffic.
6. Post-process the `tcpdump` traces to find the end-to-end delay encountered by the packets of the *test* flow through the Diffserv network.

BG-RT (Mbps)	BG-BE (Mbps)	Thruput (Mbps)	Mean Delay (μ s)	Max (μ s)	Min (μ s)	Variance	Std. Dev
0	4	3.9910	0.00123	0.00354	0.00114	5.96e-08	0.00024
	6	3.9907	0.00126	0.00398	0.00115	6.78e-08	0.00026
	8	3.9910	0.00131	0.00422	0.00119	7.79e-08	0.00028
	10	3.9907	0.00134	0.00404	0.00122	8.14e-08	0.00029
1.5	4	3.9907	0.00130	0.00340	0.00123	4.19e-08	0.00020
	6	3.9910	0.00145	0.00406	0.00134	6.67e-08	0.00026
	8	3.9907	0.00152	0.00399	0.00143	5.58e-08	0.00024
	10	3.9910	0.00160	0.00412	0.00149	7.06e-08	0.00027
4	4	3.4750	0.01550	0.05133	0.00100	5.64e-05	0.00749
	6	3.4385	0.01554	0.05234	0.00100	6.62e-05	0.00807
	8	3.4967	0.01380	0.03638	0.00102	4.73e-05	0.00688
	10	3.4490	0.01525	0.06454	0.00109	8.38e-05	0.00914
6	4	3.4607	0.01399	0.03849	0.00119	4.01e-05	0.00633
	6	3.4577	0.01395	0.03870	0.00114	3.97e-05	0.00630
	8	3.4750	0.01388	0.03815	0.00108	3.95e-05	0.00628
	10	3.4797	0.01369	0.03820	0.00103	3.95e-05	0.00628
8	4	3.5490	0.01381	0.03141	0.00099	3.58e-05	0.00598
	6	3.5430	0.01588	0.03715	0.00105	4.44e-05	0.00661
	8	3.5473	0.01384	0.03122	0.00106	3.53e-05	0.00595
	10	3.5483	0.01386	0.03082	0.00105	3.51e-05	0.00593
10	4	3.5107	0.01384	0.03142	0.00104	3.37e-05	0.00581
	6	3.5123	0.01801	0.03460	0.00411	3.32e-05	0.00576
	8	3.5153	0.01386	0.03218	0.00110	3.39e-05	0.00582
	10	3.5163	0.01389	0.03230	0.00110	3.37e-05	0.00580

Table 6.4: Results - 9-element Physical Diffserv network test

6.2.2 Conclusions from 9-element Physical Diffserv network testing

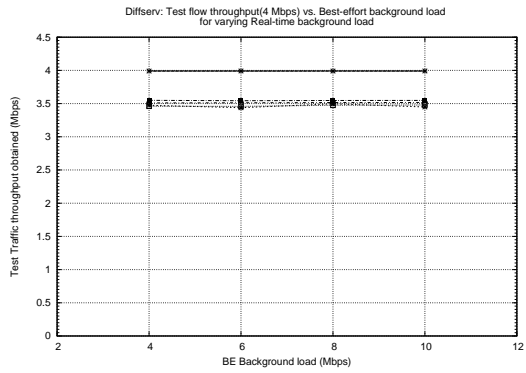
Table 6.4 shows the throughput and delay results obtained for a *test* flow of 4Mbps that was sent through the Diffserv network in the presence of varying background BE and RT traffic. Each row is a mean of the results of 3 iterations with the same traffic parameters. The graphs for throughput received by *test* flow for varying background BE and RT traffic are shown in Figures 6.5(a) and 6.5(c). The graphs for delay experienced by packets of the *test* flow for varying background BE and RT traffic is shown in Figure 6.6(a). The two throughput graphs show that for background RT traffic under 2Mbps, the *test* flow receives the expected 4Mbps throughput. As the background RT traffic increases to 10Mbps, thereby overloading the RT class inside the Diffserv core routers, some packets are reclassified with a higher drop precedence and some are dropped. As a result, the *test* flow receives reduced throughput(3.4-3.6Mbps). It has been experimentally seen that the Diffserv router provides the *test* flow a much better quality of service than a best-effort router[34, 42]. A point to be noted is that the background BE traffic does not affect the *test* traffic since it is serviced by a different queue. In the next section, we consider the results from the emulated Diffserv network and compare both the results.

6.2.3 Architecture of 9-element Virtual Diffserv network

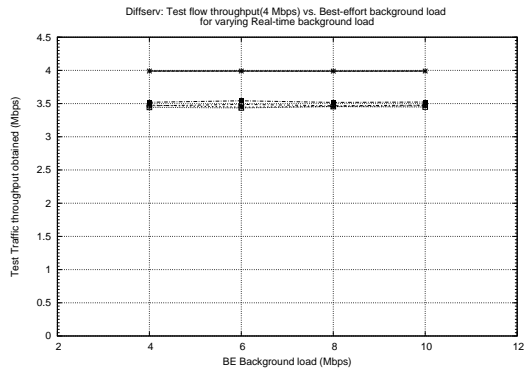
Traffic type	Source	Sink	DSCP
Best effort	TB40-veth1	TB41-veth1	0x00
Real time	TB40-veth2	TB41-veth2	0x38
Test	TB42	TB46	0x28

Table 6.5: Traffic generated in Virtual Diffserv network

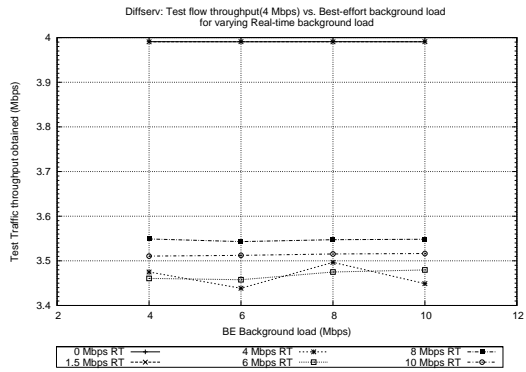
The Diffserv network architecture used for virtual network emulation is based on Figure 6.4 and is shown in Figure 6.7. This is a minimally virtualized topology in which the hosts generating the background traffic are emulated on a single pair of nodes instead of two pairs. That leads to a saving of two nodes for the experiment. TB43, TB44 and TB45 emulate one Diffserv core router each. The others machines are used to emulate



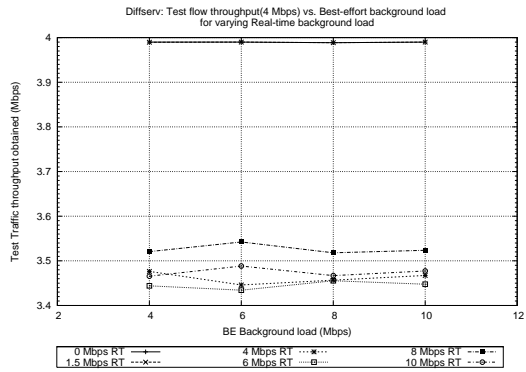
(a) Physical Diffserv Throughput



(b) Virtual Diffserv Throughput



(c) Physical Diffserv Throughput Zoomed



(d) Virtual Diffserv Throughput Zoomed

Figure 6.5: Throughput in 9-element Diffserv network

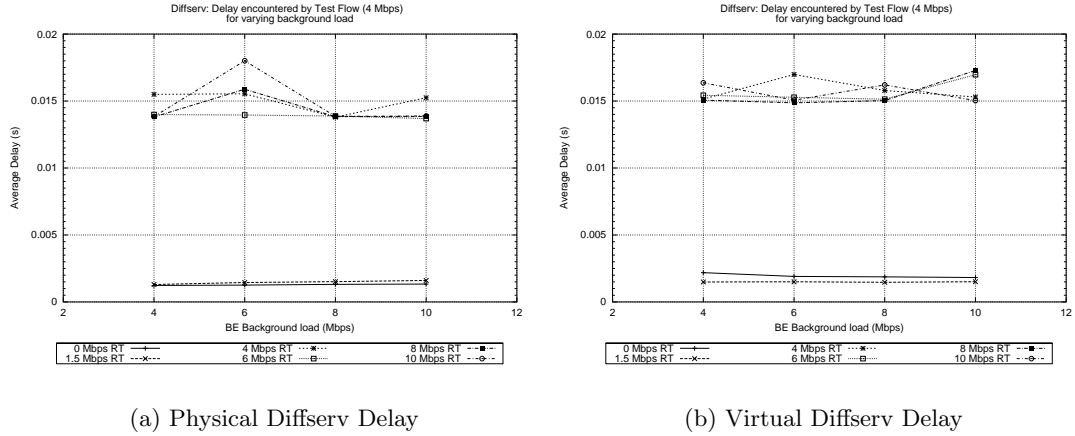


Figure 6.6: Delay in 9-element Diffserv network

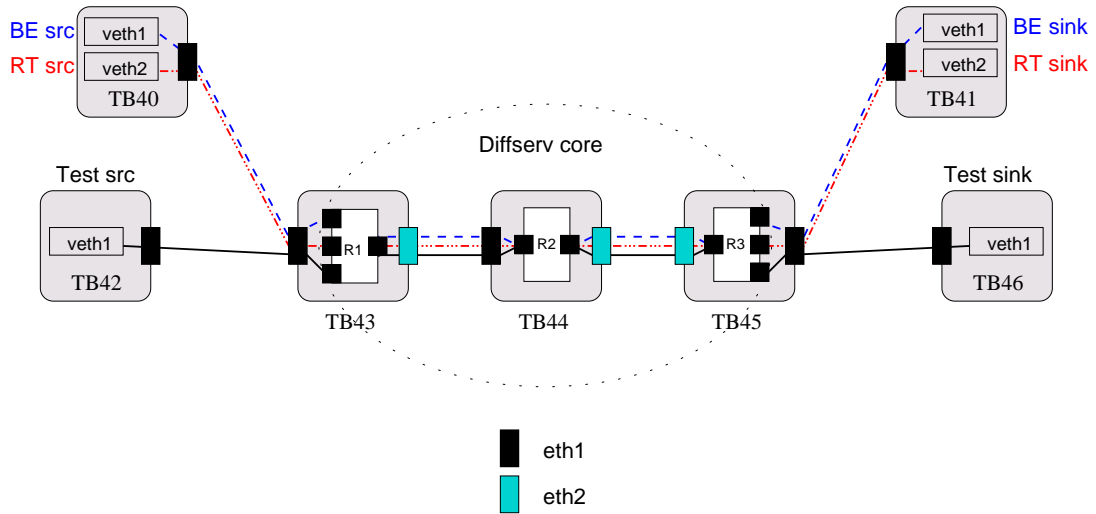


Figure 6.7: Virtual Network Architecture for 9-element Diffserv network test

hosts that generate background best-effort(BE) load, background real-time(RT) load and the *test* load as shown in Table 6.5.

The traffic load is the same as in the physical testing i.e. the *test* load is fixed at 4Mbps (Program B.8 in Appendix B) while the BE and RT loads are each varied from 0-10Mbps as shown in Programs B.6 and B.7 and Tables B.3 and B.5.

The *test* and background RT traffic streams are marked with a Diffserv code point (DSCP) at their sources (Programs B.12 and B.11) depending on whether they are in-profile (DSCP=0x28) or out-of-profile (DSCP=0x38) at a given instant. This DSCP marking is achieved due to traffic characterization using token bucket parameters which is then passed to the filter in the `tc` scripts. The traffic characterization parameters which are different from those in physical experiment are shown in Table 6.6.

Traffic type	Rate (bytes/s)	Burst (bytes)
Background RT	205817	1514
Test	548161	1514

Table 6.6: Traffic characterization - Token bucket parameters (Virtual)

The emulated Diffserv core routers are configured (Program B.14) with a queue structure similar to the one shown in Figure 4.8, so that the RT and BE traffic are serviced by different classes thus isolating them. The steps involved in automated testing of a Diffserv network are as follows:

1. Setup virtual network so that,
 - TB43 has a virtual router with four ports: three for background BE, RT and test traffic source networks and one to exchange traffic with the virtual router on TB44,
 - TB45 has a virtual router with four ports: three for background BE, RT and test traffic destination networks and one to exchange traffic with the virtual router on TB44,
 - TB44 has a two-port virtual router, one exchanging traffic with the virtual

router on TB43 and the other with the virtual router on TB45.

2. Configure Linux traffic control on the correct virtual interfaces to mark packets on TB40-`veth2` and TB42-`veth1` and apply Diffserv PHBs on core routers TB43-`r1p4`, TB44-`r2p2` and TB45-`r3p4`.
3. Setup `tcpdump` to capture transmitted packets on TB42-`veth1` and received packets on TB46-`veth1`.
4. Run a script on each machine to send MAC requests for each of the other machines in the experiment to avoid problems with the switch as discussed in Section 6.1.3.
5. Start the background BE and RT traffic sources and let them fill up the queues attached to the virtual interfaces in the network.
6. Start the *test* traffic source and measure the throughput received by the *test* traffic.
7. Post-process the `tcpdump` traces to find the end-to-end delay encountered by the packets of the *test* flow through the Diffserv network.

6.2.4 Conclusions from 9-element Virtual Diffserv network test

Table 6.7 shows the throughput and delay results obtained for a *test* flow of 4Mbps that was sent through the emulated Diffserv network on top of virtual network elements in the presence of varying background BE and RT traffic. The graphs for throughput received by *test* flow for varying background BE and RT traffic are shown in Figures 6.5(b) and 6.5(d). The graph for delay experienced by packets of the *test* flow for varying background BE and RT traffic is shown in Figure 6.6(b). The two throughput graphs show that for background RT traffic under 2Mbps, the *test* flow receives the expected 4Mbps throughput. As the background RT traffic increases to 10Mbps, thereby overloading the RT class inside the emulated Diffserv core routers, some packets are re-classified and eventually dropped. Thus the *test* flow receives only 3.4-3.6Mbps throughput.

BG-RT (Mbps)	BG-BE (Mbps)	Thruput (Mbps)	Mean Delay (s)	Max (s)	Min (s)	Variance	Std. Dev
0	4	3.9900	0.00219	0.00490	0.00197	2.29e-07	0.00047
	6	3.9900	0.00191	0.00481	0.00173	1.82e-07	0.00043
	8	3.9887	0.00187	0.00405	0.00168	1.77e-07	0.00042
	10	3.9900	0.00183	0.00534	0.00151	3.87e-07	0.00061
1.5	4	3.9897	0.00149	0.00337	0.00136	1.14e-07	0.00034
	6	3.9900	0.00151	0.00447	0.00132	1.95e-07	0.00044
	8	3.9887	0.00147	0.00381	0.00131	1.45e-07	0.00038
	10	3.9900	0.00151	0.00402	0.00131	1.94e-07	0.00044
4	4	3.4763	0.01515	0.03749	0.00133	4.49e-05	0.00670
	6	3.4460	0.01698	0.04977	0.00135	6.04e-05	0.00775
	8	3.4567	0.01579	0.04825	0.00134	5.75e-05	0.00753
	10	3.4673	0.01530	0.04078	0.00132	4.79e-05	0.00691
6	4	3.4440	0.01543	0.03887	0.00132	3.31e-05	0.00575
	6	3.4343	0.01528	0.05157	0.00128	4.70e-05	0.00678
	8	3.4553	0.01514	0.04308	0.00123	3.72e-05	0.00610
	10	3.4477	0.01697	0.04001	0.00119	4.18e-05	0.00644
8	4	3.5207	0.01507	0.04165	0.00118	4.14e-05	0.00638
	6	3.5423	0.01487	0.03451	0.00121	3.52e-05	0.00593
	8	3.5180	0.01503	0.03368	0.00128	3.40e-05	0.00583
	10	3.5237	0.01729	0.03859	0.00317	3.88e-05	0.00618
10	4	3.4657	0.01636	0.04663	0.00150	4.52e-05	0.00666
	6	3.4883	0.01506	0.03233	0.00168	3.02e-05	0.00550
	8	3.4670	0.01620	0.04143	0.00201	4.20e-05	0.00635
	10	3.4773	0.01503	0.03249	0.00180	3.07e-05	0.00554

Table 6.7: Results - 9-element Virtual Diffserv network test

6.2.5 Comparison of physical and virtual 9-element Diffserv results

The physical experiment and the virtual experiment show similar throughput and delay results for the *test* flow through a Diffserv network. The throughput results in both cases are between 3.4 and 3.6Mbps for background RT traffic over 2Mbps. Similarly, the delay results are between 1.4 and 1.8ms when background RT traffic exceeds 2Mbps. It should be pointed out, that in the virtual experiment, all the rate limiting and scheduling algorithms act on the virtual host interface or the virtual router ports. Thus the Linux Traffic control can be effectively used in network emulation experiments using virtual network elements. This similarity in results substantiates our efforts to create a faithful network emulation platform using virtual network elements.

6.3 Emulation of 17-element Diffserv network

Now that we confirmed the efficacy of using virtual network elements to emulate IP networks (a 9-element Diffserv network example in the previous section), we would like to use this powerful concept to emulate larger networks, the biggest advantage of VNE. Our goal is to emulate a 17-element Diffserv network as shown in Figure 6.8. The figure shows four autonomous networks NW1, NW2, NW3 and NW4 communicating through the Diffserv core network. NW1 sends traffic to NW2 and NW3 sends traffic to NW4. Both pairs of networks negotiate a SLA with the Diffserv core network provider under which other traffic inside the core should not affect metered traffic from their networks. In the event of any pair of networks overloading their negotiated SLA, the Diffserv provider is free to mark the overload with high drop probability and drop it too. Each pair of networks is similar to the example seen in the previous section (Figure 6.4), in that they have background BE and RT traffic streams and a *test* traffic stream. The background RT and *test* traffic streams are marked with DSCP values as shown in Table 6.8. Thus, at the Diffserv border router on TB43, the traffic for the two networks is segregated based on its DSCP and each of the networks' packets follow different paths through the Diffserv core.

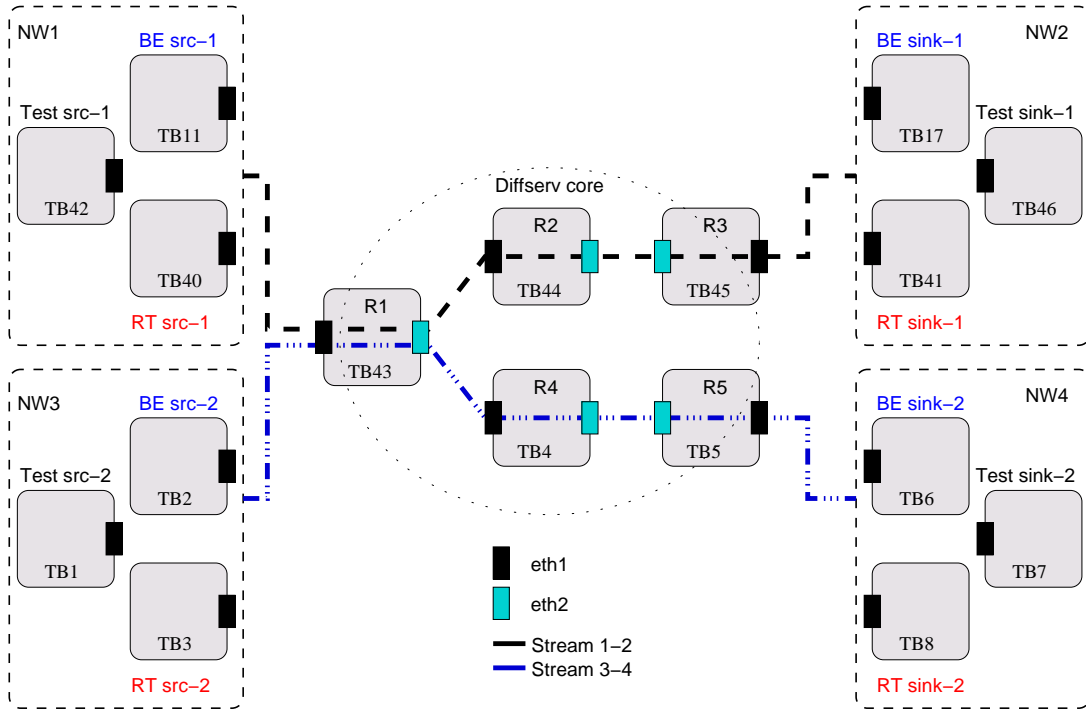


Figure 6.8: Physical Network Architecture for 17-element Diffserv network test

Traffic type	Belonging to	DSCP
In-profile	NW1-2	0x28
Out-of-profile	NW1-2	0x38
In-profile	NW3-4	0x48
Out-of-profile	NW3-4	0x58

Table 6.8: DSCP values assigned to traffic in 17-element Diffserv network

6.3.1 Architecture of 17-element Virtual Diffserv network

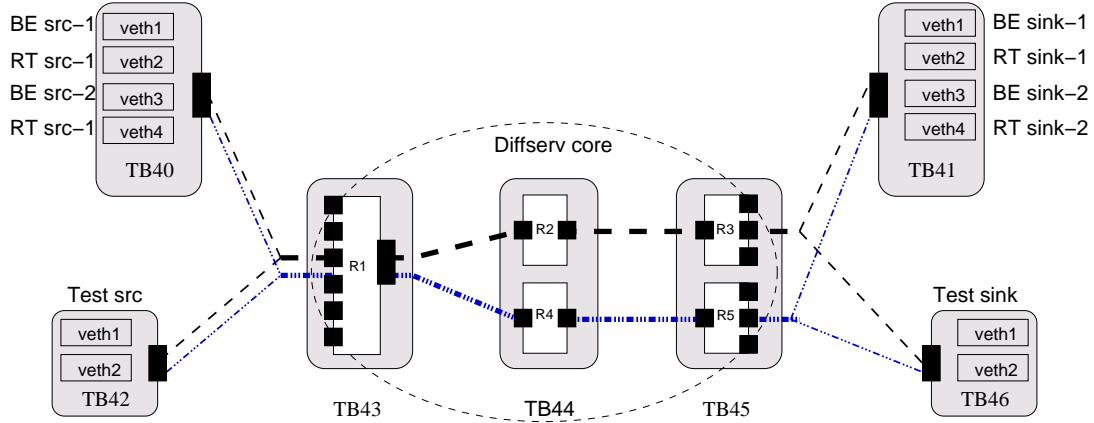


Figure 6.9: Virtual Network Architecture for 17-element Diffserv network test

We emulated the 17-element physical Diffserv network with a virtual network on 7 physical machines, the configuration of which is shown in Figure 6.9. As seen in the figure, the four background traffic sources are emulated on a single physical machine TB40, while the four sinks are emulated on TB41. Similarly TB44 and TB45 are configured with two emulated routers, one belonging to each of the paths taken by packets through the Diffserv network. TB43 carries the maximum load since all the packets from both NW1 and NW3 flow through it. Depending on the overall load and the load of the individual networks, it might drop some traffic and forward the remaining traffic to the respective routers emulated on TB44.

Each of the two source networks generates a 4Mbps *test* traffic stream. The background RT traffic is varied from zero to 6Mbps and the background BE traffic is varied from zero to 4Mbps in each of the networks. We configured the DiffServ border router emulated on TB43 to have an *outgoing* link bandwidth of 16Mbps: 12Mbps reserved for RT traffic and 4Mbps reserved for BE traffic. This should allow both the *test* flows (8Mbps) to pass through unaffected along with a total of 4Mbps background RT traffic without any packet losses. Similarly a total of 4Mbps background BE traffic will pass through unaffected. The emulated DiffServ routers R1-R5 are configured similar to the

9-element Diffserv network described in the previous section except for the DSCP which is different for the two networks as shown in Table 6.8.

The results of the experiments are shown in Table 6.9. Each result is a mean of 3 iterations of the experiment. For purposes of tabulation, NW1-2 is denoted by E2E NW 1 and NW3-4 is denoted by E2E NW 2. Results 1-6 use the same background RT and BE traffic in both the networks, so the *test* traffic stream in both networks is impacted similarly. Results 7-8 show an instance of a badly behaved neighbor, in this case E2E NW 2, that sends more traffic than specified in the SLA.

Sr.	E2E NW	BG-RT (Mbps)	BG-BE (Mbps)	Thruput (Mbps)	Mean Delay (s)	Max (s)	Min (s)
1	1	0	2	3.9901	0.00153	0.00400	0.00136
	2	0	2	3.9901	0.00151	0.00401	0.00136
2	1	0	4	3.9904	0.00156	0.00340	0.00140
	2	0	4	3.9901	0.00158	0.00358	0.00140
3	1	1.5	2	3.9901	0.00164	0.00506	0.00129
	2	1.5	2	3.9901	0.00170	0.00514	0.00129
4	1	1.5	4	3.9900	0.00219	0.00488	0.00183
	2	1.5	4	3.9897	0.00225	0.00495	0.00183
5	1	4	4	3.4735	0.01519	0.03756	0.00173
	2	4	4	3.4648	0.01686	0.04584	0.00173
6	1	6	6	3.4752	0.01505	0.03879	0.00163
	2	6	6	3.4551	0.01602	0.03991	0.00158
7	1	1.5	2	3.9901	0.00226	0.00639	0.00186
	2	4	4	3.4799	0.01558	0.03887	0.00186
8	1	1.5	4	3.9901	0.00215	0.00571	0.00181
	2	6	6	3.4591	0.01529	0.04077	0.00187

Table 6.9: Results - 17-element Virtual Diffserv network test

6.3.2 Conclusions from 17-element Virtual Diffserv network test

Table 6.9 shows the throughput and delay results obtained for the two *test* flows of 4Mbps flowing through each E2E Diffserv network. Results 1-6 show that as the background BE and RT traffic in each network is increased, the *test* flow gets impacted similarly in both networks as expected. Results 1-4 shows that when the total real-

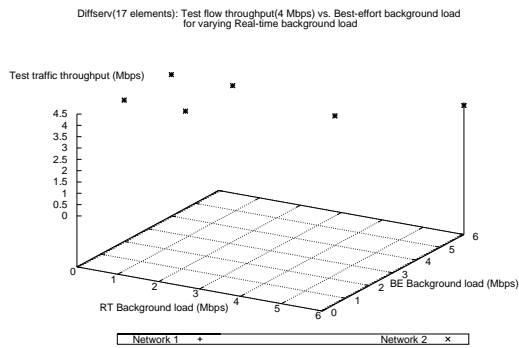
time traffic is below 12Mbps (Σ test-1, test-2, BG-RT-1, BG-RT-2), the *test* flow passes through unaffected (3.99Mbps throughput). Even overloading the background BE class in results 2 and 4 does not impact the RT traffic.

Results 5 and 6 show both E2E networks overloading their BE and RT traffic classes. This causes the Diffserv routers to drop packets from both classes. Even under such a overloaded network, the *test* flows receive QoS similar to results obtained in the 9-element Diffserv test (~ 3.5 Mbps).

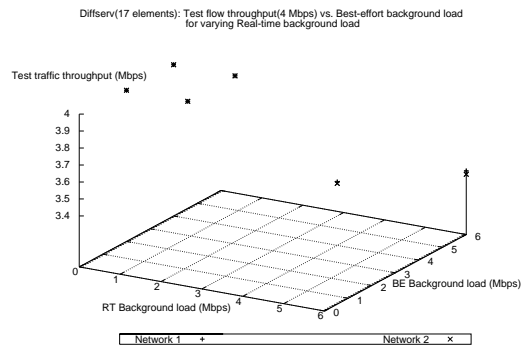
Results 7 and 8 bring out a as-of-yet untested property of Diffserv. When multiple customers share a Diffserv core network, a bad neighbor - a customer transmitting more traffic from his network than the negotiated SLA should not affect the good citizens of the network. In our case, E2E NW 2 overloads the RT and BE traffic classes, while E2E NW 1 transmits only its share of traffic. As seen from the results, the *test* flow from NW 1 passes through unaffected, while the background RT flow from NW 2 undergoes traffic conditioning at border Diffserv router on TB43. Even then the *test* flow from NW 2 receives QoS from the Diffserv core network (~ 3.5 Mbps).

Figure 6.10 shows the graphs for the throughput results 1-6 obtained for the 17-element Diffserv network. Figures 6.10(a) and 6.10(b) show the throughput results plotted against different scales on the Z-axis for different BG-RT and BG-BE traffic. Figures 6.10(c) and 6.10(d) show the same results plotted in a 3D perspective to clarify the results further. As the RT class gets overloaded when background RT traffic on both networks exceeds 2Mbps, we see a small drop in throughput received by the *test* flows in both networks due to the dropping of packets. However, as seen before in the 9-element Diffserv network, the *test* flow in the Diffserv network receives a better QoS than it would have in a best-effort IP network. This is seen by the leveling off of the throughput results plane in Figure 6.10(d) after the drop.

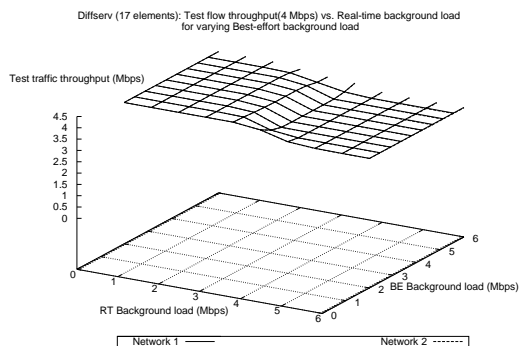
In this experiment, we *correctly* emulated a 17-element Diffserv network on just 7 physical machines. This reaffirms our assertion that virtualization of network elements is not difficult and can be used to perform larger experiments that yield correct results without a corresponding increasing in the required hardware.



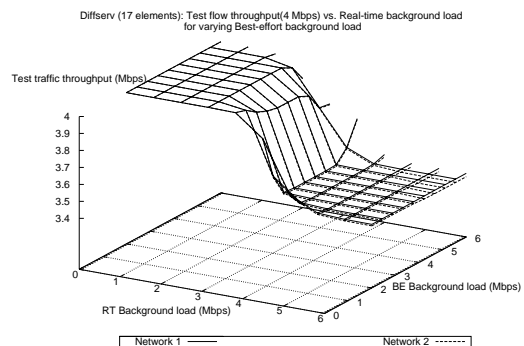
(a) Diffserv Throughput



(b) Diffserv Throughput Zoomed



(c) Diffserv Throughput - 3D



(d) Diffserv Throughput Zoomed - 3D

Figure 6.10: Throughput in 17-element Diffserv network

6.4 Emulation of Intserv network

The Intserv network architecture we emulate is very similar to the 17-element Diffserv network that we tested in the previous section. The only difference is that a RSVP daemon is configured on each network element to carry messages for a priori signaling and resource reservation before sending the data traffic. Hence each (virtual) host and (virtual) router in the emulated network is associated with a RSVP daemon. In cases where multiple virtual elements are being emulated on the same physical machine, there exist multiple RSVP daemons, each associated with one of the network elements. Although the physical testing in the research focused on control-plane complexity of this architecture, in emulation we are merely concerned about the data-plane results. Our goal is to verify that the multiple RSVP daemons are faithful to the Intserv standard and that the emulated Intserv network has the same data-plane characteristics for the real-time traffic passing through Intserv-capable routers as the physical Intserv network.

Emulation of an Intserv network over a virtual network requires major changes to the RSVP daemon as discussed in Section 5.1.4. These changes allow multiple instances of the RSVP daemon to execute on a single physical machine by being bound to the multiple virtual routers and hosts being emulated on the machine. However, RSVP has built-in code which uses the CBQ queuing discipline and class in Linux to schedule the real-time controlled load flows. As mentioned before, CBQ does not produce accurate data-plane results and was replaced in our Diffserv tests with HTB which does a better job of accurate packet scheduling. Modification of the RSVP daemon to use HTB is a significantly large project which was beyond the scope of the current work. Our goal was to show the efficacy and ease of our approach in virtualization of IP networks. Hence we will just describe the configuration of a virtual Intserv network without looking at the results of these experiments.

The emulated Intserv network looks exactly the same as in Figure 6.9, except that the Diffserv queuing disciplines are replaced by RSVP daemons for each virtual element; thus we have 17 instances of RSVP daemons in the virtual network. The following are the sequence of steps to configure a virtual Intserv network with RSVP signaling:

Program 6.1 RSVP commands

Host:

```
rsvpd -D -i veth1 -a 11000 -e 11001 -p 11002 -S 11003 -L 11004
```

Router:

```
rsvpd -U -D -i vrouter1 -a 11000 -e 11001 -p 11002 -S 11003 -L 11004
```

Identification of interfaces:

22:42:29.039 Physical, Virtual, and API interfaces are:

22:42:29.040	0 vnet	10.0.0.1/8	NoIS
22:42:29.040	1 vrouter3	10.254.3.254/8	NoIS
22:42:29.040	2 r3p1	10.20.0.254/8	NoIS
22:42:29.040	3 r3p2	10.2.0.254/8	NoIS
22:42:29.040	4 r3p3	10.40.0.254/8	NoIS
22:42:29.040	5 r3p4	10.101.1.254/8	NoIS
22:42:29.040	6 API	0.0.0.0/0	NoIS

PATH command:

```
dest udp 10.2.1.1/4444
sender 10.1.1.1/4444 [t 30000 300 30000 64 1500]
```

Actual path message sent:

```
Register sender: 10.1.1.1/4444 T=[30K(300) 30KB/s 64 1.5K]
path_set_laddr: Enter
00:11:36.311| Rcv API PATH 10.2.1.1/4444[17] <API ttl=/63
PATH: Sess: 10.1.1.1/4444[17] R: 30000 PHOP: <0.0.0.0 LIH=1>
10.1.1.1/4444 T=[30K(300) 30KB/s 64 1.5K]
Adspec( 0 hop InfBW Ous 65535B, G={br!}, CL={br!})
```

PATH message received by router R1:

```
00:27:10.384| Rcv Raw PATH 10.2.1.1/4444[17] r1p2<=3 < 10.1.1.1/64
PATH: Sess: 10.2.1.1/4444[17] R: 30000 PHOP: <10.1.1.1 LIH=1>
10.1.1.1/4444 T=[30K(300) 30KB/s 64 1.5K]
Adspec( 0 hop InfBW Ous 65535B, G={br!}, CL={br!})
```

RESV command:

```
reserve ff 10.1.1.1/4444 [c1 30000 300 30000 64 1500]
```

1. Configure the 17-element virtual network as shown in Figure 6.9.
2. Setup an RSVP daemon attached to each virtual element in the network. The command to setup RSVP on a virtual host is shown in Program listing 6.1. The `-U` option tells the daemon that it is being configured on a router, hence the RSVP seeks all the ports of the router to bind to, as shown in the Program 6.1. The `-D` option puts the daemon in debug mode so that we can interactively enter the signaling commands.
3. Send `PATH` messages from all the sources in the network an example of which is shown in Program 6.1. The `dest` message establishes a new session and the `sender` message sends the `PATH` message with the requested traffic parameters in the square brackets. A flow is uniquely identified by its source and destination IP address and port numbers and protocol. In the example shown, the flow is a UDP flow from `10.1.1.1/4444` to `10.2.1.1/4444`
4. The `PATH` message flows across the virtual network after being modified by each router in its path. An example message received by virtual router R1 is shown in the Program. In cases where there are two virtual routers on a single physical machines, the message reaches both the RSVP daemons but is processed only by the daemon which is attached to the network element whose IP address is in the `next-hop` field of the packet.
5. Once the `PATH` message reaches its ultimate destination (`10.2.1.1` in the above example), we send a `RESV` message confirming the reservation parameters to all the intermediate network elements as shown in the Program 6.1.
6. This sequence of steps is carried out for each traffic source and sink pair till all the reservations are established. All the reservation classes can be seen using the `tc` utility.
7. Once all the reservations are established, the traffic is sent through the network and the received throughput is measured.

We successfully configured an Intserv network that created reservations for the real-time traffic. But since RSVP uses the CBQ queuing discipline, the throughput results were found to be inaccurate. This in no way implies a failure of the virtual network. It merely implies limitations of the existing RSVP daemon software that shows these inaccurate measurements even in a physical network.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

The communications and networking industry is experiencing an explosion of new ideas to bring better communication facilities to the end user. This ranges from end-to-end ubiquitous IP networks to providing quality of service over these networks. Efforts are underway to evaluate these technologies to learn about their scaling properties in complex scenarios. Evaluating communication networks has been a difficult problem due to our current inability to study networks of significant size and complexity without access to the real networks. Most often, large and complex networks are *production* networks which are not available for testing and experimentation. Currently, the only alternatives available to researchers are to either simulate the networks using software (OPNET, ns2) or study scaled down versions of these networks so that they can be setup or emulated in their laboratories. Both approaches allow the researcher to get acquainted with a new technology and are useful for rapid prototyping. Both approaches, however, have their shortcomings; simulation ignores the control plane complexity of configuring a real network and makes simplifying assumptions about the interactions between network software in the simulations while scaled down networks are very often not able to highlight the performance penalties on network software as the network is scaled.

The real problem is how to efficiently simulate large, complex networks without requiring to buy expensive hardware and without resorting to simplifications in the

software or scale of the target network. We have proposed a new form of emulation which emulates the basic elements of the network, namely the host and the router. Our approach aims to create a *Scalable Emulation Framework* to emulate IP networks using virtualization.

Emulation through virtualization tries to improve on the inherent drawbacks of other methods of evaluating networks; it tries to maintain real-world complexity and scale of a network. By configuring virtual hosts and virtual routers of a set of physical machines, we understand the control plane complexity of networks, while putting multiple virtual network elements on a single physical machine allows us to carry out large-scale experiments without requiring the corresponding hardware. Furthermore, networks emulated using virtual network elements use the *real code* that operating systems and network use, without having to resort to abstractions that lead to skewing of the results. Hence, the results that we see are very close to those seen on real networks. Most user-space applications should be able to use virtual network elements with minor modifications. Complex user-space applications like RSVP that make routing decisions in user-space only require to use the virtual routing table to route packets instead. Similarly, kernel-space code simply requires the ability to recognize a new packet type for virtual networks.

This work has demonstrated the ability of virtual network elements to emulate IP networks faithfully, giving extremely similar, arguably identical results to those obtained using conventional means. It has also shown examples of emulation of QoS networks which use more software than conventional best-effort IP networks. We have successfully adapted Diffserv and Intserv code in Linux to use virtual network elements with little or moderate effort not exceeding a week's worth of coding. It is worth mentioning that due to the open source and modular nature of implementation it could be easily enhanced to support more complex network features some of which are outlined in the next section.

7.2 Future Work

We realize the potential applications of virtual network elements in researching more complex problems such as routing, congestion control, flow control, etc. Clearly, the

current implementation does not handle them currently, but could be easily enhanced to support these features. The following is a brief list of enhancements and optimizations that we have managed to come up with for newer applications of virtual network elements.

Optimizations

- Optimize the packet flow path in the current implementation by removing the redundant paths and merging paths which execute the same code.
- Study hashing mechanism inside the virtual routing code more closely for possible optimization when using a larger pool of IP addresses.
- Optimize critical paths of execution by providing more efficient methods of demultiplexing virtual network traffic and optimizing the virtual router's forwarding code.

Enhancements

- Research the possibility of maintaining the virtual routing table for each virtual host and virtual router on a physical machine as an independent forwarding information base (FIB) inside the Linux kernel. Besides optimizing the VNET code due to removal of virtual routing house-keeping code this will lead to easier adaptation of existing network applications to use virtual network elements.
- Add support for dynamic virtual route updates through routing daemons such as zebra. This would involve hacking the routing daemon to enable it to discover virtual network topology and then install this information into the virtual routing tables. This would enable emulation of more complex networks running dynamic routing software.
- Hack the IP layer code to *prevent* short-circuit delivery of packets produced on an interface and destined to another interface on the same machine. This enhancement would allow tighter virtualization.

- Currently even the VNET layer short-circuits the path of the packet, in that, if the next hop (virtual router port) is on the same machine, the packet is directly delivered to that virtual router. We can change this and allow the packet to go out of the machine through the *sending* virtual interface and then come back as a normal virtual network packet that is then sent to the *receiving* port of the virtual router.
- Add ARP protocol support to virtual network elements so that they can request and reply to ARPs from other virtual network elements. This would do away the requirement of having to send out ARP messages for all physical machines before the start of an experiment. Failing this, write a Netspec daemon to send out ARP requests to all machine every few minutes during the duration of the experiment. The switches remember the MAC addresses of attached machines for about 4 minutes. Currently, if an experiment runs for more than 4 minutes, the switch will start behaving as a hub suddenly unless we re-run the ARP script.
- Introduce the concept of a virtual link. This concept could be used to divide the bandwidth of a physical interface among all the virtual network elements using that interface to multiplex their traffic. This could be a configuration option when creating a virtual network element or an explicit option without which the element cannot use the physical interface.
- Create topology visualization and script generation tools. As the networks being emulated get larger, writing scripts to configure these networks becomes cumbersome and error-prone. Thus front-end applications can be written that allow users to specify a network topology graphically and then generates the required configuration scripts.
- Add virtual network elements to the ProTEuS[28] framework. This is considered briefly in the next section.

7.3 Virtual Network Elements and ProTEuS

ProTEuS stands for Proportional Time Emulation and Simulation. It is a network evaluation framework developed at The University of Kansas[28, 29] for simulation of ATM networks using a concept of *proportional time*. In short, ProTEuS simulates networks by modifying the time-line in the life of a running network. It uses virtual ATM devices to emulate the ATM sources and sinks and virtual switches device to emulate the ATM cell switch along with real-time extensions to Linux, KU Real Time (KURT), which allows it finer-grained control over network events and simulation execution.

ProTEuS would allow us to simulate extremely complex networks in which events are too closely spaced to be successfully emulated using simple virtual network elements. It introduces another level of virtualization: *virtual time*. ProTEuS can modify the time-scale to expand or reduce it to successfully simulate the network. The ProTEuS framework has a heartbeat: *the epoch*. This is the most basic unit of time in a ProTEuS simulation and can be modified by the user. An epoch of execution is the real time it takes to simulate a virtual time interval.

Currently, the ProTEuS code makes assumptions of an ATM network layer and is tightly coupled to the concepts of cell switching. It has been the desire of many to be able to use ProTEuS to simulate IP networks. With ProTEuS, a researcher can simulate a large network using virtual hosts, virtual routers and virtual links on a virtual time scale. Currently virtual time is the missing piece in the solution to solve the problem of emulating arbitrarily large and complex networks. As of this writing, there exists an incomplete effort to port ProTEuS for use with virtual network elements using Ethernet interfaces and an IP network layer.

Appendix A

Important Linux functions

Program A.1 lists some of the important functions in the Linux protocol stack. All the filenames are referenced relative to the top-level directory of the Linux kernel.

Function name	Task	Source filename
<code>netif_rx</code>	Network top half: Receive a packet from a device driver and queue it in <code>backlog</code> queue for the upper protocol levels (interrupts disabled).	<code>/net/core/dev.c</code>
<code>net_bh</code>	Network bottom half: Process each packet in the <code>backlog</code> queue, determine its type (IP, ARP, etc.) and hand it over to correct protocol handler (interrupts enabled).	<code>/net/core/dev.c</code>
<code>dev_queue_xmit</code>	Queues a packet on a interface's queue when it is ready to be transmitted and calls the transmit function for the network device.	<code>/net/core/dev.c</code>
<code>qdisc_wakeup</code>	Wakes up a device queue whenever a packet is enqueued on it (by <code>dev_queue_xmit</code>) and calls <code>qdisc_restart</code> .	<code>include/net/pkt_sched.h</code>
<code>qdisc_restart</code>	Dequeues a packet from the queue according to the characteristics of the attached queuing discipline and transmits it onto the wire.	<code>net/sched/sch_generic.c</code>
<code>device->hard_header</code>	Called by the <code>neighbor</code> routines to add a hardware header to the packet after having determined which interface the packet will go out on.	<code>include/linux/netdevice.h</code>

Table A.1: Important functions in the Linux network stack

Appendix B

Patches and Scripts

This appendix lists all the scripts and code patches to run the experiments described in Section 6. Listed in the following sections are the Netspec scripts used for traffic generation, tc scripts to configure the Linux traffic control system, patches to tools to get them to work with Linux and Linux kernel patches.

B.1 Patches

All the experiments were performed on 1GHZ Intel PIII machines with at least 256Mb RAM. They are running the Linux kernel version 2.2.18 with various patches as listed in Table B.1. This sections lists the patches applied to various tools or the kernel to enable certain features or to improve performance.

Patch	URL
Diffserv	http://www.ssi.bg/~ja/ds9/
HTB3	http://www.ssi.bg/~ja/ds9/
KURT	http://www.ittc.ku.edu/kurt/

Table B.1: Patches applied to Linux kernel

Patch name	Applied to	Description	Reference
iproute2 VNET patch	iproute2	Enables tc to recognize virtual network packets of type ETH_P_KUVMNET	Program B.1
Traffic control UTIME patch	Linux kernel	Allows Linux traffic control to use real-time scheduling through UTIME	Program B.2

Table B.2: Miscellaneous Patches

Program B.1 iproute VNET patch

```
diff -BburN clean-iproute2-2.2.4-now-ss991023/lib/ll\_proto.c \
iproute2-2.2.4-now-ss991023/lib/ll\_proto.c
--- clean-iproute2-2.2.4-now-ss991023/lib/ll\_proto.c  Fri Apr 26 02:40:01 2002
+++ iproute2-2.2.4-now-ss991023/lib/ll\_proto.c  Thu Apr 25 18:20:56 2002
@@ -67,6 +66,7 @@
  \_PF(CONTROL,control)
  \_PF(IRDA,irda)

+{ ETH\_P\_KUVMNET, "vnet" },
+ { 0x8001, "802.1Q" },
+ { ETH\_P\_IP, "ipv4" },
+ };
```

Program B.2 Traffic control UTIME patch

```
--- pkt_sched.h.old      Wed May 29 18:16:26 2002
+++ pkt_sched.h Wed May 29 18:16:37 2002
@@ -5,7 +5,7 @@
 #define PSCHED_JIFFIES          2
 #define PSCHED_CPU              3

-#define PSCHED_CLOCK_SOURCE     PSCHED_JIFFIES
+#define PSCHED_CLOCK_SOURCE     PSCHED_CPU

#include <linux/pkt_sched.h>
#include <net/pkt_cls.h>
```

B.2 Scripts

This section lists the various scripts used to run the experiments described in Section 6. These include the Netspec scripts used for traffic generation and `tc` scripts to configure the Linux traffic control system. In the case of traffic generation using Netspec scripts, traffic is generated using the formula:

$$\text{datarate}(\text{bps}) = \frac{\text{packetsize}(\text{bits})}{\text{period}(\text{seconds})}$$

Data Rate (Mbps)	Packet size (bytes)
4	5243
6	7865
8	10486
10	13108

Table B.3: Packet sizes for background BE traffic (period = 10ms)

Data Rate (Mbps)	Period (μs)
1.5	7487
4	2808
6	1872
8	1404
10	1123

Table B.4: Period for background RT traffic (Packet size = 1472 bytes)

To generate background best-effort (BE) traffic, we keep the period constant at 10 milliseconds. Therefore, for generating various background BE data rates, the packet size is varied as shown in Table B.3. The values of the packet size from the table are used for the value of `blocksize` parameter (`blocksize=X`) in all `Background BE` scripts.

To generate background real-time (RT) traffic, we keep the packet size constant at 1472 bytes (for physical network testing) or 1456 bytes (for virtual network testing). Therefore, for generating various background RT data rates, the period is varied as

Data Rate (Mbps)	Period (μ s)
1.5	7405
4	2777
6	1851
8	1389
10	1111

Table B.5: Period for background RT traffic (Packet size = 1456 bytes)

shown in Tables B.4(physical network) and B.5(virtual network). The values of the period from the table are used for the value of `period` parameter (`period=Y`) in the Background RT scripts.

Program B.3 Netspec script for generating Background BE traffic - Physical Network

```

cluster {
  test testbed11 {
    type = burst(blocksize=X,period=10000, duration=30);
    protocol = udp;
    own = testbed11:9009;
    peer = testbed17:9009;
  }
  test testbed17 {
    type = sink (blocksize=X, duration=30);
    protocol = udp;
    own = testbed17:9009;
    peer = testbed11:9009;
  }
}

```

Program B.4 Netspec script for generating Background RT traffic - Physical Network

```
cluster {
  test testbed40 {
    type = burst(blocksize=1472,period=Y, duration=30);
    protocol = udp;
    own = 192.168.10.10:6610;
    peer = 192.168.10.11:6610;
  }
  test testbed41 {
    type = sink (blocksize=1472, duration=30);
    protocol = udp;
    own = 192.168.10.11:6610;
    peer = 192.168.10.10:6610;
  }
}
```

Program B.5 Netspec script for generating Test traffic - Physical Network

```
cluster {
  test testbed42 {
    type = burst(blocksize=1472,period=2807,duration=10);
    protocol = udp ;
    own = 192.168.10.1:9001;
    peer = 192.168.10.4:9001;
  }
  test testbed46 {
    type = sink (blocksize=1472, duration=10);
    protocol = udp ;
    own = 192.168.10.4:9001;
    peer = 192.168.10.1:9001;
  }
}
```

Program B.6 Netspec script for generating Background BE traffic - Virtual Network

```
cluster {
  test testbed40 {
    type = burst(blocksize=X, period=10000, duration=30);
    protocol = udp;
    own = 10.10.1.1:6609;
    peer = 10.20.1.1:6609;
  }
  test testbed41 {
    type = sink (blocksize=X, duration=30);
    protocol = udp;
    own = 10.20.1.1:6609;
    peer = 10.10.1.1:6609;
  }
}
```

Program B.7 Netspec script for generating Background RT traffic - Virtual Network

```
cluster {
  test testbed40 {
    type = burst(blocksize=1456, period=Y, duration=30);
    protocol = udp;
    own = 10.30.1.1:6610;
    peer = 10.40.1.1:6610;
  }
  test testbed41 {
    type = sink (blocksize=1456, duration=30);
    protocol = udp;
    own = 10.40.1.1:6610;
    peer = 10.30.1.1:6610;
  }
}
```

Program B.8 Netspec script for generating Test traffic - Virtual Network

```
cluster {
    test testbed42 {
        type = burst(blocksize=1456,period=2777,duration=10);
        protocol = udp;
        own = 10.1.1.1:9001;
        peer = 10.2.1.1:9001;
    }
    test testbed46 {
        type = sink (blocksize=1456, duration=10);
        protocol = udp;
        own = 10.2.1.1:9001;
        peer = 10.1.1.1:9001;
    }
}
```

Program B.9 TC script for marking background RT traffic - Physical Network

```
#!/bin/sh

TC=/usr/bin/tc
DEVICE=eth1

$TC qdisc add dev $DEVICE handle 1:0 root dsmark indices 64
$TC class change dev $DEVICE parent 1:0 classid 1:1 dsmark mask 0x3 value 0x28
$TC class change dev $DEVICE parent 1:0 classid 1:2 dsmark mask 0x3 value 0x38
$TC class change dev $DEVICE parent 1:0 classid 1:3 dsmark mask 0x3 value 0x00

$TC filter add dev $DEVICE parent 1:0 protocol ip prio 1 rsvp ipproto \
udp session 192.168.10.11/6610 police rate 202355bps burst 1514 \
continue flowid 1:1

# filter for real time BG traffic
$TC filter add dev $DEVICE parent 1:0 protocol ip prio 2 rsvp ipproto \
udp session 192.168.10.11/6610 flowid 1:2

$TC filter add dev $DEVICE parent 1:0 protocol ip prio 3 u32 match ip \
tos 0x00 0x00 flowid 1:3
```

Program B.10 TC script for marking test traffic - Physical Network

```
#!/bin/sh

TC=/usr/bin/tc
DEVICE=eth1

$TC qdisc add dev $DEVICE handle 1:0 root dsmark indices 64
$TC class change dev $DEVICE parent 1:0 classid 1:1 dsmark mask 0x3 value 0x28
$TC class change dev $DEVICE parent 1:0 classid 1:2 dsmark mask 0x3 value 0x38
$TC class change dev $DEVICE parent 1:0 classid 1:3 dsmark mask 0x3 value 0x00

$TC filter add dev $DEVICE parent 1:0 protocol ip prio 1 rsvp ipproto \
udp session 192.168.10.4/9001 police rate 542078bps burst 1516 \
continue flowid 1:1

# filter for real time BG traffic
$TC filter add dev $DEVICE parent 1:0 protocol ip prio 2 rsvp ipproto \
udp session 192.168.10.4/9001 flowid 1:2

$TC filter add dev $DEVICE parent 1:0 protocol ip prio 3 u32 match ip \
tos 0x00 0x00 flowid 1:3
```

Program B.11 TC script for marking background RT traffic - Virtual Network

```
#!/bin/sh

TC=/usr/bin/tc
DEV="dev veth2"

# !!! IMPORTANT - For virtual network
# 'rsvp' filter does NOT require protocol to be changed from 'ip' to 'vnet'
# 'u32' requires it.

$TC qdisc add $DEV handle 1:0 root dsmark indices 64
$TC class change $DEV parent 1:0 classid 1:1 dsmark mask 0x3 value 0x28
$TC class change $DEV parent 1:0 classid 1:2 dsmark mask 0x3 value 0x38
$TC class change $DEV parent 1:0 classid 1:3 dsmark mask 0x3 value 0x00

# Mark traffic destined for 10.40.1.1/6610
# In-profile packets are marked with TOS=0x28
# Policing rate is used to determine whether traffic is in-profile
$TC filter add $DEV parent 1:0 protocol ip prio 1 rsvp ipproto \
udp session 10.40.1.1/6610 police rate 205817bps burst 1514 \
continue flowid 1:1

# Out-of-profile packets are marked with TOS=0x38
$TC filter add $DEV parent 1:0 protocol ip prio 2 rsvp ipproto \
udp session 10.40.1.1/6610 flowid 1:2

# All other traffic is marked with TOS=0x00
$TC filter add $DEV parent 1:0 protocol vnet prio 3 u32 match ip \
tos 0x00 0x00 flowid 1:3
```

Program B.12 TC script for marking test traffic - Virtual Network

```
#!/bin/sh

TC=/usr/bin/tc
DEV="dev veth1"

# !!! IMPORTANT - For virtual network
# 'rsvp' filter does NOT require protocol to be changed from 'ip' to 'vnet'
# 'u32' requires the change

$TC qdisc add $DEV handle 1:0 root dsmark indices 64
$TC class change $DEV parent 1:0 classid 1:1 dsmark mask 0x3 value 0x28
$TC class change $DEV parent 1:0 classid 1:2 dsmark mask 0x3 value 0x38
$TC class change $DEV parent 1:0 classid 1:3 dsmark mask 0x3 value 0x00

# Mark traffic destined for 10.2.1.1/9001
# In-profile packets are marked with TOS=0x28
# Policing rate is used to determine whether traffic is in-profile
$TC filter add $DEV parent 1:0 protocol ip prio 1 rsvp ipproto \
udp session 10.2.1.1/9001 police rate 548161bps burst 1514 \
continue flowid 1:1

# Out-of-profile packets are marked with TOS=0x38
$TC filter add $DEV parent 1:0 protocol ip prio 2 rsvp ipproto \
udp session 10.2.1.1/9001 flowid 1:2

# All other traffic is marked with TOS=0x00
$TC filter add $DEV parent 1:0 protocol vnet prio 3 u32 match ip \
tos 0x00 0x00 flowid 1:3
```

Program B.13 TC script for core Diffserv routers - Physical Network

```
#!/bin/sh -x

TC="/usr/bin/tc"
DEV="dev eth1"
BW="bandwidth 10Mbit"
AVPKT="avpkt 1514"

#####
#           Class Structure
#####
# DSMARK qdisc to classify packets according to their TOS bits
$TC qdisc add $DEV handle 1:0 root dsmark indices 64 set_tc_index

# HTB qdisc
$TC qdisc add $DEV parent 1:0 handle 2:0 htb

# Real-time class (6Mbit)
$TC class add $DEV parent 2:0 classid 2:1 htb rate 6Mbit ceil 6Mbit

# Best-effort class (4Mbit)
$TC class add $DEV parent 2:0 classid 2:2 htb rate 4Mbit ceil 4Mbit

# GRED qdisc in real time class
$TC qdisc add $DEV parent 2:1 gred setup DPs 3 default 2 prio

# 3 GRED qdiscs for the 3 drop priorities of real time packets
$TC qdisc change $DEV parent 2:1 gred limit 12KB min 3KB max 9KB \
burst 20 $AVPKT $BW DP 1 probability 0.02 prio 2
$TC qdisc change $DEV parent 2:1 gred limit 12KB min 3KB max 9KB \
burst 20 $AVPKT $BW DP 2 probability 0.04 prio 3
$TC qdisc change $DEV parent 2:1 gred limit 12KB min 3KB max 9KB \
burst 20 $AVPKT $BW DP 3 probability 0.06 prio 4

# Best effort class is served using a RED qdisc
$TC qdisc add $DEV parent 2:2 red limit 12KB min 3KB max 9KB \
burst 20 $AVPKT $BW probability 0.4

#####
#           Filters for classifying the packets
#####
# Filter to get 6 TOS bits from packet
$TC filter add $DEV parent 1:0 protocol ip prio 1 \
tcindex mask 0xfc shift 2 pass_on

# Change the skb->tcindex of packets to the correct class.
# skb->tcindex is "changed" only if the parent qdisc of the filter
# is 'dsmark'
$TC filter add $DEV parent 1:0 protocol ip prio 1 \
handle 10 tcindex classid 1:111
$TC filter add $DEV parent 1:0 protocol ip prio 1 \
handle 12 tcindex classid 1:112
$TC filter add $DEV parent 1:0 protocol ip prio 1 \
handle 14 tcindex classid 1:113
$TC filter add $DEV parent 1:0 protocol ip prio 2 \
handle 0 tcindex classid 1:1

# Filter to get first 2 bits which denote AF class
$TC filter add $DEV parent 2:0 protocol ip prio 1 \
tcindex mask 0xf0 shift 4 pass_on

# Put packets of AF1 into real time class
$TC filter add $DEV parent 2:0 protocol ip prio 1 \
handle 1 tcindex classid 2:1

# Put other packets into best effort class
$TC filter add $DEV parent 2:0 protocol ip prio 1 \
handle 0 tcindex classid 2:2
```

Program B.14 TC script for core Diffserv routers - Virtual Network

```
#!/bin/sh -x

TC="/usr/bin/tc"
DEV="dev r1p4"
BW="bandwidth 10Mbit"
AVPKT="avpkt 1514"

#####
#           Class Structure
#####

# DSMARK qdisc to classify packets according to their TOS bits
$TC qdisc add $DEV handle 1:0 root dsmark indices 64 set_tc_index

# HTB qdisc
$TC qdisc add $DEV parent 1:0 handle 2:0 htb

# Real-time class (6Mbit)
$TC class add $DEV parent 2:0 classid 2:1 htb rate 6Mbit ceil 6Mbit

# Best-effort class (4Mbit)
$TC class add $DEV parent 2:0 classid 2:2 htb rate 4Mbit ceil 4Mbit

# GRED qdisc in real time class
$TC qdisc add $DEV parent 2:1 gred setup DPs 3 default 2 prio

# 3 GRED qdiscs for the 3 drop priorities of real time packets
$TC qdisc change $DEV parent 2:1 gred limit 12KB min 3KB max 9KB \
burst 20 $AVPKT $BW DP 1 probability 0.02 prio 2
$TC qdisc change $DEV parent 2:1 gred limit 12KB min 3KB max 9KB \
burst 20 $AVPKT $BW DP 2 probability 0.04 prio 3
$TC qdisc change $DEV parent 2:1 gred limit 12KB min 3KB max 9KB \
burst 20 $AVPKT $BW DP 3 probability 0.06 prio 4

# Best effort class is served using a RED qdisc
$TC qdisc add $DEV parent 2:2 red limit 12KB min 3KB max 9KB \
burst 20 $AVPKT $BW probability 0.4

#####
#           Filters for classifying the packets
#####
# !!!!!!! IMPORTANT: For virtual network !!!!!!!
# Notice that for 'tcindex' filters, the protocol is NOT 'ip', it is 'vnet'
# This requires a special version of 'tc' with 2 lines added to the code to
# register the vnet packet type

# Filter to get 6 TOS bits from packet
$TC filter add $DEV parent 1:0 protocol vnet prio 1 \
tcindex mask 0xfc shift 2 pass_on

# Change the skb->tcindex of packets to the correct class.
# skb->tcindex is "changed" only if the parent qdisc of the filter
# is 'dsmark'
$TC filter add $DEV parent 1:0 protocol vnet prio 1 \
handle 10 tcindex classid 1:111
$TC filter add $DEV parent 1:0 protocol vnet prio 1 \
handle 12 tcindex classid 1:112
$TC filter add $DEV parent 1:0 protocol vnet prio 1 \
handle 14 tcindex classid 1:113
$TC filter add $DEV parent 1:0 protocol vnet prio 2 \
handle 0 tcindex classid 1:1

# Filter to get first 2 bits which denote AF class
$TC filter add $DEV parent 2:0 protocol vnet prio 1 \
tcindex mask 0xf0 shift 4 pass_on

# Put packets of AF1 into real time class
$TC filter add $DEV parent 2:0 protocol vnet prio 1 \
handle 1 tcindex classid 2:1

# Put other packets into best effort class
$TC filter add $DEV parent 2:0 protocol vnet prio 1 \
handle 0 tcindex classid 2:2
```

Bibliography

- [1] Opnet. <http://www.opnet.com/>.
- [2] RSVP - Resource ReSerVation Protocol. <http://www.isi.edu/rsvp/>.
- [3] The QoS Forum. <http://www.qosforum.com>.
- [4] W. Almesberger. *Scalable Resource Reservation for the Internet*. PhD thesis, EPFL, Nov 1999.
- [5] W. Almesberger. Linux Network Traffic Control - Implementation Overview. Technical Report FIXME!!!!!!!, EPFL ICA, 1999 April.
- [6] W. Almesberger, J. Boudec, and T. Ferrari. Scalable Resource Reservation for the Internet. In *Proceedings of the IEEE Conference Protocols for Multimedia Systems – Multimedia Networking (PROMSMmNet)*, (Santiago, Chile), Nov 1997.
- [7] W. Almesberger, S. Giordano, R. Mameli, S. Salsano, and F. Salvatore. A Prototype Implementation for the Intserv Operation over Diffserv Networks. Article, Institute for Computer Communications and Applications (ICA), Ecole Polytechnique Federale de Lausanne (EPFL), 2000.
- [8] W. Almesberger, J. Hadi Salim, and A. Kuznetsov. Differentiated Services on Linux. In *Proceedings of Globecom'99*, pages 831–836, December 1999.
- [9] G. Armitage. *Quality of Service in IP Networks. Foundations for a Multi-Service Internet*. Macmillan Technical Publishing, USA, first edition, 2000.

- [10] F. Baker, C. Iturralde, F. Le Faucheur, and B. Davie. Aggregation of Rsvp for IPv4 and IPv6 Reservations. RFC 3175, IETF Network Working Group, September 2001.
- [11] Y. Bernet, P. Ford, R. Yavatkar, F. Baker, L. Zhang, M. Speer, R. Braden, B. Davie, J. Wroclawski, and E. Felstaine. A Framework for Integrated Services Operations over Diffserv Networks. RFC 2998, IETF Network Working Group, November 2000.
- [12] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An Architecture for Differentiated Services. RFC 2475, IETF Network Working Group, December 1998.
- [13] R. Bless and K. Wehrle. Evaluation of Differentiated Services using an Implementation under Linux. In *Proceedings of the 7th IFIP International Workshop on Quality of Service (IWQOS'99)*, published by IEEE ???, June 1999.
- [14] B. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin. Resource Reservation Protocol (rsvp) - Version 1 Functional Specification. RFC 2205, IETF Network Working Group, September 1997.
- [15] R. Braden, D. Clark, and S. Shenker. Integrated Services in the Internet Architecture: an Overview. RFC 1633, ISI, MIT and PARC, June 1994.
- [16] L. Breslau, D. Estrin, K. Fall, S. Floyd, J. Heidemann, A. Helmy, P. Huang, S. McCanne, K. Varadhan, Y. Xu, and H. Yu. Advances in Network Simulation. *IEEE Computer Magazine*, 33(5):59–67, May 2000.
- [17] B. Buchanan, D. Niehaus, G. Dhandapani, R. Menon, S. Sheth, Y. Wijata, and S. House. The Datastream Kernel Interface (Revision A). Technical Report ITTC-FY98-TR-11510-04, Information and Telecommunication Technology Center, University of Kansas, 1994 June.

- [18] 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, IETF, March 2002.
- [19] B. Hubert et al. Linux advanced routing and traffic control. <http://www.lartc.org>.
- [20] J. Evans et al. Rapidly Deployable Radio Network.
<http://www.ittc.ku.edu/RDRN/>.
- [21] L. Torvalds et al. The Linux Kernel Archives. <http://www.kernel.org>.
- [22] W. Almesberger et al. ATM on Linux. <http://icawww1.epfl.ch/linux-atm>.
- [23] W. Almesberger et al. Diffserv on Linux. <http://diffserv.sourceforge.net>.
- [24] K. Fall. Network Emulation in the Vint/NS Simulator. Technical report, University of California at Berkeley, 1999.
- [25] T. Ferrari. *QoS support for Integrated Networks*. PhD thesis, University of Bologna, Nov 1998.
- [26] S. Floyd and V. Jacobson. Link-Sharing and Resource Management Models for Packet Networks. *IEEE/ACM Transactions on Networking*, 3(4):365–385, August 1995.
- [27] J. Heinanen, F. Baker, W. Weiss, and J. Wroclawski. Assured Forwarding PHB Group. RFC 2597, IETF, June 1999.
- [28] S. House. Proportional Time Emulation and Simulation of ATM Networks. Master’s thesis, University of Kansas, December 2000.
- [29] S. House, S. Murthy, and D. Niehaus. Proportional Time Simulation of ATM Networks. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1999.
- [30] S. House, D. Niehaus, R. Sanchez, and A. Kucheria. Virtual Network Devices. Technical Report ITTC-FY2001-TR-13200-X, Information and Telecommunication Technology Center, University of Kansas, May 2003.

- [31] IETF. Differentiated Services (diffserv) working group.
<http://www.ietf.org/html.charters/diffserv-charter.html>.
- [32] IETF. Integrated Services (intserv) working group.
<http://www.ietf.org/html.charters/intserv-charter.html>.
- [33] IP Infusion. Virtual Routing for Provider Edge Applications.
http://www.ipinfusion.com/pdf/VirtualRouting_app-note_3rev0302.pdf.
- [34] V. Jenkal. Implementation and Complexity Analysis of RSVP/Diffserv Hybrid IPQoS Models. Master's thesis, University of Kansas, January 2003.
- [35] R. Jonkman. NetSpec: Philosophy, Design and Implementation. Master's thesis, University of Kansas, February 1998.
- [36] R. Jonkman, D. Niehaus, J. Evans, and V. Frost. NetSpec: A Network Performance Evaluation Tool. Technical Report ITTC-FY98-TR-10980-28, Information and Telecommunication Technology Center, University of Kansas, December 1998.
- [37] D. Katz. IP Router Alert Option. RFC 2113, IETF Network Working Group, February 1997.
- [38] K. Kilkki. *Differentiated Services for the Internet*. Macmillan Technical Publishing, USA, 1999.
- [39] A. Kuznetsov. Iproute2. <ftp://ftp.inr.ac.ru/ip-routing/iproute2-current.tar.gz>.
- [40] A. Kuznetsov. Modifications to ISI's rsvp to run on linux.
<ftp://ftp.inr.ac.ru/rsvp/>.
- [41] D. McWherter, J. Sevy, and W. Regli. Building an IP Network Quality-of-Service Testbed. *IEEE Internet Computing*, 4(4):65–73, July/August 2000.
- [42] K. Nathillvar. Implementation and Performance Evaluation of Tunneled Aggregated RSVP Architecture. Master's thesis, University of Kansas, February 2003.

- [43] 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, IETF Network Working Group, December 1998.
- [44] NIST. NIST Network Emulation Tool. <http://w3.antd.nist.gov/tools/nistnet/>.
- [45] Kivimaki Perttu. Co-operation of Differentiated and Integrated Services in IP Networks. Master's thesis, Tampere University of Technology, February 2000.
- [46] Saravanan Radhakrishnan. Internet protocol - quality of service page. <http://qos.ittc.ku.edu>.
- [47] Vlora Rexhepi and Geert Heijenk. Interoperability of Integrated Services and Differentiated Services Architectures. Article 12/036-FCPNB 102 88 Uen, Ericsson, October 2000.
- [48] Alessandro Rubini and Jonathan Corbet. *Linux Device Drivers*. O'Reilly and Associates, second edition, 2001.
- [49] R. Sanchez. Virtual ATM Switch Driver for ATM on Linux. <http://www.ittc.ukans.edu/~rsanchez/software/vswitch.html>.
- [50] S. Shenker and J. Wroclawski. General Characterization Parameters for Integrated Services Network Elements. RFC 2215, IETF Network Working Group, September 1997.
- [51] J. Wroclawski. Specification of the Controlled-Load Network Element Service. RFC 2211, IETF Network Working Group, September 1997.
- [52] J. Wroclawski. Specification of the Guaranteed-Load Quality of Service. RFC 2212, IETF Network Working Group, September 1997.
- [53] J. Wroclawski. The Use of RSVP with IETF Integrated Services. RFC 2210, IETF Network Working Group, September 1997.

- [54] J. Wroclawski and A. Charny. Integrated Service Mappings for Differentiated Services Networks. Internet Draft draft-ietf-issll-ds-map-01.txt, IETF, February 2001.