Model-Based Quality of Service Control

by

Mohamed A. El Gendy

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2005

Doctoral Committee:
Professor Kang G. Shin, Chair
Associate Professor Sugih Jamin
Associate Professor Brian Noble
Assistant Professor Mingyan Liu

ABSTRACT

Model-Based Quality of Service Control

by

Mohamed A. El Gendy

Chair: Kang G. Shin

This thesis presents a *model-based* feedback control approach for network Quality-of-Service (QoS), that achieves accurate control, better resource utilization, and faster response to network and traffic changes than other prevalent techniques. It is a significant departure from the traditional approaches in providing network QoS, that either calculate and provision *a priori* the required network resources in a *feed-forward* way to meet certain QoS requirements, or use ad-hoc adaptive solutions, that need significant tuning and calibration efforts. Models are used as the basis for performance control of the network problems under study. We apply our control approach at several points in the network, starting with differentiated packet marking, where we introduce a new marking algorithm for assured services based on a feedback model of TCP congestion control. Then, we develop models for per-hop QoS using statistical modeling techniques and build model-based controllers for QoS over network nodes using these models. We further use control-theoretic design techniques to build QoS-controlled FIFO-based network services. These services are *reservation-less* and *self-controlled*, so they are capable of tracking reference QoS in a way that could not be easily achieved using operator-controlled or other commonly-used techniques. The problems and applications studied in the thesis demonstrate the effectiveness and the importance of using new and versatile methods such as statistical analysis, design of experiments and control-theory in studying network QoS. We evaluate the new

control approach using both simulation and experimentation on a Linux-based network testbed showing its correct functionality as well as comparing it with other available QoS schemes. We also present a two-tier *QoS Gateway* architecture that works at the first mile to facilitate the use of these controls for QoS-sensitive applications and end-user devices and enables better deployment and utilization of QoS frameworks in general. The overall research in this thesis will provide a better understanding of QoS modeling and control problems and enable wide QoS deployment.

To my Lord, the lord of heaven and earth..

ACKNOWLEDGEMENTS

I thank Allah (SWT) who gave me the strength and spirit to survive through the long and exhausting journey of my Ph.D. I always seek refuge to Him when it gets dark at the corners of my life for He is the only one Who gets me out to the light again. Ph.D. is a life cycle, in which, one learns and changes a lot, both professionally and personally. Without the help and company of many great people around me, it would have been much harder and less bearable, and I would not be able to thrive.

I would like to express my gratitude to Professor Kang Shin, my advisor and my second father, for believing in me and standing beside me along the way of my Ph.D. journey. I thank him for encouraging me all the time, teaching me how to overcome obstacles and be strong, giving me of his wisdom and long experience, motivating me, and building my self-confidence to lead a successful career after the Ph.D. I feel indebted to him for being such a great father to me and all his students, opening his kind heart and listening to us, backing us up, and never letting us worry about losing support. I owe him my success.

I am grateful to my parents for being there for me and for showering me with their never-ending love. Although they are overseas, but when I needed them they came all the way for me and they are always willing to do anything for me. They taught me patience, which I consider the most important thing in life to be able to survive. I attribute my success in life to them.

Special thanks go to my dear friends, Amgad Zeitoun and Hani Jamjoom, for being such great buddies to me along my journey. I consider them my mentors as they were always sincere to me, giving me great advices, and helping me to get over the stress and frustration of the Ph.D. process.

I would like to thank Abhijit Bose for being such a great colleague and friend. We

worked together a lot and I learned quite a bit from him. We went through a lot in our research and I wish him the best of luck in his Ph.D. and his career. I do not want to forget our previous colleague, Haining Wang, as the three of us had successful collaboration on exciting research problems.

I thank Professor Sugih Jamin for serving in my thesis committee, for his friendly support, and for giving me such a special place among his students, sharing interesting ideas and visions. I would also like to thank Professors Brian Noble and Mingyan Liu for serving in my thesis committee and for their feedback and suggestions.

I would also like to thank my friend Hosam Fathy in the Automotive Research Center for helping me with many important parts in my research and for collaborating with me. Having also good friends like Mohamed Abdel Moneum, Ahmed Omara, and Alaa El Rouby around me and giving me the psychological support I needed is a great blessing.

My appreciation goes to my colleagues at the RTCL, especially Padmanabhan (Babu) Pillai, John Reumann, and Songkuk Kim who have been of great help to me since the first day I came to Michigan. They gave me a lot of their time and were ready to answer any of my questions. I was lucky to be among such a friendly and cooperating environment. I also like to thank my officemates, Wei Sun, Zhigang Chen, Min-Gyu Cho, Shige Wang, and Kyu-Han Kim for sharing a lot of good and bad moments in my work and for giving me the hand-of-help whenever I ask for it.

I extend my sincere appreciation to BJ Monahgan for being a second mom to me with such a kind heart, and for giving us her utmost care and dedication which make our lives easier. I also like to thank Kirsten Knecht, Dawn Freysinger, Karen Liska, and Stephen Reger for their assistance and effort.

I had a great opportunity to work with Samsung corporation, especially Mathew Park, and I hope that such collaboration will never end even after my graduation.

It is always exciting to end a cycle in our life and start another one, full of unknowns and adventures, and I am asking my God, Allah (SWT), to bless me with similar great people along the way.

I hope that I did not forget anyone in my acknowledgment, and if I did, then I sincerely apologize for the mistake and hope they will accept my sincere appreciation.

Finally, I thank everyone who prayed for me.

TABLE OF CONTENTS

LIST OF FIGURES

Figure

LIST OF TABLES

Table

LIST OF APPENDICES

APPENDIX

CHAPTER 1

Introduction

The Internet has become the main communication vehicle for many of our daily-life and real-time applications. These applications, such as online shopping, banking and gaming, Voice-over-IP (VoIP) (e.g., IP telephony), and high definition TV (HDTV), depend on network parameters such as bandwidth, delay, jitter (inter-packet delay variation), and loss for their correct operation. The degree of tolerance or sensitivity to each of these parameters varies widely from one application to another. Critical applications, such as remote surgery and online trading systems, require reliability of such parameters as well as guaranteed delivery. Emerging applications such as home networking, intelligent appliances, factory supply-chain networks, as well as the vast majority of multimedia applications also require different levels of service quality from the underlying network in terms of these parameters. These applications impose throughput, delay, loss and other requirements on the Internet links and routers carrying their traffic. Providing predictable, as well as controllable, network performance at all times is still a major challenge even with the significant increase in network bandwidth. Network Quality-of-Service (QoS) is the term for describing how the network can serve these applications' traffic. It is defined in [129] as "the capability to provide resource assurance and service differentiation in a network."

1.1 Internet QoS

The Internet QoS has been the main focus for a large body of research over many years. It was motivated by the need to support the performance guarantees provided by circuit-switched networks, such as the telephone network on the packet-switched Internet. In traditional circuit-switched networks, the quality of a connection can be measured and guaranteed in terms of connection setup delays, media quality (e.g., voice or video quality), and trunk availability. On the other hand, packet-switched networks, especially IP networks such as the Internet, were not designed to provide per-connection service guarantees.[1] As a result, packets of a particular flow can reach their destination via different paths while experiencing different amounts of delay, jitter, and loss along the way. This is a critical problem in the current Internet as more applications, originally developed for circuit-switched networks, are being migrated to the Internet to take advantage of the reduced running cost. Additionally, new emerging applications rely on guarantees from the underlying IP network for their correct operation. This has led to the introduction of several network architectures that explicitly address the need to support QoS [34], such as Asynchronous Transfer Mode (ATM), IP precedence and type of service, Internet stream protocol (ST), Integrated Services (IntServ) [16] and its accompanying resource reservation and signaling protocol RSVP [135], virtual private networks (VPNs), real-time protocols (such as RTP, RTCP, and RTSP), optical networks (SONET/SDH), and more recently, the Differentiated Services (DiffServ) [13]. A common goal in all these architectures is to *control* the "quality" of forwarding traffic to satisfy the user and application service requirements and offer different levels of service quality as well. This is not available in the inherent type of service provided by the current Internet which is called "best-effort."

The need for QoS support in the Internet has been a topic for many debates. Opponents of QoS suggest that throwing more bandwidth (i.e., *over-provisioning*) is a simple and cost-effective solution for congestion, packet loss, and network delays. This is motivated by the decreasing cost of bandwidth. However, many real-time applications such as VoIP and Video-on-Demand (VoD), require different guarantees on delay, jitter and packet loss in

[1] As part of its original principles of simplicity and scalability.

addition to bandwidth.

Even with the existence of "fat" or high-bandwidth network links in today's Internet, there is still a capacity mismatch at the interconnection points between different physical network media, such as wired and wireless media, or between different network domains.[2] Significant packet delays could occur at these network points, affecting the overall end-to-end (e2e) delay performance. There is also the problem of heterogeneity in the traffic mix flowing through the Internet. At their merging points, packets of time-sensitive traffic get delayed behind other packets, and due to the non-preemptive forwarding at these points, traffic can experience varying and unpredictable amounts of delay and jitter [20]. Furthermore, the advent of advanced multimedia applications and the growing use of the Internet are expected to quickly consume the offered bandwidth, resulting in a bandwidth-limited environment as before. Another argument in favor of incorporating QoS comes from the disparity of available bandwidths in the core and edges of the Internet. The domains at the edges of the Internet tend to have more congestion than the core. Also, over the last few years, Internet dynamics have been found to lead to periods of severe congestion and resource shortage regardless of abundant resource availability under normal conditions. Therefore, QoS mechanisms are needed when network components experience their peak usage. For better network and resource utilization, simply adding more bandwidth comes as a last resort when persistent over-utilization is detected.

Throughout the thesis, the DiffServ architecture is used as a "use-case" in applying QoS over the Internet.

1.2 The Differentiated Services Architecture

The IntServ is one of the initial frameworks for supporting QoS on IP networks, especially the Internet. However, its service model, in spite of its flexibility and per-flow QoS provisioning, has not been successfully deployed in the public Internet, mainly because of the lack of scalability. In order to address the problems of IntServ, the IETF developed the Differentiated Services or DiffServ architecture [13] as a more scalable alternative.

[2] Due to technology and policy differences.

The DiffServ received considerable attention because it is considered a good match to the Internet architecture. DiffServ works in harmony with the Internet basic design principles [26] of simplicity, scalability, distribution of resource management and authority, aggregate control, and stateless packet-level functionality.[3]

The main differences between the DiffServ and its predecessor, IntServ, are as follows [129]. First, the DiffServ is based on resource provisioning rather than resource signaling and reservation as in IntServ. This in effect makes it difficult to achieve precise or deterministic guarantees, but on the other hand, it is simpler to deploy and more attractive to use. Second, the DiffServ handles traffic aggregates[4] instead of individual flows. As a result, on a per-flow basis, DiffServ provides *qualitative* instead of *quantitative* QoS guarantees provided by IntServ. Third, the DiffServ standards define forwarding treatments, not end-to-end services like the guaranteed and the controlled load services in IntServ. Finally, the emphasis in DiffServ is placed on Service Level Agreements (SLAs) between domains rather than end-to-end dynamic signaling as in IntServ.

Additionally, the Internet is made up of independently-administered entities or domains, often referred to as "clouds." Clouds may be under different authorities and may be administratively and technologically diverse. The DiffServ Working Group acknowledges this fact by only providing basic building blocks (such as Per-Hop Behaviors or PHBs [13], and Per-Domain Behaviors or PDBs [97]), rather than end-to-end services. With these basic building blocks, useful end-to-end or Internet-wide services can be constructed through agreements between neighboring clouds.

1.2.1 The Components of the DiffServ Architecture

The DiffServ, like any large framework, has several components. The following subsections describe the main components of the DiffServ architecture and their operation to provide QoS on the Internet.

[3] This is different from state-full flow-level IntServ.

[4] A traffic aggregate is a group of traffic flows handled in a similar way through a part or all of the network.

DS Field

Service differentiation requires marking of selected bits in the header of IP packets. These bits are called the "DS Field" in the DiffServ context. RFC 2474 [96] defines the DS Field coinciding with the ToS byte in the IP header. However, only six bits are used to carry the DS codepoint (DSCP), and the remaining two bits are not currently used.[5]

DSCP is the codepoint used to determine the forwarding behavior that a packet experiences in a typical DiffServ node. This forwarding behavior is called *Per-Hop Behavior* (PHB). DiffServ is based on defining a small number of PHBs implementing the necessary service differentiation at the participating routers, and marking the DSCP bits to assign incoming packets to one of these PHBs. Currently, the DiffServ WG has finalized three types of PHBs in addition to the default best-effort service. One of these PHBs, the Class Selector (CS) PHB, was defined to ensure backward compatibility with the IP precedence bits used in the ToS byte. The other two PHBs are defined as Expedited Forwarding (EF) and Assured Forwarding (AF). These two PHBs can be used to implement Premium and Assured services, respectively.

Class Selector (CS) PHB

CS PHB was defined in RFC 2474 [96] to keep backward compatibility with the IP precedence bits in the IP ToS byte. It can be used to create 8 different levels of priority with a larger value indicating a higher forwarding priority. CS-compliant PHBs can be realized by a variety of mechanisms, including strict priority queueing, WFQ [30, 72], CBQ [50], WRR [58], and their variants (RPS [122], HPFQA [9], DRR [117]).

Expedited Forwarding (EF) PHB

The EF PHB is designed to implement a service with low delay, jitter and loss in addition to an assured bandwidth, as specified in RFC 3246 [29]. The idea behind the EF PHB is to make packets marked with an EF DSCP encounter very small queues at the forwarding nodes. This is usually achieved by allocating forwarding resources with a higher rate than

[5] Recently, these two bits are used for Explicit Congestion Notification (ECN).

the arrival rate of EF packets. EF is used for services that have strict requirements on delay and jitter such as time-critical and multimedia applications. This type of service is usually referred to as *Virtual Leased Line* (VLL).

The IETF documents suggest use of priority queues to implement the EF PHB as well as other scheduling schemes such as CBQ and WRR. The latter may not result in an efficient implementation due to the nature of round-robin scheduling.

Assured Forwarding (AF) PHB

The AF PHB is used for building services with controlled loss and assured bandwidth. Such services do not have any delay or jitter guarantees. The IETF DiffServ WG defined a PHB group[6] for AF in [59]. The AF PHB group consists of three forwarding behaviors, AFx1, AFx2 and AFx3 in increasing order of drop precedence. This can also be interpreted as AFx1 being the most important and AFx3 being the least important. The notation 'x' represents the AF class. The WG recommends the use of four independent AF classes with three drop precedences per class. Packet reordering between AF classes is not allowed. The AF PHB defines two rates: a Committed Information Rate (CIR), which is the minimum bandwidth from the network to be assured, and a Peak Information Rate (PIR) for a rate above the CIR to accommodate bursts.

The AF PHB is usually implemented in terms of buffer-management schemes such as RED In/Out (RIO) [24] and Weighted RED (WRED) [80]. It is worth mentioning that the effect of the AF PHB becomes most prominent in case of network congestion when some packets have to be discarded.

Per-Domain Behavior (PDB)

PHBs are designed to be installed on individual nodes or routers. A group of packets that experience the same forwarding behavior at every node while traversing a domain is called a *Behavior Aggregate* (BA). The term BA later became synonymous with *Per-Domain Behavior* (PDB) — one of the basic building blocks of creating a DiffServ-enabled

[6] A PHB group is a set of one or more PHBs that can be meaningfully specified only if they are implemented simultaneously.

network. A PDB [97] is used to define a certain edge-to-edge service that has measurable network parameters as experienced by a set of packets with the same DSCP as they traverse a DiffServ domain.[7] Therefore, PDBs are used to construct services between ingress and egress points of a domain. The attributes of PDBs (throughput, drop rate, delay bound, etc.) are advertised as Service Level Specifications (SLSs) at the edges of the domain. They are usually listed as statistical bounds or percentiles and not as fixed values.

PDBs can be constructed by concatenating a set of PHBs[8] between the edges of a domain. Traffic conditioning (marking/remarking, shaping and policing) on all incoming packets is done so that the PDB can meet the service level for which it was designed. An example of PDBs proposed by the WG so far is the Virtual Wire (VW) PDB [69] based on the EF PHB, and is suitable for delay-sensitive applications. Other PDBs such as Assured Rate (AR) [114] and Bulk Handling (BH) [19] PDBs have also been proposed.

Note that there is no one-to-one relationship between PHBs and PDBs. This means that more than one PDB can be based on the same PHB. On the other hand, the inverse is not currently supported, i.e., a PDB can only be based on one PHB within a single domain. This means that a large number of PDBs can be constructed from a small number of PHBs, depending on many factors, such as PHB characteristics, available routes between each ingress–egress pair, and policy. An important consideration for the scalability of any PDB is that its attributes should be independent of the amount of traffic entering the domain or the path taken by this traffic inside the DS domain. Furthermore, the edge-to-edge attributes of a PDB should hold regardless of any splitting or merger of the traffic aggregates inside the domain.

With the basic building blocks defined, we now describe how DiffServ can be deployed in practice. Fig. 1.1 illustrates a typical architecture of a DiffServ network. A DiffServ-aware network consists of multiple DiffServ domains that can be viewed as Autonomous Systems (ASs). The boundary routers of each domain perform the necessary traffic conditioning at the edges, while the interior or core routers implement the necessary traffic forwarding treatments for different PHBs supported by the DS domain. Every DS do-

[7] By DS domain, we mean part of the network under a single administration and compliant with DiffServ standards.

[8] Usually, from the same type or group.

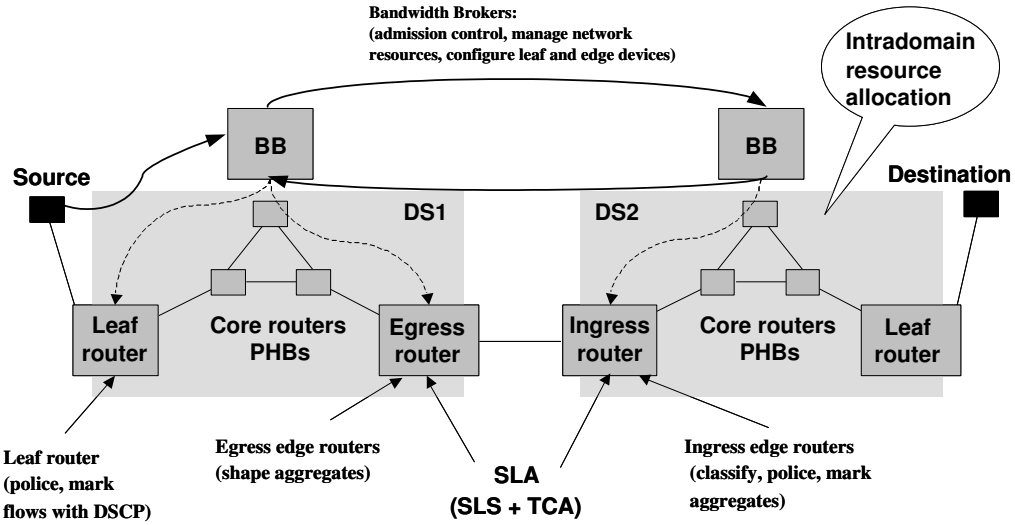


Figure 1.1: DiffServ architecture

main makes two agreements with each of its neighboring domains: a SLA specifying the services (in terms of SLSs) that this domain will provide, and a Traffic Conditioning Agreement (TCA) that incoming traffic to this domain will be subjected to. Adjacent domains negotiate SLAs among themselves and with customers accessing their network. Each DS domain configures and provisions its internal nodes such that these SLAs can be met. The interior or core routers implement the necessary traffic forwarding treatments for different PHBs supported by the DS domain. This distribution of configuration responsibilities adds to the flexibility of the DiffServ architecture. It is important to emphasize here again that, unlike IntServ, DiffServ employs "resource provisioning" with the help of SLAs and does not use "resource reservation" in setting up different services across domains.

From the above discussion, it is clear that the DiffServ architecture pushes the complexity of managing the network to the edges, and leaves packet handling and forwarding in the core of the network as simple and fast as possible. This is a major improvement in scalability of DiffServ over other schemes such as IntServ. Although dealing with aggregated traffic only reduces the flexibility of providing QoS guarantees to individual flows, it improves the overall scalability of the architecture.

1.3 Model-Based QoS Control: Motivations

Although modularity is one of the key principles behind DiffServ (and the Internet in general), it is still unclear how to use such feature to provide "well-defined" services to the Internet user. This lack of clarity, arises from both technical and business aspects. The technical aspect is concerned with how to form "well-defined" and "well-controlled" services using the basic DiffServ building blocks and principles, while the business-related aspect is concerned with the absence of a business model behind the DiffServ QoS support which is necessary for its success. In this thesis we explore new directions and efforts addressing the first aspect which we call "QoS control."

QoS control design can be characterized along two dimensions [48], *time* and *space*. The first dimension specifies the *time scale* at which the control mechanism operates. This can be at the packet, round-trip time, session levels, or even longer (such as hours, days, weeks, months, or even years). On the other hand, the *location of control* can be at the very edge of the network (traffic sources and customer sites) or at the very core (backbone routers and links). Accordingly, this can be interpreted into the *granularity of control*, which varies between per-flow and aggregate, forming the space dimension of QoS control. These two dimensions together define a broad design space from which QoS control architectures can be built, reflecting various tradeoffs in service performance, operations, management complexity and implementation cost. For example, control granularity has a direct impact on the operation and management complexity of a network data plane (e.g., to deal with per-packet, per-flow, or per-aggregate processing cost). On the other hand, time scale determines how frequently control information must be conveyed and control actions must be taken. These two dimensions also determine the scalability and deployability of the QoS control mechanism.

While considerable attention has been paid to building techniques and mechanisms for realizing QoS in packet-switched networks (e.g., packet schedulers, queueing disciplines, filters, as well as others), much less attention has been paid to how to globally control these techniques to provide predictable QoS. The traditional approach in providing network QoS is to calculate and then provision the required network resources (e.g., link bandwidth,

processing power at routers, buffer allocation, and traffic scheduling) *a priori* to ensure that the network can meet certain QoS requirements. This is usually referred to as a *feed-forward control* strategy, and it relies on *a priori* knowledge of workload and resource usage and it can work well for some cases [75]. However, as we mentioned before, the increased dynamics of the Internet suggests for QoS control mechanisms based on accurate models and a *feedback control* strategy that yield better resource utilization and faster response to network and traffic dynamics. According to this strategy, the output QoS is measured and fed back through a control algorithm (controller) to form the control actions to be applied to the network, thus forming a *feedback control loop* [44].

In order to use this feedback control strategy, a model for the system under control is required. Usually modeling is used to extract bounds on QoS performance metrics such as delay and jitter [10, 22] or in admission control as in [21]. Most of previous QoS work has relied on heuristic and ad-hoc approaches. We can find only a few efforts for using control theory in QoS problems such as in [3, 4, 82, 107], but most of them are for application QoS adaptation. However, the majority of modeling and control work has focused on congestion control problems in TCP [87, 103, 104] and Active Queue Management (AQM) [14, 89].

An important and difficult process for the success of applying the control-theoretic approach is the ability to find an accurate model that represents the system under control. By model here we mean a mathematical representation for the system that describes its behavior and also the relationships between its inputs and outputs. Models can be either stochastic or deterministic [71]. For example, in modeling network traffic, deterministic models are usually constructed using fluid-based analysis while stochastic models are constructed based on the Markovian process[9] like what is used in queueing theory [12]. Stochastic models are usually used for studying the long-term or static case where it gives only average estimates for the network parameters. On the other hand, deterministic models are usually preferable for transient analysis. Therefore, Floyd in [51] identified that "models should be specific to the research questions being investigated."

[9]Poisson-based random process.

1.4 Applying Feedback Control to Networking and QoS

Recently, there has been an increasing interest in applying fundamental and classical theories such as Systems and Feedback Control theories [53, 100] to analyze and solve networking and computing problems (see, for example, [63, 64, 71, 95]). These theories have been used for decades in many other engineering and scientific disciplines such as mechanical and aeronautical engineering, and their effectiveness in solving control problems in networking has been realized as well. Significant efforts have been made on flow control problems in either high-speed networks like ATM, or IP networks (specifically TCP). TCP congestion control is built based on a feedback from the network (or more accurately between the source and destination) to sense the condition of the network and adjusts the sending rate accordingly [67]. Also, Keshav [71] applied a control-theoretic approach in studying the general flow control problem. He considered network conversations that do not reserve any bandwidth in advance, and used the control approach to determine how these conversations can adapt their transfer rate to varying network conditions in order to satisfy throughput and queueing delay requirements. After that, a considerable amount of work has been directed toward ABR rate control in ATM (see for example [56, 74]). More recently, attentions were paid to TCP due to the predominance of TCP traffic and due to its complex reaction to network conditions. For example, [52, 105] presented congestion control and rate control mechanisms that are based on a validated TCP model derived in [104], and in [95] a study of fair bandwidth allocations in core-stateless networks is presented. On the other hand, and along with the efforts on TCP, the control-theoretic approach has been applied to Active Queue Management (AQM) as well. In [63, 89], the authors presented a fluid-based analysis for AQM supporting TCP flows and followed by a control design of two-color marking for throughput differentiation of TCP flows. RED has also been analyzed using control theory in [14, 64] as a well-known example of AQM.

On the server and software applications side, Abdelzaher *et al.* [3, 4] applied the control-theoretic approach to web servers to control the user-perceived utility under varying server workload. In [107], the authors used the control theory to design a controller for the Lotus Notes server, thus achieving service-level objectives in performance management.

They used a stochastic autoregressive modeling technique based on ARMA which proved to be an effective technique for modeling without knowing the details of the system under control. Steere *et al.* [121] developed a feedback-based CPU scheduler that synchronizes the progress of consumer and supplier processes. Finally, a middleware control framework was used in [82] to enhance the effectiveness of QoS adaptation decisions of distributed multimedia applications. The objective was to meet both system-wide properties like fairness and application-specific requirements like critical performance needs.

1.5 Research Goals

The lack of a formal theory for QoS modeling and control has hindered qualitative understanding of the dynamics and quantitative assurance of performance. The resemblance of network QoS problems to other physical (such as mechanical and fluid) systems problems motivated us to investigate the use of fundamental theories such as the systems theory and control theory in studying network QoS control.

In this thesis, we develop new QoS control mechanisms that provide more dynamic, and at the same time, reliable solutions for today's Internet without the usual complexity of the traditional QoS frameworks. Our mechanisms feature the use of model-based feedback control to achieve the required QoS in the network. We present the use of versatile and rigorous techniques such as statistical modeling [70, 93] and system identification [84] in modeling network QoS elements, like per-hop and edge-to-edge QoS. Then, we show the use of model-based feedback and control-theoretic approaches in designing control algorithms for these elements that optimize resources for conflicting QoS requirements. Collectively, we will be able to look at network QoS in the same way we look at a mechanical system or fluid mechanics. Figure 1.2 illustrates our goal of modeling and control network services. Generally speaking, applying a control-theoretic approach to a system requires the following three steps:

System identification: to construct a *transfer function* which relates the inputs to the outputs of the system. This transfer function constitutes a model of the system. In order to use classical linear control theory, linearization techniques may be applied

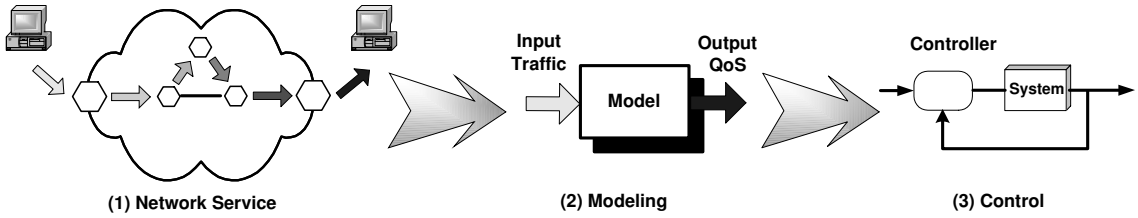


Figure 1.2: Modeling and control of network QoS

to non-linear systems to obtain a linear transfer function, and simplify the analysis and controller design.

Controller design: based on the properties of the transfer function and the desired objectives, a particular *controller* is designed. Control theory is used to predict how the system will behave once the chosen controller is added to it.

Control action: in distributed systems, such as computer networks, it is important to specify the location and mechanism of control decisions to be applied. In the terminology of classical control theory, this action is usually taken by an entity called an *actuator*. We will identify the control action in each part of our study.

1.6 Thesis Contributions

The model-based control approach can be applied at many places in the network ranging from hosts (or servers) running applications all the way down to the network protocol layers as well as the network components such as routers, switches, and network links. This involves traffic management, traffic conditioning, traffic scheduling, buffer management, routing, resource allocation, admission control, as well as other numerous network functions. Figure 1.3 illustrates our contributions in modeling and controlling different network QoS components that are explained in the following subsections. We start by studying differentiated packet marking control that is used at the host or gateway level. Then, we present modeling and control of QoS-enabled network routers (per-hop QoS), and finally, we study domain-level network services or edge-to-edge QoS across network clouds. We

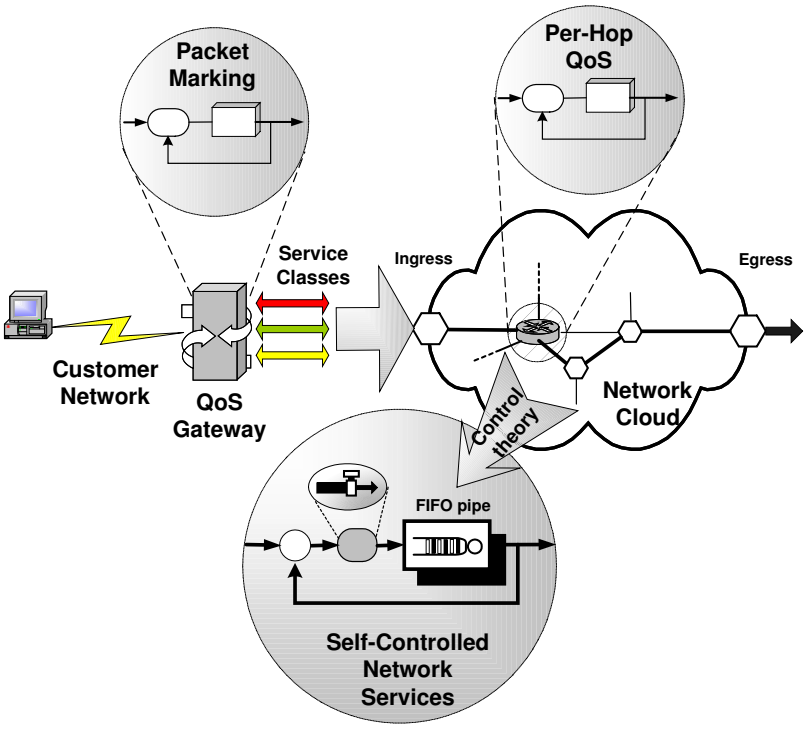


Figure 1.3: Contributions

also present an architecture, called *QoS Gateway*, for promoting QoS support to hosts and applications.

1.6.1 Packet Marking Control

The first piece of work in the thesis focuses on the mechanism that creates different traffic classes in the network, which is called "packet marking." Packet marking is considered as a basic mechanism of creating aggregate traffic classes to meet different QoS requirements. It is used in most recent QoS architectures such as DiffServ, MPLS, IEEE 802 QoS, as well as others. Usually packets from different traffic or service classes are marked with different tags to identify their relative importance. Techniques for QoS along the network differentiate handling of packets according to their tags. In this way, per-flow classification inside the network (which suffers the scalability problem) is avoided.

To show the effectiveness of applying the model-based control approach to network QoS components, we start by using a recently-developed model at the University of Mas-

sachusetts for TCP [104] in creating a better marking algorithm. We show how to use this model for controlling the output throughput, as one of the QoS parameters, of the network traffic. Specifically, we consider the edge of the network and study the problem of packet marking within the context of the *Assured Forwarding* (AF) services in DiffServ. We introduce a new packet marking algorithm called *Equation-Based Marking*, or EBM [37, 38]. It uses the TCP model in [104] in a tight feedback control loop that controls the output throughput. EBM handles the problems found in other marking schemes regarding fairness among heterogeneous TCP flows through adaptation and control of the packet color marking probabilities. Although we do not use a control-theoretic approach in designing the controller for this problem, we show how an iterative model-based control approach can result in a better and predictable QoS performance than other ad-hoc approaches. The *control action* used here is "differentiated packet marking." We design a packet marker that uses EBM as the marking algorithm, and evaluate its performance using in-depth simulation. We also prove analytically the correctness of the marking algorithm and compare it with other marking schemes for different network scenarios. Our evaluation results demonstrate the effectiveness of EBM in controlling the output throughput of the assured traffic and in providing the required fairness among heterogeneous flows as well as ensuring protection against non-assured traffic.

1.6.2 Per-Hop QoS Modeling and Control

This time we apply control inside the network, and extract the models from scratch. We take a typical QoS-enabled network node (specifically a PHB-enabled node) in isolation and find the functions that relate the inputs to the outputs of this node. We call the output of the QoS-enabled node as the "Per-Hop QoS" or PHQ [35, 38], which is measured in terms of throughput, delay, jitter, and loss rate experienced by traffic crossing this PHB. We use a statistical approach that is based on experiments on a real network testbed to characterize the per-hop QoS of a given PHB. Specifically, we employ a full factorial statistical "design of experiments" to study the effects of different PHB configurations and input traffic scenarios on per-hop QoS. We use Analysis of Variance (ANOVA) to identify the input

and PHB configuration parameters that have the most significant influence on per-hop QoS. Then, a polynomial regression analysis is applied to construct models for the per-hop QoS with respect to these parameters. The overall approach is shown to be effective and capable of characterizing any given PHB, within the ranges of the experiments, and for construction of functional relationships for the PHB output parameters. We are also able to identify the operational differences between different realizations of a given PHB. These functional relationships are considered as off-line (static) models for the PHB. Once these models are found, we study how to use them in achieving a required PHQ within a simple feedback controller following our "model-based control" approach.

1.6.3 CONNET: Self-Controlled Network Services

Despite the attractive features of the modeling framework used in PHQ, it does not cover all possible performance ranges or configurations for a particular PHB. Instead, it predicts the PHB performance under the levels of factors used in the experiments which represent only a certain range of the PHB operation. We would like to think of these "off-line derived" models as *static* models that capture only a snapshot of the per-hop QoS, and in order to be able to apply the control theory we need to find the dynamic version of these models. PHQ control proves that model-based control is working, but we need to use more reliable feedback control.

To meet this need, we develop a new *reservation-less* mechanism for delay and jitter guarantees across "single FIFO queue" network services by controlling interfering non-premium traffic. The mechanism, called *CONNET* (CONtrolled NETwork) [32], capitalizes on the dependence of the delay/jitter of premium traffic on the amount of non-premium traffic sharing the same network service. Specifically, it uses feedback regulation to create *self-controlled* network services capable of tracking delay and jitter references specified either online or off-line with minimal management overhead. The difference between the measured delay/jitter and the reference value is fed back to control the rate of non-premium traffic entering the network service. We use a control-theoretic approach in designing the feedback loop based on a "black-box" model of the access link in both cases of single

and multiple nodes. For robust control, we use an optimal Linear Quadratic Gaussian (LQG) regulator that achieves a good balance between the system response and the control effort required. We evaluate the performance of the CONNET algorithm using a real testbed implementation, and illustrate ways of deploying it on the Internet. This evaluation demonstrates CONNET's superiority to other rate-based schemes (such as CBQ and WFQ) in terms of simplicity, deployability, robustness, accuracy, and fault-tolerance. We also present the use of the same control approach in Delay-controlled Queue Management (DQM) [31] for delay-sensitive traffic.

1.6.4 Paving the First Mile for QoS-dependent Applications and Appliances

To realize the full potential of the proposed QoS control mechanisms in creating a QoS-aware environment, applications should be able to incorporate them. Therefore, paving the first mile of QoS support becomes an essential step for full deployment and utilization of QoS techniques that are introduced in this thesis and even to existing techniques. Usually, the efforts have been focused on providing network services and control without paying much attention to how applications can use these services. Until recently, a few operating systems have provided APIs for new applications to access network QoS. The main problem with this scheme is that it provides support only for those applications that are programmed to use these interfaces. For legacy applications that are not aware of these interfaces, they will not be able to use them.

To address these issues, we propose the design and implementation of a two-tier architecture, called *QoS Gateway* or QoSGW [33], that can support both types of applications in addition to small embedded network devices that need to use network-provided QoS support. Our system, once fully integrated, is composed of two main components: (i) an agent that either resides on the end-host or as a stand-alone entity, and it provides adequate interface to QoS-dependent applications, and (ii) a QoS manager for network services that provides interfaces to these agents. Using two components achieves both generality and scalability in providing QoS support for applications and end devices. The agent receives

the application requirements either directly or indirectly, map them to an intermediate form, and consults the manager to map these requirements to an appropriate network service. The manager, on the other hand, serves a number of agents in providing adequate network services from the underlying network in addition to other management responsibilities. This system promotes QoS deployment and facilitates the building of QoS-aware customer networks. For example, EBM is implemented within the marking module of the manager. The manager can also use the self-controlled network services mentioned above to build edge-to-edge or end-to-end QoS for applications with pre-specified QoS.

1.7 Organization of the Thesis

The thesis is organized as follows. Chapter 2 introduces the Equation-Based Marking (EBM), which shows the effectiveness of using model-based control for predictive QoS as a first example. Chapter 3 presents the modeling and control of per-hop QoS for typical QoS-enabled nodes using statistical methods. The chapter covers the modeling framework, validation of the models and the model-based control algorithm. CONNET is covered in Chapter 4, where we present the motivations behind using the control-theoretic approach and a detailed analysis of the feedback control design. It also presents a thorough experimental evaluation to show the effectiveness of CONNET. Chapter 5 presents our approach for paving the first mile of QoS to QoS-depended applications and appliances. It describes the architectural design of the QoS Gateway, points to its operation, and finally presents the evaluation of a prototype of the design to show its correctness and effectiveness. Finally, Appendices A, B, and C include the TCP model used in EBM, a brief review of the statistical tools used in PHQ modeling and control, and an excerpt of the prototype implementation of the QoS Gateway.

Note that our analysis techniques are tailored throughout the thesis to best suit the problem at hand. This allows us to maximize the effectiveness of each proposed technique in our study of QoS modeling and control, and to gain different experiences during the course of the thesis research. In particular, we use simulation and simple mathematical proofs in the packet marking study. Then, we use a full experimental approach (based

on a real network testbed and implementation) along with statistical analysis in studying the per-hop QoS which gives hands-on experience. Finally, we use MATLAB design and simulation tools in designing CONNET followed by an evaluation using a Linux-based testbed network.

CHAPTER 2

Equation-Based Packet Marking for Throughput Guarantees

Packets entering a DiffServ-enabled network are marked with different DiffServ Code Points (DSCPs). Based on this marking, packets are subject to classification, traffic conditioning (such as metering, shaping, and policing), as well as to a certain PHB. As mentioned before, one of the defined PHBs, the Assured Forwarding (AF) PHB [59], is proposed to offer different levels of forwarding assurance and throughput guarantees.

The idea behind the DiffServ's AF PHB in controlling throughput is to identify "less important" packets and let the network buffer management drop them upon detection of congestion to protect "more important" packets. Packets are identified by their marking, based on conformance to their target throughput. Non-conformant packets are called *out-of-profile* (OUT), while conformant packets are called *in-profile* (IN). Then, by using a differentiated random drop gateway like RIO [24] or a more general form thereof, at the time of congestion, OUT packets are more likely to be dropped than IN packets. This, in effect, protects IN packets from OUT ones, giving applications their required bandwidths. For three-color AF, IN packets are labeled *Green*, and OUT packets are divided into *Yellow* and *Red* in order to have more control on the available bandwidth of the network. These three colors correspond to the AF's three drop precedences, AFx1, AFx2, and AFx3, respectively. Associated with any marking algorithm, there is one or more values of "marking probabilities." Each probability determines the ratio of the traffic to be marked with a cer-

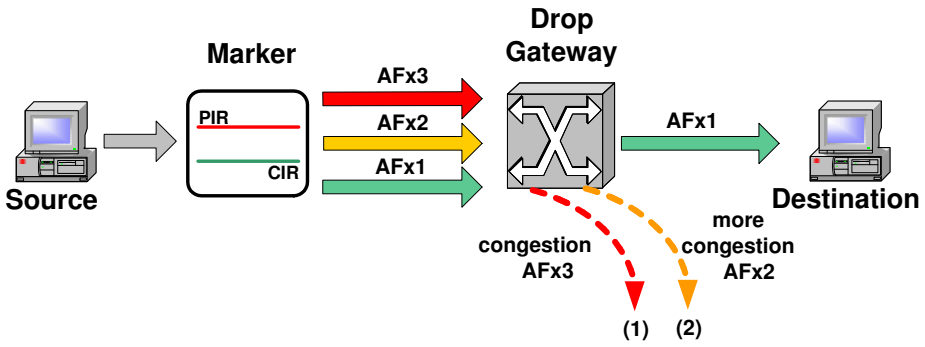


Figure 2.1: AF PHB marking process

tain color. They all add to one, which is the total traffic sending rate. Figure 2.1 shows this idea of marking for the AF PHB.

2.1 Fairness in Bandwidth Distribution

AF works best with TCP because of responsiveness to packet drop. The interaction between the packet drop rate (also known as loss rate), and TCP effective throughput is a complex process and is known as "congestion control." Recently, significant efforts have been invested into the performance evaluation of TCP congestion control for a random drop gateway like RED [49] and RIO [24] with an emphasis on AF services. The authors of [85, 115] pointed out unfairness issues among TCP aggregates that have different round-trip times (RTTs), average packet sizes, target rates, or numbers of micro-flows in the aggregate. This unfairness occurs in sharing the extra bandwidth in under-subscribed networks, or in the degraded performance in over-subscribed networks. They considered different models for the drop gateway configuration, such as RIO-C, WRED, overlapped, non-overlapped, single and multiple averages. Similar results have been reported in [109] and [65], the latter of which has also evaluated the difference between using token bucket and average rate estimator policing (marking). Fairness between responsive (e.g., TCP) and non-responsive (e.g., UDP) traffic was also investigated in [115, 116].

Fairness has also been an important problem in the performance of AF services[1] within

[1]Because of AF's dependency on congestion control.

the DiffServ framework [13, 98]. We define the fairness term as follows: *"In an under-subscribed network, all flows should get a share of the excess bandwidth proportional to their target rates (an equal share for equal target rates). In an over-subscribed network, all flows should experience throughput degradation proportional to their target rates (equal degradation for equal target rates)."*

Several packet marking algorithms were proposed to work with AF, such as the *Two Rate Three-Color Marker* (TCM) [60], and the *Time Sliding Window Three Color Marker* (TSWTCM) [39, 77]. TCM is based on token bucket metering, while TSWTCM is based on the average rate estimator algorithm in [24]. An enhanced version of the TSW, called *Enhanced Time Sliding Window* (ETSW), was proposed in [83] to handle TCP dynamics and over-marking in the original TSW marker. We found that these marking algorithms do not handle the unfairness issues mentioned above, nor have they been evaluated with respect to these issues in the first place. For most of these marking schemes, two target rates are defined: *Committed Information Rate* (CIR) and *Peak Information Rate* (PIR). The former is the minimum requirement to be achieved, and the latter is for a surplus of bandwidth when the network is lightly-loaded.

The authors in [85, 115] found that all these marking algorithms suffer from the same afromentioned fairness issues. To emphasize, TCP flows with different RTTs cannot achieve throughput proportional to their target rates, and the same phenomenon occurs also for TCP flows with different target rates, different mean packet sizes, or aggregates with different numbers of micro-flows. This was detected in our extensive simulations as well, and it is not acceptable in the operation of AF. Also Non-responsive flows usually get more bandwidth than their fair (proportional) share, leaving responsive flows with less than their fair share, and sometimes starve for bandwidth (because of greedy non-responsive flows).

Several remedies have been proposed to overcome these fairness problems, such as the *TCP-Friendly* marker [43] and the *Fair Marker* in [73], but they suffer from their inherent complexity, and their performance has not been evaluated either. Two adaptive marking algorithms have been proposed in [41] and [92]. They are called *Adaptive Packet Marking* (APM) and *Intelligent Traffic Conditioners* (rtt-aware and target-aware), respectively. APM is found to perform well in tracking the dynamics of TCP and preserving the target rate, but

it is based on an inaccurate feedback model which causes performance fluctuations. Moreover, APM has to be implemented inside the TCP code itself, requiring modification of all TCP agents to be able to use this marking algorithm. The Intelligent Traffic Conditioner tries to use the simple TCP model in [87] to handle the unfairness associated with different RTTs and different target rates. This is somewhat similar to the approach proposed in this chapter, except that these conditioners require external inputs and cooperation among markers for different traffic aggregates, which tend to be very complex in implementation and deployment. Moreover, several researchers attempted to mimic TCP congestion control using models in [87, 104, 133] and introduced model-based rate control [105], and equation-based congestion control [52]. On the other hand, an excellent analytical study of the achievable performance for TCP using token bucket marking is reported in [112] where the authors prove that token bucket markers cannot achieve all values of assured rates and the achieved rate is not proportional to the assured rate. They also introduced a method of choosing the correct profile to achieve a given service level.

In this chapter we address these issues in the AF PHB performance by introducing a new marking algorithm that draws from the model-based control approach we described in Chapter 1. Our marking algorithm is called *Equation-Based Marking* or EBM, and it is based on a model (or equation) for the TCP congestion control process that was developed in [104], hence the name "Equation-Based." Next, we describe the design, operation, analysis and evaluation of EBM and compare it with the previously-mentioned marking algorithms showing its superiority in achieving required throughput guarantees and solving the fairness issues of those algorithms.

2.2 Equation-Based Packet Marking

Our solution for the fairness problem in AF is two-part: The first part is EBM, which works similar to TCP, but on the packet-marking level. It senses the current network conditions and adapts the packet marking probabilities (IN and OUT for two-color marking; Green, Yellow, and Red for three-color marking) accordingly. This adaptation adjusts the loss rates of heterogeneous TCP flows, thus providing an equal or proportional share of

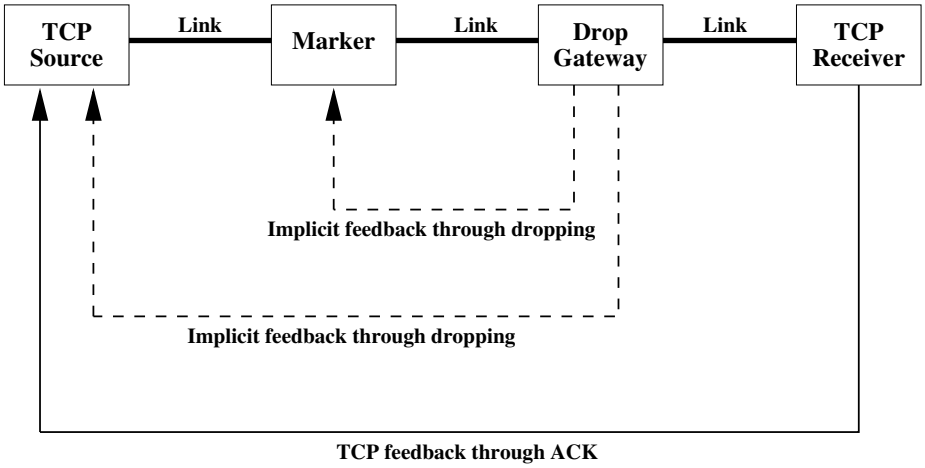


Figure 2.2: Feedback loop operation of EBM

extra bandwidth in under-subscribed networks, and an equal or proportional degree of throughput degradation in over-subscribed networks. The new marking algorithm predicts the behavior of the TCP sender and adjusts the marking probabilities accordingly, distinguishing itself from the other approaches to AF marking. This behavior gives a close interaction between the marker and the TCP sender.

The second part is called "Packet Separation" mechanism and it handles the fairness between responsive and non-responsive traffic. It is mainly a mechanism for isolating different traffic types at routers or nodes that are performing buffer management and packet dropping.

EBM employs a compact feedback control loop based on the TCP model mentioned before and summarized in Appendix A for completeness. The model encodes all the previously-mentioned factors affecting performance, in a single equation, Eq. (A.1). EBM is different from other marking algorithms because it is based on TCP's reaction to packet losses as we will show shortly. The marker works in the feedback loop shown in Figure 2.2 by sensing the losses that the TCP connection experiences, hence it tries to estimate the current network conditions and adjusts the packet marking probabilities accordingly. EBM works just like TCP, which adjusts the sending rate by sensing the level of congestion in the network via observation of packet losses. EBM forms another feedback loop, as shown in Figure 2.2, which provides appropriate marking to achieve the required target rate. This,

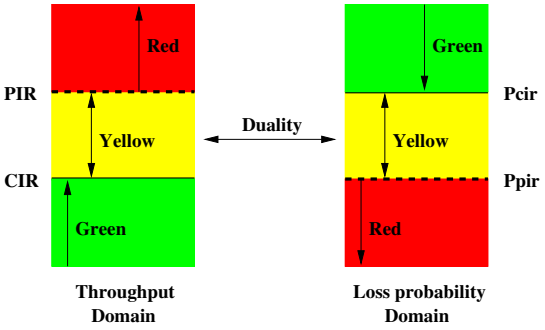


Figure 2.3: Duality between throughput and loss probability

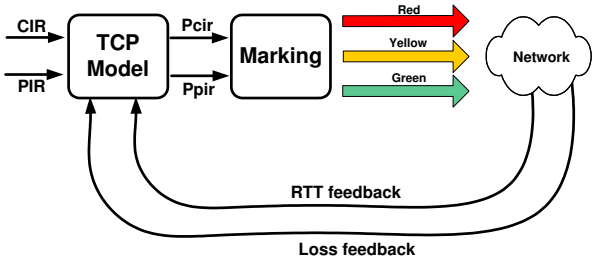


Figure 2.4: Operation of EBM

in effect, provides fairness among different TCP flows.

One basic difference between EBM and the previous marking algorithms, is that EBM uses estimated loss rate, instead of estimated average throughput, in calculating the required marking probabilities. This was drawn from the duality between throughput and loss probability when using the TCP model in Eq. (A.1), as shown in Figure 2.3. In this figure, we identify the target loss probabilities, p_{cir} and p_{pir}, corresponding to target throughput rates, CIR and PIR, respectively. EBM uses these two probabilities in its operation, and calculates them from the TCP model in Eq. (A.1) using the current network conditions like RTT, mean packet size, and maximum TCP window size. Then, as described later, it uses the current loss rate seen by this TCP flow as well as these target loss probabilities to calculate the packet-marking probabilities.

2.3 The Design of EBM

The operation of EBM is shown in Figure 2.4, where the marking engine uses the current estimated RTT (Section 2.3.2) as well as the current estimated loss rate (Section 2.3.3) in its marking decision. It plugs the estimated RTT into the inverse of the TCP equation, $T_p^{-1}(r_t, M, RTT, T_o, W_{max})$[2], to get the target loss probabilities from the target throughput rates. This calculation is the key, as it adapts to the current values of RTT, packet size, and the other derived parameters, reflecting different values for different TCP flows/aggregates.

[2]We call T_p^{-1} the inverse of T w.r.t. p.

```
Each periodic interval:

1.  Use current estimate of round-trip time and time-out,
      RTT and T_o, respectively
2.  Use value of the maximum TCP window size, W_max
3.  Calculate maximum achievable throughput, p = 0,
      r_max = T(0, M, RTT, T_o, W_max)
4.  if (CIR > r_max)
5.       Exit with error ``Not able to achieve CIR and PIR''
6.       if (PIR > r_max)
7.            Warning ``Not able to achieve PIR''
8.       end if
9.  end if
10.  Calculate p_cir = T_p^-1(CIR, M, RTT, T_o, W_max)
         and p_pir = T_p^-1(PIR, M, RTT, T_o, W_max)
11.  Use current estimate of loss rate, loss_rate
12.  Calculate packet marking probabilities P_yellow and P_red
13.  Mark packets
```

Figure 2.5: EBM algorithm

The target loss probabilities are used through a special marking function to get the packet marking probabilities for the three colors: Green, Yellow, and Red. Figure 2.5 illustrates the detailed algorithm of EBM. The steps in the algorithm are executed periodically, and the execution period is tunable to achieve the best performance. In the following sections we detail each of the building blocks of the EBM and the steps of the algorithm.

2.3.1 Calculation of the target loss probability

Eq. (A.1) indicates a one-to-one relationship between throughput, r_t, and loss probability, p, for a given maximum window size, RTT, and mean packet size. EBM uses this relationship to derive the loss probabilities corresponding to the target throughput rates, CIR and PIR. As mentioned earlier, these loss probability values are called *target loss probabilities*, $p_{cir} = T_p^{-1}(CIR, M, RTT, T_o, W_{max})$ and $p_{pir} = T_p^{-1}(PIR, M, RTT, T_o, W_{max})$. This calculation requires the inverse of the TCP equation with respect to loss probability, p, however, there is no closed form for this inverse. So, EBM numerically calculates the required values. The calculation process is tuned by changing the rate and/or accuracy of calculation.

```
At each estimation interval:

Record and timestamp a packet of the TCP flow
Upon receiving an acknowledgment
/*Check if it is for the previously-recorded packet*/
if (seqno of the ack ≥ recorded seqno)
      current rtt := now − recorded timestamp
      /*Calculate the WMA of the measured RTT values*/
      srtt = w × rtt + (1 − w) × srtt
      delta = |rtt − srtt|
      rtt_var = w_rto × delta + (1 − w_rto) × rtt_var
      T_o = srtt + 4 × rtt_var
end if
```

Figure 2.6: Estimation of RTT

One important step to be taken by the EBM algorithm is to ensure that the numerical iterations converge to a correct value; otherwise, there is no need to iterate in the first place. This is the role of Steps 3-9, where the algorithm calculates the maximum achievable TCP throughput, r_{max}, with the current values of RTT, T_o, and W_{max} by putting $p = 0$[3] when evaluating T. Then, it compares the target throughput rates, CIR and PIR, to this maximum value, and if it is smaller than any of the target rates, then this target rate can not be achieved under the current circumstances. Accordingly, the numerical iteration will not converge to a correct value. The proof of this convergence criterion is given in Section 2.4.3.

2.3.2 Estimation of RTT and T_o

This module uses a method similar to the one used in TCP's estimation of RTT with two differences. First, the estimation procedure uses the timestamp option in the TCP header in order to estimate RTT and T_o with a high resolution. Second, the module is located outside the sender TCP, so it reads and modifies the packets' TCP headers in order to use the timestamp option. The estimation procedure is depicted in Figure 2.6, where *srtt* is the estimated RTT and T_o the estimated retransmit time-out.

[3] In fact, there is no value for T at $p = 0$, but we use a very small value like 10^{-15}, instead.

2.3.3 Estimation of the current loss rate

Using the "average loss interval" method [52], EBM estimates the current loss rate of the network seen by a TCP flow. For convenience, we describe the method here and give the details of detecting loss events[4] from the marker side, which is located outside the TCP source. The loss interval, s_i, is defined as the number of packets transmitted correctly between two loss events $(i,\ i-1)$. The estimated loss interval $\hat{s}_{(1,n)}$ is calculated as the weighted average of the last n intervals:

$$\hat{s}_{(1,n)} = \frac{\sum_{i=1}^{n} w_i s_i}{\sum_{i=1}^{n} w_i}$$

for weights w_i:

$$w_i = \begin{cases} 1 & 1 \leq i \leq n/2 \\ 1 - \frac{i-n/2}{n/2+1} & n/2 < i \leq n \end{cases}$$

For the reasons mentioned in [52], EBM uses $n = 8$, giving weights of 1, 1, 1, 1, 0.8, 0.6, 0.4 and 0.2 for w_1 through w_8, respectively. The reported loss rate is $loss\_rate = 1/\hat{s}$.

Detecting loss events from the marker side and outside the TCP sender requires some knowledge of packet losses. In TCP, a packet loss is detected when the sender receives three duplicate ACKs, or when the retransmit time-out expires, whichever occurs first. In both cases, and according to the TCP Reno's fast retransmit procedure [68], the lost packet is retransmitted. Detecting three duplicate ACKs at the marker is not a problem when using similar state variables to the ones used in TCP, however, detecting packet loss that causes a time-out is harder, especially when a time-out timer is not used. Also, we ignore any loss within one RTT from a previous one, as mentioned in [52]. After taking several tuning procedures, a reasonable behavior of the loss estimator has been reached.

2.3.4 Marking Function

Based on the target loss probabilities, EBM uses a linear function to calculate the marking probabilities, P_{yellow} and P_{red} for the three-color marking case, proportional to p_{cir}

[4] A loss event is different from packet loss, as the former may consist of several packet losses within a round-trip time.

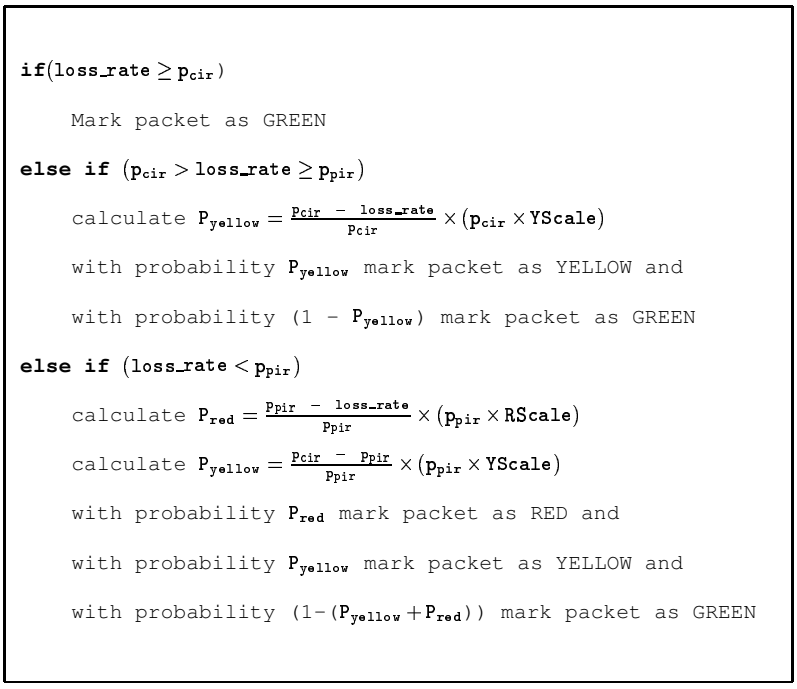


Figure 2.7: Marking function

and p_{pir} as shown in Figure 2.7. This marking function is similar to the one used in the TSWTCM marking scheme, but with throughput replaced by loss rate, and CIR and PIR replaced by p_{cir} and p_{pir}, respectively. The *YScale* and *RScale* are design parameters that take values which depend on the current network conditions.

To show how the marking function works, we present the case of four similar TCP flows but with different RTTs. We plot their p_{cir} and p_{pir} in Figure 2.8 and the resulting marking probabilities for the three-color case in Figure 2.9. We identify here that the marking function tries to adjust the packet marking probabilities to equalize the loss rates for the four different flows, and hence equalizing throughput (as there is a one-to-one relationship between throughput and loss probability). The flow with a higher percentage of Yellow, and Red will have more losses, and hence less throughput. One should see the same behavior for the other factors, or for a combination thereof as well. This validates the operation of EBM in providing the required fairness among heterogeneous TCP flows or aggregates.

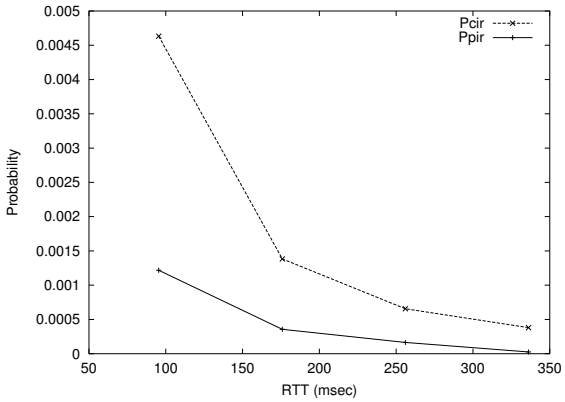


Figure 2.8: Pcir and Ppir

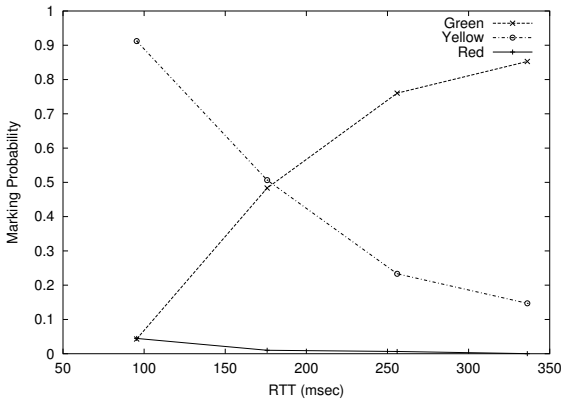


Figure 2.9: Marking probabilities

2.4 Analysis of EBM Operation

In this section, we analyze the operation of EBM to show the correctness of the operation and examine the convergence of the numerical iterations taken to calculate the target loss probabilities. Here we choose a steady-state analysis, and we will consider the dynamics of EBM in our future work. First, we present a correctness proof based on the feedback model of the marker with both TCP and the drop gateway. The approach to this proof has been inspired by the study in [47]. We present the proof here for the case of flows with different RTTs only. Using a similar procedure, the other cases of different target rates or different packet sizes can also be proved.

2.4.1 Assumptions

In order to make the proof tractable and simplify the mathematics involved, we make the following assumptions.

A1. We consider a system of n TCP flows passing through a common link ℓ with capacity C as shown in Figure 2.12, except for using a single-hop network. In Section 2.5, we use a multi-hop network as a generalization, and the results are valid for that case as well. All other access links have enough capacity, so that link ℓ is the only bottleneck for all flows.

A2. All TCP flows have the same parameters except for RTTs, i.e., each flow sees a dif-

ferent RTT. We assume that the system is in the steady state (no slow start), and all TCP flows have unlimited data to send. We also assume that the system is under-subscribed, i.e., there is a surplus of bandwidth that can be allocated to each flow.

A3. Without loss of generality, we will use a two-color version of the RED drop gateway, called RIO [24], to make the proof simpler and tractable, but the results work for the three-color case as stated in Section 2.5.

A4. Unlike the study in [47], we do not use the average queue size model of the RIO. Instead, we assume that a typical RIO drop gateway gives a fixed loss probability per class (IN and OUT) for all flows passing through this gateway. So, all flows see the same value of loss probabilities, $loss_{in}$, for IN packets, and $loss_{out}$ for OUT packets. Note that different flows may still see different total losses, depending on their parameters, but we only assume that loss probabilities in a class are the same for all flows. A justification for this assumption is given in the following:

In RED, and similar Active Queue Management (AQM) schemes like RIO, the drop rate is a linear function of the average queue size. So, for example, for a two-color drop gateway based on RIO, the drop probabilities (or drop rates) are given as follows. For the IN packets:

$$loss_{in} = \begin{cases} 0, & 0 \leq \bar{q}_{in} < min_{th_{in}} \\ \frac{\bar{q}_{in} - min_{th_{in}}}{max_{th_{in}} - min_{th_{in}}} p_{max_{in}}, & min_{th_{in}} \leq \bar{q}_{in} < max_{th_{in}} \\ 1, & max_{th_{in}} \leq \bar{q}_{in} \leq B \end{cases} \tag{2.1}$$

where:

B: is the maximum queue size or buffer size.

$\bar{q}_{in}$: is the average queue size for the IN packets calculated as WMA of the instantaneous queue samples.

$p_{max_{in}}, min_{th_{in}}, max_{th_{in}}$: are the RIO parameters for IN packets.

A similar formula exists for OUT packets obtained simply by replacing every *in* with *out*. The loss probabilities, $loss_{in}$ and $loss_{out}$, will be in the range $(0, p_{max_{in}})$, and $(0, p_{max_{out}})$, respectively. It is important to note that the calculation does not depend on per-flow information, so packets from different flows will see the same instantaneous values of loss probabilities for IN and OUT packets as functions of the same average queue size. However, total losses for IN and OUT packets for each flow depends on the number of IN and OUT packets in the flow's packet stream, which is directly related to the marking technique used. The values of $loss_{in}$ and $loss_{out}$ are proportional to the average traffic load on this gateway. If the load is high, the loss probabilities will be high; and if the load is low, the loss probabilities will be low, but will still be confined in range $(0, p_{max_{in/out}})$.

As mentioned before, the TCP model in Appendix A does not have a closed-form inverse with respect to loss probability p; neither does the approximate model given in [47], so we use numerical methods for this proof and discretize the problem as explained next.

2.4.2 Proof of Correctness

Theorem 1 (Fairness). *For a set of TCP flows, f_i, $1 \leq i \leq n$, with same parameters and same target rate,* CIR*, but with different RTTs, RTT_i, when EBM is used for AF marking, they will get approximately the same goodputs, r_{t_i}. In other words, for all $i \neq j$, $1 \leq i, j \leq n$, if $RTT_i \neq RTT_j$, then using EBM will result in $r_{t_i} = r_{t_j}$ $\forall$ i and j.*

Proof. Listed below are the steps taken for the proof.

1. For each value of RTT_i, calculate the TCP throughput $r_{t_i}(L_i, RTT_i)$ from Eq. (A.1), using the loss probability, L_i, seen by flow f_i.

2. The value of L_i is calculated from the RIO loss probabilities for each flow, depending on its packet marking probability, P_{m_i}.

$$L_i = P_{m_i} \times loss_{out} + (1 - P_{m_i}) \times loss_{in} \tag{2.2}$$

3. The value of the marking probability, P_{m_i}, is calculated for each flow using a similar marking function to the one in Figure 2.7 but for two colors only (IN and OUT).

$$P_{m_i} = \frac{p_{cir_i} - L_i}{p_{cir_i}} \times (p_{cir_i} \times Scale) \tag{2.3}$$

4. The value of p_{cir_i} is calculated numerically for each flow using its own RTT_i from the inverse of Eq. (A.1) with respect to p as:

$$p_{cir_i} = T_p^{-1}(CIR, RTT_i) \tag{2.4}$$

5. Solving for L_i using Eqs. (2.2) and (2.3), we get:

$$L_i = \frac{p_{cir_i} \times (loss_{out} - loss_{in}) \times Scale + loss_{in}}{1 + (loss_{out} - loss_{in}) \times Scale} \tag{2.5}$$

Then, substituting in Step 1 for each value of RTT_i, we get the final TCP throughput, r_{t_i}, for each flow f_i. Next, by plotting the values of r_{t_i}, we numerically show that they are equal for the same target rate.

We show one instance of the results using these steps is in Figure 2.10 for $M = 576$ bytes, $W_{max} = 256$, $Scale = 5000$, $C = 15$ Mbps, $CIR = 0.5$ Mbps, $n = 20$, $loss_{in} = 0.000066$, $loss_{out} = 0.00066$, and RTT is uniformly-distributed from 200ms to 900ms. The figure also shows the throughput values without using EBM, as well as the CIR value. The figure clearly shows the correct operation of EBM with respect to the required fairness among TCP flows with different RTTs. □

2.4.3 Convergence of EBM Iterations

As mentioned in Section 2.3.1, Eq. (A.1) describes a continuous and one-to-one relationship between r_t and p as shown in Figure 2.11, where T is plotted as a function of p for different RTTs ranging from 0.07s to 0.7s, $M = 576$ bytes, and $W_{max} = 256$. We use the bisection method for finding the inverse of T w.r.t. p numerically [7], and to iterate for the values of p_{cir} and p_{pir}, we start from $p = 0$ and $p = 1$. These two values bound the whole scale of possible loss probability, meaning that a valid value of r can be calculated by the Intermediate Value Theorem [7]. However, T is a function of other variables like RTT, M,

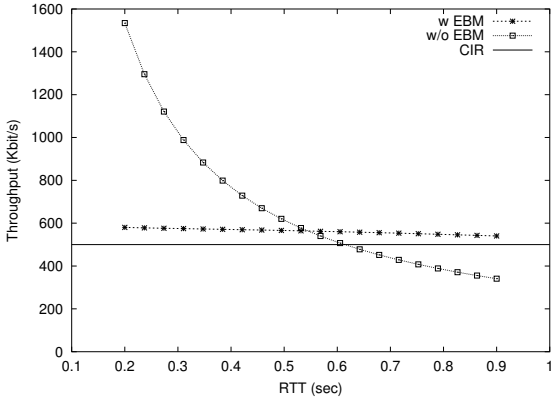


Figure 2.10: Output of the analytical proof — throughput vs. RTT

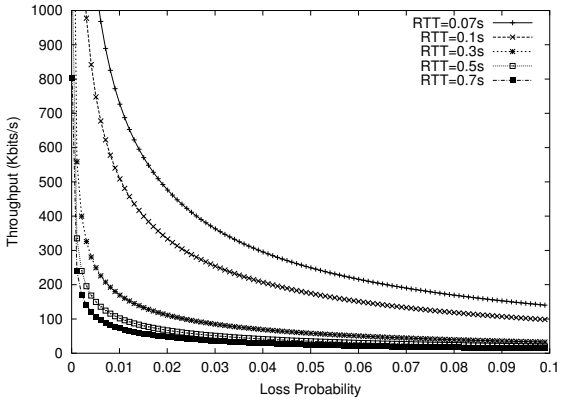


Figure 2.11: Throughput vs. loss probability for different RTT

and W_{max}, which impose other limitations on the throughput achieved at certain p as shown in Figure 2.11. So, given some values for RTT, M, and W_{max}, there is a maximum value for the throughput, r_{max}, that can be achieved. This is the value calculated in Section 2.3.1 to compare with. By definition, r_{max} should be found at very small p, and this value is bounded for certain values of RTT, M, and W_{max}. Therefore, we first test for this condition in the EBM algorithm, and if the required target rates are below this maximum value, then the numerical iteration will always converge.

2.5 Evaluation

We use `ns`-2 [118] to evaluate the performance of EBM and compare it with other marking algorithms. The network in Figure 2.12 is used in our evaluation. Different scenarios have been simulated on this network to measure the performance for different parameters. In all these scenarios, we use ten TCP sources (i.e., $n = 10$) along with two UDP sources as background traffic. This background traffic represents the best-effort traffic in the Internet and uses Red-colored CBR (Constant Bit Rate) packets. Edge routers do the metering and packet marking (represented as squares in Figure 2.12) and core routers implement the Multi-RED (MRED) buffer management as a realization of the AF PHB. MRED uses a single average, non-overlapped (staggered) model and has the parameters

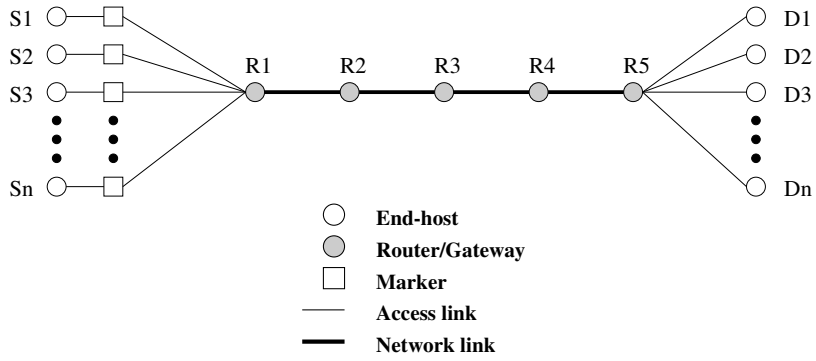


Figure 2.12: Simulation topology

listed in Table 2.1 where thresholds relative to the total queue length L are used. Each TCP source generates an infinite FTP bulk data transfer, with its own target rates, CIR and PIR. The subscription level of the network is set by properly adjusting CIRs and the background rate. All access links have a capacity of 100 Mbps and a latency adjustable to the simulation scenario, while the links between core routers have a 10 Mbps capacity and a 10ms latency. The bottleneck is made to occur at the links between the core routers. Unless otherwise stated, a packet size of 576 bytes is used. In all the simulations, we measure the goodput achieved by each TCP flow using the Weighted Moving Average (WMA) technique with a 1-second window and a weight of 0.5–0.8. We then calculate the average goodput over the whole simulation period. Each simulation scenario is repeated 10 times, and then an average is taken over all runs. The EBM uses a 10sec interval between two successive calculations of p_{cir} and p_{pir}, and an accuracy of 0.005%, while using a *YScale* of 500 and a *RScale* of 1000. These values have been set empirically for the best performance of EBM with the current network configuration used in the simulation.

For different marking algorithms, we use the following notations in the graphs: TCM for Token bucket Three Color Marker, TSW for Time Sliding Window three color marker, ETSW for Enhanced Time Sliding Window marker, RPM for Random Packet Marking, APM for Adaptive Packet Marking, and finally EBM for Equation-Based Marking. Also, CIR is used for Committed Information Rate, and PIR for Peak Information Rate.

Parameters for	Green	Yellow	Red
Queue length	L	L	L
$\texttt{Max}_{\texttt{th}}$	0.875L	0.625L	0.3125L
$\texttt{Min}_{\texttt{th}}$	0.625L	0.3125L	0.025L
$\texttt{Max}_{\texttt{p}}$	0.02	0.05	0.1
$\texttt{w}_{\texttt{q}}$	0.002	0.002	0.002

Table 2.1: MRED parameters

2.5.1 Subscription Level

In the first scenario, we investigate the effect of the network subscription level, or load, on the performance of EBM, comparing it with other marking schemes. The subscription level is changed from a light load (45%) to a heavy overload (200%), and the results are plotted in Figure 2.13. A CIR value of 500 Kbps and a PIR value of 700 Kbps are used for all markers, and by changing the background rate, we achieve the required subscription level in the network.

From the figure, we see that EBM has better protection and adherence to CIR than other marking schemes under a broad range of network loads. Of course, under a very light load, all the markers can achieve good throughput, as there is a large surplus of capacity in the bottleneck links. On the other hand, for heavy network loads, EBM is superior to others in protecting the AF traffic from background traffic to sustains its CIR and get as much bandwidth as they can from the extra capacity toward PIR. To show the fairness among different flows, Figure 2.14 plots the goodput per flow for a 120% subscription level. All marking schemes have the same fairness in this case for similar flow parameters. Note that APM also yields similar performance on a large time scale, but on a small time scale, APM causes more fluctuations in the achieved throughput than EBM. This characteristic is also experienced in all other scenarios.

2.5.2 RTT

Using a range of RTTs from 100ms to 1520ms for different TCP flows, by changing the latencies of access links, we evaluate the fairness and performance of EBM against the other marking schemes in moderately- and heavily-loaded networks. The results are plotted in

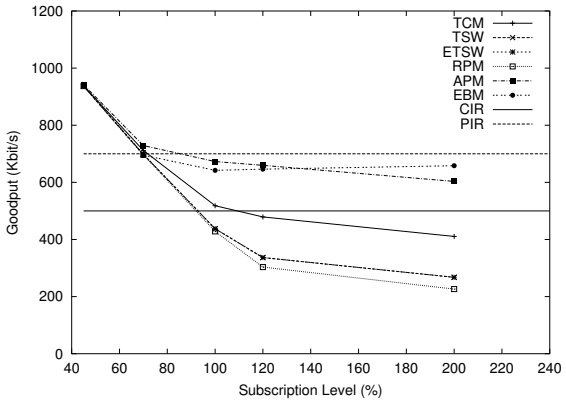


Figure 2.13: Goodput vs. subscription level

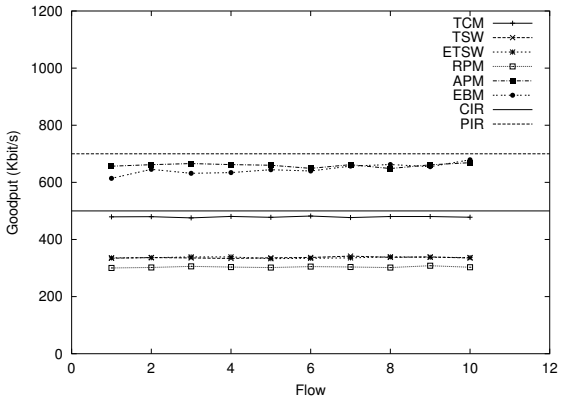


Figure 2.14: Goodput per flow — over-subscription

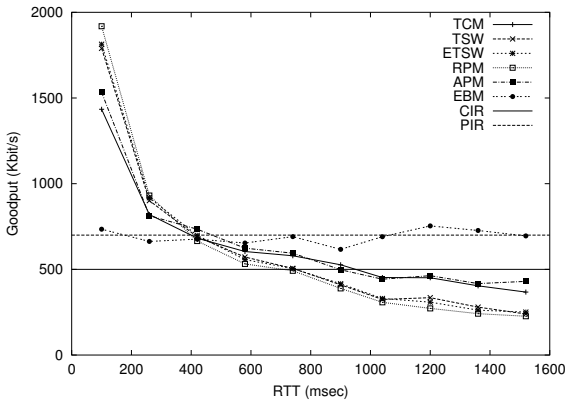


Figure 2.15: Goodput vs. RTT — under-subscription

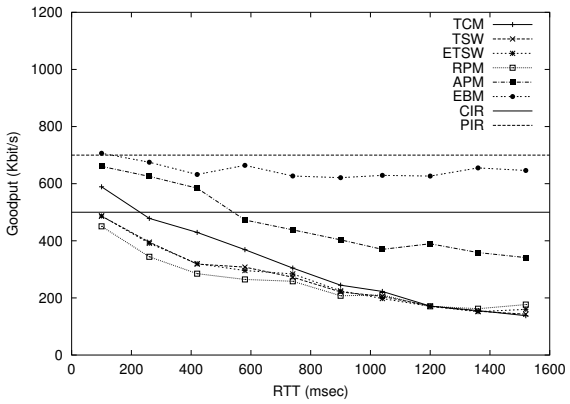


Figure 2.16: Goodput vs. RTT — over-subscription

Figures 2.15 and 2.16 for 80% and 130% load, respectively. One can see form these figures how EBM equalizes the throughput among different TCP flows while satisfying the CIR requirement under both conditions, whereas the other marking schemes can not even reach CIR under heavy loads and large RTTs.

2.5.3 Target Rate

TCP flows with different target rates should get proportional shares of excess bandwidth in under-subscribed networks [92, 115]. We evaluate EBM for this case using a range of CIR from 0.5 Mbps to 2.3 Mbps and a PIR equal to twice the CIR. Each flow has a different

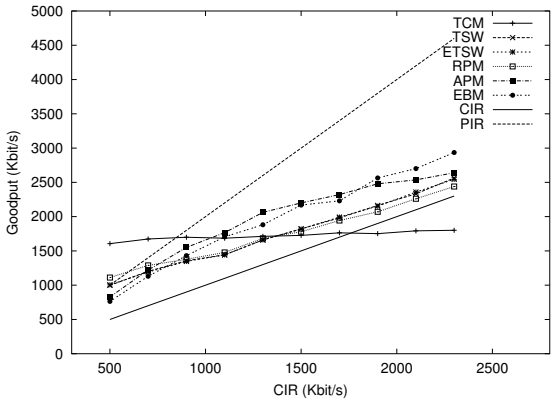


Figure 2.17: Goodput vs. target rate — under-subscription

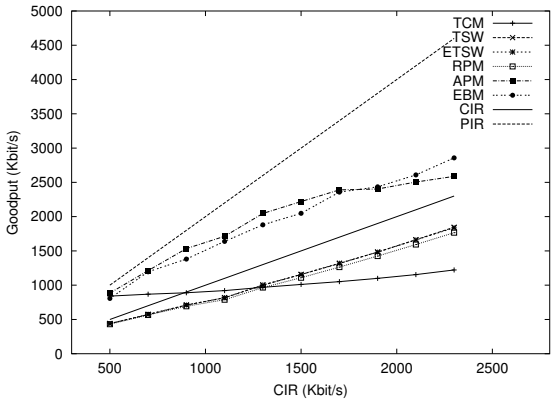


Figure 2.18: Goodput vs. target rate — over-subscription

CIR and PIR values. A link capacity of 20 Mbps is used between core routers instead of 10 Mbps. Results are plotted in Figure 2.17 for 80% load and in Figure 2.18 for 120% load. One can see that EBM provides each connection its proportional share of excess bandwidth for both under-subscribed and over-subscribed cases.

2.5.4 Packet Size

We now evaluate the performance of EBM along with the other marking schemes for flows of different packet sizes. We use a range of packet sizes from 100 bytes to 1500 bytes and show the results for 65% load (under-subscribed) in Figure 2.19, and 120% load (over-subscribed) in Figure 2.20. CIR and PIR values are 1 Mbps and 1.5 Mbps, respectively, and we use a link capacity of 20 Mbps between core routers.

Although Figure 2.19 shows a significantly fairer behavior of EBM than other marking algorithms, we still see some slight increase in the goodput with the increase of packet size. To quantify this increase, we compare the average slope of the EBM performance with the other markers. The average slope in the EBM graph is calculated to be 0.32, while the ratio between the maximum and minimum value of the goodput is 1.35 which is close to the ideal case of 1 for equal target rates, CIR and PIR. TCM, as the basic marking algorithm, has a slope of 1.97 and the ratio between the maximum and minimum values of the goodput is 17. Knowing that the ratio between the maximum and minimum packet sizes in this

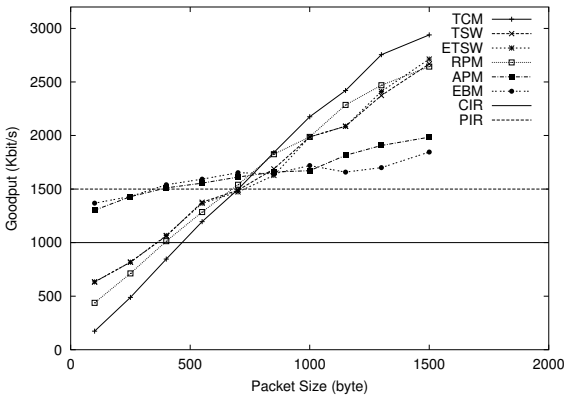


Figure 2.19: Goodput vs. packet size — under-subscription

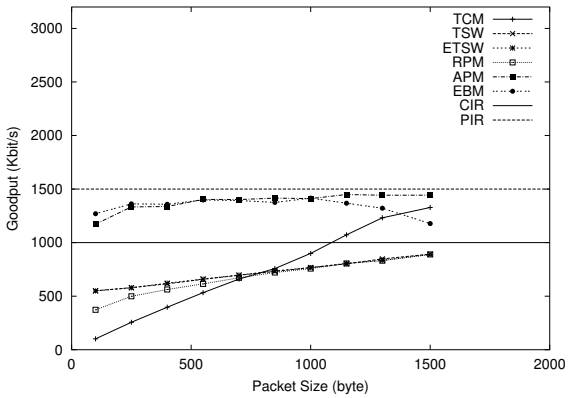


Figure 2.20: Goodput vs. packet size — over-subscription

experiment is 15 (1500/100), TCM goodput is directly proportional to packet size, which makes it significantly unfair with respect to EBM. For completeness, the slope in case of TSW is found to be 1.45, for ETSW it is 1.48, while for RPM it is 1.57. This means that EBM achieves about 76.5% improvement in fairness than the lowest-slope marker, TSW.

2.5.5 A Closer Look into EBM and APM

As it was clear in the previous set of experiments, the performance of EBM and APM was close to each other. This motivated us to conduct another experiment to distinguish between EBM and APM operations.

APM is designed using a simple feedback loop based on measurement of the achieved throughput and changing the marking probabilities accordingly [41]. However, this simple feedback loop, which does not resemble the behavior of the original transport algorithm, can lead to overshoots and undershoots in the achieved throughput, especially when dealing with long-lived TCP flows. Figure 2.21 shows the throughput achieved as a function of time using APM, and also shows the share of each color in this throughput. A large amount of fluctuations in the achieved throughput is observed and the main reason for this is that the APM adaptively adjusts the marking probabilities, depending on the current position of the measured rate. For cases in which the measured rate is always above the target rate, the marking probability will decrease continuously until it reaches the value of 0.0, i.e.,

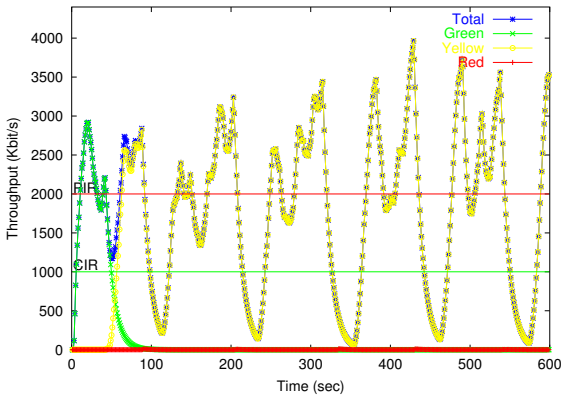


Figure 2.21: Throughput vs. time for APM

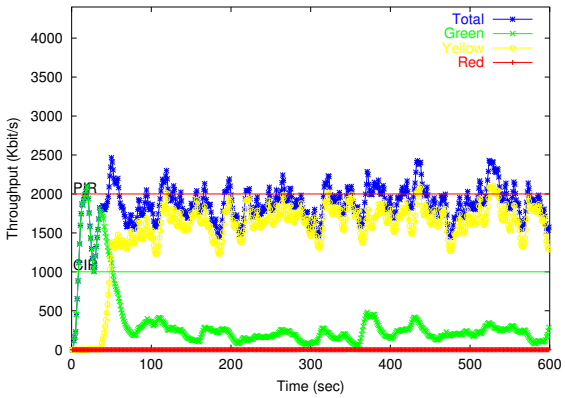


Figure 2.22: Throughput vs. time for EBM

marking all packets as out-of-profile, and causes the TCP flow to have a high probability of packet drop which, in turn, results in fluctuations in throughput. The behavior of the EBM under the same situation is depicted in Figure 2.22. Another difference between APM and EBM is that APM needs to be implemented inside the TCP code itself on the host as it makes use of internal TCP state variables. This means modifying TCP in every host machine which will induce large overhead. EBM, on the other hand, can be implemented within the host, but as a separate module from TCP, or can be implemented on a completely separate device called a *marker* (this can be found in leaf routers or gateways).

2.5.6 Overhead

We evaluate the overhead caused by the operation of EBM which is expressed in two terms. The first is the time taken in packet classification, marking, and state variable updates. This term is common to all other packet markers and is not unique to EBM, hence we do not evaluate it. The second overhead is for the calculation of p_{cir} and p_{pir}. We measured this overhead to be 1.5 to 2ms on a Pentium II, 450MHz processor with 192MB RAM machine. This calculation is performed every 10 seconds resulting in 0.015% to 0.02% overhead, which is considerably small, hence EBM introduces very low overhead.

2.6 Fairness between Responsive and Non-responsive Traffic

As pointed out earlier, buffer management using packet drop schemes are always biased against responsive (TCP) traffic sources. This biased performance results from the action taken by responsive traffic sources when they experience packet losses. Usually, these losses are interpreted as congestion and network overflow, and accordingly, responsive sources tend to reduce their transmission rate. Unfortunately, this does not take place in non-responsive (UDP) traffic sources. They keep sending packets at the required rate regardless of the packet losses they may experience. An equal treatment for both traffic types at the buffer management node is not appropriate and another solution has to be found. The fairness between TCP and UDP traffic has been studied in [115, 116] and the results there confirm the existence of the unfairness problem. In the following two subsections we present a solution that alleviates this unfairness and also demonstrate its effectiveness in achieving the required fairness.

2.6.1 Packet Separation

The key issue of the fairness problem between responsive and non-responsive traffic is putting all the packets of the two types into the same queue at routers where all the buffer management functionalities apply on them indiscriminately. A serious result of putting all packets in the same queue is that when the queue overflows, mainly because of the continuous unaffected transmission rates of the non-responsive flows, packets belonging to responsive traffic are dropped blindly irrespective of their relative importance, especially when using packet marking. This cannot protect responsive traffic from non-responsive traffic.

One solution is to calculate the virtual queue lengths separately between responsive and non-responsive traffic and try to punish non-responsive traffic more, but this solution turns out to be too complex. Another simpler, and at the same time effective, solution is to use separate queues for responsive and non-responsive traffic at buffer management nodes. This

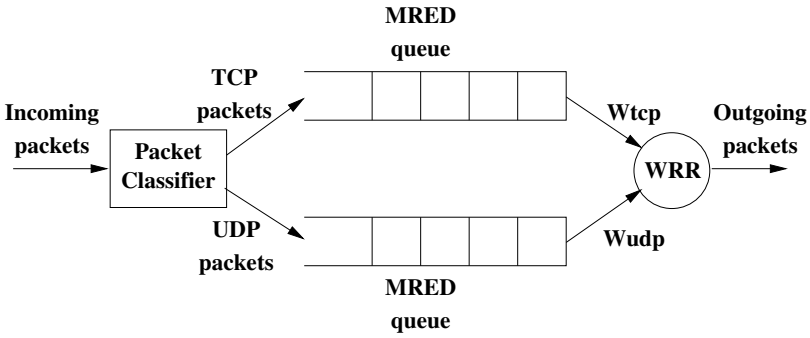


Figure 2.23: Separate handling for TCP and UDP packets at router

isolates performance between the two types, and the correct drop ratios can be calculated without changing the original buffer management algorithm. The idea of using separate queues was mentioned before in [92], but here we put it in action with EBM. As shown in Figure 2.23, two MRED queues are used, one for TCP (as an example of responsive traffic) packets and the other for UDP (as an example of non-responsive traffic) packets. These two queues are served using a Weighted Round Robin (WRR) scheduler. Both queues have the same MRED parameters. W_{tcp} and W_{udp} are the scheduler weights assigned to TCP and UDP queues, respectively, and they should be proportional to their target rates, or can be set according to a certain policy. This separation scheme can be used to handle fairness issues between responsive and non-responsive traffic in general and can be extended to any number of queues depending on the types of traffic handled in the network.

2.6.2 Evaluation

To evaluate the performance of using two separate queues for TCP and UDP traffic at routers, we conduct simulation experiments using the same network topology used in previous evaluation. We compare the two cases of using a shared queue for both TCP and UDP traffic, and using two separate queues. Four TCP and four UDP sources share a link of 10 Mbps capacity with 40 ms latency. Access links have 100 Mbps capacity with 10 ms latency and MRED parameters are the same for both TCP and UDP queues. Both TCP and UDP flows have the same CIR and PIR values of 0.875 Mbps and 1.75 Mbps, respectively, and hence, they use equal weights at the WRR scheduler.

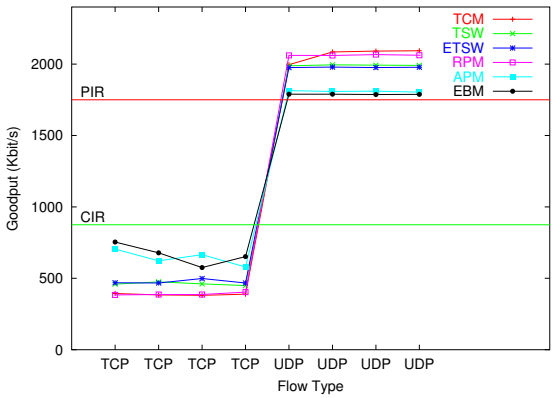


Figure 2.24: One shared queue

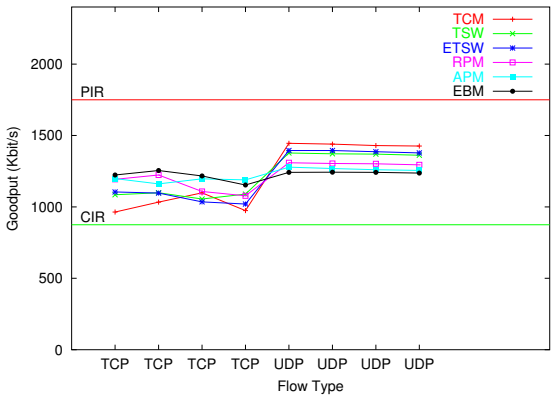


Figure 2.25: Two separate queues

Figures 2.24 and 2.25 show the performance for the two cases. We can see the difference in fairness between TCP and UDP flows in the achieved throughput even when packet marking is used. When using a single shared queue, none of the TCP flows could achieve their CIR, while UDP flows exceeded their PIR. This is not the case when using two separate queues, where both TCP and UDP flows could achieve their CIR. Moreover, we notice enhanced performance when the EBM algorithm with TCP traffic is used.

2.7 Concluding Remarks

In this chapter we presented a working example of using the model-based control approach in achieving throughput guarantees and fairness as two parameters of QoS. Although the model used in this example is not our original contribution, we were keen to choose an accurate model that can be trusted in generating the control decision. We used packet marking as the control action in this study which is currently specific to the DiffServ framework.[5] The EBM marking scheme solves the problems associated with previously-proposed schemes in providing fairness between heterogeneous TCP flows and protection of target rates under diverse network conditions.

We compared the performance of the new marking algorithm with other ad-hoc schemes and found it to be superior to others. Even with other adaptive marking schemes that

[5]We expect marking to be a common technique for achieving QoS in the future.

are based on less accurate models, a close look at the performance difference shows an advantage of our EBM technique.

There are several lessons learned from this work. First, using an accurate model of the controlled system gives better results and stable control actions. We showed this in the difference between EBM and APM. Second, using feedback control is an effective technique in controlling QoS in computer networks. Third, choosing the right control action is important for realizing the control decision and achieving the required output.

CHAPTER 3

Statistical Modeling and Control of Per-Hop QoS

The ability to provide predictable network services, to support, for example, Voice-over-IP and Video-on-Demand, is an important goal for most network providers on the Internet. Such services should have well-defined, and at the same time, well-controlled, network-level QoS. In IntServ [16] and DiffServ [13, 98], QoS is achieved by applying different traffic management and control techniques at all, or a subset, of the network nodes that are handling traffic. A major difference among these frameworks lies in the granularity of traffic control. For example, in DiffServ, a small set of PHBs have been proposed [29, 59] to provide different classes of services to different traffic aggregates. The PHB is the key building block of the DiffServ architecture end-to-end (e2e) QoS. In a typical PHB, service differentiation is achieved by allocating different amounts of network resources, such as link bandwidth and buffers, to different types of traffic aggregates traversing the DiffServ-enabled routers. The output of this service differentiation at each router with respect to throughput, delay, jitter, and loss of the output traffic, is called *per-hop QoS* (PHQ). On the other hand, under IntServ, this control is applied on a per-flow basis, which is fine-grained but with less scalability. In both cases, providing a certain PHQ guarantee along the traffic path is the fundamental requirement in realizing network-level QoS.

Given the PHQ for every node along an e2e path, the e2e QoS perceived by users can be pre-computed or even estimated and controlled at runtime.[1] Each QoS-aware node has a variety of traffic-management components, such as queues, schedulers, buffer managers,

[1] Edge-to-edge Per-Domain Behaviors (PDBs) can also be quantified.

policer, and filters. These components must be properly designed and configured to ensure PHQ guarantees. However, it is a challenging task to integrate these building blocks together to meet the QoS requirements, especially in view of the complex interactions among the various components.

It is becoming more important for network operators to be able to adjust the configuration parameters of network nodes within their domains to meet the needs of specific services being offered, and to respond to the dynamic changes in input traffic. Accurately modeling and controlling a network node is, therefore, an important step to achieve network-level QoS in corporate networks as well as the Internet.

We showed the effectiveness of the model-based control approach in Chapter 1, and now, we take it further and build the models to be used in the control algorithm. We use a generic quantitative approach in modeling the PHQ for a typical QoS-aware network node (e.g., PHB) under a wide range of input traffic and configuration parameters. We consider the DiffServ PHBs (EF and AF) as "use-cases" for our study, but the approach itself is general enough to be applied to *any* QoS-aware subsystem (a single node or the entire network). We demonstrate that a careful statistical analysis in conjunction with an experimental framework can effectively characterize any QoS-aware node or PHB, and extract the functional dependencies of PHQ on the input traffic and configuration parameters. These functional relationships point out the differences in performance guarantees provided by different PHB implementations [35].

Once these statistical models are extracted through the analysis, we show how to use them to predict and control the PHQ [36]. The control is applied to the inputs of a QoS-aware node to achieve the required PHQ level following our "model-based control" approach. We evaluate the overall modeling and control with several experiments on our network testbed, showing its effectiveness and accuracy.

3.1 Statistical Modeling

To model the PHQ of a given QoS-aware node, we take a statistical approach following an experimental analysis. We use Design of Experiments (DOE) [70] along with statistical

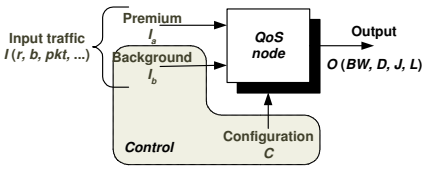


Figure 3.1: Model of per-hop QoS

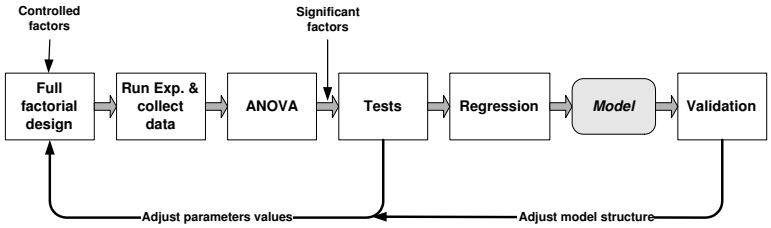


Figure 3.2: Steps of statistical modeling

modeling such as Analysis of Variance (ANOVA) and regression analysis to get a characterization of the PHQ under different conditions and parameters. Formally, the following major steps are taken to model the PHQ of a given QoS-aware node or PHB.

S1: Our key idea is to abstract a given QoS-aware node or PHB[2] as a "black-box" system with input (I) and output (O) traffic characteristics, as well as a set of configuration parameters (C), as shown in Figure 3.1. These parameters can be controlled and/or measured via experiments. The details of the internal implementation and states of the QoS node are not considered explicitly, but their effects on the overall performance (i.e., the output traffic characteristics) are modeled.

S2: Design a set of full factorial experiments [70], by expanding the inputs and the configuration parameters, to measure the effects of the input and configuration parameters on the output PHQ.

S3: Perform ANOVA on the measurement results to identify the most significant parameters, and apply a polynomial regression analysis to extract the functional relationships — statistical models of the node under study — between the PHQ and input parameters within the range and domain of our experiments.

S4: Evaluate the extracted models against real experimental results, adjust their structures accordingly, and validate their correctness as the conceptual representation of a given node.

[2]The terms "QoS-aware node" and "PHB" will be used interchangeably throughout the rest of the chapter.

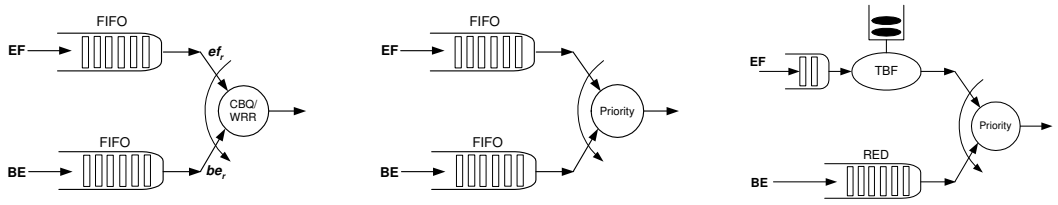
In the context of our experimental design, we refer to the PHQ attributes experienced by traffic traversing the corresponding node as O or the *response variables*. The parameters of I and C are termed *input factors* and the values each factor takes are called *factors levels*. The basic steps of our approach are illustrated in Figure 3.2. For certain experiments, due to the large number of combinations of input factors and their levels, it is useful to start with a reduced number of levels for the input factors. Once the most significant/influential factors are identified, the number of levels corresponding to these factors are increased to capture the detailed dependency on their particular levels. This allows us to construct higher-order regression models in the second stage of the analysis. This repetition of factorial analysis is shown as the "adjust parameters values" step in Figure 3.2.

The statistical models extracted in our approach describe the steady-state behaviors of a QoS-aware node, instead of specific "time-detailed" or "transient" behavior. Note that the purpose of these models is to control the overall behavior of the node against changing levels of inputs and configuration parameters regardless of the specific transient dynamics of the node's behavior. For this reason, the models do not carry any notion of time dependency.

We investigate three different realizations of the EF PHB, as an example of a QoS-aware node, as illustrated in Figures 3.3(A) and (B). These realizations show different choices for premium (EF) and best-effort (BE) traffic that shares a single physical link using two scheduling algorithms: Class-Based Queueing (CBQ) [50] and Priority Queueing (PRIO). We also consider the case of an edge node with policing support using Token Bucket filters (TBFs) as in Figure 3.3(C). While these different realizations are expected to provide similar qualitative behavior of the aggregated output traffic, the functional relationships among the parameters are different due to the internal construction of the traffic handling blocks. We will later demonstrate this through experiments.

3.1.1 Factors and Response Variables

The parameters of I include both premium (EF), I_a, and best-effort (BE), I_b, input traffic specifications. We use the Dual Leaky Bucket (DLB) representation for the input traffic, as



(A) A CBQ-based EF PHB (EF-CBQ)

(B) A priority-based EF PHB (EF-PRIO)

(C) A edge-based EF PHB (EF-EDGE)

Figure 3.3: Three different realizations for EF PHB sharing the a link with best-effort (BE)

	Factor	Symbol
I_a	Premium traffic average rate	a_r
	Premium traffic peak rate	a_p
	Premium traffic burst size	a_b
	Premium traffic packet size	a_{pkt}
	Premium traffic number of flows	a_n
I_b	Best effort traffic rate ratio	R_{ab}
	Best effort traffic rate	b_r
	Best effort traffic burst size	b_b
	Best effort traffic packet size	b_{pkt}
	Best effort traffic number of flows	b_n

Table 3.1: Symbolic representation of the factors in I

shown in Table 3.1. This lets the characterization process be independent of the application traffic type as long as the traffic can be described in the general form of DLB representation, which is a common practice. In some cases, we also use the ratio, R_{ab}, of EF traffic rate to BE traffic rate, i.e., $\frac{a_r}{b_r}$, instead of the absolute value of the BE rate (b_r). The reason behind this is that we are primarily interested in the performance of premium traffic, and hence, it is more meaningful to consider the relative load of the BE traffic with respect to premium traffic.

The set of configuration parameters (C) usually consists of parameters such as queue length, drop probability, allocated forwarding rate, and scheduling parameters. Choosing the parameters of C requires either knowledge of the traffic control components of the node, or the use of vendor-supplied specifications. Different choices of schedulers and traffic controls give rise to different functional configurations, and therefore, constitute different sets of parameters in C. We use routers based on the open-source Linux operating system and, therefore, we have access to all the configuration parameters. The Linux traffic control module provides a flexible way to realize various PHBs with the help of a number

Factor	Symbol
EF service rate	ef_r
BE service rate	be_r

(A) EF-CBQ

Factor	Symbol
Token bucket rate	ef_r
Bucket size	ef_b
Max Transfer Unit	ef_{mtu}

(B) EF-EDGE

Factor	Symbol
Min threshold	min_{th}
Max threshold	max_{th}
Drop probability	$prob$

(C) AF PHB

Table 3.2: Configuration parameters - set "C"

of queueing disciplines and traffic conditioning modules.

For the three different realizations of the EF PHB shown in Figure 3.3, EF-CBQ significant configuration parameters (set C) are listed in Table 3.2(A). EF-PRIO employs absolute priority[3] scheduling, and hence, no significant parameters are used, while for EF-EDGE, Table 3.2(B) lists the considered parameters. We also repeat similar performance analysis for the AF PHB. We use a multi-color RED (or GRED) queue for each AF class served by a CBQ scheduler. Table 3.2(C) lists the C set for the AF PHB. It includes the parameters for the AF11 RED virtual queue only. We choose the AF PHB as another instance of a DiffServ PHB to demonstrate the generality of our measurement framework and analysis approach.

Since most QoS schemes do not provide any guarantee for best-effort traffic, we are primarily interested in the performance of the premium traffic. Therefore, the response variables in O used in this chapter are throughput (BW), per-hop delay (D), per-hop jitter (J), and loss rate (L) of the premium traffic.

An important issue is how to choose the levels for the factors in designing a set of experiments. It may not be possible to cover all possible ranges and various modes of operation of a specific PHB. However, the experiments should capture the expected ranges of operation for the node.[4] A *full factorial design* utilizes every possible combination of all the factors [70] at all levels. If we have k factors, with the i-th factor having n_i levels, and each experiment is repeated r times, then the total number of experiments to perform will be $\prod_{i=1}^{k} n_i \times r$. One of the drawbacks of the full factorial analysis is, therefore, the number of experiments growing exponentially with the number of factors and their levels. Moreover, in the context of network measurements, the total duration of an experiment can

[3] Absolute priority means processing EF packets as soon as possible in a non-preemptive work-conserving manner.

[4] Given by a network administrator, for example.

be prohibitively long, and often taking several days.

To reduce the number and the execution time of experiments, we use a combination of *factor clustering* and *semi-automated* experimental execution techniques. In factor clustering, the input and configuration parameters having similar effect on the output, are grouped together, while the semi-automated experiments enables investigation of a large number of scenarios in one execution.

3.1.2 Tools for Statistical Modeling

A detailed description of the statistical methods we used, namely, ANOVA and regression analysis can be found in [70, 93]. However, we include a brief description of them in the Appendices B.1 and B.2 for completeness. The linear model used in ANOVA is based on the following assumptions [70]: (1) the effects of the input factors and the errors are additive, (2) errors are identical and independent, normally distributed random variables, and (3) errors have a constant standard deviation. Therefore, an important step after the ANOVA analysis is to validate the model by inspecting the results. This can be done using two basic visual tests:[5] (1) the scatter plot of the residuals (errors) versus the predicted response should not demonstrate any trend, and (2) the normal quantile-quantile (Q-Q) plot of residuals should be approximately linear (after removing the outliers). The ANOVA method itself, however, does not make any assumption about the nature of the statistical relationship between the input factors and the response variables [93].

To obtain the required relationships between each output response variable and the most significant input factors, we use a variant of the multiple linear regression method called the *polynomial regression* [93]. The reason for this is to approximate any complex, nonlinear relationship into an expansion of piecewise polynomials of an enough number of terms. We use a number of simple transformations [70] such as inverse, logarithmic, and square root, to capture non-linearity in these relationships, and convert them into linear ones. However, more complex transformations [70, 93] can be used to capture complex dependencies. We choose the transformation that best satisfies the visual tests, minimizes the error percentage

[5] These tests can be automated as well.

in ANOVA, and maximizes the coefficient of determination (R^2) in the regression model.

3.2 Experimental Setup for Modeling

A controlled environment is used for extracting the PHQ models. The values of the input factors are chosen according to specific scenarios, and for each scenario, the corresponding models are extracted. This controlled execution of scenarios is automated, and in the following we describe the scenarios used and the way they are automated.

3.2.1 Network Setup

The testbed network used, illustrated in Figure 3.4, consists of Linux-based software routers and end-hosts. The traffic conditioning and handling mechanisms built into the Linux kernel [6] enable building QoS-aware nodes such as those under study. A Fast-Ethernet based ring network topology, with link capacity of 100 Mbps, is used so that the one-way delay can be measured without sophisticated time synchronization techniques such as GPS. Since the objective of our experiments is to characterize PHQ of a single PHB, we need only one router (*M*) implementing the PHB. We are primarily interested in determining how a single EF or AF flow, which we refer to as the *designated flow*, is influenced by PHB configuration parameters as well as other traffic sharing the router with it. This *designated flow* is always generated and terminated at host *H* (as a result of the ring topology). In order to build the ring topology, we add a second router *S* to (i) forward outgoing packets from *M* back to host *H* using *iptables* in Linux, and (ii) act as a destination for the background or best-effort[6] traffic. Hosts 1 through 6 are used to generate background traffic (both premium and best-effort) that share the links between routers *M* and *S* along with the *designated flow*.

[6]Unless otherwise mentioned, we use terms "background" and "best-effort" interchangeably onwards.

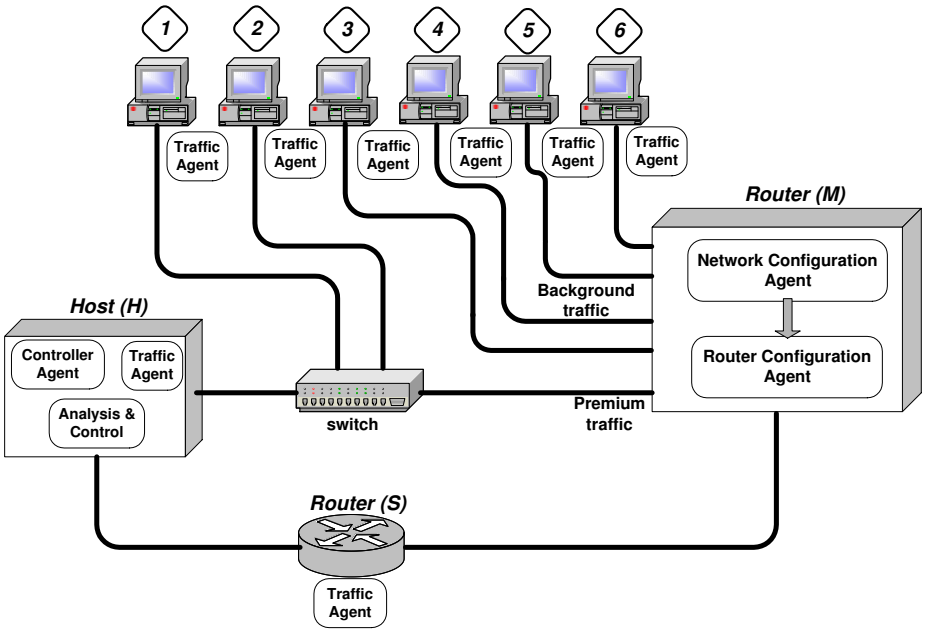


Figure 3.4: Testbed network used for PHQ modeling

3.2.2 Components of the Automated Experiments

The software components used to conduct the automated experiments and their locations are illustrated in Figure 3.4. We use a traffic generation agent for UDP traffic, based on a modified version of Iperf [99], that is policed with a built-in leaky bucket to produce output traffic following a particular DLB specification. The response variables, namely, BW, D, J, [7] and L are measured within the agent itself. The network and router configuration agents are placed on the QoS subsystem under evaluation to configure the traffic control blocks on that subsystem. For our QoS node, we use the traffic control (tc) APIs in the Linux kernel [6]. The controller agent executes and keeps track of the experiment steps. It resides on host H, as shown in Figure 3.4, reads in a scenario file, that defines the parameters (factors) and their values (levels), and runs the experiments accordingly. It communicates with the other agents on a request-reply basis. The statistical analysis and computation of the extracted models are performed on host H as well.

[7] J is calculated as $J = J + (|D(i-1,i)| - J)/16$, where $D(i-1,i)$ is the delay variation between packets i and $(i-1)$.

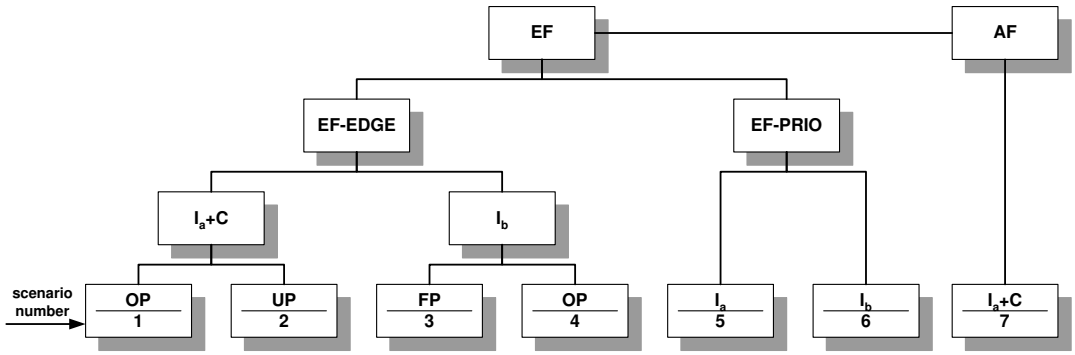


Figure 3.5: First set of experimental scenarios used in modeling

3.2.3 Design of Experiments

As discussed earlier, we cluster the input factors and the configuration parameters to reduce the parameter space and duration of the experiments. The I set is partitioned into two subsets: the premium traffic factors (I_a), and the best-effort traffic factors (I_b) as depicted in Table 3.1. Along with these, we also choose a few parameters from the configuration set (C).[8]

There are usually three different operating modes for a given PHB configuration with respect to the premium input traffic: the PHB can be *over-provisioned* (OP), *under-provisioned* (UP), or *fully-provisioned* (FP). In the OP mode, the total premium input traffic rate is less than its allocated rate; It is larger than the allocated rate for the UP mode, and in the FP mode, they are nearly equal. These three modes can be further investigated for different scenarios of input traffic type and the degree of flow aggregation. We also study two other scenarios, *saturated* (SAT) and *not-saturated* (NOT-SAT) network depending on the relationship between the background traffic sending rate and its allocated rate.

We consider two sets of experimental scenarios, the organization of the first set is illustrated in Figure 3.5, which has been presented in [35], while the second set, illustrated in Figure 3.6, has been presented in [36]. Each cell is associated with a scenario number to distinguish them in the following sections.

[8]If the size of C is large, it can be partitioned as well.

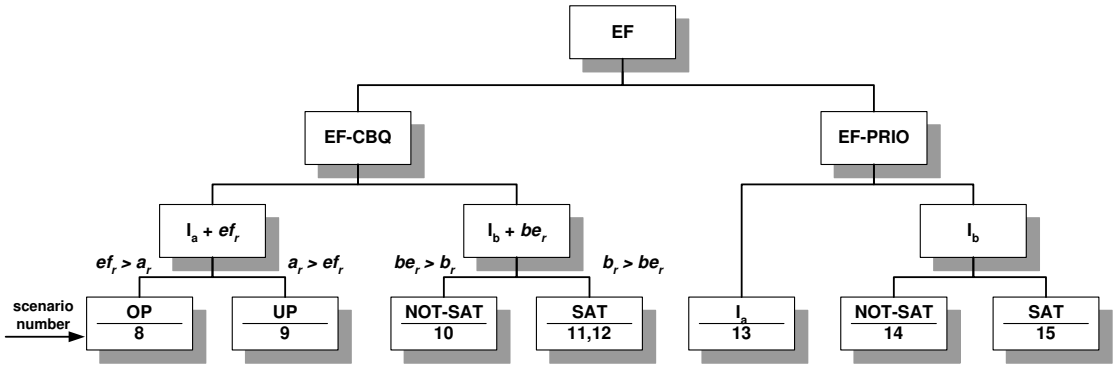


Figure 3.6: Second set of experimental scenarios used in modeling

First Set of Scenarios

In this set, we have 7 scenarios to study EF-EDGE, EF-PRIO, and AF PHBs and characterize their behaviors. We start by investigating the interaction between sets I_a and C in the absence of any best-effort traffic. This experiment is performed for the two modes of OP and UP. The purpose of these two scenarios is to identify the most important factors in sets I_a and C that affect the output PHQ. Next, we target a specific type of premium traffic, e.g., fix set I_a at certain constant values and vary the factors in I_b around these constant values. The factors in the best-effort traffic (such as rate, burst size, packet size and number of flows) range in values that include both higher and lower levels than their corresponding parameters in the premium traffic. The purpose is to investigate the effect of best-effort traffic on the premium traffic within a given range. These scenarios are investigated for OP and FP modes only, since the PHQ of an already-overloaded PHB is not going to change significantly due to changes in the best-effort traffic. We repeat these scenarios for each one of the EF PHB realizations in Figure 3.3. We also study the AF PHB in this set to show generality of the approach.

Second Set of Scenarios

In this set, we start by investigating the effect of I_a factors with the change of ef_r for EF-CBQ. This experiment is performed for the two modes of OP ($ef_r > a_r$) and UP ($a_r > ef_r$) as shown above. The purpose of these scenarios is to identify the most important

factors in I_a that affect the output PHQ under different network provisioning conditions. This is used to adjust the premium traffic parameters according to the network conditions to achieve certain QoS, and it can be done by either signaling traffic sources, or by using other coordinated mechanisms. Next, we target a specific instance of premium traffic, i.e., certain values of I_a, (similar to the first set) and vary the factors of I_b and C sets (specifically be_r). Here, we distinguish between the two cases, NOT-SAT ($be_r > b_r$) and SAT ($b_r > be_r$), where b_r is the background traffic sending rate, which equals a_r/R_{ab}. The purpose is to investigate the effect of best-effort traffic and the configuration parameters on a specific premium traffic flow. Again, this scenario is investigated for OP mode only. For the EF-PRIO realization, we consider three scenarios, one for modeling the effect of I_a on the PHQ, and other two scenarios for modeling the effect of I_b at SAT and NOT-SAT conditions of the PHB.

3.3 Model Extraction and Validation

The results from our experiments are presented in the same order of the scenarios described in Section 3.2.3 and the charts in Figures 3.5 and 3.6. Note that the results presented in this section are only examples of illustrating the applicability of our approach, and are not meant to be standard models for any QoS-aware node. Throughout the experiments, and unless mentioned otherwise, each experiment is repeated at least 5 times and each traffic trace is collected for 20 seconds. We use a rest period of 3 seconds between successive runs so that the network drains packets of a previous run. Both premium and best-effort traffic are ON/OFF with periods of 5/0.5 sec, and 4/0.3 sec, respectively.

3.3.1 First Modeling Set

For the EF-EDGE and EF-PRIO realizations, we group the results into two categories: the interaction between sets I_a and C, and the effect of background traffic (I_b) on the fixed premium traffic (I_a). The input factors (I_a, I_b, and C) used in each experiment scenario are listed in Tables 3.3, and 3.4 for EF-EDGE and EF-PRIO, respectively. For the AF PHB,

	Factor (unit)	Scenario 1	Scenario 2	Scenario 4
Set I_a	a_r (Mbps)	0.5,1,2,4	3,3.5,4,4.5	1 (X)
	a_p (Mbps)	6 (X)	8 (X)	2 (X)
	a_{pkt} (bytes)	100,400,600,900	800,900,1000,1200	1000 (X)
	a_b (bytes)	20000 (X)	40000 (X)	20000 (X)
	a_n (flows)	1,2,3,5	1,2,3,5	1 (X)
Set C	ef_r (Mbps)	20 (X)	2,2.2,2.5,3	3 (X)
	ef_b (bytes)	40000 (X)	20000 (X)	40000 (X)
	ef_{mtu} (bytes)	1000,1200,1400,1520	1520 (X)	1520 (X)
Set I_b	R_{ab}	(X)	(X)	2,1,0.1.0.01
	b_{pkt} (bytes)	(X)	(X)	200,1000,1200,1470
	b_b (bytes)	(X)	(X)	10000,20000,30000,40000
	b_n (flows)	(X)	(X)	1,2,3,4

Table 3.3: Factors and their levels for EF-EDGE

	Factor (unit)	Scenario 5
Set I_a	a_r (Mbps)	0.5,1,2
	a_p (Mbps)	(X)
	a_{pkt} (bytes)	100,800,1470
	a_b (bytes)	5000,10000,60000
	a_n (flows)	1,3
Set I_b	R_{ab}	(X)
	b_{pkt} (bytes)	(X)
	b_b (bytes)	(X)
	b_n (flows)	(X)

Table 3.4: Factors and their levels for EF-PRIO

	Factor (unit)	Scenario 7
Set I_a	a_r (Mbps)	50,100
	a_p (Mbps)	60,100
	a_{pkt} (bytes)	600,1200
	a_b (bytes)	20000,40000
	a_n (flows)	1 (X)
Set C	min_{th} (Kbytes)	10,20
	max_{th} (Kbytes)	40,55
	$prob$	0.02 (X)
Set I_b	R_{ab}	0.01 (X)
	b_{pkt} (bytes)	600,1200
	b_b (bytes)	20000,40000
	b_n (flows)	2 (X)

Table 3.5: Factors and their levels for AF PHB

the values of the input factors used in the modeling experiments are listed in Table 3.5. The factors marked with (X) are not active in the corresponding experiment. An inactive factor is a factor with one level only, and therefore, has no effect on the ANOVA results.

Over-Provisioned (OP) EF-EDGE PHB without Background Traffic

This corresponds to scenario 1 in Fig. 3.5. The PHB is over-provisioned (OP), i.e., the throughput of the premium traffic is less than the configured rate for the PHB. The effects of significant factors and their significant interactions[9] are identified by using ANOVA and listed in Table 3.6. The transformation applied to each response variable, satisfying the basic assumptions of the linear ANOVA model, is indicated in the second column. The mean, standard deviation (SD), and 90% confidence interval (CI) are listed in Table 3.7. The loss (L) in this experiment is basically zero (no loss) since this is an over-provisioned

[9]We neglect any factor or interaction with a percentage of variation less than 2%.

Response	Trans.	a_r	a_{pkt}	a_n	(a_r,a_n)	(a_r,a_{pkt})	Error
BW	linear	48.55%	$\approx 0\%$	34.23%	17.21%	$\approx 0\%$	$\approx 0\%$
D	linear	$\approx 0\%$	83.15%	$\approx 0\%$	$\approx 0\%$	$\approx 0\%$	13.53%
$1/J$	inverse	4.12%	12.52%	46.58%	$\approx 0\%$	2.88%	25.34%

Table 3.6: ANOVA results for scenario 1

Response	Mean	SD of Mean	90% CI for Mean
BW	948.81 Kbps	0.125	(948.60,949.015)
D	0.262199 msec	0.001088	(0.260409,0.263989)
$1/J$	311.21	3.74	(305.05,317.37)

Table 3.7: Mean, standard deviation (SD), and 90% confidence interval (CI) for the response variables in scenario 1

case.

To verify the linearity of the ANOVA model, the visual tests for jitter (J), as an example, are shown in Fig. 3.7 and 3.8. After the inverse transformation, there is no trend in the residual versus predicted response scatter plot. Moreover, the errors are normal since the normal Q-Q plot is almost linear. The tests for the throughput and the delay show linearity as well.

The polynomial regression models for the response variables are calculated, and the model for the BW is given in Eq. (3.1). The surface plot corresponding to this model is shown in Fig. 3.9. The coefficient of determination (R^2) is 96%.

$$\begin{aligned} BW &= 299733.48 + 3246957.34a_r - 2318542.4a_n - 32371.95a_r^2 + 2223351.57a_n^2 \\ &\quad -2815325.9a_r a_n \,. \end{aligned} \tag{3.1}$$

Eq. (3.2) represents the model for jitter, with the corresponding value of $R^2 = 84\%$. Note that we use the same transformation from ANOVA while performing the regression analysis to maintain consistency of results. We also normalize the input factors by their maximum values to provide a scaled version of the model equations.

$$\begin{aligned} 1/J &= 620.62 + 1343.58a_r - 1024.12a_{pkt} - 2212.85a_n - 1586.15a_r^2 - 231.39a_r a_{pkt} \\ &\quad -574.60a_r a_n + 913.48a_{pkt}^2 + 717.55a_{pkt}a_n + 3485.72a_n^2 + 281.31a_r^3 \end{aligned}$$

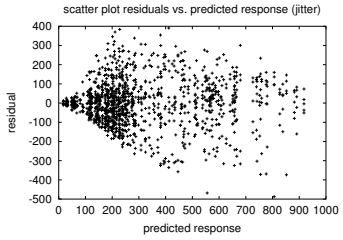


Figure 3.7: Residuals vs. predicted response for J (scenario 1)

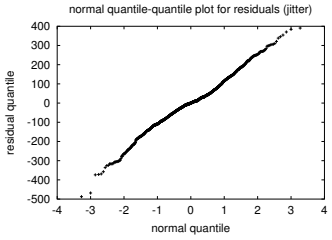


Figure 3.8: Normal Q-Q plot for J residuals (scenario 1)

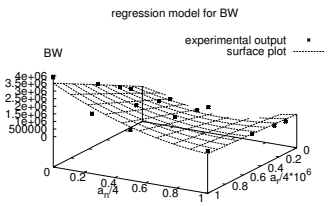


Figure 3.9: Response surface for BW (scenario 1)

$$+583.23a_r^2a_{pkt} + 700.89a_r^2a_n - 133.74a_ra_{pkt}^2 - 249.64a_ra_{pkt}a_n$$
$$-51.79a_ra_n^2 - 268.81a_{pkt}^3 - 307.76a_{pkt}^2a_n - 106.48a_{pkt}a_n^2 - 1768.39a_n^3\,. \quad (3.2)$$

As shown in Table 3.6, the throughput (BW) depends mostly on the sending rate of the premium traffic (a_r) and number of EF flows (a_n). The reason for the dependency of BW on a_n is that, in this experiment, we divide the total EF rate (a_r) specified in Table 3.3, by the number of flows (a_n) to get an equal share for each EF flow. Because we only measure a single designated EF flow, the resulting throughput depends on the number of flows. The per-hop delay is mainly affected by the EF packet size. On the other hand, jitter is affected mostly by a_n and a_{pkt}, but little by a_r. This follows our intuition about jitter: the packets from the monitored flow have to wait in the queue because of other EF flows sharing the same queue, as well as due to the relative difference of their packet sizes. Larger values of a_n and a_{pkt} result in larger jitter. This can be seen in the regression results as well, since the corresponding coefficients are negative in the model equation given that an inverse transformation is applied to jitter. The statistical analysis also provides us with confidence intervals for each of the parameter estimates in the regression equation. The confidence intervals determine the accuracy of the models extracted, and therefore constitute an important part of model checking.

Response	Trans.	a_r	a_n	ef_r	(a_r,a_n)	(a_r,ef_r)	(a_n,ef_r)	(a_r,a_n,ef_r)	Error
$1/L$	inverse	6.06%	20.81%	6.63%	15.1%	8.9%	16.21%	25%	$\approx 0\%$

Table 3.8: ANOVA results for scenario 2

Response	Trans.	b_{pkt}	b_n	R_{ab}	(b_{pkt},b_n)	(b_{pkt},R_{ab})	(b_n,R_{ab})	(b_{pkt},b_n,R_{ab})	Error
BW	linear	$\approx 0\%$	36.6%	2.65%	$\approx 0\%$	$\approx 0\%$	2.87%	4.4%	38.76%
$\log(D)$	log	3.28%	8.98%	36.3%	3.54%	10.01%	27.07%	10.69%	$\approx 0\%$
J	linear	3.36%	7.5%	39.14%	3.7%	9.72%	12.13%	22.77%	1.4%

Table 3.9: ANOVA results for scenario 4

Under-Provisioned (UP) EF-EDGE PHB without Background Traffic

In scenario 2, the EF PHB is under-provisioned (UP), i.e., the throughput of the premium traffic is greater than the configured rate for the PHB. Here we present the results for loss (L) only. The effects of significant factors and their significant interactions are listed in Table 3.8. The transformation used is indicated in the second column as before. The polynomial regression model for loss is given in Eq. (3.3) with R^2 =74%.

$$\begin{aligned} 1/L \;=\; & 0.06 - 0.63a_r - 0.2a_n + 0.63ef_r + 1.48a_r^2 + 0.79a_ra_n - 2.05a_ref_r + 0.3a_n^2 \\ & -0.69a_nef_r + 0.43ef_r^2 - 1.14a_r^3 - 0.73a_r^2a_n + 2.37a_r^2ef_r - 0.42a_ra_n^2 + 1.23a_ra_nef_r \\ & -1.69a_ref_r^2 - 0.18a_n^3 + 0.46a_n^2ef_r - 0.57a_nef_r^2 + 0.57ef_r^3 \,. \end{aligned} \tag{3.3}$$

The model for loss clearly shows that by increasing the sending rate, the loss increases (due to the negative sign of a_r and the inverse of L). A similar effect was observed with the number of flows (a_n). We can see also that the loss can be reduced by increasing the EF service rate (ef_r). The positive coefficient of ef_r in Eq. (3.3) clearly indicates this and it agrees with our intuition.

Over-Provisioned (OP) EF-EDGE PHB with Background Traffic

Next, we investigate the effect of background (e.g., best-effort) traffic on the premium traffic for the EF PHB. As shown in Table 3.9, the throughput (BW) of the EF traffic is affected heavily by the number of background traffic flows sharing the link with it.

Since the PHB is over-provisioned, the effect of the ratio R_{ab} or the size of the back-

Response	Trans.	a_{pkt}	a_n	(a_{pkt},a_n)	Error
$1/J$	inverse	29.37%	18.27%	6.97%	31.78%

Table 3.10: ANOVA results for scenario 5

ground traffic itself is not dramatic. As a result of over-provisioning too, the percentage of variation due to experimental error is high. This is because of the weak interaction between the inputs and the output BW. The results also indicate that the priority scheduler does a good job in isolating the EF traffic from the background flows. There are no significant EF losses for the same reason above. Both delay and jitter values show very little experimental errors due to repetition, and strongly depend on the background traffic characteristics, such as the packet size, number of background flows, and their interactions. Note that a linear transformation fits jitter well with respect to the background traffic parameters, where a logarithmic transformation is appropriate for delay. This is different from what we found in Section 3.3.1 with EF-EDGE, where jitter is inversely related to the EF traffic parameters. Similar results are to be observed in Section 3.3.1.

EF-PRIO PHB without Background Traffic

The purpose of scenario 5 is to demonstrate how different implementations of a specific PHB can differ in their extracted input/output relationships. We present the results for jitter (J) as an example. The most important factors affecting J are listed in Table 3.10. Note that there is no effect due to a_r, unlike the scenario in Section 3.3.1 using EF-EDGE PHB.

The visual tests are shown in Fig. 3.10 and 3.11 to test the validity of the ANOVA model. The jitter model is given in Eq. (3.4) with a coefficient of determination (R^2) of 64%. The response surface is shown in Fig. 3.12. The inverse relationship also holds here with respect to the EF traffic parameters as mentioned earlier.

$$1/J = 727.74 - 810.85a_n - 748.17a_{pkt} + 425.13a_n^2 + 322.99a_{pkt}^2 + 189.18a_{pkt}a_n \,. \quad (3.4)$$

The negative coefficients for both a_n and a_{pkt} in the above model indicate that an increase in either of them will increase the value of jitter. A very important result can be

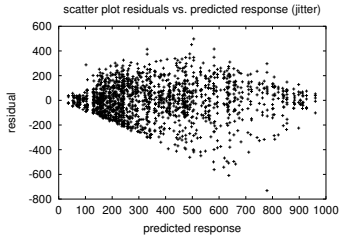


Figure 3.10: Residuals vs. predicted response for J (scenario 5)

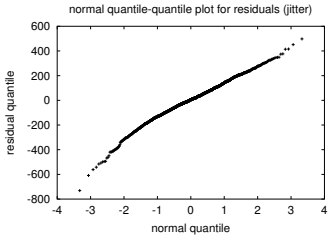


Figure 3.11: Normal Q-Q plot for J residuals (scenario 5)

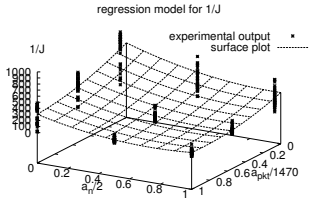


Figure 3.12: Response surface for J (scenario 5)

Res.	a_r	a_p	a_{pkt}	max_{th}	min_{th}	(a_r,a_p)	(a_r, max_{th})	(a_r, min_{th})	(a_p, max_{th})	(a_{pkt}, b_{pkt})	(a_r, a_p, max_{th})
BW	51.38%	12.51%	$\approx 0\%$	2.67%	2.38%	13.27%	3.27%	2%	2%	$\approx 0\%$	2.37%
D	38.2%	25.42%	2.85%	$\approx 0\%$	3.1%	25.06%	$\approx 0\%$	$\approx 0\%$	$\approx 0\%$	$\approx 0\%$	$\approx 0\%$
$1/J$	14.75%	17.35%	34.82%	$\approx 0\%$	$\approx 0\%$	18.94%	$\approx 0\%$	$\approx 0\%$	$\approx 0\%$	4.09%	$\approx 0\%$
L	29.32%	17.12%	$\approx 0\%$	7.14%	4.17%	16.98%	7.4%	3.89%	3.91%	$\approx 0\%$	4.03%

Table 3.11: ANOVA results for AF PHB

deduced accordingly — the larger the number of flows in an EF traffic, the larger the jitter.

Modeling AF PHB

We present results for the AF PHB (scenario 7) to show the generality of our approach and to provide an insight into the behavior of the AF PHB. The relevant PHQ attributes, according to the IETF standard, are throughput (BW) and loss (L). However, some applications may also impose delay and jitter requirements on AF-based services, so we present results for them as well. In this experiment, we mark the designated flow with AF11 DSCP; and the background traffic (filling up the rest of the network capacity) with AF12 and AF13 DSCPs. The input factors and their levels are listed in Table 3.5. Table 3.11 lists[10] the ANOVA results for the four QoS attributes. The results indicate that the throughput (BW) depends mostly on the average and peak sending rates, and their interactions. They also exhibit some dependency on the threshold levels of the AF11 virtual queue. In another experiment (not shown here), we find this dependency increases with the increase of the drop probability for the virtual queue.

[10]The error column is not shown due to space limitation.

	Factor (unit)	Scenario 8	Scenario 9	Scenario 10	Scenario 11	Scenario 12
Set I_a	a_r (Mbps)	0.5 $\longrightarrow$ 2	2 $\longrightarrow$ 10	1 (X)	1 (X)	1 (X)
	a_p (Mbps)	3 (X)	11 (X)	2 (X)	2 (X)	2 (X)
	a_{pkt} (bytes)	100 $\longrightarrow$ 1470	100 $\longrightarrow$ 1470	1000 (X)	1000 (X)	1000 (X)
	a_b (bytes)	20000 (X)	20000 (X)	20000 (X)	20000 (X)	20000 (X)
	a_n (flows)	1 $\longrightarrow$ 5	1 $\longrightarrow$ 5	1 (X)	1 (X)	1 (X)
Set C	ef_r (Mbps)	10 (X)	1 (X)	2 (X)	2 (X)	2 (X)
	be_r (Mbps)	80 (X)	80 (X)	100 (X)	5 $\longrightarrow$ 50	50 (X)
Set I_b	R_{ab} or b_r (Mbps)	70 (X)	40 (X)	0.02 $\longrightarrow$ 0.2	50 (X)	0.01 $\longrightarrow$ 0.02
	b_{pkt} (bytes)	1000 (X)	1000 (X)	100 $\longrightarrow$ 1470	100 $\longrightarrow$ 1470	100 $\longrightarrow$ 1470
	b_b (bytes)	100000 (X)	100000 (X)	100000 (X)	100000 (X)	100000 (X)
	b_n (flows)	1 (X)	1 (X)	1 $\longrightarrow$ 8	1 $\longrightarrow$ 8	1 $\longrightarrow$ 8

Table 3.12: Factors and their levels for EF-CBQ

	Factor (unit)	Scenario 13	Scenario 14	Scenario 15
Set I_a	a_r (Mbps)	0.1 $\longrightarrow$ 16	1 (X)	1 (X)
	a_p (Mbps)	20 (X)	2 (X)	2 (X)
	a_{pkt} (bytes)	100 $\longrightarrow$ 1470	1000 (X)	1000 (X)
	a_b (bytes)	100000 (X)	20000 (X)	20000 (X)
	a_n (flows)	1 $\longrightarrow$ 8	1 (X)	1 (X)
Set I_b	R_{ab} or b_r (Mbps)	50 (X)	0.0125 $\longrightarrow$ 0.2	0.01 (X)
	b_{pkt} (bytes)	1000 (X)	300 $\longrightarrow$ 1470	300 $\longrightarrow$ 1470
	b_b (bytes)	100000 (X)	100000 (X)	100000 (X)
	b_n (flows)	2 (X)	1 $\longrightarrow$ 6	1 $\longrightarrow$ 6

Table 3.13: Factors and their levels for EF-PRIO

The results for loss (L), delay (D), and jitter (J) show similar dependencies, although the percentages vary with each of the QoS parameters. We notice a small, but non-trivial, interaction between the packet size of the designated and that of the background traffic contributing to the jitter of the premium traffic.

3.3.2 Second Modeling Set

The levels of input factors (I_a, I_b and C) used in each experiment scenario are listed in Tables 3.12 and 3.13 for EF-CBQ and EF-PRIO, respectively. The values listed for background rate are for R_{ab} if it is less than 1.0; otherwise, the absolute values of b_r are listed instead (i.e., if the value listed in R_{ab} column is 10Mbps, then b_r equals to 10Mbps, not R_{ab}). We focus primarily on delay, D, and jitter, J.

Over-Provisioned (OP) EF-CBQ

In the first experiment scenario of this set (scenario 8), we characterize the EF-CBQ realization with respect to the factors in I_a, i.e., the premium traffic parameters themselves,

Response	Trans.	a_r	a_{pkt}	a_n	(a_r,a_n)	Error
$\sqrt{BW}$	square	38.40%	$\approx 0\%$	58.30%	3.30%	$\approx 0\%$
D	-	$\approx 0\%$	49.54%	44.53%	$\approx 0\%$	4.49%
$\log(J)$	log	$\approx 0\%$	$\approx 0\%$	94.02%	$\approx 0\%$	$\approx 0\%$

Table 3.14: ANOVA results for scenario 8

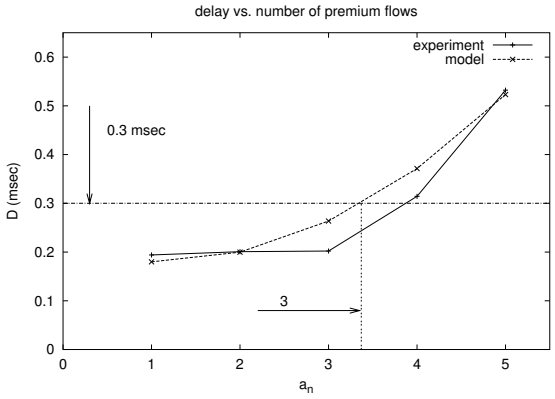


Figure 3.13: Model validation for D with a_n (scenario 8)

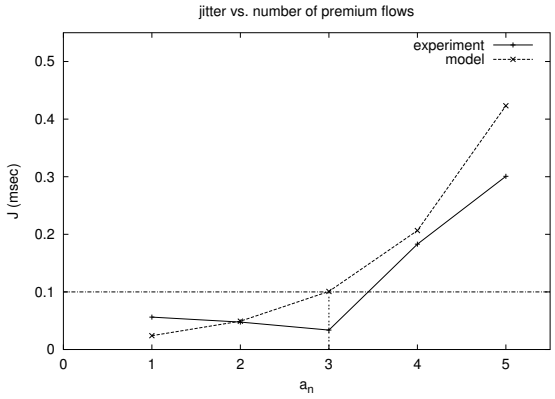


Figure 3.14: Model validation for J with a_n (scenario 8)

and its allocated service rate at the CBQ scheduler. This scenario studies the PHQ in case of $ef_r > (a_r \times a_n)$, i.e., over-provisioned PHB. In order to mimic a real situation, we also inject best-effort background traffic at a rate of 70 Mbps into the network with fixed parameters. The ANOVA results for this scenario is listed in Table 3.14. The transformation applied to each response variable, satisfying the basic assumptions of the linear ANOVA model, is indicated in the second column. The loss (L) in this experiment is basically 0 (no loss) since this is an over-provisioned case.

The regression models for the delay and jitter are given in Eqs. (3.5) and (3.6), with the coefficient of determination (R^2) of 90% and 76%, respectively.

$$D = 0.15 - 0.0028a_n + 0.00028a_{pkt} + 0.022a_n^2 + 2.2 * 10^{-8}a_{pkt}^2 + 6.2 * 10^{-6}a_{pkt}a_n \,. \quad (3.5)$$

$$\log(J) = -3.73 + 0.72a_n \,. \quad (3.6)$$

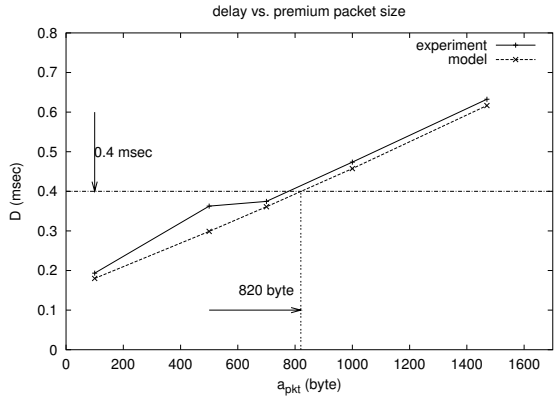


Figure 3.15: Model validation for D with a_{pkt} (scenario 8)

As shown in Table 3.14, the per-hop delay (D) depends mostly on the packet size (a_{pkt}) and the number of flows (a_n) of premium traffic, while the jitter is mostly affected by a_n. This is consistent with our intuition: per-hop delay depends on the time taken to send the packet, and is proportional to the packet size. Besides, both the delay and jitter depend on the number of flows sharing the same premium traffic queue, as packets have to wait for a longer time with a larger number of flows. We already pointed out to this in the first set of scenarios. To validate the extracted models, we conduct two experiments: one to validate the models with the number of premium traffic flows, and the other with its packet size. For the first experiment, we use a fixed packet size of 100 bytes while changing the number of flows from 1 to 5. Figure 3.13 shows the output delay calculated from the model against the experimental measurements for the same values of input a_n, and Figure 3.14 shows the same for the jitter. Clearly, the jitter model is not as accurate as the delay model due to a lower R^2. In the second experiment, we use a single premium flow, i.e., $a_n = 1$, and change the packet size from 100 to 1470 bytes. Figure 3.15 shows the model validation for delay. Section 3.4 will show how to use these models in limiting the per-hop delay and jitter, i.e., control of PHQ.

Under-Provisioned EF-CBQ

In this scenario, we set ef_r to be less than the total premium traffic rate ($a_r \times a_n$), and hence, we get an under-provisioned PHB. We also use a background stream sending at 40

Response	Trans.	a_r	a_{pkt}	a_n	(a_r,a_n)	(a_{pkt},a_n)	Error
D	-	$\approx 0\%$	98%	$\approx 0\%$	$\approx 0\%$	$\approx 0\%$	$\approx 0\%$
$\log(J)$	log	$\approx 0\%$	44.16%	13.38%	2.16%	35.82%	$\approx 0\%$
L	-	87.68%	2.86%	3.35%	$\approx 0\%$	$\approx 0\%$	$\approx 0\%$

Table 3.15: ANOVA results for scenario 9

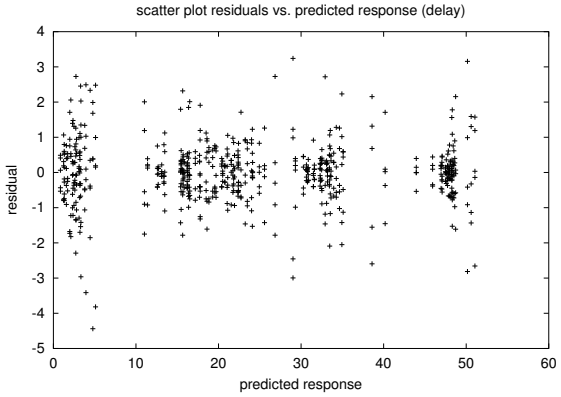


Figure 3.16: Residuals vs. predicted response for D (scenario 9)

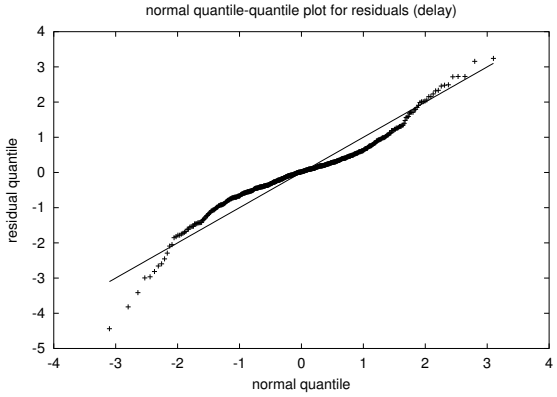


Figure 3.17: Normal Q-Q plot for D residuals (scenario 9)

Mbps to load the network. Table 3.15 shows the ANOVA results for this scenario, and we observe that loss L is significant as the arrival rate is greater than the service rate, and hence, overflow packets are dropped. The visual tests used in judging the correctness of the ANOVA delay models are shown in Figure 3.16 where the residuals-response scatter plot does not show any strong trend, and in Figure 3.17 where the normal Q-Q plot is almost linear.

The regression model for delay is given in Eq. (3.7) with coefficient of determination (R^2) of 98%. We validate the model against premium packet size (a_{pkt}) in Figure 3.18, where we keep a_r at 6 Mbps and use only one premium flow. The jitter model is given in Eq. (3.8) with R^2 of 86%, and its validation is plotted in Figure 3.19 against the number of flows (a_n) while keeping a_r at 2 Mbps and a_{pkt} at 100 bytes.

$$D = -0.77 + 0.033 a_{pkt} . \tag{3.7}$$

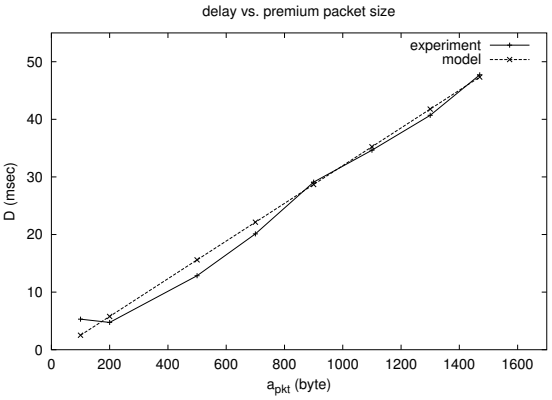


Figure 3.18: Model validation for D with a_{pkt} (scenario 9)

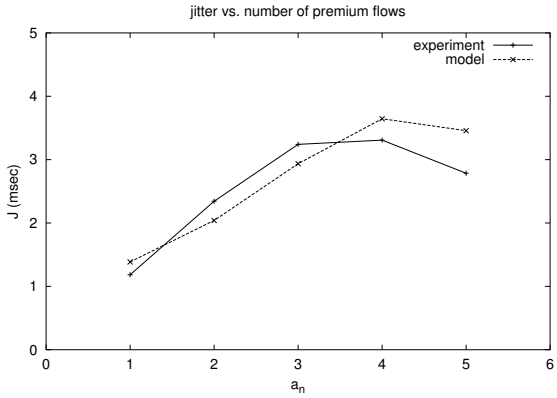


Figure 3.19: Model validation for J with a_n (scenario 9)

$$\begin{aligned}\log(J) &= -0.29 + 0.52a_n + 0.0039a_{pkt} + 1.6*10^{-7}a_r + 0.035a_n^2 - 0.0014a_n a_{pkt} \\ &\quad -1.8*10^{-8}a_n a_r - 1.7*10^{-6}a_{pkt}^2 - 1.8*10^{-10}a_{pkt}a_r - 1.3*10^{-14}a_r^2 \\ &\quad -0.02a_n^3 + 1.2*10^{-4}a_n^2 a_{pkt} + 1.4*10^{-9}a_n^2 a_r + 6*10^{-7}a_{pkt}^2 a_n \\ &\quad +3.2*10^{-14}a_{pkt}^2 a_r - 5.4*10^{-11}a_{pkt}^3 + 6.4*10^{-16}a_r^2 a_n \\ &\quad +7.1*10^{-18}a_r^2 a_{pkt} + 3.4*10^{-22}a_r^3 + 1.6*10^{-12}a_n a_{pkt} a_r \,. \end{aligned} \tag{3.8}$$

$$L = 6.83 + 4.15a_n + 1.8*10^{-5}a_r - 1.55a_n^2 - 1.04*10^{-12}a_r^2 + 9.31*10^{-8}a_n a_r \,. \tag{3.9}$$

We note here that the delay depends only on the premium packet size and shows no dependency on the number of flows. This is because, in the under-provisioned case, the load on the PHB is high enough so that packets are randomly dropped from different flows and the number of flows becomes insignificant for the packet waiting time in the queue. We also present the extracted loss model in Eq. (3.9) as this is the only scenario that has significant losses in the premium traffic (because of under-provisioning). From the ANOVA results, we find no dependency of loss on the premium packet size (a_{pkt}), and the model has a coefficient of determination (R^2) of 90%. From the model, we notice that loss increases with the increase of sending rate (due to the positive coefficient), and this is also consistent

Response	Trans.	b_{pkt}	b_n	R_{ab}	(b_{pkt},b_n)	(b_{pkt},R_{ab})	(b_n,R_{ab})	(b_{pkt},b_n,R_{ab})	Error
$1/D$	inverse	24.43%	3.63%	40.8%	$\approx$ 0%	15.27%	4.08%	4.41%	5.8%
$log(J)$	log	18.83%	15.97%	41.68%	6.22%	2.95%	5.06%	4.79%	4.5%

Table 3.16: ANOVA results for scenario 10

with our intuition.

EF-CBQ with Background Traffic—Not Saturated

To study the effects of the background traffic on the premium traffic, we present scenario 10 where there is a unsaturated (NOT-SAT) network node, i.e., best-effort allocated service rate is higher than the traffic sending rate. The factors and their values for the background traffic are listed in Table 3.12, while Table 3.16 contains the ANOVA results for the delay and jitter.

The polynomial regression model for the jitter is given in Eq. (3.10) with a coefficient of determination (R^2) of 87%. The comparative behavior of the model against the experimental results is plotted in Figure 3.20 with background traffic packet size (b_{pkt}) while using 6 background flows ($b_n = 6$) sending at a total rate of 25 Mbps ($R_{ab} = 0.04$). In Figure 3.21 we plot the jitter (model output as well as experimental results) with the number of flows (b_n) while using b_{pkt} of 1000 bytes and R_{ab} of 0.04. The jitter model is validated against R_{ab} in Figure 3.22, where we use 6 background flows ($b_n = 6$) sending with packet size (b_{pkt}) of 1000 bytes. We also validate the delay model (not shown) against R_{ab} in Figure 3.23 for the same scenario.

We notice from Figure 3.20 that the jitter is higher for small background traffic packet sizes. This holds true for the same background traffic sending rate, i.e., same b_r, and this is due to the way the CBQ implementation works in the router.[11] When the BE packet size is much smaller than EF packet size (1000 bytes in this scenario), the variability in delay due to waiting for BE packets is significantly higher. On the other hand, and according to our intuition, the jitter increases with the increasing number of BE flows due to the increasing variability in processing BE packets coming from different flows before EF packets. This is clear from Figure 3.21. These results show that, even in an over-provisioned and

[11] A variant of non-preemptive Weighted Round Robin (WRR).

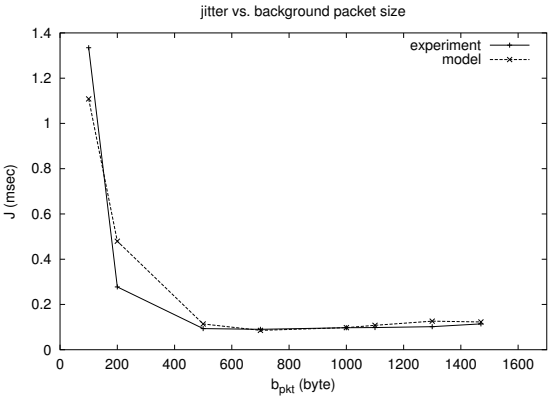


Figure 3.20: Model validation for J with b_{pkt} (scenario 10)

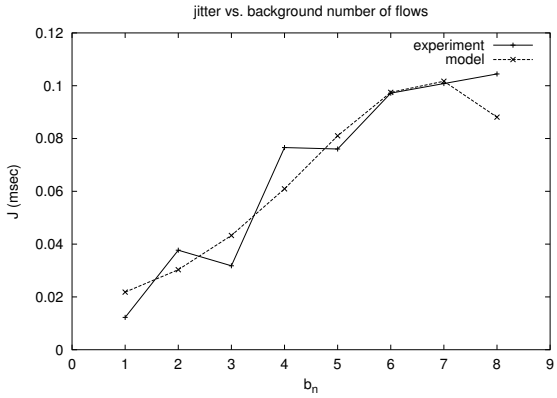


Figure 3.21: Model validation for J with b_n (scenario 10)

unsaturated network node, QoS metrics can vary and still need some control mechanism to stabilize their values.

$$\begin{aligned}\log(J) \quad = \quad & -2.12 - 52.7R_{ab} + 1.21b_n - 0.0077b_{pkt} + 400.1R_{ab}^2 - 7.45R_{ab}b_n \\ & +0.026R_{ab}b_{pkt} + 0.008b_n^2 - 0.001b_nb_{pkt} + 1.05 * 10^{-5}b_{pkt}^2 \\ & -993.82R_{ab}^3 + 11.96R_{ab}^2b_n - 0.036R_{ab}^2b_{pkt} + 0.336R_{ab}b_n^2 \\ & +3.57 * 10^{-5}b_n^2b_{pkt} - 0.0071b_n^3 - 1.02 * 10^{-5}R_{ab}b_{pkt}^2 \\ & +3.2 * 10^{-7}b_nb_{pkt}^2 - 3.7 * 10^{-9}b_{pkt}^3 + 5.85 * 10^{-5}R_{ab}b_nb_{pkt} \ . \end{aligned} \tag{3.10}$$

For delay, we experience a smaller range of change with the background sending rate as the network is not saturated. However, it is clear that for lower values of b_r (higher values of R_{ab}), the per-hop delay gets smaller.

EF-CBQ with Background Traffic—Saturated

When the background sending rate (b_r) is higher than the allocated rate (be_r) at the CBQ, we have a saturated node. We consider two experimental scenarios in this case; in the first (scenario 11), we change be_r with b_r fixed at a higher value all the time. We want to investigate the effects of controlling the PHQ by changing one of the node's configuration parameters, which is be_r. In the second scenario (scenario 12), we fix be_r and change b_r,

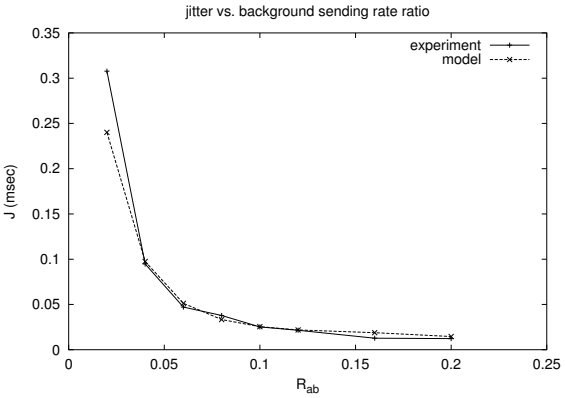


Figure 3.22: Model validation for J with R_{ab} (scenario 10)

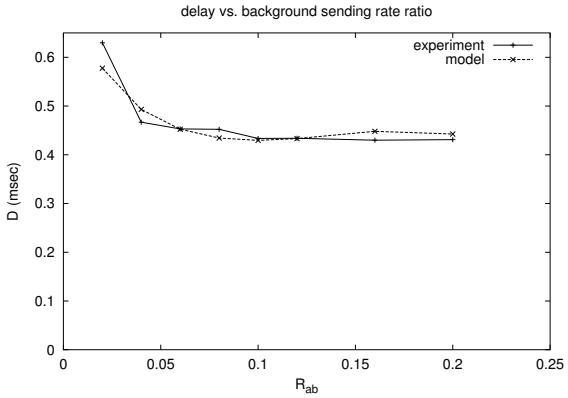


Figure 3.23: Model validation for D with R_{ab} (scenario 10)

or equivalently R_{ab}, within higher value ranges. In the first scenario, the jitter model, given in Eq. (3.11) with R^2 value of 71%, and also the ANOVA output (not shown), both suggest a strong dependency on be_r and b_{pkt} and a weak dependency on b_n. Therefore, we conduct validation experiments with respect to be_r and b_{pkt} only.

$$\begin{aligned} J \;=\; & 0.2 + 7.1*10^{-8} be_r + 0.089 b_n - 0.0023 b_{pkt} + 6.3*10^{-16} be_r^2 + 1.5*10^{-8} be_r b_n \\ & -4.9*10^{-11} be_r b_{pkt} + 0.0066 b_n^2 - 0.00032 b_n b_{pkt} + 4.1*10^{-6} b_{pkt}^2 \\ & -2.3*10^{-22} be_r^3 - 4.8*10^{-16} be_r^2 b_n - 9.1*10^{-18} be_r^2 b_{pkt} - 8.3*10^{-10} be_r b_n^2 \\ & +4.3*10^{-7} b_n^2 b_{pkt} - 0.0003 b_n^3 + 8.3*10^{-14} be_r b_{pkt}^2 + 1.7*10^{-7} b_n b_{pkt}^2 \\ & -1.9*10^{-9} b_{pkt}^3 - 2.7*10^{-12} be_r b_n b_{pkt} \; . \end{aligned} \tag{3.11}$$

In Figure 3.24, the calculated jitter is compared against the experimental output with b_n fixed at 4 flows, sending at 10 Mbps and changing background packet size from 100 to 1470 bytes. We see behavior similar to the NOT-SAT case in scenario 10 with small differences in sizes around 1000 bytes. We refer this difference to the default average packet size used in the CBQ configuration which is set to 1000 bytes. In Figure 3.25, on the other hand, we change be_r while fixing b_{pkt} at 1000 bytes and b_n at 4. This result is very important for controlling the per-hop jitter of the premium traffic by controlling the allocated rate for the BE traffic at the network nodes. We will investigate more on this result in Section 3.4.

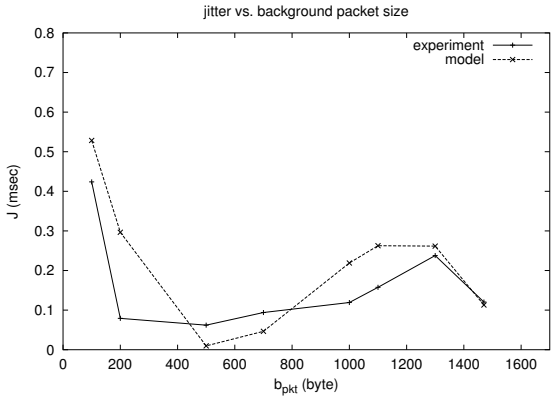


Figure 3.24: Model validation for J with b_{pkt} (scenario 11)

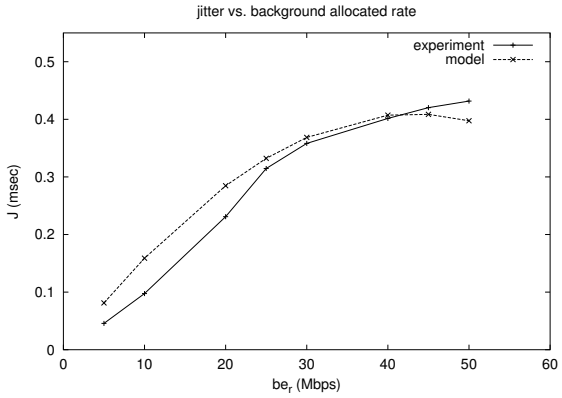


Figure 3.25: Model validation for J with be_r (scenario 11)

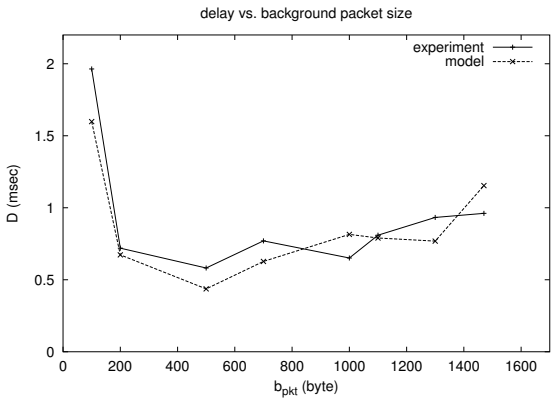


Figure 3.26: Model validation for D with b_{pkt} (scenario 12)

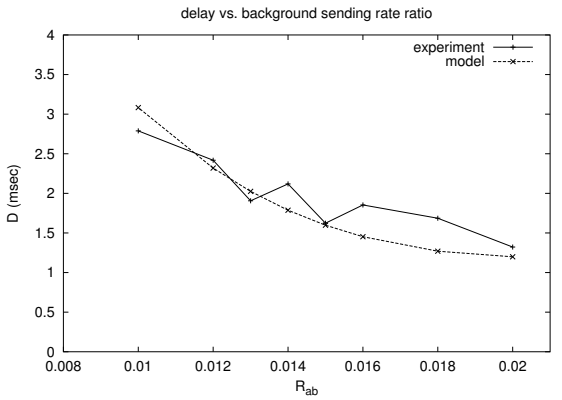


Figure 3.27: Model validation for D with R_{ab} (scenario 12)

In scenario 12, the delay model (not shown) is validated against b_{pkt} in Figure 3.26, where we keep R_{ab} at 0.016 and b_n at 6 flows, and against R_{ab} in Figure 3.27, where we set b_{pkt} at 100 bytes and b_n at 8 flows. From the first figure, we notice a slow increase of the delay with b_{pkt}, except for small packet sizes such as 100 bytes, where it causes a high delay for the premium traffic. We refer this to as the realization of the PHB itself since it has been observed in all other experiments as well. On the other hand, the delay can be reduced by lowering b_r, or equivalently, increasing R_{ab}. This follows our intuition that the smaller the background traffic, the lower the delay of premium traffic.

Response	Trans.	a_r	a_{pkt}	a_n	(a_r,a_{pkt})	(a_r,a_n)	(a_{pkt},a_n)	(a_r,a_{pkt},a_n)	Error
D	inverse	3.15%	76.72%	4.65%	5.64%	$\approx 0\%$	3.29%	$\approx 0\%$	4%
$\log(J)$	log	3.51%	$\approx 0\%$	59.46%	2.69%	6.43%	3.64%	15.51%	6.78%

Table 3.17: ANOVA results for scenario 13

EF-PRIO with Fixed Background Traffic

For the second realization of the EF PHB, which is based on priority scheduling, we conduct the 13th experimental scenario, which models the effects of the premium traffic parameters on the PHQ it receives. This is similar to what we did in scenarios 8 and 9, except that for absolute priority scheduling, there are no configuration parameters to change.[12] The ANOVA results are listed in Table 3.17 for both delay and jitter, i.e., the most significant response variables. To show the effect of the model's order on the accuracy of the model, we confirm that the higher the order, the more accurate the model becomes. From Table 3.17, we observe that the delay depends weakly on a_n and a_r. We build two delay models: the first is a 1^{st}-order model that incorporates a_{pkt} only, in Eq. (3.12) with $R^2 = 75\%$, while the second is a 3^{rd}-order model, in Eq. (3.13) with $R^2 = 95\%$, which incorporates all of the three factors.

$$1/D = 4.13 - 0.002a_{pkt}\,. \tag{3.12}$$

$$\begin{aligned} 1/D = {} & 6.13 - 0.67a_n - 0.0067a_{pkt} + 3.5*10^{-8}a_r + 0.09a_n^2 + 0.0006a_n a_{pkt} \\ & -3*10^{-8}a_n a_r + 2.7*10^{-6}a_{pkt}^2 + 1.69*10^{-10}a_{pkt}a_r - 9.6*10^{-15}a_r^2 \\ & -0.0015a_n^3 - 5.7*10^{-5}a_n^2 a_{pkt} + 1.1*10^{-9}a_n^2 a_r - 1.3*10^{-7}a_{pkt}^2 a_n \\ & -1.2*10^{-13}a_{pkt}^2 a_r - 1.9*10^{-10}a_{pkt}^3 + 4.2*10^{-16}a_r^2 a_n \\ & +3.1*10^{-18}a_r^2 a_{pkt} + 1.7*10^{-22}a_r^3 + 1.2*10^{-11}a_n a_{pkt} a_r\,. \end{aligned} \tag{3.13}$$

The difference of the two models is shown in Figure 3.28, along with the experimental output. Clearly, the 3^{rd}-order model is more accurate than the lower-order model. There

[12] This is the case only for our particular realization.

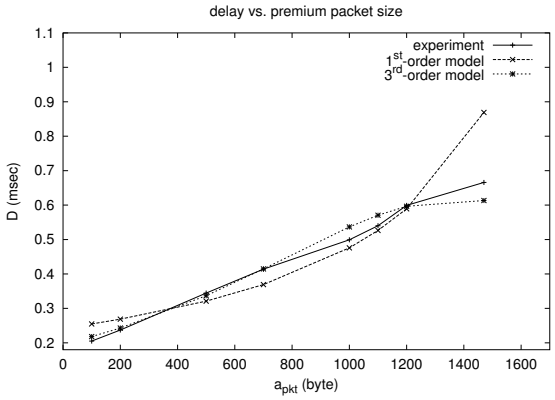


Figure 3.28: Model validation for D with a_{pkt} (scenario 13)

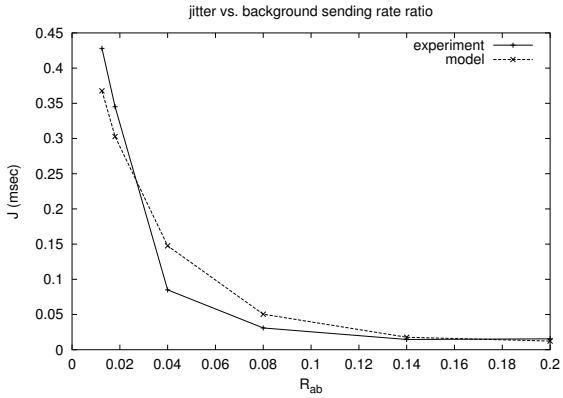


Figure 3.29: Model validation for J with R_{ab} (scenario 14)

is a tradeoff between the order of the model, hence the involved input factors, and the complexity of the model. This should be determined by the type of traffic expected to traverse the node under study.

EF-PRIO with Background Traffic—Not Saturated

In scenario 14, according to Figure 3.6, we investigate the effects of the background traffic parameters on the premium traffic PHQ of a priority-based node. We look at an unsaturated PHB, and extract a 2^{nd}-order model for the jitter in Eq. (3.14) with R^2 of 83%. The model is plotted with R_{ab} against the experimental output in Figure 3.29 with b_{pkt} of 1000 bytes and 6 background flows. It is also used as an example of control in Section 3.4.

$$log(J) \quad = \quad -3.27 - 21.3R_{ab} + 0.33b_n + 94R_{ab}^2 + 0.02b_n^2 - 2.8R_{ab}b_n \; . \tag{3.14}$$

EF-PRIO with Background Traffic—Saturated

We study the saturated case of Section 3.3.2 in scenario 15. Saturation here is relative to link capacity as there is no allocated rate to be configured in the priority-based node. The delay model is depicted in Eq. (3.15) with R^2 of 91% and is plotted against the experimental output with b_{pkt} in Figure 3.30 and with b_n in Figure 3.31.

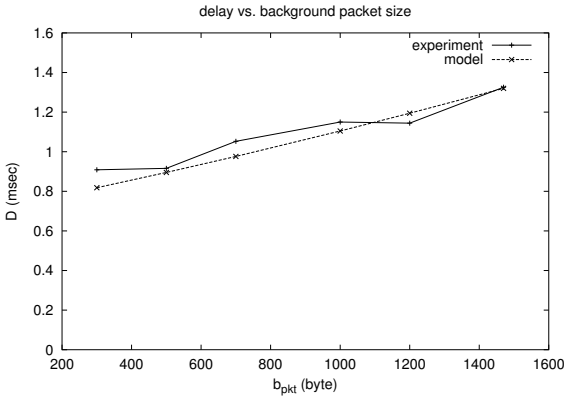


Figure 3.30: Model validation for D with b_{pkt} (scenario 15)

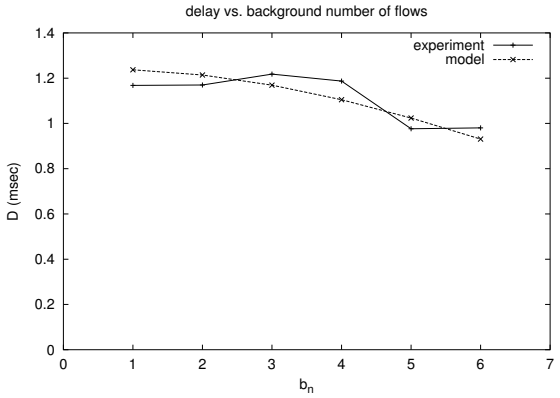


Figure 3.31: Model validation for D with b_n (scenario 15)

$$log(J) \quad = \quad -0.28 + 0.023b_n + 0.0005b_{pkt} - 0.009b_n^2 - 4.3 * 10^{-8}b_{pkt}^2 - 1.2 * 10^{-5}b_n b_{pkt} \; . \quad (3.15)$$

3.4 QoS Control

In this section, we put the extracted models in action and show how we can use them to control the PHQ of the afromentioned PHBs. We use the same experimental network shown in Figure 3.4.

3.4.1 Choosing I_a Parameters to Control Delay and Jitter

Here we show how to use the models extracted in Sections 3.3.2 and 3.3.2 in controlling the per-hop delay and jitter of the premium traffic. We assume that all other network parameters are within the range of our modeling experiments as mentioned before. In Figure 3.13, where the delay is plotted against the number of premium flows in an over-provisioned (OP) network, in order to achieve the delay ≤ 0.3 msec, the number of flows for the premium traffic should not exceed 3 flows according to the model. Similarly, keeping the number of flows below 4 will achieve jitter ≤ 0.1 msec from Figure 3.14. On the other hand, for the application to achieve per-hop delay ≤ 0.4 msec, it has to use packet

sizes $\leq$ 820 bytes, as indicated in Figure 3.15. This can be done using different encoding techniques within today's real-time applications. To achieve a delay less than 0.4 msec per node, both the number of flows and the packet size should be restricted to 3 and 820 bytes, respectively.

The results obtained from this scenario as well as from the ninth scenario, can be used as recommendations for applications to regulate their traffic to achieve certain PHQ. Of course, this has to be done in coordination with the network to compose the required overall e2e QoS. They can be used as per-hop data sheets that predict the PHQ for each hop along the path according to the available premium traffic characteristics. If the network is under-provisioned (UP), the models in Eqs. (3.7) and (3.8) and depicted in Figures 3.18 and 3.19 from the ninth scenario for delay and jitter, respectively, are used instead.

3.4.2 Using R_{ab} and b_n to Control Jitter

We demonstrate how to use the jitter model, developed in scenario 10 and validated in Figure 3.22, in PHQ control. Given similar network configurations such as the ones used in Section 3.3.2, we control b_r and hence R_{ab} to achieve the jitter of 0.3, 0.097 and 0.014 msec. Using the jitter model in Eq. (3.10), and substituting with these jitter values along with other configurations used in that scenario, we get the corresponding values of R_{ab} to be 0.02, 0.04, and 0.2, respectively.[13] We conduct an experiment with a single premium flow running at 1 Mbps with packet size of 1000 bytes and 6 background flows each with packet size of 1000 bytes. We also change R_{ab} from 0.02 to 0.04, and then to 0.2, each lasting for 20 seconds. The resulting premium jitter at the PHQ is plotted[14] against the experimentation time in Figure 3.32. This figure shows that using the models extracted in this chapter, the input background traffic can be controlled at runtime to provide predictable PHQ for premium traffic. The number of background traffic flows, b_n, can also be controlled (e.g., through admission control) to yield the desired per-hop jitter.

This is illustrated in Figure 3.33, where we attempt to achieve per-hop jitters of 0.0218, 0.061, and 0.101 msec. The corresponding values of b_n from the models in Eq. (3.10)

[13] We can also look up the validation graph to find these values.

[14] We removed strong outliers.

are 1, 4, and 7 flows, respectively. We plot the actual measured jitter as well as the reference values of the jitter mentioned above, while keeping each b_n value (of 20 seconds) unchanged.

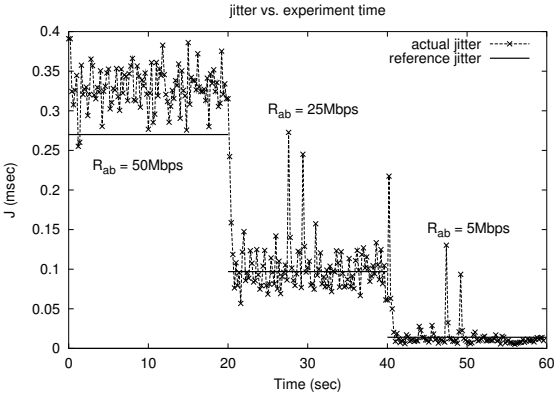


Figure 3.32: Controlled jitter by R_{ab} (scenario 10)

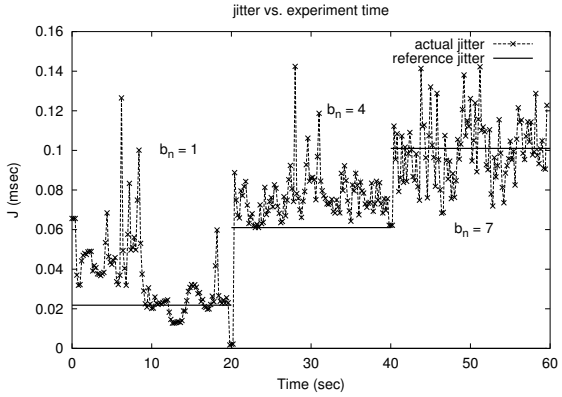


Figure 3.33: Controlled jitter by b_n (scenario 10)

3.4.3 Using be_r to Control Jitter

As depicted in Figure 3.1, not only the input traffic but also the configuration parameters of the PHB can be controlled to achieve specific PHQ values. Here we control the allocated service rate for the background traffic, be_r, to get the pre-specified values of per-hop jitter using the models developed in Section 3.3.2 (shown in Figure 3.25). We control be_r at three values, 5, 20, and 40 Mbps, calculated from the model, to give 0.081, 0.285, and 0.407 msec, respectively. We run each value for 20 sec, and measure the jitter of the premium traffic. We use a single premium flow running at 1 Mbps with packet size of 1000 bytes, and 4 background flows with the same packet size. The output jitter is plotted in Figure 3.34 with the experimentation time along with the reference values of jitter.

3.4.4 Delay and Jitter Control in EF-PRIO Scenarios

Using the priority-based scenarios, we show how to control the per-hop delay by choosing the right premium packet size, a_{pkt}. As before, we choose three values of delay, 0.215,

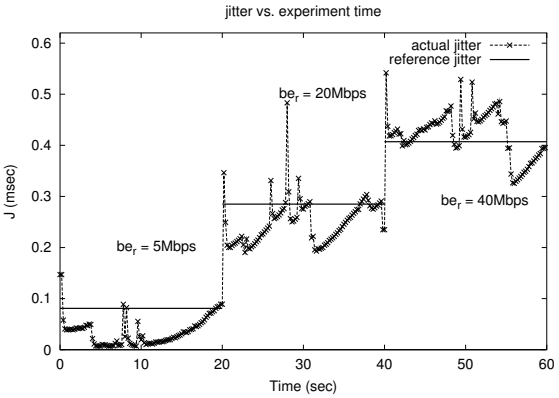


Figure 3.34: Controlled jitter by be_r (scenario 11)

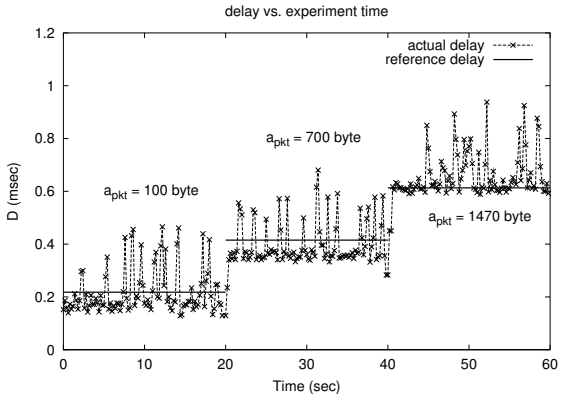


Figure 3.35: Controlled delay by a_{pkt} (scenario 13)

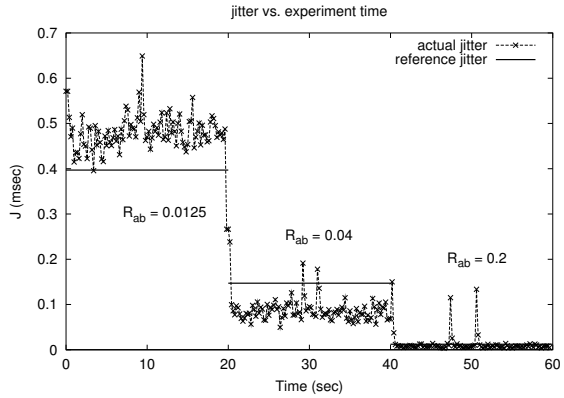


Figure 3.36: Controlled jitter by R_{ab} (scenario 14)

0.415, and 0.615 msec, and use the model in Eq. (3.12) to get the corresponding values of a_{pkt} to be 100, 700, and 1470 bytes, respectively. We compare the actual delay measurements with the reference values in Figure 3.35, where we plot the delay versus experimentation time, and each of the controlled packet sizes is held for 20 sec of the total time. This experiment corresponds to the one shown in Figure 3.28 with all other factors kept at the same values used before. Using scenario 14, the jitter can also be controlled by controlling R_{ab}. This is illustrated in Figure 3.36 where we use the values of 0.0125, 0.04, and 0.2 for R_{ab} calculated from the model in Eq. (3.14) to give jitter values of 0.367, 0.147, and 0.0122 msec, respectively.

3.4.5 Model-Based PHQ Controller

We have presented several examples for PHQ control through either limiting the premium traffic parameters, or controlling the background traffic as well as node configuration parameters. These examples demonstrate the efficacy of the model-based PHQ control scheme. A schematic of the model-based controller is illustrated in Figure 3.37. In this figure, the control action can be applied to either the premium traffic parameters as we have seen in Section 3.4.1, non-premium traffic parameters as in Section 3.4.2, or configuration parameters of the node as in Section 3.4.3.

Among these different control actions, we recommend the use of non-premium traffic parameters in control. This can be done by using traffic controllers that limit the amount of input non-premium traffic of the QoS subsystem (PHQ) to the values calculated from the models. In such a case, there is no need to limit the input premium traffic or to re-configure the parameters of the QoS subsystem which may be an unwanted task.

Although the simple model-based control approach works well, it has some limitations. First, during real-time control, the inputs have to stay within the modeling range; otherwise, the models used in control would not be valid anymore. Second, the scheme works best by changing one control parameter (factor) at a time with the others fixed. This is because the reverse calculation of the model equations, i.e., from a reference PHQ to input factors values, is complex and not possible for multiple factors calculation. Third, the scheme is suitable for controlling one output at a time, and this is a result of using a separate model equation for each output response. Controlling multiple outputs simultaneously within this scheme would be unwieldy and impractical. The last two points are referring to the necessity of studying the problem as a Multiple Inputs Multiple Outputs (MIMO), instead of Single Input Single Output (SISO) systems.

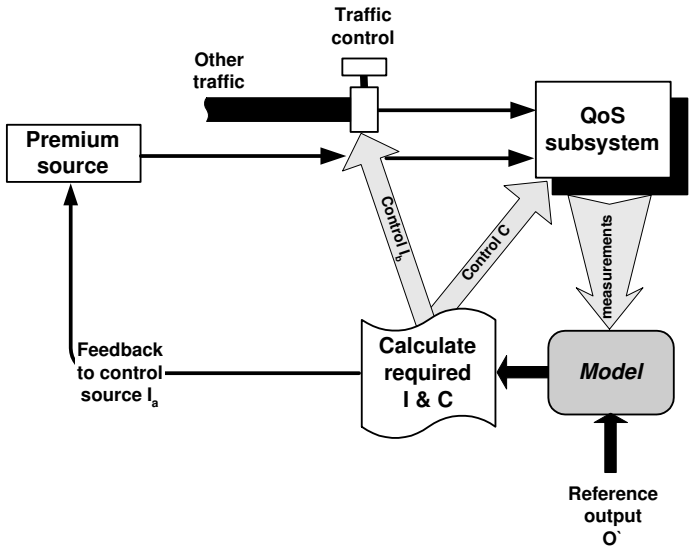


Figure 3.37: Model-based PHQ controller

3.5 Previous Work on the Use of Statistical Modeling in Networks

A very few previous studies reported the use of ANOVA for experimental analysis and modeling of IP QoS networks. For example, the authors in [106] applied a full factorial design and ANOVA to compare a number of marking schemes for TCP acknowledgments in a DiffServ network. The performance of AF PHB was also analyzed in [57] using ANOVA where the authors compared the performance of different techniques for bandwidth and buffer management. Our approach uses same statistical tools, but our goal is different from the above, i.e., modeling and control of the PHQ of any QoS-aware node. The idea of factor clustering has been mentioned before in [61], where the authors studied the congestion control in TCP Vegas. They clustered 10 congestion and flow control algorithms in TCP Vegas into 3 groups, according to the 3 phases of TCP protocol, and deduced which of the algorithms are the most effective in the overall operation which helps later reduce the complexity of realization. We use a similar approach in clustering our factors, but in the context of modeling. There have also been previous efforts in using statistical modeling techniques to extract models for computer systems to be used in the feedback control of these systems. The authors in [107], in designing a controller for the Lotus Notes server,

used a stochastic autoregressive modeling technique based on ARMA. It proved to be an effective technique for modeling without knowing the details of the system under control. To calculate the model parameters, they used the least-square regression estimator which is similar to our case except we deal with steady-state polynomial models. Another example of using statistical modeling (based on experimental characterization) and applying feedback control using these models was presented in [3]. The authors modeled a web server as a difference equation and estimated the model parameters using experiments.

On the other hand, the authors of [45, 46] utilized a *ring* network topology, which is similar to the one used in this chapter for experimental studies on the EF PHB [29]. Their findings can be summarized as follows. The EF burstiness was greatly affected by the number of EF streams and packet loss. Moreover, WFQ was found to be more immune to burstiness of traffic than priority queueing, when used for scheduling, but had less timely delivery guarantees. Our work complements these studies and provide a more rigorous way to identify these effects. The authors of [90] conducted an experimental analysis of the EF PHB by incorporating a part of the Internet2 QBone and using the QBone Premium Service (QPS). They used the same QoS metrics (throughput, delay, jitter and packet loss) as in our study. The difference, however, is that their metrics are measured as end-to-end quantities, while ours are per-hop quantities. Moreover, they did not provide any functional relationships or models for designing premium services over Internet2. A rigorous theoretical study to find probabilistic bounds for EF was presented in [126]. The authors used a combination of queueing theory and network calculus to obtain delay bounds for backlogs of heterogeneous traffic as well as traffic regulated by a leaky bucket. They also derived bounds on loss ratio under statistical multiplexing of EF input flows. Our approach is purely experimental; instead of providing bounds on delay or packet loss, we seek to identify the parameters necessary to construct simple performance models of PHQ mechanisms, and eventually to control their run-time behavior.

3.6 Concluding Remarks

In this chapter we presented a novel approach for PHQ control based on design of experiments and statistical modeling. We used ANOVA and regression analysis to extract the required models used in steady-state control, and verified the thus-developed models against the actual behavior of the PHQ. We were able to find strong functional relationships between input traffic and output QoS for the PHQ. We pointed out operational differences among the different PHB implementations, and explained these differences based on the internal structure of the PHBs. The realization of measurement and experimental design is automated – the configuration of the PHBs and the measurement of the corresponding output parameters are automatically performed – which is important to the success of the approach.

The proposed approach has two important features. First, it is a general modeling approach, that can be applied on a variety of QoS components along the network, not only on per-hop QoS. As long as the QoS component can be modeled as a system with input traffic and configuration parameters, and output QoS, then our framework can be used to get the functional relationships required for the operation. Second, it is simple and does not involve complex mathematical analysis and relaxing assumptions. Moreover, it is a convenient vehicle for modeling systems with a small number of input/output parameters. It can be extended to more complex systems, e.g., edge-to-edge QoS building blocks such as a Per-Domain Behavior (PDB) in DiffServ [97], or LSP in MPLS [110]. Most importantly, we demonstrated the effectiveness of using these models in PHQ control.

As pointed out in Section 3.4.5, there are some limitations in our model-based control scheme. In Chapter 4, we use dynamic feedback control within a control-theoretic approach to address them. We will show how to build MIMO models to control both delay and jitter at the same time. We will also show the dynamics of the control algorithm in tracking reference QoS and extend the analysis to the multi-node case and hence, an edge-to-edge or even end-to-end network services.

CHAPTER 4

CONNET: Self-Controlled Network Services for Delay and Jitter Requirements

In first-in-first-out (FIFO) networks, such as the current Internet, and with the absence of any QoS architecture such as the DiffServ or the IntServ, it is still a challenging task to provide timely packet delivery for time-sensitive and real-time applications. Without protecting their traffic, the performance of these applications will likely be unpredictable with respect to delay and jitter because of different traffic crossing the network. To accommodate network unpredictability, many of the delay- and jitter-sensitive applications use a limited form of adaptivity, such as changing their traffic parameters according to the availability of network resources, or they use extra buffering to deal with transient congestion. This approach sometimes provides acceptable, but limited, performance for the above-mentioned applications, especially in case of large amounts of competing traffic at bottleneck network segments. In the absence of smart bandwidth management, applications usually have limited control over the network performance. For this reason, application adaptivity fails when the traffic distribution across the network exceeds a certain limit, especially at bottleneck links. Therefore, a better-informed, network-centric approach is required, where *controllable* network services can support *premium* traffic from delay- and jitter-sensitive applications. These network services can be deployed within a single domain, or across multiple domains.

In this chapter, we present a new scheme, called CONNET (CONtrolled NETwork), for

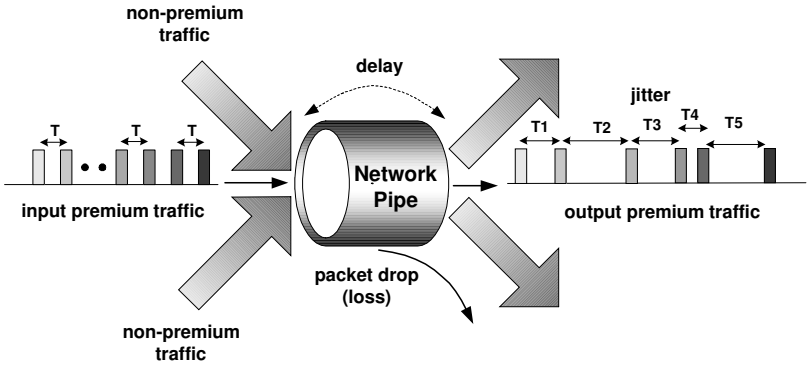


Figure 4.1: Delay and jitter across a network pipe

providing controllable network services for delay- and jitter-sensitive *premium* traffic. It is based on a simple, yet intelligent bandwidth control for traffic going through these network services. Consider the network pipe[1] shown in Figure 4.1, which carries two classes of traffic, premium and non-premium. Assuming static link capacity and FIFO queueing (used in the current Internet), the delay and jitter, as well as the throughput, of premium traffic will depend directly on the amount of non-premium traffic sharing the network pipe with it. This is because, for single FIFO-queue networks, non-premium packets in front of premium packets cause them to experience variable queueing and service delays.

Based on this observation, we propose to achieve desired delays and jitters for premium traffic by controlling non-premium traffic. We first find a model for the relationship between the amount of non-premium traffic and the premium traffic's delay and jitter. We then use this model in controlling the competing non-premium traffic to achieve the required premium delay and/or jitter. We take a control-theoretic — specifically model-based control — approach to the network service control problem, and present a full design and analysis of robust estimation and control.

Our contributions are two-fold: (i) development of a *reservation-less* approach for delay and jitter control that does not require prior knowledge of input traffic specification, and works well with currently-prevalent FIFO routers, and (ii) creating dynamic as well as *self-controlled* network services that react to changing traffic loads while maintaining high link utilization. We evaluate this scheme using an implementation on a real testbed net-

[1] A network pipe may represent a single or multiple hops.

work, demonstrating the correctness, accuracy, robustness, and fault-tolerance of the thus-designed controller. Its performance is compared against other well-known schemes, such as classed-based queueing (CBQ) [50] and weighted-fair queueing (WFQ) [30]. Based on this evaluation, we found CONNET to make a significant improvement (more than 40% in some cases) in preserving low delay and jitter for premium traffic.

4.1 Rationale and Main Features of CONNET

Before delving into the design, implementation and evaluation of our network service control, we describe the rationale, and outline the main theme, of CONNET. As mentioned earlier, CONNET provides two important salient features, making it more appealing than other QoS or bandwidth-management frameworks whose deployment has been hindered by their complexity and configuration burden. These features are detailed next.

4.1.1 Reservation-less Delay and Jitter Control

The traditional approach in providing network QoS is to calculate and provision *a priori* the required network resources (e.g., link bandwidth, processing power and buffer at routers) to ensure that the network can meet certain QoS requirements [16, 124]. CONNET requires neither advance resource reservation nor *a priori* traffic parameterization. In other words, it uses a reservation-less technique to control delay or jitter. Furthermore, it does not assume any special queueing discipline other than single-FIFO-queue routers, unlike common scheduling algorithms that serve multiple queues for different traffic classes. This adheres to the basic principles of the Internet: simplicity and best-effort [26]. Most routers in the current Internet are either FIFO or configured to be FIFO.[2]

For a typical FIFO network path with two types of input traffic, premium and non-premium, sharing the same FIFO queue as shown in Figure 4.1, when we vary the rate of non-premium traffic while keeping all other configurations fixed, and plot both the measured premium delay (d) and jitter (j) along with the rate of non-premium traffic (R), we

[2]Multiple queueing disciplines are usually turned off because of their configuration complexity.

get a relationship similar to the one in Figure 4.2(a).[3] Increasing the rate of non-premium traffic is found to increase the premium traffic delay and jitter, and vice versa. This figure can also be interpreted as: "one can effectively control the delay and jitter of premium traffic by simply controlling the rate of non-premium traffic."

The relationship in Figure 4.2(a) represents the basis for our reservation-less control as well as the "physics" of the system under study. This figure is also considered as the "characteristic curve" for the network subsystem under study. It determines the feasible delay and jitter under the current configuration and with the available range of non-premium traffic rate.

This relationship has also been captured in [28], where Cruz considered the case of first-come-first-served (FCFS) multiplexers with two input links (or traffic classes) and one output link, shown in Figure 4.3(a), as an example application of delay calculus theory. If we call one input class *premium* and the other *non-premium*, then the maximum delay for the premium traffic is governed by the following relationship:

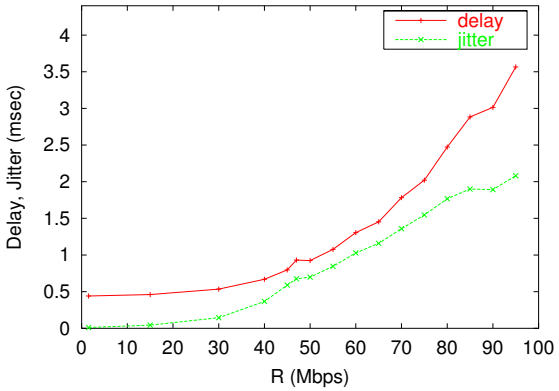
$$D_{max,p} = \frac{1}{C}\min\{b_p + r_p(\frac{b_{np}}{C-R}) + (C-r_p)\frac{L}{C}, \\ b_{np} + R(\frac{b_p}{C-r_p}) + R\frac{L}{C}\} \tag{4.1}$$

where $D_{max,p}$ is the maximum delay for the premium traffic, C is the link capacity,[4] r_p and b_p are the arrival rate and the burstiness of the premium traffic while R and b_{np} are the arrival rate and the burstiness of the non-premium traffic, and L is the packet size for all traffic. We observe that the delay increases with the increase of the non-premium traffic (R), and if we evaluate Eq. (4.1) with respect to R while fixing all the other parameters, we get a behavior like the one depicted in Figure 4.3(b), which is showing the same trend in Figure 4.2(a).

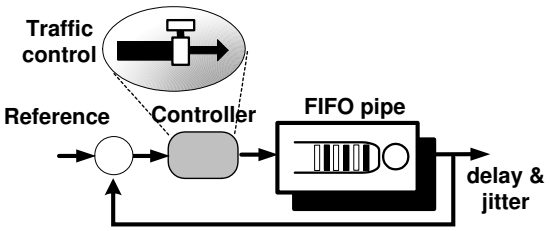
We control the rate (R) of non-premium traffic sharing the network resources with the premium traffic in order to achieve the required delay and jitter for premium traffic. This is illustrated in Figure 4.2(b). Note that R is not the only factor affecting the premium traffic

[3] Measurements were averaged over the experiment time to remove the transients.

[4] All links have the same capacity.

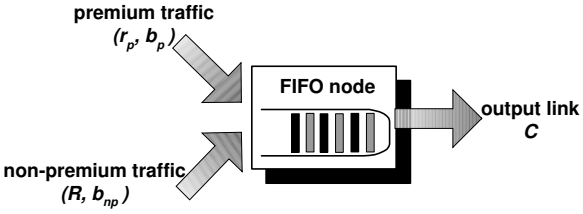


(a) Effect of non-premium rate, R, on premium delay and jitter

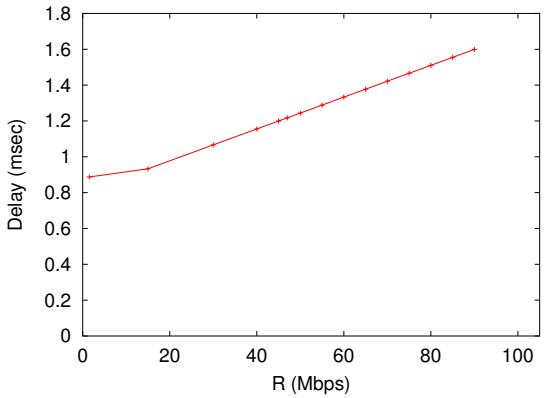


(b) FIFO network control

Figure 4.2: Control criterion



(a) Two input multiplexer node



(b) Premium delay depends on non-premium traffic rate

Figure 4.3: FIFO multiplexer with two input traffic

delay and jitter. However, Per-Hop QoS control in Chapter 3 has shown that it has the greatest effect, and hence, suffices to be the control input.

4.1.2 Dynamic and Self-Controlled Services

For better resource utilization, CONNET employs *feedback control* that yields faster response to traffic changes. It adapts to traffic dynamics through a feedback loop that takes the difference between actual and required (reference) delay/jitter and adjusts the control action accordingly. This creates self-controlled network paths that do not require the operator's intervention to tune their performance each time the network workload changes. The network operator only needs to set references for the output, and then, the feedback controller will be able to track these values. Reference tracking also works with variable

references, meaning that the operator can provide a reference signal reflecting the scenario of operation during a certain time interval (time-of-day or day-of-week). This would not be easy with operator-controlled links or other commonly-used techniques.

4.1.3 A Control-Theoretic Approach to Network Service Control

We now outline the approach we take to achieve the above-stated goals of CONNET. The resemblance of this network control problem to fluid mechanic problems motivated us to investigate the use of a control-theoretic approach to designing the control algorithm for the delay and jitter across the network service. Following this approach, we first obtain a model for the relationship between the amount of non-premium traffic, in terms of average bit rate, and the premium delay and jitter, and then use *state-space* [53] methods in the controller design. State-space is recommended for digital control as in our network problem, and also for multiple-input-multiple-output (MIMO) systems [54]. It is also an attractive design method as it consists of a sequence of independent steps that can be executed separately. This is important when we need to change the systems parameters and adjust the control loop.

We use a Linear Quadratic Regulator (LQR) controller along with Kalman filtering as a combination of robust estimation and control [54]. The main advantage of following this systematic approach is the generality of control design. We will study single- and multiple-node network pipes using the same approach, which exemplifies its generality. In what follows, we describe the steps taken for control design and evaluate the performance of the self-controlled network services using both simulation and experimentation on a network testbed.

4.2 Three Types of Self-Controlled Network Services

CONNET can be used in different places in the network where we need delay and jitter guarantees by just controlling the interfering traffic. We present three examples of CONNET in three different places in network: (i) a self-controlled pipe within a net-

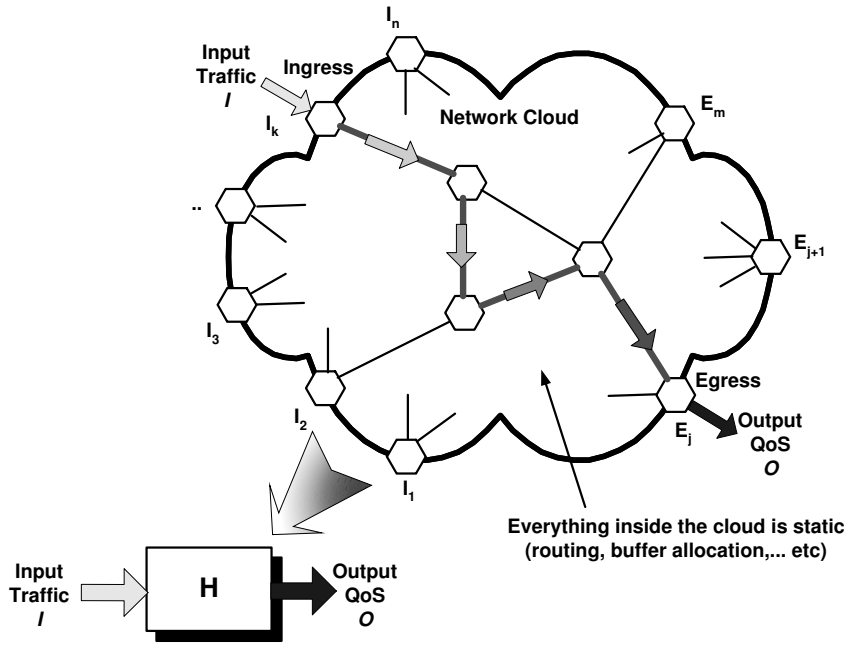


Figure 4.4: Edge-to-edge pipe in a network cloud

work cloud (edge-to-edge), (ii) self-controlled access links, and (iii) delay-controlled active queue management.

4.2.1 Self-Controlled Network Pipes in a Network Cloud

Providing predictable *edge-to-edge* or per-domain[5] services has been discussed before in many contexts such as in IntServ [16] RSVP, in DiffServ Per-Domain Behavior (PDB) [69, 97], in MPLS Label Switched Path (LSP) [40, 110], or more generally Virtual Private Networks (VPNs). The idea behind these services is to enable building predictable end-to-end services by just concatenating per-domain services. We show here how to use CONNET to build edge-to-edge services with delay and jitter guarantees by just controlling the input traffic at the domain ingress. In this way, we create a form of "core-stateless" delay and jitter scheme that requires the minimal change to today's Internet. We illustrate the idea in Figure 4.4 where we look at traffic entering the domain at a certain ingress point I_k and leaving at an egress point E_j. The routers inside the domain are FIFO routers and we assume the following:

[5]We use network cloud and network domain interchangeably.

- The traffic matrices of all incoming and outgoing traffic into and from the network cloud, respectively, are known. This includes traffic specifications at ingress points and traffic paths inside the network cloud.

- The configuration of the network cloud does not change during the course of analysis. This includes routing, buffer allocation, etc.

- Link capacities and resource allocation are known inside the network cloud and they do not change as well.

Then, as shown in the figure, we can abstract the per-domain service in terms of input traffic and the measured delay and jitter across the domain. We will illustrate how to realize this case in Section 4.5.1.

4.2.2 Self-Controlled Access Links

Access links are typically the bottleneck between a high bandwidth LAN and a high bandwidth IP network. Currently, customer networks employ high bandwidth technology ranging from 10/100 Mbps Ethernet to optical links, such as OC3 (155 Mbps) and OC12 (622 Mbps), to even Gigabit Ethernet. On the other hand, Internet Service Provider (ISP) networks, or more generally IP backbones, are usually equipped with even higher capacity using OC48 (2.4 Gbps), OC192 (10 Gbps) or 1 Tbps WDM fibers [78]. Access links (e.g., cable, xDSL, T1) between customer networks and their ISPs are still at most a few megabits per second. This creates a bottleneck between these two high-bandwidth networks, and hence, makes a strong negative impact on the performance of delay- and jitter-sensitive applications at medium- and high-utilization levels of access links [120, 125, 130]. Moreover, with the current Internet's first-in-first-out (FIFO) access links/routers there is not much to do in order to protect time-sensitive traffic from being delayed and jittered at these bottlenecks.[6] Large background HTTP downloads can cause an annoying delay to an interactive or a real-time streaming application running through the same network. Without *a priori* resource provisioning or reservation, this tends to have a serious effect on premium

[6]Note that this situation is found at both sender and receiver sides.

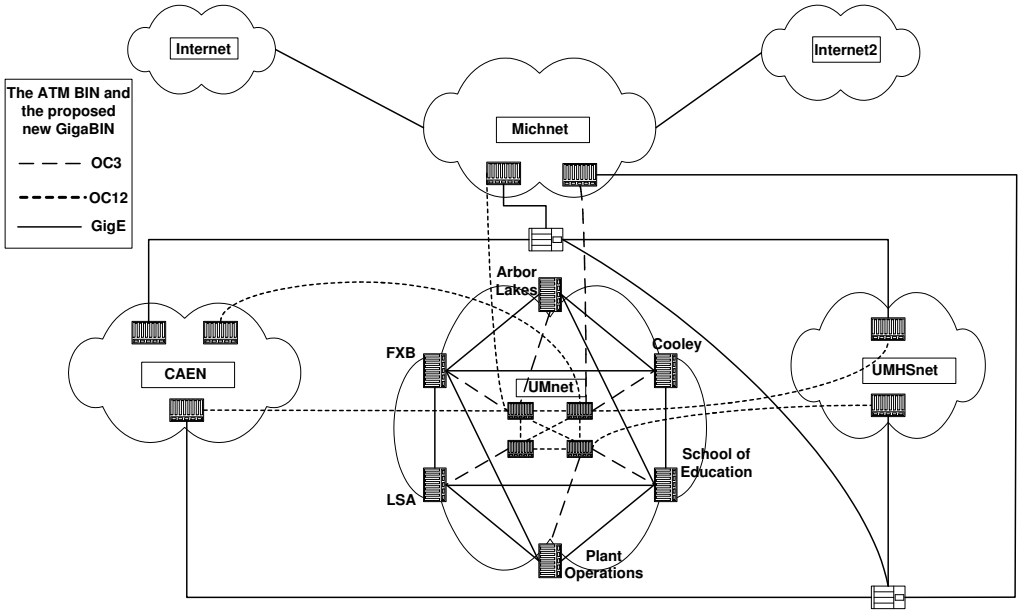


Figure 4.5: The University of Michigan campus network topology

traffic from these applications. Static resource reservation is denounced for poor resource utilization and lack of efficient handling of traffic dynamics.

To develop a feel for the need of access link control, we consider our campus network, shown in Figure 4.5. From the EECS Department, we used `traceroute` to `www.google.com`, and found it is taking 5 hops to reach `mich.net`, the regional ISP for the University of Michigan, at access router AA1 (`ge-1-1-0x984.aa1.mich.ne`). Specifically, we obtained the following result:

```
> 1  eecs2n-gw 0.488 ms
> 2  141.213.127.37 0.608 ms
> 3  caen-bin.r-bin-seb.umnet.umich.edu 0.711 ms
> 4  bin-arb.r-bin-arb.umnet.umich.edu  0.842 ms
> 5  ge-1-1-0x984.aa1.mich.net 0.922 ms
> 6  ge-1-2-0x25.nl-chi3.mich.net  7.138 ms
> 7  198.110.131.78  7.717 ms
```

Within each segment LAN environment, there is enough bandwidth—e.g., the link speed of CAEN (Computer-Aided Engineering Network) varies between OC12 and OC48,

hence a small delay. However, the delay increases significantly ($\sim$ 7 ms) at the 6-th hop between access router AA1 and the next hop in `mich.net`. This indicates that the access link (equivalently, the access router) has a significant percentage of the end-to-end delay and must, therefore, be controlled in order to achieve overall predictable delay and jitter performances. We show how to use the same idea of CONNET to provide delay- and jitter-controlled access links [32], and we illustrate the realization of this control in Section 4.5.2.

4.2.3 DQM: Delay-Controlled Active Queue Management

Current Active Queue Management (AQM) techniques are mainly concerned with bandwidth requirements of the Internet traffic. This does not suffice for certain current and future applications that require network nodes to be aware of delay and jitter in addition to bandwidth. With the advent of many delay-sensitive applications on the Internet such as Voice-over-IP (VoIP) (e.g., IP telephony), high-definition TV (HDTV), on-line gaming, and home/office remote control, future IP networks must meet a new requirement for *delay-aware* queue management.

Existing approaches to Active Queue Management (AQM) have mainly been using either the average queue length (as in RED [49]) or packet loss (as in BLUE [42]) as the indicator for network congestion. Although this is considered as a direct measure for network overload and hence longer queues, it does not reflect the actual delay of traffic while passing through network routers. Per-node delay can vary significantly even when the queue length is still moderate. Moreover, in first-in-first-out (FIFO) networks such as the current Internet, different packet sizes from different traffic aggregates can have a significant effect on the delay behavior of each network node and might not be captured by the existing queue-management approaches.

Figures 4.6 illustrates our way of applying CONNET inside a network node to realize DQM. Both the delay/jitter feedback and the control signal are applied inside the node. The premium traffic is stamped with TS_i at the ingress interface and with TS_o at the egress interface. The delay, $TS_o - TS_i$, as well as the delay variation (jitter) are used in the feed-

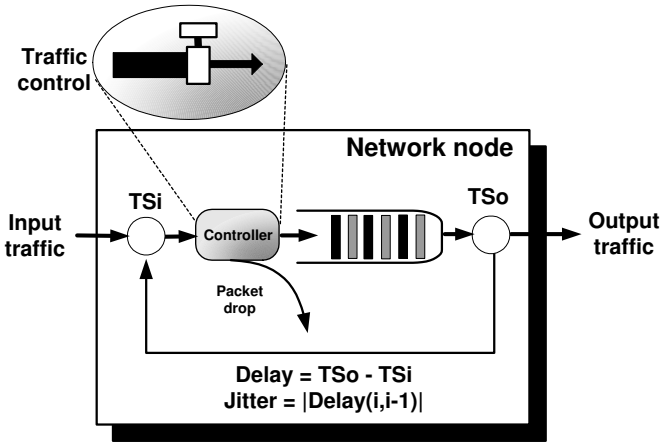


Figure 4.6: Internal control of a FIFO network node

back loop. At the same time, traffic control is applied to non-premium traffic using a Token Bucket Filter (TBF) inside the node itself. We present further details on the DQM implementation in Section 4.5.3.

4.3 System Modeling and Validation

We are mainly interested in a "black-box" model that describes the system behavior in terms of its inputs and outputs only, and this process is called *system identification*. This can be viewed as a dynamic extension of curve fitting. Models developed using this approach have some uncertainties/errors that are acceptable as long as the robustness of the overall control system is ensured. Despite their limited validity and working range, such models are relatively easy to obtain and use. Since we use the state-space method for the design and analysis of the network-control algorithm, we need a state-space model for the system under study. We use a state-space system identification method called the *subspace modeling* [101], which extracts a system model from traces of its inputs and outputs. A discrete-time "state innovation" model

$$\begin{aligned} \mathbf{x}_{k+1} &= \mathbf{A}\mathbf{x}_k + \mathbf{B}R_k + \mathbf{G}\mathbf{e}_k \\ [d_k, j_k]^T &= \mathbf{C}\mathbf{x}_k + \mathbf{D}R_k + \mathbf{e}_k \\ \mathbf{V} &= E(\mathbf{e}_p\mathbf{e}_q) = \mathrm{cov}(\mathbf{e}_p) \end{aligned} \tag{4.2}$$

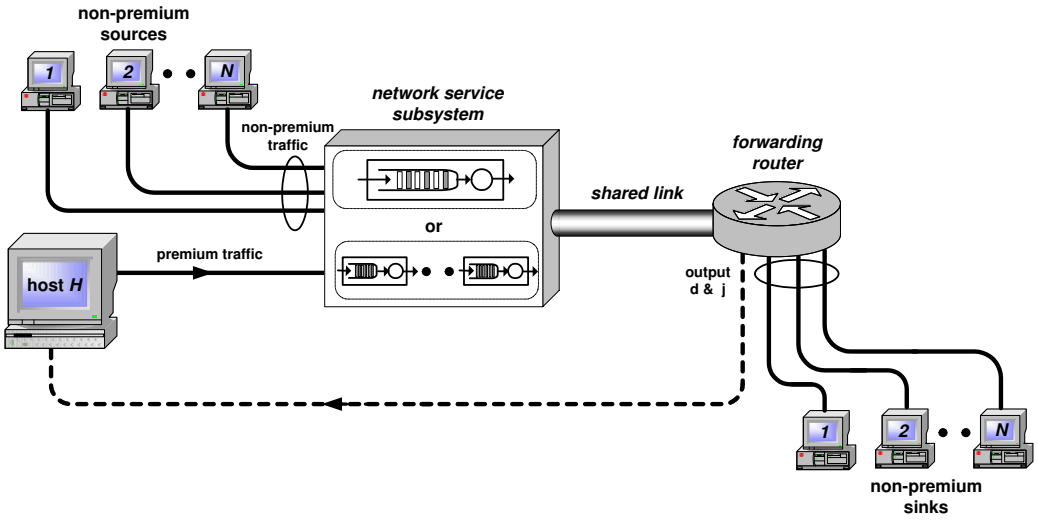


Figure 4.7: Network setup

is the output of the subspace modeling, where the input is $u_k = R_k$, and the output is $\mathbf{y}_k = [d_k, j_k]^T$. The state vector, $\mathbf{x}_k$, consists of variables that describe the internal condition of the network subsystem. For example, the state variables could be the length of the queue inside the network service node(s) and the rate of change of this length. For any dynamic system, there exists an infinite number of choices of state variables. System identification does not necessarily select state variables with physical meaning. Therefore, we will not be able to assign a particular meaning to $\mathbf{x}_k$. Only the input and output vectors, $\mathbf{u}_k$ and $\mathbf{y}_k$, will have well-defined physical meanings in this chapter. The subspace system identification provides the system matrices, **A**, **B**, **C**, and **D**, that describe how the input affects the state variables and the output. Moreover, the subspace modeling quantifies the the measurement error, $\mathbf{e}_k$, and system noise, $\mathbf{G}\mathbf{e}_k$ in terms of the covariance (cov) matrix of the measurement error, **V**. Subspace modeling allows us to specify a particular, as well as a recommended, system order n.[7] We also denote the number of inputs to the system as m and the number of outputs as ℓ. In this chapter we focus on systems with one input, the non-premium traffic bit rate (R) and two outputs, the delay (d) and jitter (j) of premium traffic.

[7] By calculating the singular values of the system.

4.3.1 Experimental Network Setup

We use the experimental network testbed shown in Figure 4.7 for both modeling (i.e., calculating the above system matrices) and evaluation. It consists of Linux-based software routers and end-hosts. The traffic controls built into the Linux kernel [5] enable construction of FIFO queues as well as traffic regulators that are used to enforce the control signal. A fast Ethernet-based ring network topology, with link capacity of 100 Mbps, is used so that the one-way delay may be measured without sophisticated (and sometimes inaccurate) time synchronization such as NTP or GPS. The ring topology was built using Linux *iptables* installed on the forwarding router. Premium traffic is generated at host H, going through the network service pipe (consisting of single or multiple routers) under study, then the forwarding router, and finally, terminates at host H again (i.e., a ring topology). Hosts 1 to N are used to generate non-premium traffic that shares the links and FIFO router(s) with the premium traffic. We use multiple non-premium sources to mimic a real network where traffic comes from multiple subnets sharing the same network service. All measurements, analysis, modeling and control calculations are done on host H.

We use "non-responsive" UDP traffic sources that can be instructed to generate traffic according to a specific input signal chosen based on a given experiment scenario regardless of the losses or delays at the bottleneck. This allows us to control the exact non-premium rate without interference from either congestion- or flow-control mechanisms. However, in Section 4.6, we evaluate CONNET with TCP traffic as well and show its effectiveness even with the TCP congestion control algorithm.

4.3.2 Model Extraction

In the first phase of experiments, our main goal is to calculate the system matrices of the model in Eq. (4.2). The input signal used for modeling has to cover most of the operational range of the system under study and avoid saturation regions. Saturation regions are governed by the link capacity between network nodes, which is 100 Mbps each. Besides, it should contain enough frequencies to excite most of the system dynamic modes and resembles the actual network workload expected in a real deployment. We use an input signal

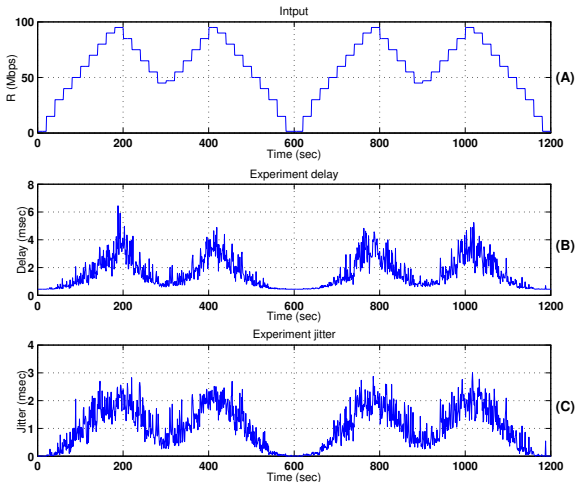


Figure 4.8: Input and output signals applied for modeling

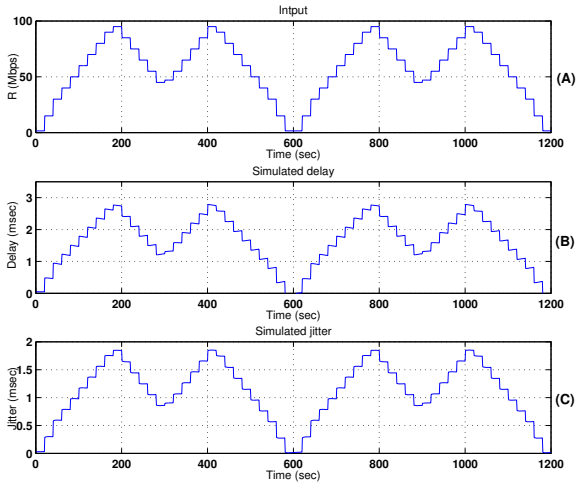


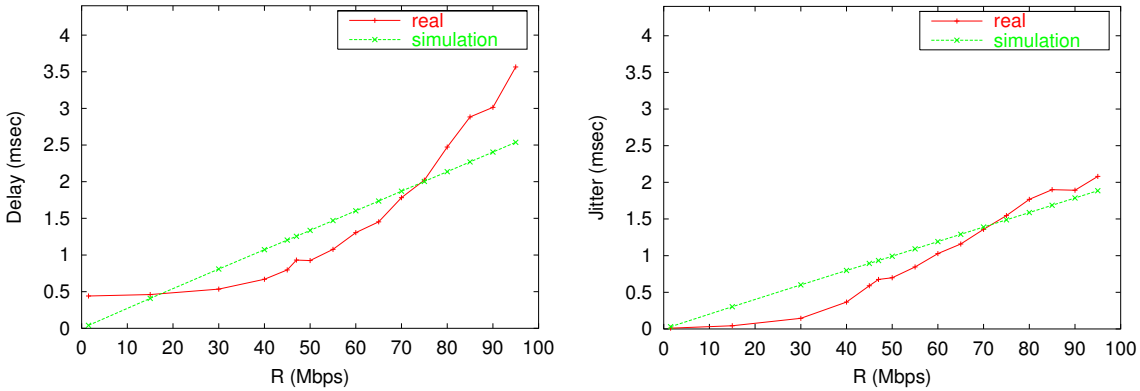
Figure 4.9: Simulated output using the same input

R depicted in Figure 4.8(A) for estimating the model parameters. It consists of a ramp-up-ramp-down traffic sending rate that ranges from 1.5 Mbps to 95 Mbps of non-premium traffic during a small constant-rate period. This rate is always higher than that of premium traffic which is a 1 Mbps CBR.[8] Each constant-rate period of the input signal lasts for 20 sec, and each cycle has 30 such periods for a total cycle time of 600 sec. Using two cycles of this makes the total duration of the input signal 1200 sec. All traffic sources use the same packet length of 1000 bytes. The corresponding measured output delay and jitter for the single FIFO node case are depicted in Figures 4.8(B) and (C).

The one-way delay (d) and jitter (j) are measured at host H, and sampled every T_s by a timer-operated traffic monitor listening on the receiving network interface. We use $T_s = 1$ sec, or equivalently a sampling frequency of 1 Hz. The UDP header[9] in each packet carries time-stamps to enable delay and jitter calculations. Delay and jitter measurements also go through a weighted moving average (WMA) filter calculated for jitter, for example, as $j = j + (|d(i-1,i)| - j)/8$, where $d(i-1,i)$ is the delay variation between packets i and $(i-1)$. To determine the order of the system model, we tried several choices, and found from experiments that choosing a second order model (i.e., $n = 2$) is good enough to capture

[8] Real-time applications usually send constant-bit-rate UDP traffic.

[9] We use RTP-like headers.



(a) Delay vs. R (b) Jitter vs. R

Figure 4.10: Real and simulated model behaviors

the dynamics of the system and achieve the required goal. Accordingly, the thus-acquired systems matrices for the single-node case are:

$$A = \begin{bmatrix} 0.7978 & 0.6260 \\ 0.0790 & 0.2525 \end{bmatrix}, \quad B = \begin{bmatrix} -0.0243 \\ 0.0275 \end{bmatrix}$$

$$C = \begin{bmatrix} -0.5002 & 0.3134 \\ -0.1988 & -0.1957 \end{bmatrix}, \quad D = \begin{bmatrix} 0.0108 \\ 0.0250 \end{bmatrix}$$

$$G = \begin{bmatrix} -0.2041 & -0.3298 \\ 0.2015 & -1.2981 \end{bmatrix}, V = \begin{bmatrix} 2.8306 & 0.0694 \\ 0.0694 & 0.0747 \end{bmatrix}.$$

From this model, we calculate the open-loop system's discrete-time poles to be 0.8770 and 0.1733. The corresponding poles of the equivalent continuous-time system are -0.1313 and -1.7526 which indicates a stable system model. The two natural frequencies (corresponding to the poles) of the system are 0.0208 and 0.2785 Hz, respectively. The model is then simulated with MATLAB to compare with the original system outputs. The simulation output is plotted in Figure 4.9 and indicates a correct behavior of the model.

In order to get a better view of the difference between the model behavior (from simulation) and the real system behavior (from experimental data), we plot the average values of the output delay and jitter versus the input R for both outputs in Figures 4.10(a) and 4.10(b); the resulting model is linear, while the real system is not. However, the model is the best-fit for the experimental data and is good enough for building the feedback control.

4.4 Design of Feedback Control

The feedback-control loop around a network service is to enforce the user-/application-specified delay and jitter *references* of the premium traffic. This reference-tracking control must meet two design criteria. First, the feedback loop is designed using the model extracted from particular input–output scenarios, and should be able to cope with the changes and uncertainties of the model under different and/or real input–output scenarios. Second, the controller is required to have a smooth response at an adequate speed. Large overshoots or undershoots would have a negative effect on the network traffic performance, especially with responsive congestion- and flow-control protocols such as TCP. A surge increase in the allowed non-premium input traffic can drive the network service to saturation,[10] hence a large transient delay and even traffic loss that may generate and send an incorrect (false-negative) feedback signal to traffic sources. Conversely, a surge decrease in the allowed non-premium input traffic reduces the utilization[11] of the network service as well as causing unnecessary drop of non-premium application packets. Third, within the context of network delay and jitter control, it is always required not to exceed certain bounds. Therefore, the delay and jitter references represent upper bounds, and the feedback-loop's output cannot exceed them, but there will always be a limit for how much lower (than the bounds) they can be. This lower bound should be determined so as (i) not to unduly lower the utilization, and (ii) to yield feasible output delay and jitter for the real system under control according to Figure 4.10.

In state-space digital control design, unmeasurable states ($\mathbf{x}$) of the system, as in the case of network services, are estimated ($\hat{\mathbf{x}}$) using the measured input–output samples of the system.[12] The estimated states are then used to generate the feedback-control signal to act as input to the system for the next sampling period. In order to overcome inaccuracies in the system model (e.g., caused by linearization), robust control is preferred to other regular control designs. Robust control can also deal with changes and uncertainties in the system model and input conditions. The LQR (Linear Quadratic Regulator) controller [53, 54]

[10] In the presence of large unregulated non-premium traffic in the network.
[11] Defined as the percentage of the capacity used by the running traffic.
[12] This is possible because the system's dynamic model is observable.

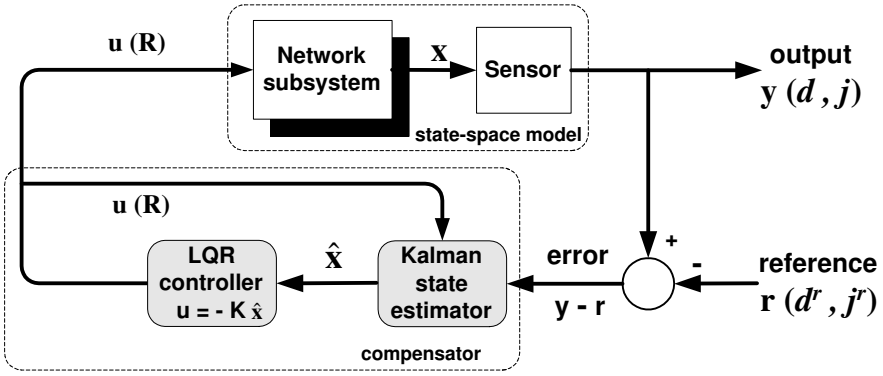


Figure 4.11: Feedback control loop

fits well this design criterion and achieves a good balance between system response and the control effort required. For state estimation, we employ the Kalman filter [54], which works well with experimentally-derived models.[13] Moreover, intelligent reference setting is used to adjust the input to the closed feedback loop according to the model behavior plotted in Figures 4.10(a) and (b). More on intelligent reference setting will be discussed in Section 4.4.3.

By putting all of these blocks together, the feedback control-loop is formed as illustrated in Figure 4.11. The loop takes both the delay and jitter references as input, compares them with the measured values, and produces an error signal. This error signal drives the compensator to produce an input signal to the network service that brings the output of the overall system closer to the reference values. The following subsections detail the design and analysis of each part of the closed loop. Then, we will describe the implementation of the algorithm on real networks and evaluate its effectiveness.

4.4.1 Design of the LQR Controller

The control law, based on the system's estimated state vector, $\hat{\mathbf{x}}$, is given by:

$$R_k = -\mathbf{K}\hat{\mathbf{x}}_k \tag{4.3}$$

where $\mathbf{K}$ is the control gain to be designed based on the system matrices, $\mathbf{A}$ and $\mathbf{B}$, and two weighting matrices, $\mathbf{Q_1}$ and $\mathbf{Q_2}$, that minimize a certain cost function [54]. Using

[13] It accounts for process noise and measurements errors.

MATLAB, we calculated the controller gain as:

$\mathbf{K} = [-15.7017, -9.3098]$.

4.4.2 Design of the State Estimator

The states estimates ($\hat{\mathbf{x}}$) are iteratively calculated in terms of successive samples of output $\mathbf{y}_k$ or $[d_k, j_k]^T$, reference $\mathbf{r}_k$ or $\left[d_k^r, j_k^r\right]^T$, previous step input, and previous step state estimate as:

$$\hat{\mathbf{x}}_{k+1} = (\mathbf{A} - \mathbf{LC})\hat{\mathbf{x}}_k + (\mathbf{B} - \mathbf{LD})\mathbf{u}_k + \mathbf{L}(\mathbf{y}_k - \mathbf{r}_k) \tag{4.4}$$

where $\hat{\mathbf{x}}_{k+1}$ and $\hat{\mathbf{x}}_k$ are the state estimates at step $k+1$ and k, respectively. **A**, **B**, **C**, and **D** are the system matrices from the model (i.e., Eq. (4.2)), while **K** is the controller gain derived above and **L** is the estimator gain. This formula is used to update the states in real time during control. The initial state, $\hat{\mathbf{x}}_0$, is set to zero, and the closed-loop starts building up from that point. The estimator gain, **L**, is optimally chosen to reduce the effects of both the process noise and the measurement error [54]. Using the MATLAB built-in Kalman filter function, we calculated the required estimator gain as:

$$\mathbf{L} = \begin{bmatrix} -0.1020 & -0.1649 \\ 0.1008 & -0.6491 \end{bmatrix}.$$

4.4.3 Intelligent Setting of Feasible References

As mentioned in Section 4.1.1, the characteristic curve of a network service subsystem governs the feasible delay and jitter achievable from the underlying control algorithm. Although the developed model is linear and, theoretically, can be controlled to any arbitrary value, we use only one input to control two output signals. So, it would not be possible to get arbitrary delay and jitter values. However, this does not mean that our control approach is incomplete; we chose to use one input only for the ease of realization in real networks.

In addition to following the feasible values in Figure 4.10, we also use an intelligent reference-setting algorithm that utilizes these feasible values as upper bounds for the required premium delay and jitter. A key process takes place when the delay and jitter refer-

```
Given the characteristic curve of the network service:
 1.  Estimate R(d^r)
 2.  Estimate R(j^r)
 3.  if (R(d^r) ≤ R(j^r))
 4.    choose d^r and the corresponding j^l as reference
 5.  else
 6.    choose j^r and the corresponding d^l as reference
```

Figure 4.12: Intelligent references-setting algorithm

ences are set. One can observe from Figure 4.10 that both delay and jitter increase with the increase of R. If we require a certain pair of delay (d^r) and jitter (j^r), then we can imagine a vertical line sweeping from left to right on Figure 4.10 until it meets the first value of either delay or jitter. At this value of R, either delay or jitter will attain its required value, while the other will be less than, or equal to, the required value, and this would be considered advantageous for the application. Figure 4.12 illustrates a simple algorithm used to realize this functionality, given the characteristic curve of the network service subsystem, which can be determined during the modeling phase as discussed in Section 4.3.2.

4.4.4 Simulation and Verification

In order to simulate the closed-loop system along with its controller and estimator, we manipulate Eqs. (4.2), (4.3), and (4.4) to get the closed-loop dynamics in terms of the original and estimated states:[14]

$$\begin{aligned}
\begin{bmatrix} \mathbf{x}_{k+1} \\ \hat{\mathbf{x}}_{k+1} \end{bmatrix} &= \begin{bmatrix} \mathbf{A} & -\mathbf{BK} \\ \mathbf{LC} & \mathbf{A}-\mathbf{BK}-\mathbf{LC} \end{bmatrix} \begin{bmatrix} \mathbf{x}_k \\ \hat{\mathbf{x}}_k \end{bmatrix} \\
&\quad + \begin{bmatrix} 0 \\ -\mathbf{L} \end{bmatrix} \begin{bmatrix} d_k^r \\ j_k^r \end{bmatrix}, \\
\begin{bmatrix} d_{k+1} \\ j_{k+1} \end{bmatrix} &= [\mathbf{C} \quad -\mathbf{DK}] \begin{bmatrix} \mathbf{x}_k \\ \hat{\mathbf{x}}_k \end{bmatrix}
\end{aligned} \tag{4.5}$$

This system of closed-loop equations has the references $[d^r, j^r]^T$ as the input, and $[d_{k+1}, j_{k+1}]^T$ as the output, and represents the closed-loop system in Figure 4.11. The

[14] Derivation is straightforward.

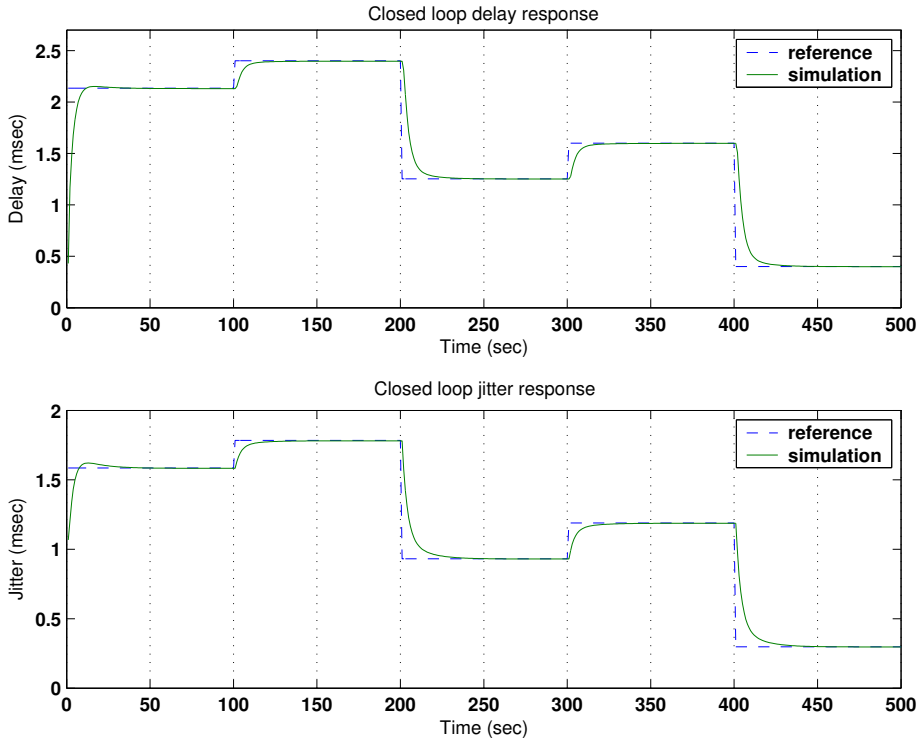


Figure 4.13: Closed-loop response

closed-loop system is simulated with a time-varying reference input using MATLAB, and the resulting delay and jitter are plotted in Figure 4.13. From this plot we observe the controller's efficiency in tracking the delay and jitter references without overshoots or undershoots. The resulting discrete-time closed-loop poles of the single-node system are 0.9169, 0.7143, 0.0081, and 0.0936, which are equivalent to continuous-time poles of -4.8196, -0.0868, -0.3364, and -2.3688, respectively, hence a stable closed-loop. The closed-loop bandwidth (control frequency) is 0.7671 Hz, indicating that a sampling frequency of 1 Hz is suitable when compared to the closed-loop bandwidth.

4.5 Implementation of Control and Actuation Algorithms

To control the rate of non-premium traffic, we employ traffic regulators in the form of token bucket filters applied at the non-premium sources as shown in Figure 4.14. This figure also illustrates the control steps executed inside host H, as described in the control

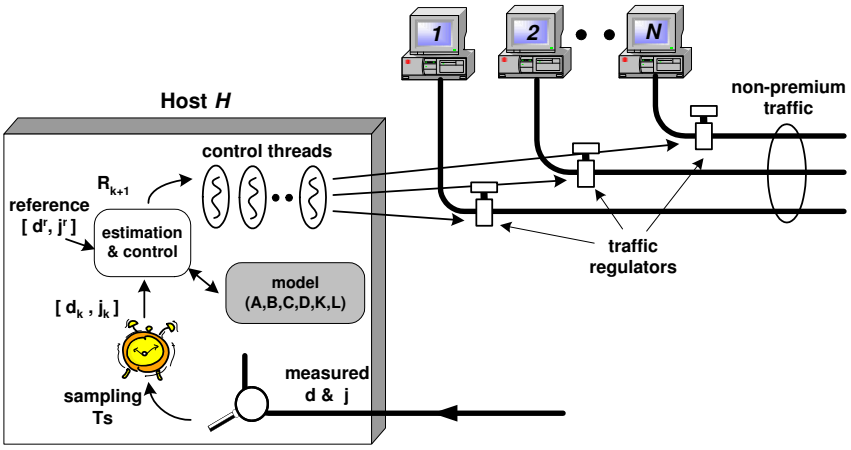


Figure 4.14: Control implementation

```
At each sampling instance, k:
Begin
1.  Read measured d_k and j_k
2.  Use x̂_k, u_k from previous step (k−1)
3.  Apply Eq.(4.4) to get x̂_{k+1}
4.  Apply Eq.(4.3) for step (k+1) and get R_{k+1}
5.  Check control signal saturation (0 ≤ R ≤ 100 Mbps)
6.  Create control threads to apply signal,
    one thread per source
7.  Each control thread sends control signal to
    traffic regulators
8.  Traffic regulators adjust the total non-premium
    rate to R_{k+1}
9.  Store values of x_{k+1} and R_{k+1} for next step
End
```

Figure 4.15: Control algorithm

algorithm of Figure 4.15. The traffic regulators are controlled by signals communicated from H. The computation and communication of the control signal are to be completed within one sampling period (T_s) and hence, a separate thread is assigned to communicate the control signal to each of the non-premium traffic regulators. The measurement thread is interrupted every T_s to sample the current delay and jitter values, and calculate the control signal using a combination of Eqs. (4.3) and (4.4).

Figure 4.16 shows the control cycle where T_c is the time taken to compute the control signal from the current sampled measurements, and T_d is the delay in delivering the control

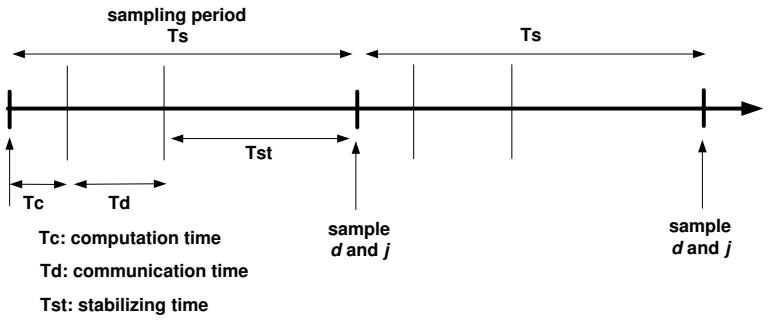


Figure 4.16: Control cycle

signal to traffic regulators. For the correct control cycle, both computation and communication of the control signal must end before the current sampling period expires. In our testbed network, T_d was the dominant factor and measured to range from 0.02 sec to 0.12 sec which was the main reason for choosing a value of 1 sec for the sampling period to allow enough time (T_{st}) for stabilizing the traffic regulators.

In a real network situation, we recommend using a traffic regulator on each non-premium traffic trunk coming from different subnets. Traffic classification (into premium/non-premium) can be done using packet marking or more simply header-based classification/filtering (can be done easily using Linux-based gateways as the one described in Chapter 5). In the following subsections, we illustrate how to realize the three suggested cases of CONNET described in Section 4.2.

4.5.1 CONNET Pipes in a Network Cloud

In the case of a network service extending across a network cloud or a domain as shown in Figure 4.17, the traffic regulators can be installed at ingress points while the measurements of the premium delay and jitter are made at the egress. For one-way delay measurements, we suggest the use of a timestamp in the header of each premium packet passing through the network service under control. This can be done by using existing frameworks like MPLS [110]. The timestamps are checked at the egress, and given a synchronized egress and ingress, the one-way delay can be calculated within a certain accuracy. Synchronization between ingress and egress nodes has also been mentioned in [21], the au-

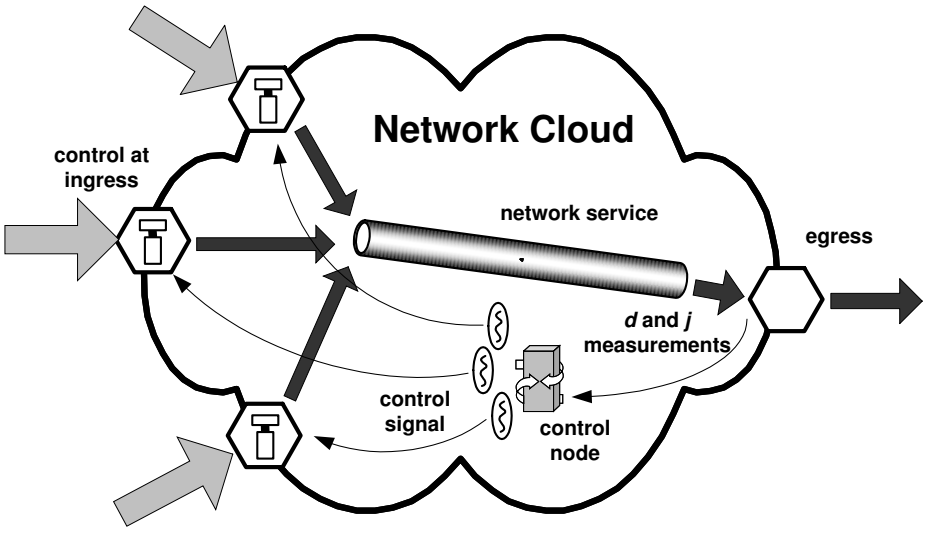


Figure 4.17: Network service control in a network cloud

thors of which suggested the use of cumulative queueing times as a substitute for clock synchronization. This is used in FIFO+ service discipline [25] for coordinated scheduling using packet headers. The controller node receives these measurements at a rate equal to the sampling frequency, computes the control decision (a new value of the control input), and communicates the new allowed rate to traffic regulators which control the input rate of non-premium traffic to the network service. We assume that, for each network service, an associated list of ingress and egress points is available for the controller node based on the routing policy. The controller node employs multiple threads, similar to the case in our testbed network, where each works with a certain ingress point. This is done to minimize the overhead of applying the control action within the sample period as explained earlier. One interesting aspect of this approach is its ability of applying different control rates on different ingress points. This may take into consideration the importance of each ingress point (if they have different degrees of importance) or the amount of total traffic into each ingress, so that the traffic control rate can be proportional to this amount.

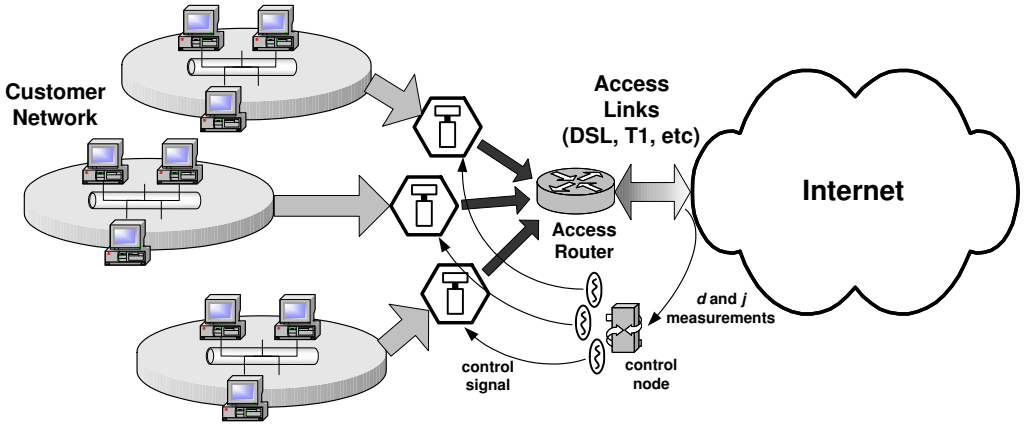


Figure 4.18: Access link control

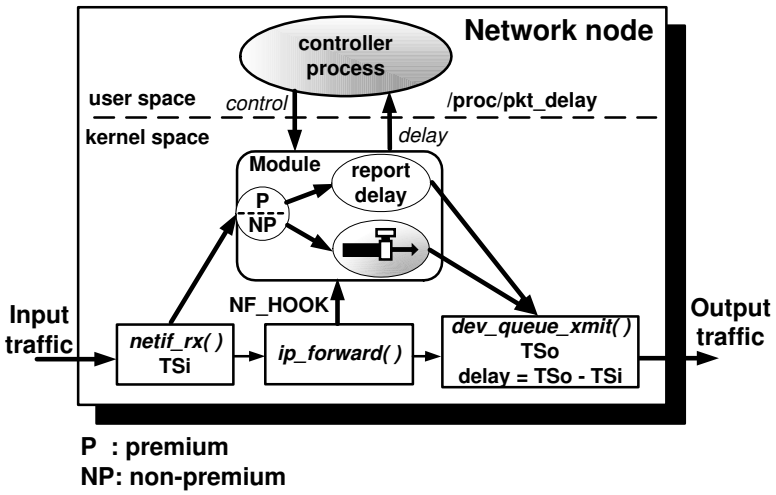


Figure 4.19: Implementation inside a Linux router

4.5.2 CONNET for Access Links Control

Very similar to the network cloud case, CONNET can be implemented as an access link control by applying traffic regulators at the output trunk of each part of the customer network(s) going to the ISP. This is illustrated in Figure 4.18 where a controller node is responsible for delivering the control signal (non-premium traffic rate) to TBFs acting on the non-premium traffic trunks coming from each customer network subnet. Again, this has the flexibility of applying different policies when throttling the non-premium traffic coming from different subnets according to their relative importance.

4.5.3 DQM Implementation

We implemented DQM internally to a Linux network node as a combination of a kernel module and a user-space controller, as shown in Figure 4.19. First, we had to patch the Linux kernel to timestamp premium packets as they pass through a delay-controlled node for delay and jitter measurements across the node. Due to the coarse time resolution in the Linux kernel (i.e., jiffies) [15], which has a default value of 10 ms, we generate timestamps using the more accurate Time Stamp Counter (TSC) register.[15] Incoming packets are timestamped (TS_i) once they are received from the ingress network interface inside the `netif_rx()` function. Timestamps are placed in the `stamp` field of the `sk_buff` structure holding the packet inside the kernel. Before the packet is sent back to the egress network interface in `dev_queue_xmit()`, another timestamp (TS_o) is taken and the packet delay ($TS_o - TS_i$) is stored in a global variable called `pkt_delay`.

Using NetFilters [1] hooks inside the Linux kernel, we register the kernel module to the `ip_forward()` function, which is responsible for forwarding IP packets. Each forwarded packet is sent to our module in its way inside the kernel. The module runs a simple classifier based on the IP header of the packet to tell premium packets from non-premium.[16] For premium packets, the module reads the `pkt_delay` variable and reports it to the user-space using the `/proc` filesystem interface. On the other hand, for non-premium traffic, it applies a traffic regulator in the form of a Token Bucket Filter (TBF).

The controller process samples `pkt_delay` every T_s through the `/proc` filesystem interface, evaluates the required feedback control, and finally pushes the required control signal to the kernel module through the `/proc` filesystem to apply it on the non-premium traffic. The reason for using an additional user-space process is to give the algorithm more flexibility in configuration (e.g., providing a variable reference signal) and operation. Additionally, the user-space controller offloads the kernel module from floating-point calculations in the feedback control loop given in Section 4.4.

[15] A 64-bit register found in recent Intel 80x86 microprocessors that counts at the processor speed.
[16] Currently, we use the Type-of-Service (ToS) field.

4.6 Experimental Evaluation

We evaluate CONNET experimentally for different scenarios, reflecting several aspects of the controller performance. We divide the experimental results into three sets: one for a single FIFO node, one for a multi-node FIFO pipe, and the third for DQM. Important and interesting points in the results for each scenario are identified and discussed.

4.6.1 Experimental Setup

We use the same testbed network illustrated in Figure 4.7 for experimental evaluation with 3 non-premium traffic sources, each running 2 flows[17] with a total of 6 non-premium flows. We use 3 traffic regulators, one for each traffic source, all controlled by the control algorithm running on host H. During evaluation, unless otherwise mentioned, each non-premium traffic source sends constant-bit-rate (CBR) UDP data at the maximum rate allowed by the network, which is 100 Mbps. This allows for actual testing of the controller when the network encounters high loads.

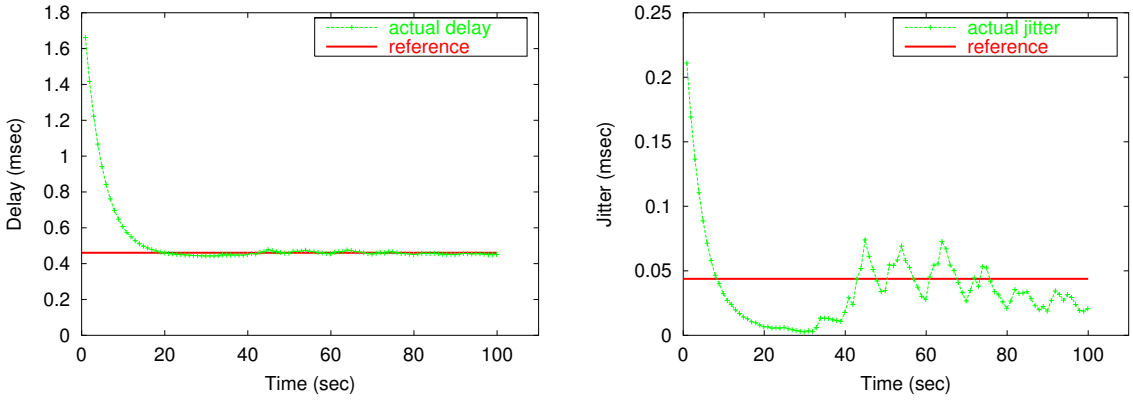
4.6.2 A Single FIFO Node

For the case of a single FIFO node shared by premium and non-premium traffic, we considered several experimental scenarios to show the effectiveness of the controller designed in Section 4.4.

Constant Reference Tracking

In the first scenario, we study the controller's performance in tracking a constant reference for both delay and jitter. The experiment is started with traffic regulators at non-premium sources (see Figure 4.14) turned on while each source is sending data at the maximum link capacity of 100 Mbps, and then they are kept on for the rest of the experiment duration of 100 seconds. We use a delay reference of 0.46 msec and a jitter reference of 0.043 msec as an example of driving the performance of the access link all the way from

[17] To increase statistical multiplexing between packets.



(a) Controlled delay response

(b) Controlled jitter response

Figure 4.20: Constant reference tracking

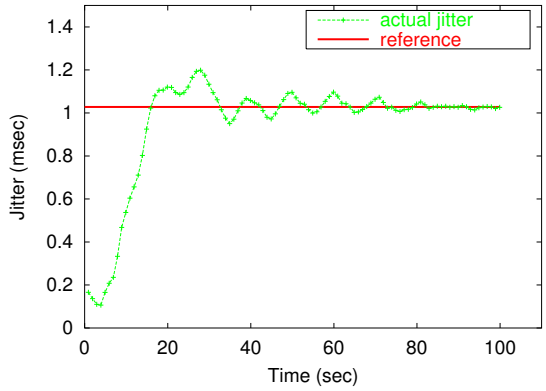


Figure 4.21: Step response for jitter

high to very low non-premium traffic loads. Figures 4.20(a) and (b) plot the measured delay and jitter as well as their reference values. Clearly, the controller can achieve the required delay and jitter together, despite some oscillation in the jitter due to the system dynamics. We also observe here the dynamics of the controller when it tries to achieve a lower jitter first and then closely tracks the reference value.

Step Response

In the second scenario plotted in Figure 4.21, the step response of the system is plotted with respect to jitter as an example. A step jitter reference of magnitude 1.028 msec is applied to the controller shortly after starting the experiment, and then kept constant for the rest of the experiment duration of 100 sec. This is an example scenario of allowing

Time (sec)	d^r (msec)	j^r (msec)
0 - 200	0.70	0.42
201 - 400	1.10	0.63
401 - 600	0.60	0.36
601 - 800	0.92	0.60
801 - 1100	0.46	0.0437

Table 4.1: Reference values for the third scenario

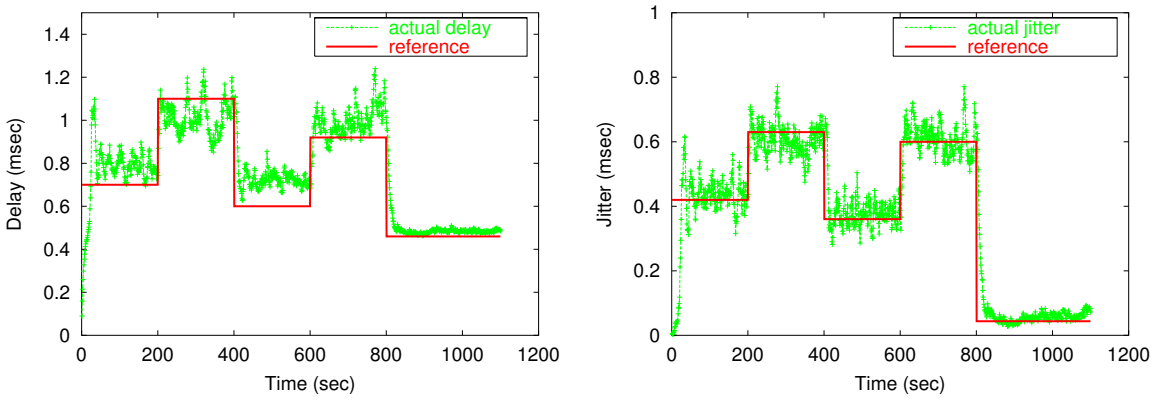
more non-premium traffic in the network, given that the application does not require small jitters. It also indicates the rise time of the controller output (about 15 sec), which depends primarily on the sampling frequency and controller gain. These are all design parameters that can be tuned to achieve faster response.

Variable Reference Tracking

To further illustrate the dynamics of the delay and jitter controllers using non-premium traffic regulation, we provide the third scenario where a variable reference signal is used to reflect a certain scenario when demands and requirements change throughout the day/hour. It also compares the actual and simulated controller performances plotted in Figure 4.13. The experiment lasts for 1100 sec, and the reference values of delay and jitter change every 200 sec. Table 4.1 shows the reference values used in this scenario for both delay and jitter as an example of a variable reference signal. The output delay is plotted in Figure 4.22(a), while the output jitter is plotted in Figure 4.22(b). The real controller performance is not as smooth as the simulated performance due to noise and un-modeled factors in the network. However, the choice of robust control proves to be successful in overcoming all these unaccounted-for factors and still yields a reasonable performance in controlling delay and jitter.

To get an idea about how the control signal, R, would look like during the application of control, we present Figure 4.23. We observe here that during the periods from time 246 to 400 sec, the control signal gets saturated,[18] and this is a very common case in real feedback control systems. However, due to the use of state-space design, which includes a Kalman state estimator (or observer), the controller succeeds in avoiding saturation and returns to

[18]We set saturation level at 95 Mbps for all experiments.



(a) Variable delay reference

(b) Variable jitter reference

Figure 4.22: Variable reference tracking

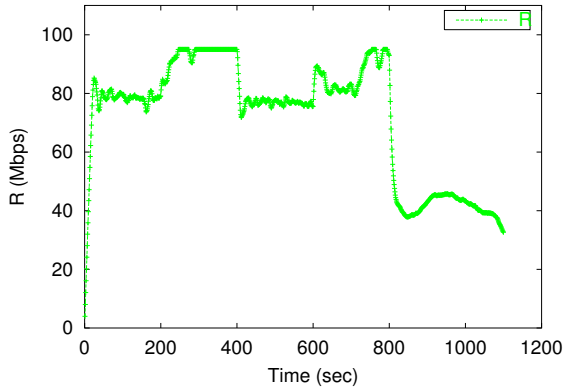


Figure 4.23: Control input R with a variable reference

normal operation in the next period, giving correct delay and jitter values. This technique is called "observer-based anti-windup." Note that the R values plotted in this figure represent the total control rate applied to all traffic regulators in the network and divided equally among them.

Controller Activation During Run-time

The fourth scenario, plotted in Figure 4.24, illustrates what happens when the controller is turned on during a heavily-loaded network operation. The experiment is started while all traffic regulators were turned off and non-premium sources are sending data at a rate of 70 Mbps resulting in a delay of around 1.1 msec and a jitter of 0.6 msec. At 100 sec (the total experiment time is 400 sec), the controller is turned on with the reference values of

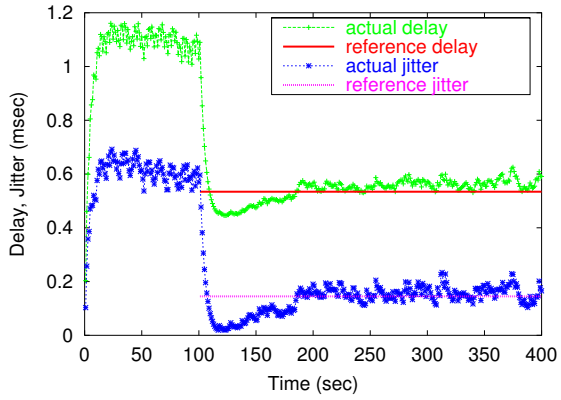


Figure 4.24: Controller is turned on at time 100 sec

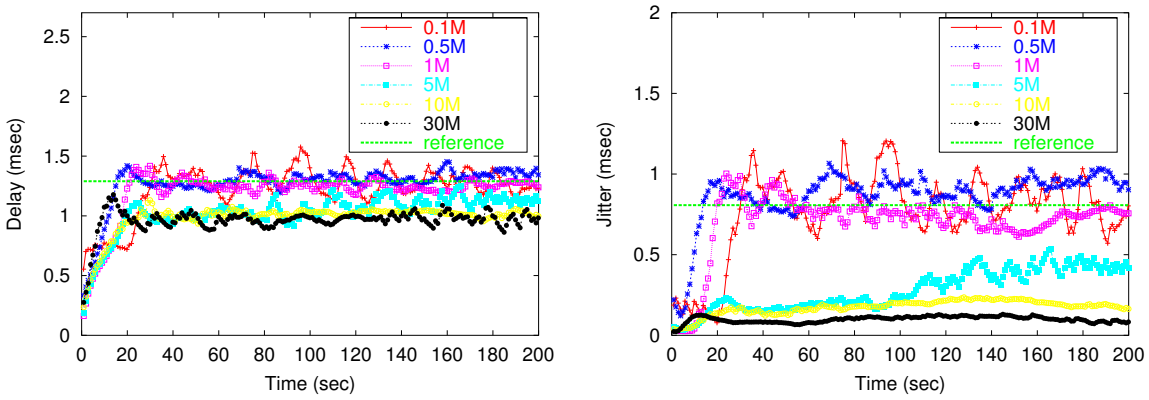
0.534 msec and 0.145 msec for delay and jitter, respectively. As indicated in the figure, the controller reacts instantaneously to regulate the non-premium traffic so that the delay and the jitter may follow the reference values as closely as possible. This scenario reflects the situation when a new premium application traffic is activated in the middle of a network operation and requires the adjustment of a network service performance to meet the new application's requirements. This is a good example of the effectiveness of self-controlled network services.

4.6.3 Multi-Node FIFO Networks

The above scenarios demonstrate the correctness and accuracy of CONNET. We obtained similar results for the multi-node FIFO pipe, but we only present scenarios different from those discussed so far. For all the following experiments scenarios, we use a FIFO pipe that consists of 3 FIFO nodes connected to each other. All premium and non-premium traffic enters the pipe only via the first node, and exit from the third node. No cross traffic is introduced (see Section 4.6.5 for further comments).

Varying Premium Traffic Rates

In Section 4.3, we used a premium traffic source with a constant sending rate of 1 Mbps during system modeling. In the fifth scenario, we test the robustness of the control algorithm with different premium traffic rates: 0.1, 0.5, 5, 10, and 30 Mbps. Although the



(a) Delay performance

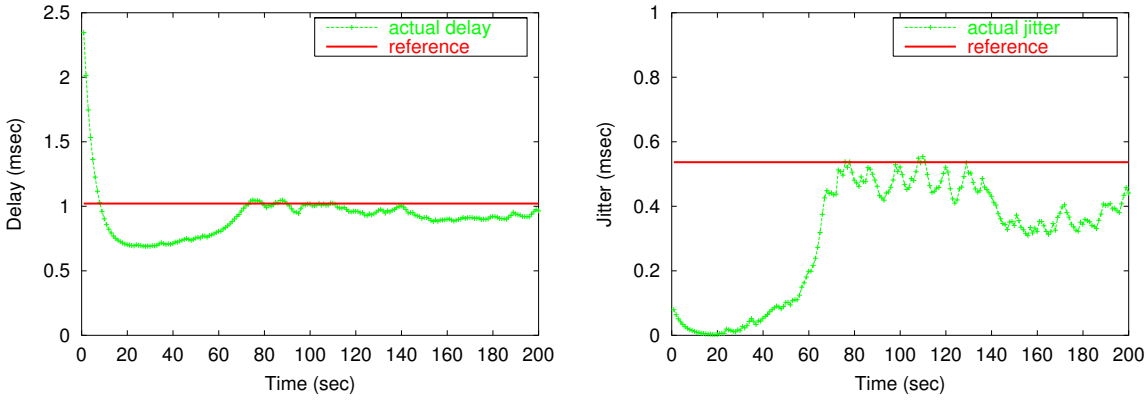
(b) Jitter performance

Figure 4.25: Different premium rates

controller was designed for a specific premium traffic rate, it can deal with rate changes, thus showing its robustness. Figures 4.25(a) and (b) illustrate the controller's performance in keeping the delay at 1.29 msec and the jitter at 0.81 msec even when the premium rate varies. However, we can still see a weak trend: for higher rates than 1 Mbps, delay and jitter are kept at lower values than the reference values, which is a good behavior. This is because the higher the premium traffic rate, the higher its queue occupancy, and at the same time, the weaker it is affected by the non-premium traffic. However, the controller reacts positively to this condition and keeps the rate of non-premium traffic at an appropriate level.

Dealing with Non-premium TCP Traffic

Instead of using non-premium UDP traffic, in this scenario we use TCP sources to investigate the TCP friendliness and effectiveness of CONNET in dealing with TCP congestion control. The LQR controller works carefully not to have large overshoots or undershoots, and this feature works well with TCP. Figures 4.26(a) and (b) plot the performance of delay and jitter control, bringing both to their reference values. Figure 4.27 shows the resulting throughput of 6 TCP sources used in comparison with a target rate of 9.167 Mbps. The target rate indicates the actual rate needed to achieve the reference values of delay and jitter taken from Figure 4.10, divided by 6. The regulated TCP traffic could reach the same target rates and did not suffer congestion collapse or any loss of utilization, indicating that CONNET works well with TCP as well as UDP background traffic.



(a) Delay with non-premium TCP traffic

(b) Jitter with non-premium TCP traffic

Figure 4.26: Controller performance with TCP non-premium traffic

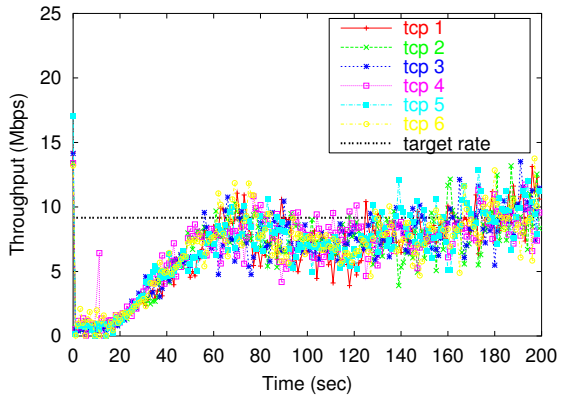
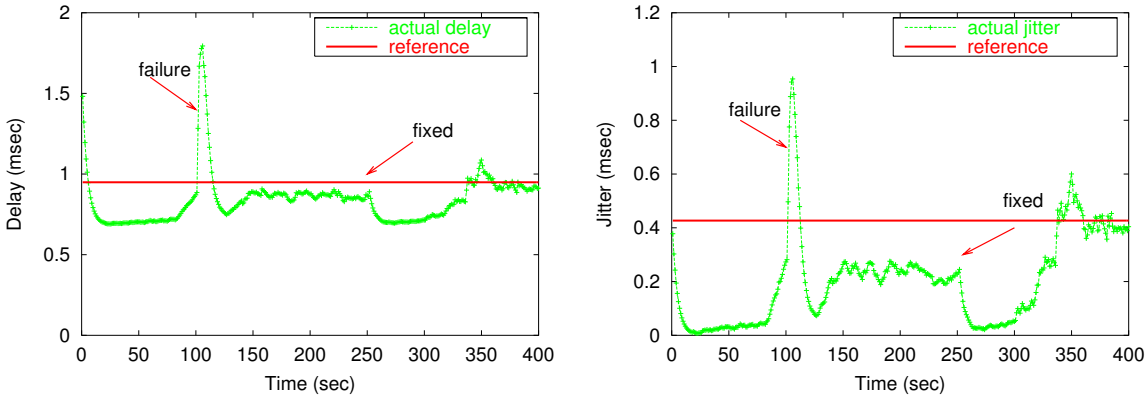


Figure 4.27: TCP throughput and target rate

Fault-tolerance and Uneven Load Distribution to Regulators

In the following two scenarios, we present a very important and powerful feature of CONNET, which is "fault-tolerance." Here we examine the case when one or more traffic regulators fail and open the door for unregulated non-premium traffic. Recall that we use a total of 3 traffic regulators as the actuator, and the control rate is divided equally among them. We first test the case of a single regulator failure, and then the case of 2 failures. In the first case, the experiment is started with all 3 traffic regulators working, then at time 100 sec, one of the 3 regulators fails, allowing unregulated traffic at 50 Mbps to enter the network pipe from the corresponding non-premium source. Then, after another 150 sec, the failed traffic regulator is fixed and back to work again for the rest of the experiment duration of 400 sec. Figures 4.28(a) and (b) show the delay and jitter during this scenario



(a) Delay performance

(b) Jitter performance

Figure 4.28: Fault-tolerance with 1 regulator failure

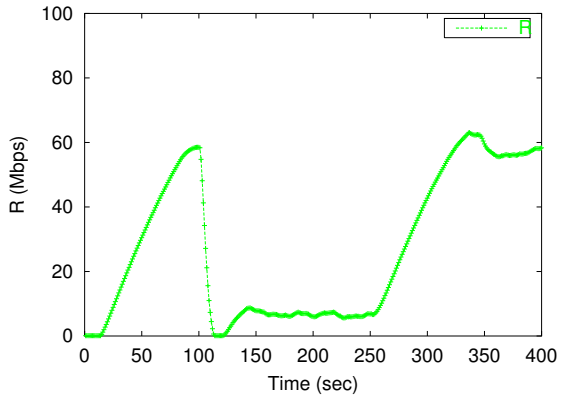
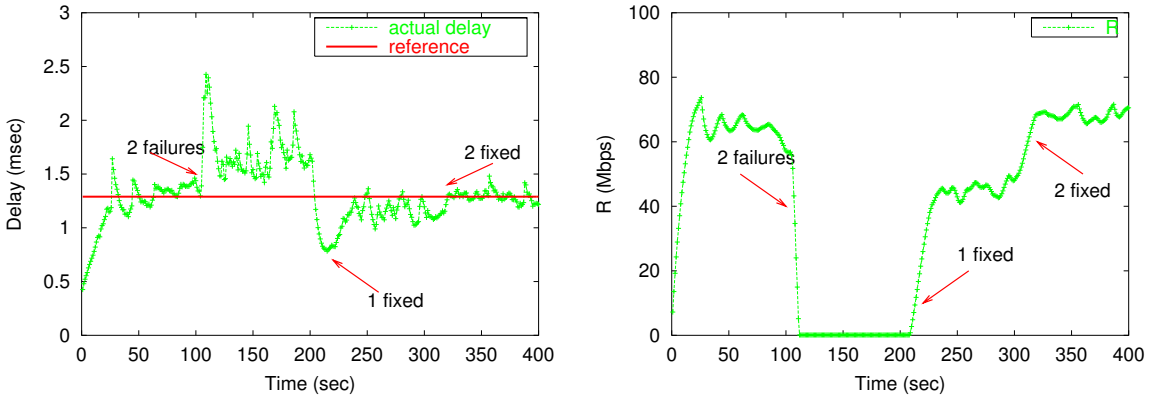


Figure 4.29: Controlled non-premium traffic rate

with the times of failure and repair indicated. We observe a spike in both delay and jitter at the time of failure due to the unregulated non-premium traffic, but the controller acts instantaneously and brings back both delay and jitter to their required (reference) values. When the failed regulator is repaired at 250 sec, the network returns back to normal again and keeps delay and jitter at their reference values. Figure 4.29 shows the rate sent to the remaining operating regulators during this failure, and the controller reacts to this failure by almost shutting off the other two traffic sources to keep premium traffic performance at the required level.

We investigate further what happens when 2 out of 3 regulators fail at time 100 sec, then one of them comes back to work at time 200 sec, and all three of them come back to work at time 300 sec. This is illustrated in Figures 4.30(a) and (b). The first figure plots the



(a) Delay performance

(b) Controlled non-premium traffic rate

Figure 4.30: Fault-tolerance with 2 regulator failures

delay, showing a similar behavior at the time of failure. However, this time the controller was not able to drive the delay to the reference value during the time of 2 failures. To see why this happened, the second figure plots the output control rate from the controller. Since the control rate cannot get below zero, the controller cannot change the control signal any more and gets helpless. When one of the failed regulators was fixed to work again, the controller catches up and sends a correct control signal to bring the output delay back to its reference value. Finally, when all the three regulators work again, the network returns back to normal. These two scenarios indicate that CONNET can survive up to one third (1/3) of regulator failures in case of our testbed network, and we also expect a similar performance when it is deployed on larger networks. This, of course, depends on the operating condition and the non-premium traffic rates running through the network service.

Another feature of the control algorithm, is its ability to apply uneven control rates at different traffic regulators. We demonstrate this ability with the following scenario, where the control rates are distributed unevenly among 3 traffic regulators. The first one gets 50% of the control rate, the second one 35%, and the third one 15%. Again, the controller was able to work well in this case and brings both the delay and the jitter to their reference values as shown in Figure 4.31. This is an attractive and important feature because of the relative importance of the non-premium traffic trunks entering the network services from different sources/applications. This allows for setting up a control policy to favor some trunks over others.

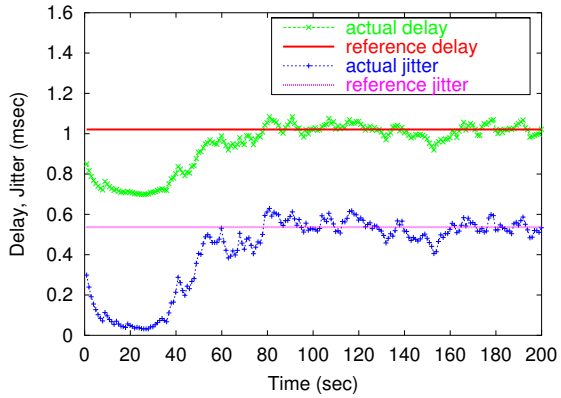


Figure 4.31: Performance with uneven regulator rates

Comparison with CBQ, WFQ, and Priority Queueing

We now compare CONNET with other well-known scheduling algorithms, such as Class-Based Queueing (CBQ) [50], Weighted Fair Queueing (WFQ) [30], and Priority Queueing (PQ). It is worth mentioning that these scheduling algorithms are different from our delay/jitter control scheme, and hence, they are not expected to behave in a similar way. However, we include this comparison to show the differences between the behavior of the two schemes. One of these differences is the ability of CONNET to achieve a specified delay/jitter reference while both CBQ and WFQ can only provide an upper bound on the delay values with no specific run-time value. CONNET can also provide this reference tracking behavior in a simple and practical way without the sophisticated behavior of these scheduling algorithms and their dependency on several factors.

CBQ is known for its capacity distribution and bandwidth allocation, but it does not pay direct attention to delay and jitter. This is exactly what we found from the experiment. We compare the delay and jitter performance while applying a variable non-premium traffic rates to the network service pipe. In this scenario, instead of FIFO queues, we set up the CBQ discipline on all of the three router nodes, where a bandwidth of 2 Mbps is allocated to premium traffic and the rest is given to non-premium traffic. Now, we start a varying non-premium input traffic to the pipe in Figure 4.32, and the delay and jitter are plotted in Figures 4.33(a) and (b), respectively. On the same delay and jitter figures, we also plot the performance when CONNET is activated with FIFO nodes in place of CBQ. Clearly,

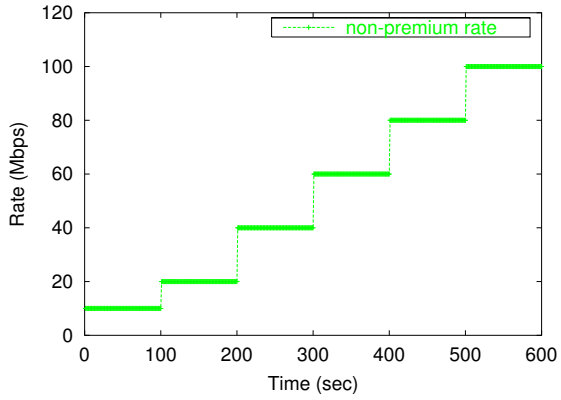
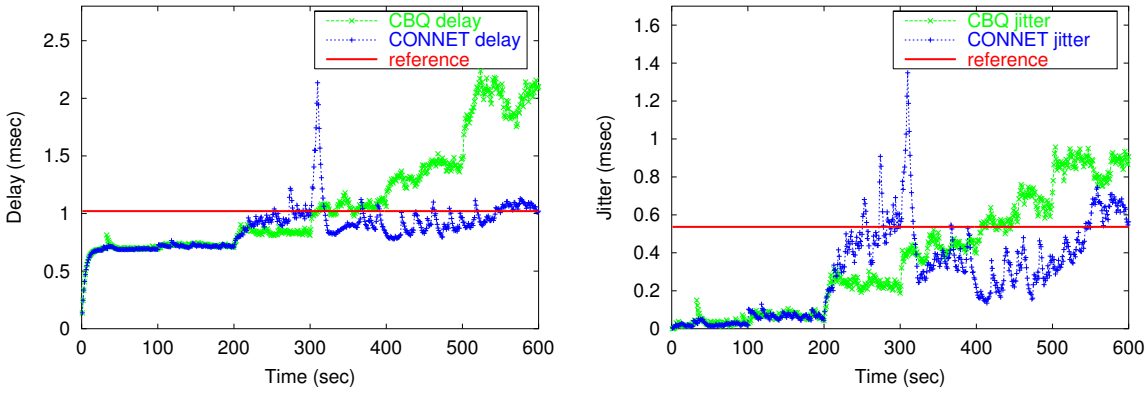


Figure 4.32: Varying non-premium input traffic rate



(a) Delay performance

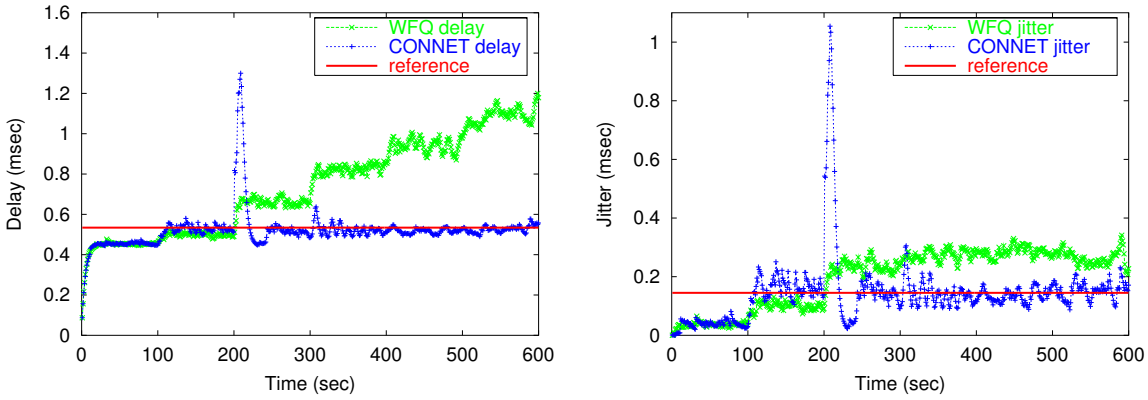
(b) Jitter performance

Figure 4.33: Comparison with CBQ

the CBQ performs worse than CONNET, and both delay and jitter increase with the input rate, while under CONNET, they follow the reference values. CONNET's performance improvement at a high non-premium rate is 100% for delay, and 54% for jitter.

WFQ is used to achieve a combination of good bandwidth allocation as well as delay/jitter protection. We compare the proposed control scheme with WFQ,[19] showing that CONNET can still outperform WFQ in tracking low reference delay/jitter values. Figure 4.34(a) shows a significant improvement for the controlled delay over WFQ (about 80%) using the same input in Figure 4.33(a). Although WFQ can achieve low jitter even with high loads, CONNET could achieve lower jitter with an almost 70% improvement as shown in Figure 4.34(b). The jump at time 200 sec is due to the controller transients.

[19]Using WFQ implementation on Free-BSD ALTQ [23].



(a) Delay performance

(b) Jitter performance

Figure 4.34: Comparison with WFQ

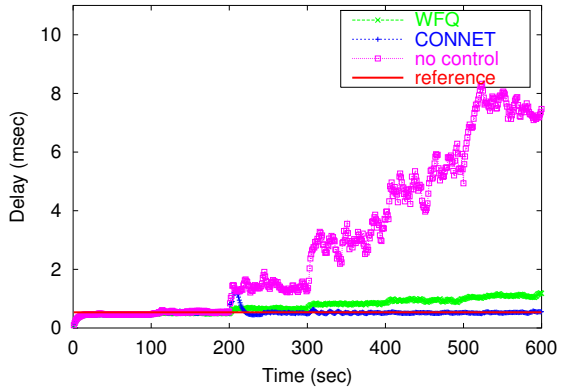
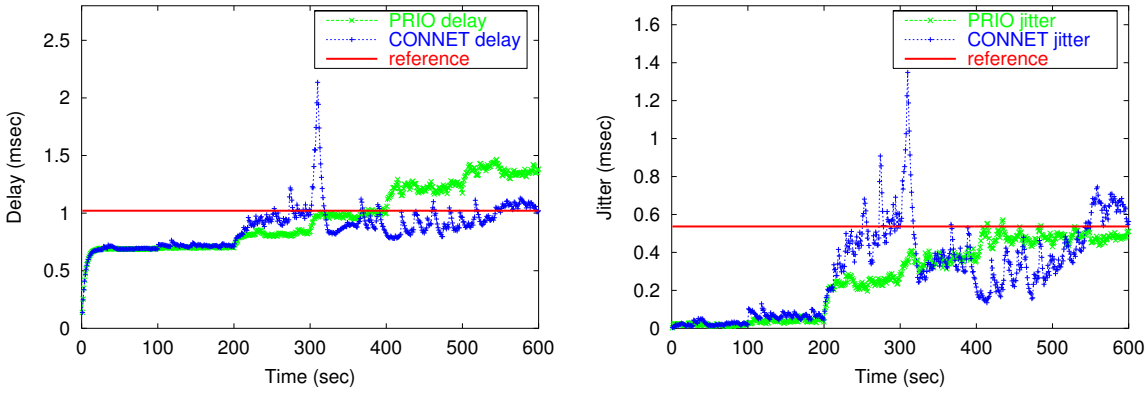


Figure 4.35: Comparison with uncontrolled delay

We also compare the delay performance with the case of uncontrolled delay in Figure 4.35 to show the degree of the absolute improvement of CONNET. Uncontrolled delay on our FIFO network could reach values up to 8 msec without WFQ or CONNET.

To compare CONNET with a stronger scheduling algorithm, we use priority queueing on the router nodes instead of FIFO, and repeat the above experiment scenario. In case of priority queueing, the delay was not as bad as in the case of CBQ, but CONNET exhibited a 40% improvement in delay control under high loads, as illustrated in Figure 4.36(a). The jitter performances of CONNET and the priority queueing are very close to each other as shown in Figure 4.36(b).

Finally, we do not claim that CONNET is absolutely better than any of the scheduling algorithms mentioned in this section. Rather, we show that our approach is more practical



(a) Delay performance

(b) Jitter performance

Figure 4.36: Comparison with priority queueing

and simpler than these scheduling algorithms and gives the ability to specify a delay/jitter reference besides a delay/jitter bound.

4.6.4 Experiments using DQM

For the DQM case, we used a different modeling signal illustrated in Figure 4.37(A) with the delay and jitter are plotted in Figures 4.37(B) and (C), respectively. It consists of a sawtooth-like traffic sending rate that ranges from 1.5 Mbps to 95 Mbps of non-premium traffic during small constant-rate periods. The behaviors of the delay and jitter with R in both simulation and real experiments are plotted in Figures 4.38(a) and (b), respectively. Accordingly, the thus-acquired systems matrices are:

$$A = \begin{bmatrix} 0.9699 & 0.0289 \\ -0.0115 & 0.9441 \end{bmatrix}, B = \begin{bmatrix} -0.0000525 \\ -0.0006954 \end{bmatrix}$$
$$C = \begin{bmatrix} -0.5309 & 0.0940 \\ -0.2454 & -0.1839 \end{bmatrix}, D = \begin{bmatrix} 0.0124 \\ 0.0104 \end{bmatrix}$$
$$G = \begin{bmatrix} -0.6023 & -0.0874 \\ 0.3929 & -0.8789 \end{bmatrix}, V = \begin{bmatrix} 0.1329 & 0.0600 \\ 0.0600 & 0.0622 \end{bmatrix}.$$

From this model, we calculate the open-loop system's discrete-time poles to be $0.957 \pm 0.0129i$, which indicates a stable system model,[20] but because the magnitude of the poles

[20]Poles are inside the unit circle.

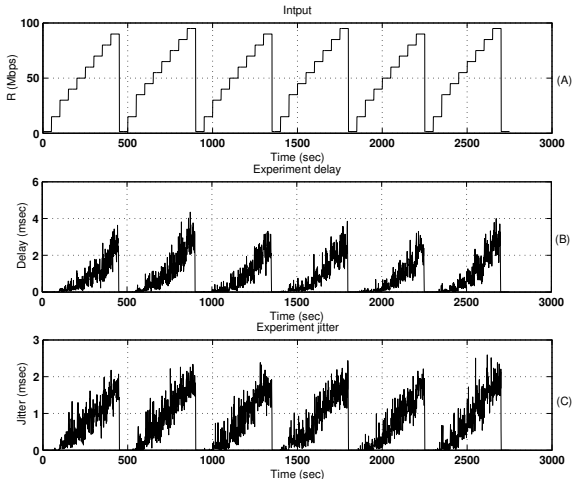
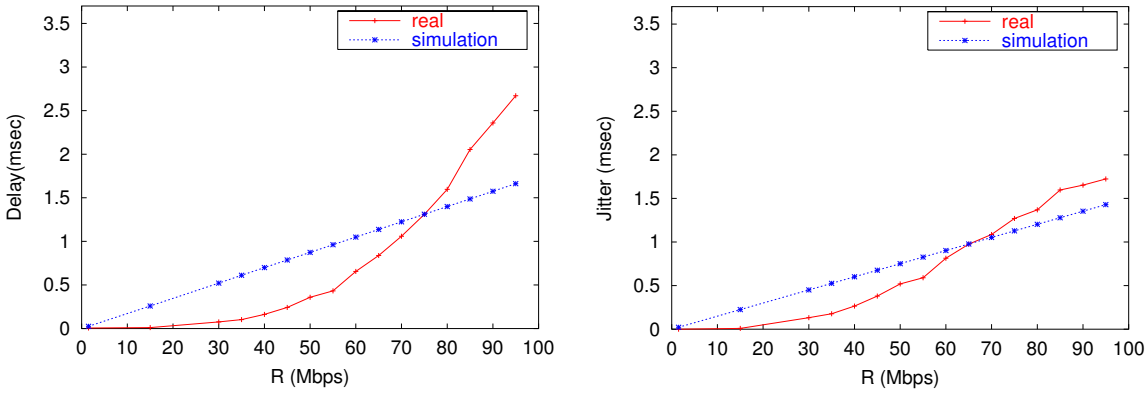


Figure 4.37: Input and output signals applied for DQM modeling



(a) Delay vs. R (a) Jitter vs. R

Figure 4.38: Real and simulated DQM model behavior

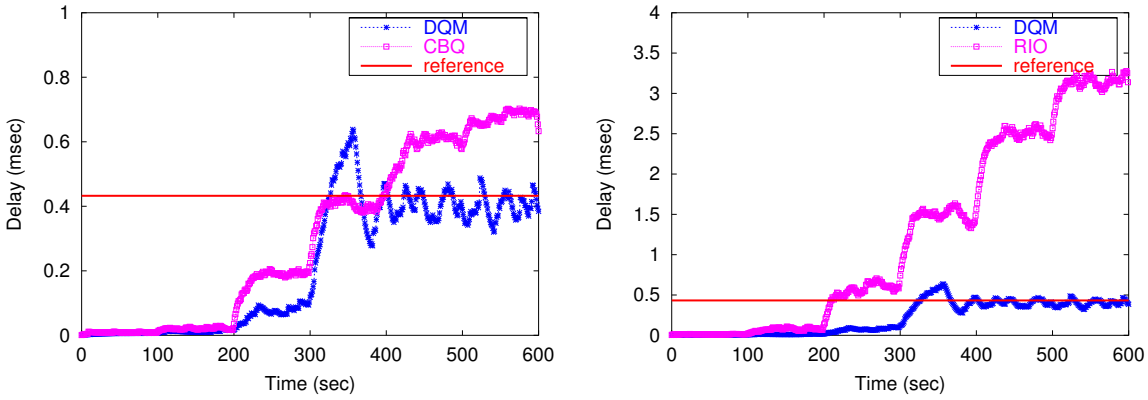
is close to 1, it is very close to the unstable region. This adds a level of difficulty to the controller in bringing a smooth response as we will show in the experiments shortly. The natural frequency (corresponding to the poles) of the system is 0.0073 Hz.[21] Using MATLAB, we calculate the controller gain as:

$$\mathbf{K} = [-36.9558, -24.0288].$$

and the required estimator gain as:

$$\mathbf{L} = \begin{bmatrix} -0.3012 & -0.0437 \\ 0.1965 & -0.4395 \end{bmatrix}.$$

[21] One frequency only as conjugate poles.



(a) CBQ delay performance

(b) RIO delay performance

Figure 4.39: Comparison with CBQ and RIO

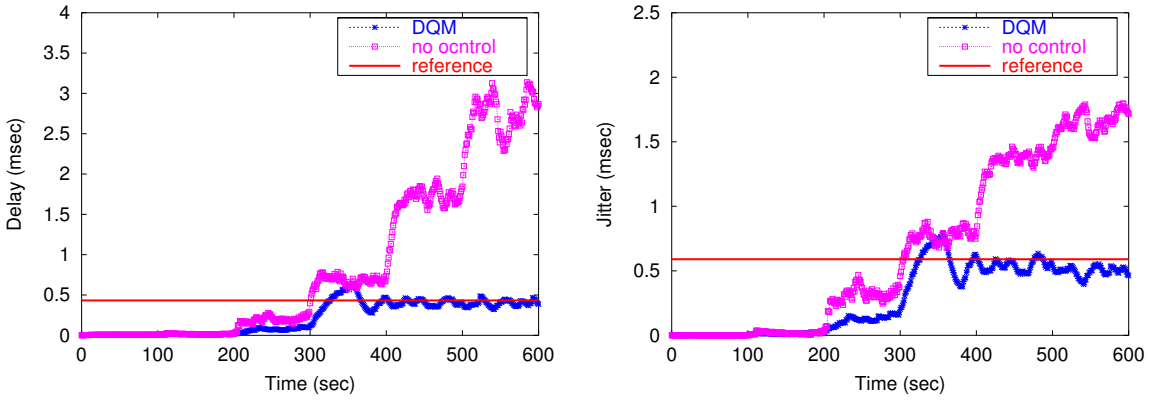
Comparison with CBQ and RED

In this scenario, we compare DQM with Class-Based Queueing (CBQ) [50] and a two-class version of RED called RED In/Out (RIO) [24]. The configuration of the experiment is similar to the one in Section 4.6.3, and the non-premium input traffic signal is the same as in Figure 4.32. The corresponding delay is plotted in Figure 4.39(a). On the same delay figure, we also plot the performance when DQM is activated on the FIFO node in place of CBQ. Clearly, the CBQ performs worse than DQM, and delay increases with the input rate, while under DQM, it follows the reference value. DQM's performance improvement at a high non-premium rate is about 60% for delay.

In place of CBQ, we installed RIO (with premium traffic classified as In and non-premium as Out), and we conducted the same experiment to show the difference in delay performance. The output delay is plotted in Figure 4.39(b) with a significant delay improvement for DQM over RIO.

Comparison with Uncontrolled FIFO

To show the difference between the uncontrolled delay and the controlled delay using DQM, we present the following scenario. We apply an input signal in the form of increasing staircase (similar to modeling input signal but only one cycle) at values of 10, 20, 40, 60, 80, and 100 Mbps to the two cases of uncontrolled FIFO node and DQM-enabled node. For the DQM case, we set the delay and jitter references at 0.43 msec and 0.59 msec, respectively.



(a) Delay performance

(b) Jitter performance

Figure 4.40: Controlled and uncontrolled cases

In Figure 4.40(a) we plot the delay performance in the two cases while in Figures 4.40(b) we plot the jitter. From these figures, we see the effects of DQM on keeping both the delay and jitter at their references even with increasing non-premium traffic in the network. Figure 4.41 shows the actual staircase input along with the control signal R and the target rate corresponding to the reference delay and jitter from Figure 4.38. From this figure we show the effectiveness of the controller in keeping the utilization at an adequate level during network operation. For example, before the input rate exceeds the target rate (i.e., before time 300 sec), the DQM controller is opening the traffic controls all the way to allow maximum non-premium traffic at 100 Mbps. This is true as long as the delay and jitter are lower than their reference values. At time 300 sec, the input rate starts exceeding the target, and accordingly, the delay and jitter start exceeding their reference values, and therefore, the controller starts throttling the non-premium traffic to keep the premium delay and jitter at reference values and continues at a target rate of 55 Mbps until the end of the experiment regardless of the increase in the actual input.

4.6.5 Assumptions, Limitations and Extensions

The various experiments we conducted have verified the correctness, robustness, and performance enhancement of CONNET, but it is worth discussing the assumptions we made, as well as the limitations of CONNET.

The first assumption is that no "uncontrolled" cross traffic enters the FIFO pipe. We

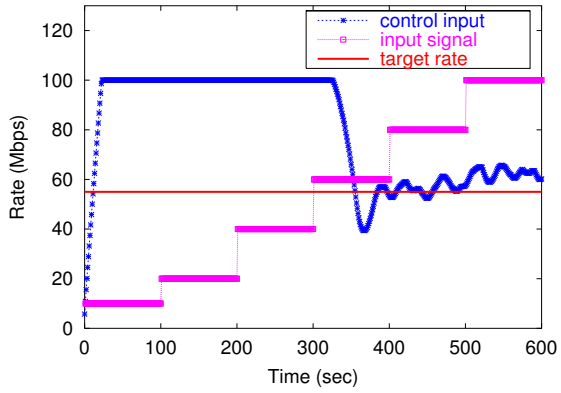


Figure 4.41: Actual input and controlled rates

assume that all non-premium traffic is controlled and enters the FIFO pipe via the front-end while exiting the rear-end. This reflects the actual case where network services are usually short enough (one to three hops) to have little cross traffic. Second, we assume that there is an adequate method for measuring run-time delay and jitter across the access link/pipe in order to feed the error back to the controller. We also assume throughout the chapter that the premium traffic cannot by itself exceed the network service capacity or available resources. In other words, we do not consider admission control on premium traffic, but we proved in 3 that premium delay and jitter are affected by the premium traffic parameters themselves. We plan to investigate ways of controlling these parameters to develop a more generic control mechanism in our future work.

One of the limitations of the current version of CONNET is not dealing with burst control along with rate control. We use only one control input, which is the non-premium traffic rate, but the burstiness of non-premium traffic also affects delay and jitter in the network. Ideally, we would like to design a two-input controller that can control both the rate and the burst of non-premium traffic. This can be implemented using token-bucket regulators with a controllable bucket size. Another possible limitation one may find is controlling only two classes of traffic, premium and non-premium. However, we designed CONNET to have only two traffic classes for simplicity and easy configuration. A hierarchy of controllers can be used for more than two classes, if needed.

4.7 Related Work

While feedback control theory [53, 54] has been used for decades in many engineering and scientific disciplines, such as mechanical and aeronautical engineering, only recently its effectiveness in solving network control problems has been realized (for example, see [63, 74, 95]).

The authors of [64] presented a thorough control-theoretic analysis of RED, and hence, they designed an AQM that is more robust to variations in system loads. Another more recent example can be found in [62] where the authors used feedback compensation in designing an AQM algorithm that eliminates the sensitivity to the change in system and configuration parameters, which has been a problem associated with the original RED for some time. Most of these studies focused on congestion control and active queue management, which presents a wide field to apply control-theoretic approaches. CONNET is following a similar path, but its main goal is different from other rate-based schemes since we are looking at delay and jitter guarantees in networks with two or more service classes.

Previous studies on delay jitter control (for example, see [86, 124]) mainly change fixed-packet flows to reconstruct the original timing characteristics in the traffic or use rate control (such as traffic shaping) to eliminate jitter. These solutions usually use a large buffer enough to smooth variations in delay and delay jitter. CONNET provides a direct solution for delay and jitter by just eliminating the original cause of delay and jitter, which is the interfering traffic in shared queues, so the need for large buffers does not exist anymore.

CONNET shares some common features with the flow-control algorithm of [71] in that both deal with best-effort networks and use a control-theoretic approach. However, the scheme in [71] was not designed for explicit delay and jitter control. Moreover, it does not treat the network as a black-box, which is a more general and simpler representation.

More recent studies that use control-theoretic approaches in controlling performance guarantees were reported in [3, 55, 107]. These studies focused on server-side control, such as web servers or Lotus Notes email servers. Although the domains of these studies are different from those of CONNET, network and traffic control, they are useful examples for us to illustrate many aspects in computer-based control design.

The idea of delay-controlled AQM has been cited before in TSQ [76] and RED-Worcester [108], as both dealing with delay-sensitive versus throughput-sensitive traffic. They both use delay hints to differentiate between different delay-sensitive traffic when doing AQM, and the latter suggests an AQM scheme that can handle both delay and throughput. DQM provides a more dynamic and more robust solution for delay-sensitive traffic specifically through the use of a robust control algorithm.

The type of control closer to that used in this chapter is Adaptive Bandwidth Control (ABC) [119] and the references thereof. CONNET differs from this type in that it does not change the service rate, but rather uses traffic controllers to police the non-premium traffic before entering the queue, which is much easier for controlling queue occupancy than when controlling the service rate. This works well for better deployment. CONNET is also different from rate-based scheduling techniques such as the ones in in [134], as we study single FIFO queues and do not employ any particular scheduling mechanism (other than FIFO).

4.8 Concluding Remarks

In this chapter we have presented a new simple, yet robust delay and jitter control mechanism, called CONNET. It is designed to protect premium traffic that shares bottleneck links and nodes with non-premium traffic, and provides a predictable performance following desired delay and jitter guarantees.

CONNET is a *reservation-less* mechanism and is based on the relationship observed between the delay/jitter of premium traffic and the amount of non-premium traffic sharing the same link. This relationship has been experienced before in Chapter 3, where we found that both delay and jitter increase if the competing traffic rates inside a network node increase. We control the competing non-premium traffic at the input of the network pipe to achieve the required premium delay/jitter.

In order to do this in a more dynamic way than the approach in Chapter 3, we took a control-theoretic approach to studying the delay and jitter control on FIFO-based network pipes. Again, we first obtained a "black-box" model for the FIFO pipe under study using

system identification (SysId) methods. Then, we presented the design and implementation of feedback control on a testbed network using the LQG regulator. CONNET creates automatic and *self-controlled* network services that require a minimal operational effort. Accompanied with intelligent reference-setting algorithm, it can achieve "below-or-equal" delay or jitter values compared to the requirements. We also introduced a delay-controlled AQM (DQM), based on the same principle, for delay- and jitter-sensitive applications. It provides a "delay-aware" replacement for the AQM schemes on the Internet.

We evaluated the performance of CONNET under various experimental scenarios to show its correctness, accuracy and robustness. We also compared CONNET with other well-known rate-based disciplines such as CBQ, WFQ, and RED, revealing its significant improvement in delay and jitter performance over the other schemes. Since it is designed to handle single-FIFO-queue networks, CONNET provides a better solution for today's FIFO-based Internet without requiring any modification or any special scheduling mechanism in the network routers. In future, we would like to address the limitations of the current version of CONNET, and explore ways of integrating it into a more general traffic control framework that creates self-controlled end-to-end (e2e) network services across the Internet.

CHAPTER 5

Paving the First Mile for QoS-dependent Applications and Appliances

Building network services that support QoS and controlling QoS inside the network does not guarantee that the end-user applications will be able to use them or at least feel their existence. This is one of the reasons for delaying wide QoS deployment because usually end-users have no access to these high elevated services and they end up being not used or even not required by the network service providers. Our work in the thesis so far presents solutions for QoS control in the network without emphasis on how to use them. For this reason, we were encouraged to introduce a system that provides QoS accessibility and support to QoS-dependent applications (or QoS applications in short) and devices in customer networks. It also promotes the use of the techniques proposed in this thesis in providing real benefit to end-user network applications.

The past few years have witnessed the convergence of digital technologies such as television, telephony, publishing, and computers. This convergence has yielded the emergence of many *QoS-dependent* applications on the Internet [123], that require certain service quality from the underlying network such as throughput guarantee, reliability, timeliness, and guaranteed delivery. For example, a wide variety of home devices and computing facilities, such as personal desktop computers, voice over IP (VoIP) phones, home telemetry and automation devices, home entertainment (mostly multimedia), shared data storage devices, as well as others, are getting connected via a Home Area Network (HAN) in what is called

"smart homes." A HAN, as well as other examples of customer networks, are usually connected to the external network via cable modems, digital subscriber line (xDSL), ISDN, optical fiber, IEEE 802.11 wireless, as well as other standards. Due to the wide range and the dynamic operation of applications running on these devices, they require different network resources, and these requirements vary from local (within a HAN or customer network) to global (end-to-end) resources. Additionally, different physical media connecting the customer network to the external network affect the overall performance of such applications. Maintaining a satisfactory QoS level for applications in such varying network connectivity and complex application interactions poses a significant challenge that requires intelligent resource allocation and admission control as well as adequate QoS support to handle diverse link media and dynamic link quality associated with each medium.

On the other hand, with the introduction of new and different multi-service network technologies to the Internet, it becomes a tedious job to upgrade every application in order to accommodate the new technology. Moreover, managing service contracts with different network service providers as well as resource usage accounting increase complexity in customer network management and end-host devices. Therefore, an abstraction layer is needed to provide an interface between the QoS-enabled network and QoS-dependent applications. This abstraction layer hides all the complexity of Service Level Agreement (SLA) management, QoS mapping, admission control, and even data plane traffic handling such as packet marking and policing.

Previous approaches depend on either putting all of these functionalities in the end-host operating system (such as IBM AIX [88] and Windows 2000 [2]) or deploying hardware devices, called *QoS appliances* [27, 94, 102]. The former approach does not address the QoS needs of other architectures, like small-memory devices and embedded systems, and also creates heavy dependency on the end-host operating systems. This has the disadvantage of relying on the QoS support "canned" in the underlying operating system, hence yielding inflexible solutions and limiting deployability. On the other hand, the latter approach provides only bandwidth management for different applications traffic. However, it does not provide enough flexibility to export network services up to the application level. Usually, some kind of mapping or translation is required between the application QoS requirements

and available network services. The intelligence found in current QoS appliances is not sufficient enough to provide such QoS mapping.

To address the above challenges and, at the same time, to provide QoS accessibility and support to QoS-dependent applications and devices, we present a two-tier architecture, called *QoS Gateway* (QoSGW), that acts as a QoS mediator between these applications/devices and the underlying network QoS infrastructure. The QoSGW also acts as an abstraction layer that hides the complexity of the QoS-enabled network from QoS devices and applications. This has the advantages of shielding the device or the end-host application from the details of the underlying QoS infrastructure and facilitating both management and deployment. Our architecture provides middleware QoS support for applications, which does not depend on the end-host operating system and does not assume QoS-aware applications.[1] It also provides a transparent support for current QoS applications without the need to modify their source code or operation. We employ the Differentiated Services (DiffServ) [13] architecture as the underlying QoS infrastructure, but the approach is general enough to be used with other QoS infrastructures. We implement a prototype system using the Linux operating system to prove the functionality and evaluate the performance of our proposed approach.

5.1 Basic Architecture and Operation

The QoSGW architecture is composed of two components: an agent that is working as close as possible to QoS applications and devices in the access network (tier 1), called *QoS Agent*, and a *QoS Manager* that is working closer to an external provider network before the access router (tier 2). Depending on the size and capability of the supported device or host, the *QoS Agent* is to be installed[2] to manage applications traffic. The *QoS Agent* provides a transparent QoS support for end-host applications by intercepting their network connections and directing QoS requests to the *QoS Manager* so that it can assign suitable service classes for applications traffic. The host agent also provides Application Program-

[1] QoS-aware applications are the ones that use QoS APIs to convey their QoS requirements.
[2] Small embedded devices will not be able to accommodate a QoS Agent.

ming Interfaces (APIs) to register applications for QoS support. These APIs basically add a QoS profile for each application to be run on the host in the host database, which defines all supported applications on that host. QoS profiles are used to map application QoS requirements into their equivalent network-level QoS representations that are later used by the *QoS Manager* in assigning service classes. This two-level QoS mapping allows more flexibility and degrees of freedom in meeting application QoS requirements through available network services. The host agent keeps track of running QoS applications on the host as well as their QoS profiles.

QoS Managers, on the other hand, process QoS requests sent by *QoS Agents* and allocate network resources and suitable service classes to applications' traffic. The managers keep track of the SLA with the network service provider and apply appropriate policies and admission control to applications' traffic accordingly. They keep two types of SLAs: application SLA (*appSLA*), and network SLA (*netSLA*). Application SLA carries per-flow specific parameters. Each *appSLA* has a mapping to one *netSLA*, which, in turn, defines the associated network service class. Network SLAs are negotiated with the network service provider (ISP), but the negotiation process is outside the scope of this thesis. Applications traffic is mapped to network service classes through packet marking.

Once the *QoS Manager* has been instructed to support a particular application traffic, it starts the admission control process that checks resource availability as well as policy rules. It also verifies the feasibility of meeting the e2e QoS requirements for the requested flow through the specified service class. This process can be summarized as follows: the manager compares the requested QoS against the available service classes defined in the list of *netSLAs*. If a class matches, then the capacity of that class is examined to see whether there is enough bandwidth to accommodate the new request. If there is enough capacity, then the manager starts an e2e QoS verification procedure that involves a signaling protocol like RSVP [135]. The result of this e2e verification procedure determines if the request can be admitted or not. If the request is admitted, the manager installs a new *appSLA*, which defines QoS parameters for that request, and starts installing traffic classifiers and traffic markers for the traffic flows.

Finally, a reply is sent back to the host's *QoS Agent* reporting the result of the admission

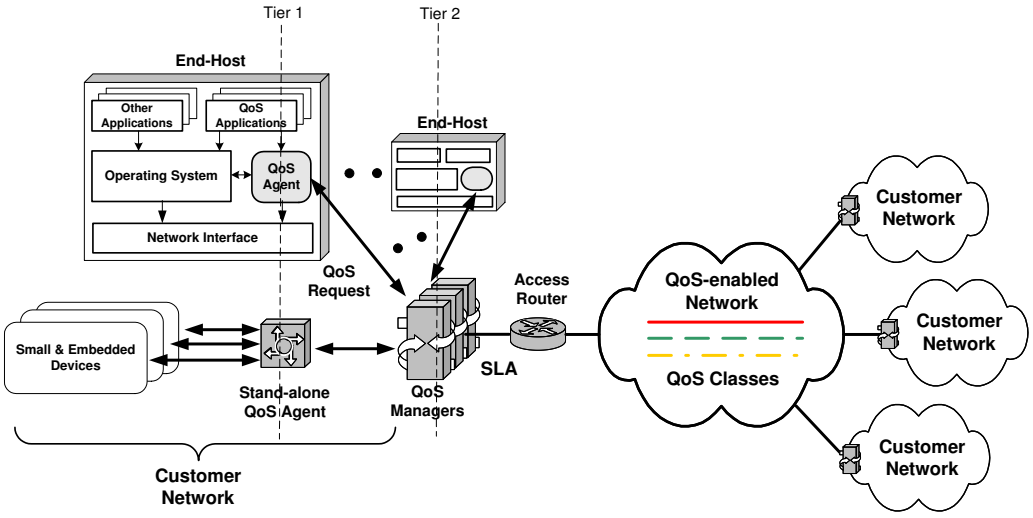


Figure 5.1: Basic architecture of the QoS Gateway

and the QoS mapping process. Then, the application can either start sending traffic under the specified network service class or choose to modify its QoS requirements if the original specifications could not be met by the network. Figure 5.1 shows a general architecture of using *QoS Agents* and *QoS Managers* to establish services for applications running on hosts as well as on small and embedded devices. In the figure we also see two types of agents, *host-based Agent* like the one we already described, and *stand-alone Agent* which provides support for small and embedded devices through a heterogeneous device interface. More details about the design of these two agents are to follow later in this chapter.

5.2 Design Considerations

The design of the *QoS Gateway* is meant to support various types of working environments such as small-scale networks (LANs), corporate networks, campus networks, and even a metropolitan network. At the same time, application support may vary from electronic mails (e-mails) to file transfer to multimedia streaming to even more critical and complex real-time applications that are motivated by the current network technological advances. In the process of designing the *QoS Gateway* architecture, we were guided by a list of design considerations that also represent the unique features of the QoS Manager/Agent

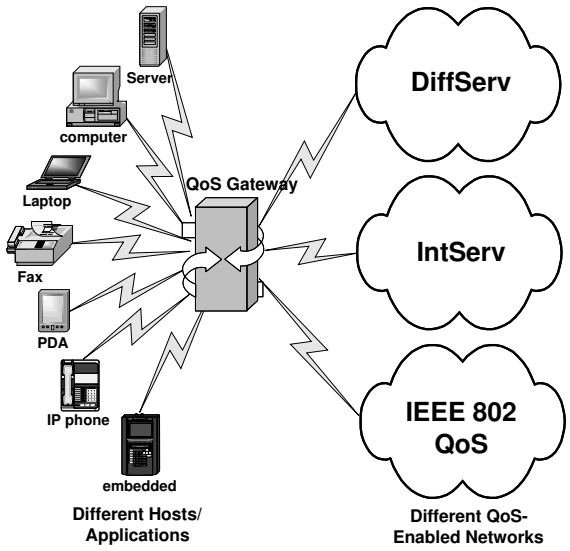


Figure 5.2: Abstract interface to heterogeneous devices and networks

approach. In the following we list these considerations:

Isolation: The QoSGW provides an abstraction level that hides the complexity of the underlying network QoS infrastructure. It provides a uniform interface for applications to external networks such as IntServ, DiffServ, or even IEEE 802 network QoS (Figure 5.2 illustrates this).

Heterogeneous interface: Different devices use different connectivity standards and links to connect to the external network. LAN, DSL, PPP, IEEE 802.11, USB, RS-232, Bluetooth, infrared are a few examples. It is necessary for the proposed architecture to support multiple interfaces to heterogeneous hosts/applications as well as being compatible to communicate with them on different processing scales. Figure 5.2 also illustrates this concept.

Closeness and transparency to applications: The use of an agent that resides on the host and interacts closely with QoS applications has the virtue of capturing the application requirements accurately. At the same time it provides a transparent QoS support (through a middleware) without the need to modify the applications' source code or the underlying end-host operating system. This is also required for the stand-alone

agent that works close to devices.

Flexibility: For lightweight end-hosts, such as embedded and hand-held wireless devices, most or all QoS management functions and network support are provided by the stand-alone agent. For more sophisticated and larger capacity devices such as personal computers, a special agent, *QoS Agent* is to be placed on such devices to perform part of the QoS management function, making the *QoS Manager* simpler and more compact. The reason for having a *QoS Agent* is scalability as well as cost-effectiveness. Consider the case of a subnetwork of 100 hosts on each of which 10 different applications are running. If the *QoS Manager* is to do the entire job, then it has to handle 1000 applications' QoS simultaneously and in real-time. This is really a burden on the *QoS Manager* while part of the functionality can be performed on the host close to the application. In a typical apartment, personal computers are more commonly deployed and used than tiny and embedded devices, resulting in a compact design for the *QoS Manager*, and this is desirable as this device is usually provided by the network provider or the site administrator.[3]

Easy administration: Administration of a few QoSGWs in a customer network is relatively easier than managing every host and device in the network. This includes policy, resource, or SLA updates. Moreover, controlling the network policy from a trusted device, *QoS Manager*, is securer than distributing the policy among various trusted and untrusted devices and applications.

Easy upgrade: Separating functionality inside the QoSGW is very important to facilitate the upgrade and update of the architecture components in order to support future and new network QoS frameworks. Both entities of the QoSGW follow a modular design that allows replacement, addition, or even removal of some of the functionality and protocols involved in the operation.

Multi-level of QoS mapping: QoS mapping is done at both the agents and the managers levels. This allows more freedom in meeting QoS-application requirements and pro-

[3]Further on this point will be discussed later in Section 5.4.3.

vides a larger adaptivity margin to accommodate transient and dynamic changes in the external network.

5.3 QoSGW Design and Functions

We first detail the *QoS Agent* design, specifying the main building blocks, functions, and their interactions, and then present the detailed design of the *QoS Manager*.

5.3.1 Host-based QoS Agent

The main functions of the *QoS Agent* is to capture applications' QoS requirements and other traffic characteristics, perform initial QoS mapping, and then communicate the application requirements to the *QoS Manager*. The host-based *QoS Agent* is designed as a middleware that resides on the end-host and enables applications running on this host to access QoS in the network. Using a middleware design eliminates the need for modifications of the host operating system. The *QoS Agent's* functionality is divided into two parts, a QoS module (*QoSmod*), which works in the operating system's kernel space,[4] and a user-space process, which is called QoS daemon (*QoSd*). The functions of the user-space daemon include QoS mapping, exporting APIs, traffic monitoring, and communication with the manager. The kernel module is employed for intercepting applications' network connections and managing the QoS application processes (mainly bookkeeping and controlling the start and end of application processes). The daemon and the module communicate via a special channel or a device driver.

The *QoSmod* intercepts applications' network connections and contacts the daemon to perform QoS mapping, and admission control with the *QoS Manager*. The application has to be registered with the host agent in order to intercept its network connections. Specific APIs are used to register new applications' QoS profiles in the host database. Before storing the profile in the database, the agent maps the application's QoS requirements to an intermediate form that is composed of bandwidth, delay, jitter, and loss. These parameters

[4] A kernel module in our Linux implementation.

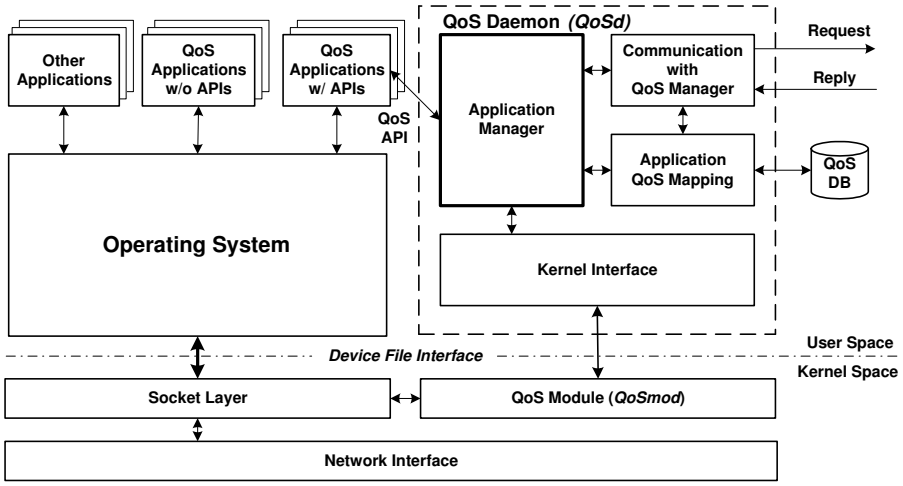


Figure 5.3: *QoS Agent* main blocks

are called "network-level QoS." This is the first level of QoS mapping done to the application requirements. Figure 5.3 shows the main building blocks of the host-based *QoS Agent* as well as their interactions, and in what follows, we give details of the functionality of each building block.

Socket-Call Capturing Module

QoSmod provides a transparent interface to registered QoS applications through intercepting their socket system calls. For each intercepted socket call, the module notifies the user-space daemon, *QoSd*, and delivers important information about the call to the daemon. This information includes the type of the call, the calling application name, the process identifier, network addresses, ports, and protocol type. This information is used by *QoSd* to locate the QoS profile of the application in the database and performs QoS mapping. The network addresses and ports as well as protocol type are sent within the QoS request to the *QoS Manager* to be used in traffic classifiers to be built for this application traffic on the manager. The *QoSmod* blocks the application (similar to waiting for a device [15]) until a reply comes back from the daemon. Once a reply is received from the daemon, the module decides whether to continue the connection normally (if the application's QoS can be met) or to drop the request if the application's QoS is rejected). Then, the module unblocks the application with either normal or error code. Figure 5.4 shows how *QoSmod* processes a

system call from a QoS application.

In our Linux-based prototype, the module communicates with the user-space daemon through a special character device file called `/dev/diff0`. The module, when loaded, attaches itself to this device file, and provides read, write, poll, as well as other necessary functions to serve the file in the kernel space [15, 111]. The module also creates two message queues, a "send" queue (`sndq`) and a "receive" queue (`recvq`), for sending and receiving messages to and from the *QoSd*, respectively.

The daemon, on the other side, listens to the device file waiting for any message sent from the kernel module. Messages sent to the daemon are either reporting a socket call, informing the daemon of a process state change, or replying to a request from the daemon. A message reporting a socket call carries the user and the process identifiers of the calling process. It also carries the necessary information about the socket call like the protocol family (e.g., AF_INET), the type of the call (e.g., accept, connect, send), the protocol (e.g., TCP, UDP), and the `sockaddr` structure that have the addresses and port numbers of the two ends of the socket call. A message reporting a process state change contains the process identifier, and the current status of the process. Finally, a reply message to a daemon request contains the result of executing such a request.

Messages received from the daemon contain a message type, process identifier, size, and a command from the application manager. There are two types of commands, either terminating a certain process (or application) or replying to a socket call message from the kernel module. Upon receiving a terminate command or a terminate reply, the kernel module terminates the process immediately. Figure 5.5 illustrates the kernel interface in both the daemon and the kernel module and shows how they work together.

Application Interface and Management

The *QoSd* provides a simple API, called `RequestQoS()`, to specify application's QoS requirements to *QoS Agent* and hence to the *QoS Manager*. The API supports three functions, `addQoS`, `modQoS`, and `delQoS`. The first function, `addQoS`, is used to create a new QoS profile for an application's traffic and store it in the host's QoS database.

We use the term "QoS profile" to describe an entry in the QoS database. A typical

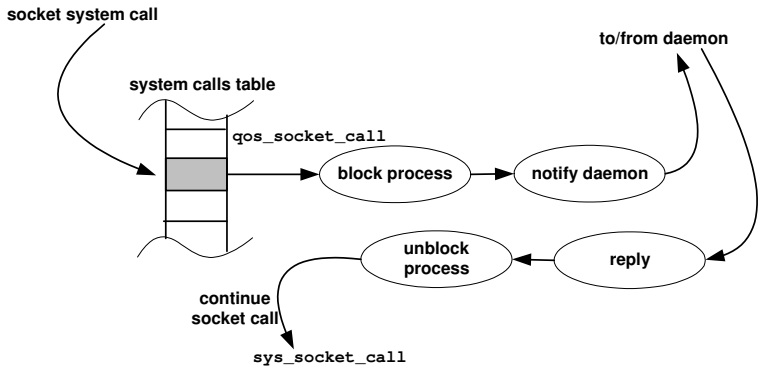


Figure 5.4: *QoSmod* operation

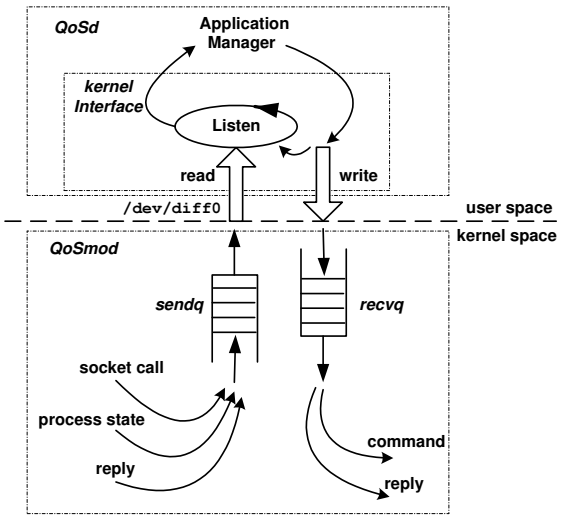


Figure 5.5: Kernel interface

application's QoS profile consists of the application's name (as a unique identifier), its type, number of flows generated by the application, traffic profiles of the flows, flows' identifiers (addresses, ports, and protocol of the traffic flows), user authentication information (user and group identifiers[5]), and canonical QoS object containing the required QoS for each flow.

Application's QoS requirements are mapped first to their equivalent network-level QoS before being stored in the host database. We refer to this procedure as "application registration." Figure 5.6 illustrates the main steps for registering an application with *QoS Agent*, where the *QoS Manager* is contacted only to find a matching network service class. How-

[5]We assume a UNIX-based or other multi-user system.

ever, the manager does not install traffic controls until the application starts sending traffic as will be shown later. After successful creation of the application's QoS profile, it can be modified or deleted altogether using `modQoS` and `delQoS` functions, respectively. The API functions can be used by QoS-aware applications or a manual configuration interface for other legacy QoS applications.

The Application Manager keeps track of registered QoS applications running on the host. When an application starts, *QoSmod* intercepts the application's socket call and notifies the Application Manager while blocking the application. If the application is registered in the QoS database, then the corresponding QoS profile is fetched and a QoS request is built to be sent to the *QoS Manager*. The *QoS Manager* is contacted to perform admission control, network service class selection, *appSLA* installation, and traffic control. When the QoS Manager replies back to the agent, the Application Manager adds this application to the active list and unblocks the application to continue normal socket call processing. This procedure is called "application invocation." If the application is not registered, a normal socket call will continue without contacting the *QoS Manager*. When an active application terminates,[6] the Application Manager notifies *QoS Manager* to delete the associated *appSLA* and free the resources allocated to the application's flows. Figure 5.7 illustrates the application invocation procedure.

QoS Mapping

QoS applications may have their own definitions for QoS and one of the main functions of *QoS Agent* is to translate these various definitions to a common QoS form to be sent to the *QoS Manager* and hence map it to a specific network service class. The QoS mapping module takes care of implementing different mapping algorithms for this translation process. The module also keeps track of the QoS database stored on the host, which records all QoS applications that can run on this host. Records in this database carry a common form for applications QoS which is referred to as the "canonical" QoS. This canonical form specifies QoS requirements for bandwidth, delay, jitter and loss. Specifically, it consists of a committed information rate (CIR), a peak rate (PIR), mean delay, maximum delay, jitter,

[6] *QoSmod* notifies the daemon with application closing connection.

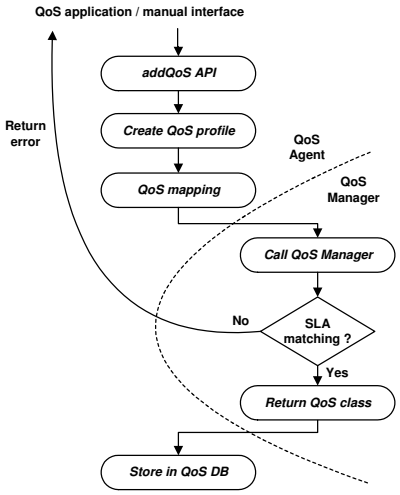


Figure 5.6: Application registration procedure

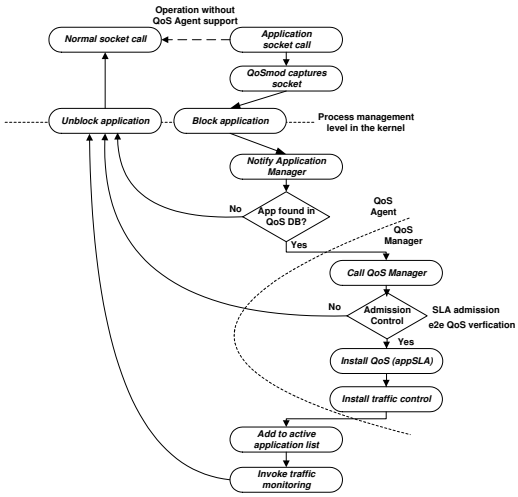


Figure 5.7: Application invocation procedure

and loss. We believe that any application's QoS can be mapped to this finite set of parameters using the appropriate algorithms. The QoS database carries also information about the traffic characteristics of these applications as well as information about their users. The latter information enables the agent to treat the same application differently depending on the user.

In this chapter we only give an example of this mapping process; the full set of mapping algorithms is a topic for future work. Consider a video transmission application that wishes to transmit frames at a rate of 30 frames/sec. Each frame can be sent at two different resolutions, a high resolution of 1280×1024 and a low resolution of 1024×768, and the application can tolerate the loss of 10% of frames every 30 consecutive frames.[7] At the other end, frames have to be played back at a specific speed and according to a play-out timer. If a frame is late by $1/2$ of a frame time, it is considered as lost, and is not played back. Consider also that the video received is used to activate a control action based on its content, and this control action has to be taken within 100 msec.[8] The algorithm for this type of applications is straightforward. The CIR is the *rate* (in bits/sec) for transmitting lower resolution frames, and the PIR is the rate for transmitting higher resolution frames.

[7] This depends on the codecs used to send frames.

[8] Video sensors are examples of such a situation.

The delay would be equal to the control delay (100 msec). The e2e jitter equals $1/2 \times frame\_size/rate$ and the loss rate would be a $(1/10 \times frame\_size)/(30 \times frame\_time)$ in bits/sec, where $frame\_time$ is the $frame\_size$ divided by the $rate$. This shows an example of how to extract the canonical QoS form from the application's QoS specifications. However, the job is not always easy like this example, and it can be a very complex operation and may also involve the user's perspective, which is still an open issue.

Traffic Monitoring

QoS Agent monitors applications traffic to make sure that it conforms to the profile specified during the application registration. Traffic monitoring is done using the packet filters user library (libpcap)[9] [79]. A monitoring thread is started for each application traffic flow and each thread installs a filter to select packets from that flow. Collected measurements are compared to the registered traffic profile of the application flow and any violations are reported to the application manager to take necessary actions.

Communication with QoS Manager

QoS Agent communicates with *QoS Manager* to convey applications' QoS requests and to set up the required support for these applications on the Manager. Communication is done on a request-reply basis as follows. The request (called *QoSrequest*) that is sent from the Agent to the Manger consists of a header and a body. The header includes mainly the type of the request and the application flow identifier to be used by the Manager to set up the required classifiers and traffic support functions. The body consists of a canonical QoS object specifying the required application QoS after being mapped, and a traffic profile (*Tprof*) description of the traffic flow to be supported. The traffic profile follows the Dual Leaky Bucket (DLB) model and includes the average rate, peak rate, burst size, average packet size, and maximum packet size of the flow. This information is to be either given by the application in the registration process described before, or by the QoS mapping module since each application has a QoS profile. Each member in *QoSrequest's* QoS object is

[9] In Linux implementation.

represented in terms of a value and a time window for measuring this value.[10]

Once *QoS Manager* finishes processing the request, a reply is sent back to the Agent that contains the result of the admission control procedure done by the Manager, an error code if something went wrong, and a handle to the newly-installed record on the Manager to be used later if any update is needed to be sent to the Manager. Upon occurrence of error or rejection in the admission control procedure, the communication module reports back to the Application Manager to take the necessary action. The *QoS Agent* sends four types of messages to the Manager.

check SLA: During application registration, *QoS Agent* asks the manager if there is a matching network service class to the application. This is shown in Figure 5.6.

install QoS: Likewise, during application invocation, the Agent requests the manager to do admission control and install the necessary support for the application traffic. This is illustrated in Figure 5.7.

delete QoS: When an application terminates, the agent sends a notification to the manager with the traffic id to be deleted from the manager's support. The traffic control setup initially for the traffic is uninstalled from the manager as a result of this request.

modify QoS: The agent sends this message to the manager when it needs to re-adjust the traffic control components according to a changing application characteristics. An example of this situation includes change of QoS requirements which requires the *QoS Manager* to redo admission control, e2e QoS verification, and traffic control. As another example, a change of traffic profile or traffic id (such as port numbers) requires change of traffic control components.

5.4 Stand-alone Agent

The main function of the stand-alone Agent is to support connecting devices (running applications) through heterogeneous interfaces. Instead of the socket interception layer in

[10]This way we can apply a moving average in measuring these quantities.

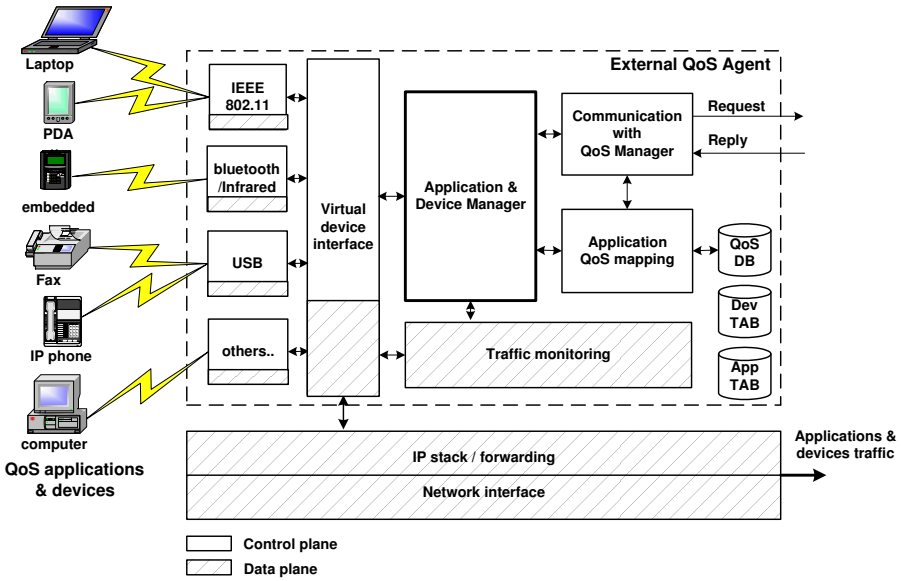


Figure 5.8: Stand-alone *QoS Agent* main building blocks

the host-based Agent, the stand-alone Agent captures the application/device traffic and QoS requirements through a multi-protocol interface layer that can connect to a wide variety of today's home/office devices. This layer, as shown in Figure 5.8, is composed of a number of "protocol-specific" modules or device drivers for different interfaces (e.g., IEEE 802.11, Infrared, USB, as well as others). These modules are all abstracted by a second layer of *virtual device interface* that provides a single interface to all physical media. The rest of the functional building blocks in the figure are similar to the ones in the host-based Agent including all the device/application management and first level of QoS mapping.

5.4.1 Protocol-specific Modules

These modules are responsible for direct interaction with devices/applications through different protocols. The main functionalities of these modules focus on providing a protocol-specific link layer to the Agent. They deal with protocol data units in lower layers than the IP layer including packetizing/depacketizing of application/device data. They also act as device drivers for the virtual device interface and thus for the whole Agent as well.

5.4.2 Virtual Device Interface

The virtual device interface provides an abstraction level for the heterogeneous protocol-specific device interconnection modules. It hosts the common functionality to be executed for all connected devices. Among this functionality, we list the following:

- Application/device connection interceptor
- Device communication primitives: discovery, registration, description, eventing, control, deregistration, blocking, errors, etc.
- Flow facilitators (forwarding): address translation/assignment, forwarding tables and fire-walling, security
- Traffic monitoring: although not really a part of the virtual device interface, it is a part of the data plane of the Agent
- For voice over IP (VoIP) devices, voice gateway functionality should be supported. This needs to connect with a Subscriber Line Interface Circuit/Coder-Decoders (SLIC/CODECs) units to connect directly with voice-enabled devices such as IP phones. The control plane has to provide functionality for the following as well: SIP, MGCP, PSTN proxy, and DSP capability.

Figure 5.9 illustrates the main steps taken to support a device through the virtual device interface. This interface layer assumes a physical connectivity to the device through the protocol-specific modules.

5.4.3 QoS Manager

The *QoS Manager* acts as a QoS server for the set of agents in the local customer network, and provides an interface for applications to access various network QoS facilities and service classes. For flexibility and upgradeability, the *QoS Manager* is composed of two planes, a control plane (called *Gateway*) and a data plane (called *Diff Agent*). The *Gateway* handles high-level QoS control functions such as SLA management, QoS mapping, admission control, and network QoS signaling. These functions are independent of

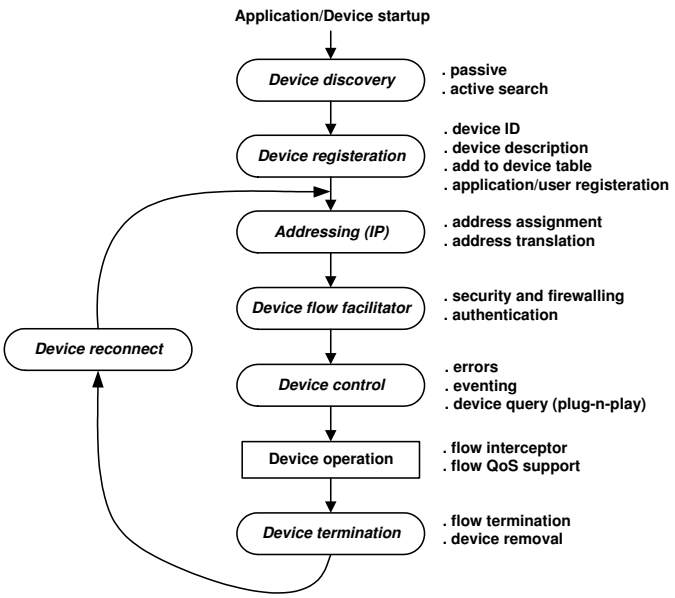


Figure 5.9: Device management and device lifecycle

the underlying network environment and can work with various QoS network infrastructures. The *Diff Agent*, which handles traffic marking, traffic classification, and traffic control, is network-specific and tailored to match the supported QoS network infrastructure such as DiffServ or MPLS. These two components communicate with each other using standard socket APIs. Therefore, these components can be located on different machines for fault-tolerance and scalability.

Our current architecture uses the DiffServ framework as the underlying QoS network infrastructure, and hence, the *Diff Agent* is DiffServ-specific. Figure 5.10 illustrates the main building blocks of the *QoS Manager* as well as their interactions, and the following subsections give details of the functionality of each building block.

SLA Management

The *QoS Manager* acts as an interface between the offered network service classes and the application's QoS requirements, and manages SLA with the network service provider. It maps application requirements to appropriate network service classes in the SLA. As mentioned in Section 5.1, the *QoS Manager* keeps track of two types of SLAs, network

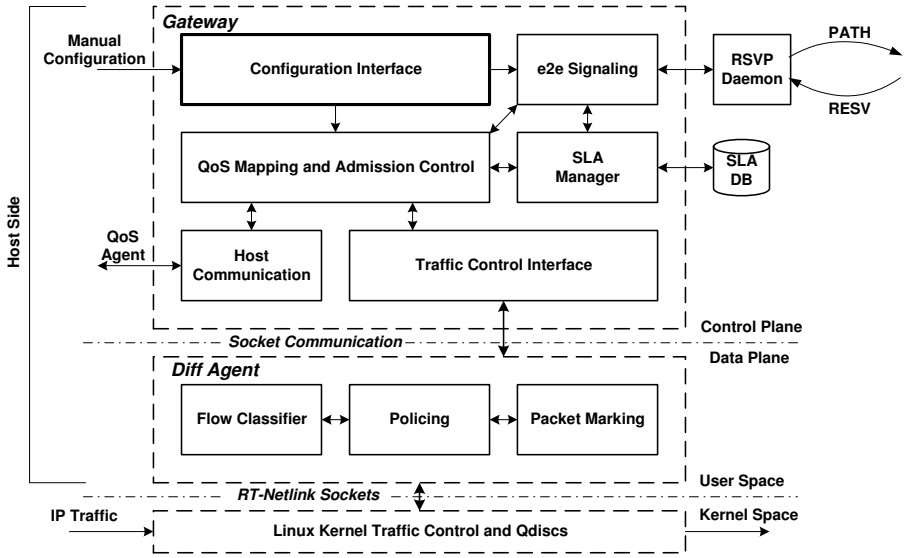


Figure 5.10: *QoS Manager* main blocks

SLA (*netSLA*) for each network service class, and per-flow SLA (*appSLA*) for each active QoS application as illustrated in Figure 5.11. This is the second level of QoS mapping, which maps *appSLAs* to *netSLAs* and is illustrated in Figure 5.12.

Each *netSLA* entry consists of a unique identifier, a DiffServ service class, a list of DSCPs (each with 8 bits) associated with this service class, allocated bandwidth, available bandwidth, and a bandwidth sharing indicator (for bandwidth borrowing from other classes). Entries for *netSLAs* are stored in a database "SLA DB" as shown in Figure 5.10. On the other hand, each application flow is associated with an *appSLA* entry. Each entry consists of a unique identifier, a flow identifier (containing source, destination, ports, and protocol type), a traffic profile of the flow, application's QoS in the canonical form, and a pointer to the containing *netSLA*. As application flows are admitted, *appSLAs* are created for each flow and mapped to suitable *netSLAs*. Available bandwidth in *netSLAs* is adjusted according to subscribing applications.

QoS Mapping

Typically, network service providers specify services classes in their SLAs in terms of capacity (amount of bandwidth available), latency (delay), delay variation (jitter), supported packet or frame sizes, availability (percentage of time available), reliability (some-

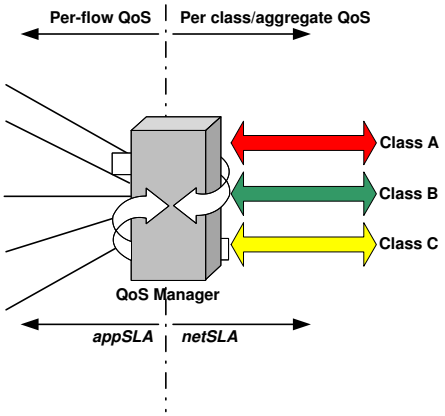


Figure 5.11: Network SLA (*netSLA*) and Application SLA (*appSLA*)

appSLA1	netSLA1
appSLA2	
....	
appSLAi	
appSLAj	netSLA2
....	
....	
appSLAk	
	netSLAm
....	
appSLAn	

Figure 5.12: *netSLA* $\leftrightarrow$ *appSLA*

times referred to as loss rate), and schedule of operations. The *QoS Manager* maps the QoS of the application (*appSLA*) to one of the available services in the SLA (*netSLA*) by matching the values of its canonical QoS parameters to the service specifications. A simple example is the mapping of an application's QoS to one of the available service classes in a DiffServ-based network. For an *appSLA* that requires delay and jitter guarantees, it is assigned to a suitable Expedited Forwarding or EF-based service [29]. For an *appSLA* that has requirements on bandwidth and loss, it is assigned to a suitable Assured Forwarding or AF-based service [59]. By "suitable" we mean matching the values of individual QoS parameters with the values of the service specifications. Services based on class selectors have more freedom for implementing services with different semantics of EF and AF services. This makes the canonical QoS form very close to network SLAs and hence simplifies QoS mapping at this stage. In most cases, this mapping process is simpler than the first level of application's QoS mapping done by the host agent, and mapping algorithms for other types of networks can be found in [11, 123].

Admission Control

Based on the service classes mentioned above, Figure 5.13 illustrates the main sequence for the admission control process inside the *QoS Manager*. Basically, the manager admits the *appSLA* if there is enough available bandwidth in the corresponding *netSLA*. The pro-

cess starts when a *QoS Agent* communicates with the manager to request QoS for some application flows. A new *appSLA* template is created to carry that request, then a suitable DiffServ service class, that matches the requested QoS, is chosen. Currently, the basic decision on selection of a service class depends on if there are delay or jitter requirements as shown in the sequence. If a matching class can be found[11] and it has enough capacity to accommodate the new application flow, then the gateway starts an e2e admission control process which we call "e2e QoS verification." This process is described in Section 5.4.3.

Depending on the result of the e2e signaling process the gateway either registers a new *appSLA* for this started flow, or rejects the request. In both cases, it replies back to the agent indicating successful or unsuccessful admission. Both the traffic profile (Tprof) and the canonical QoS object of the application flows are used in this process.

The *QoS Manager* then installs flow classifiers that distinguish packets from this flow, and installs packet markers with the correct DSCP assigned to this application flow as described in Section 5.4.3.

End-to-end QoS Verification

To provide the application with a predicted QoS level, the *QoS Manager* has to verify that the e2e path of the application traffic has enough resources and suitable service classes to support this QoS level. This verification process is also called "e2e admission control", and is done by the *QoS Manager* using a well-known signaling protocol, like RSVP [17, 131, 132, 135].

The gateway invokes the RSVP admission by communicating with a co-located RSVP daemon to handle the protocol message processing and management. The communication between the gateway and the daemon is enabled through the use of RAPI [18]. The *QoS Manager* serving the sending host (also called RSVP sender) initiates a RSVP session on behalf of the host toward the receiving host manager (also called RSVP receiver). RSVP processing along the e2e path is similar to the one described in [11, 132]. We use RSVP in our system for admission control only and not for resource reservation. Moreover, processing RSVP messages is done on a per-domain basis, not per-hop like IntServ/RSVP.

[11] This matching is done during application registration procedure.

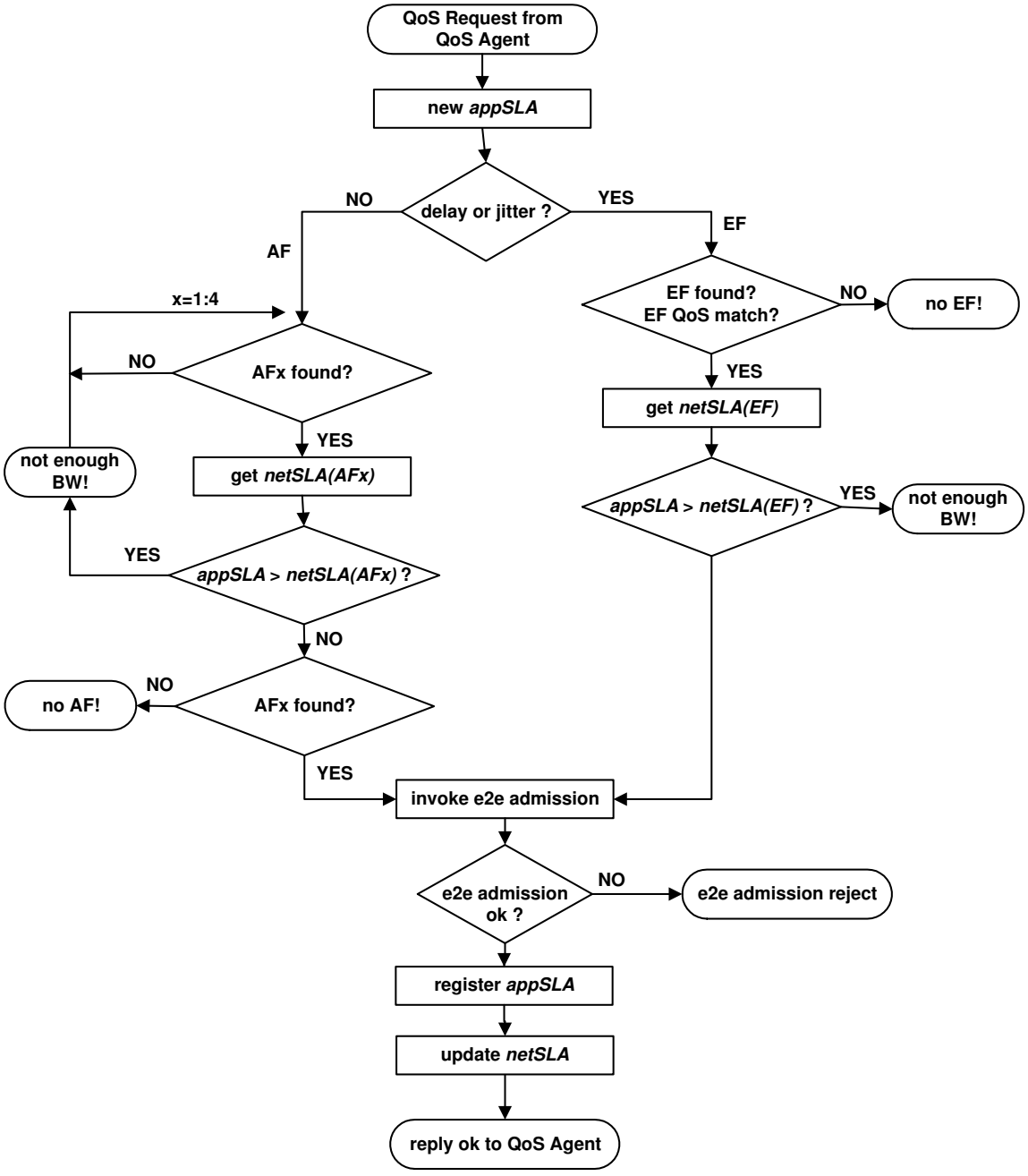


Figure 5.13: Admission control sequence

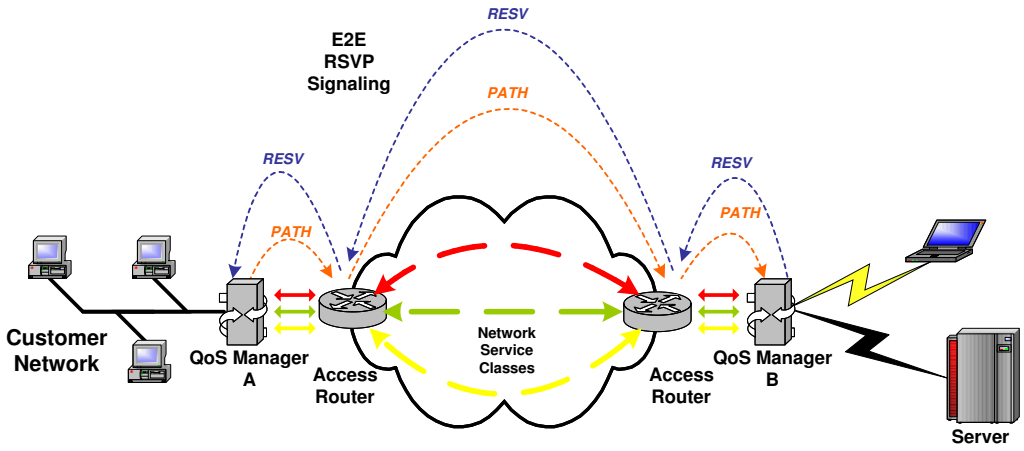


Figure 5.14: E2E QoS verification via RSVP

The process is reviewed in Figure 5.14 where *QoS Manager* A is the RSVP sender and *QoS Manager* B is the RSVP receiver. Edge routers process RSVP messages to include edge-to-edge QoS for service classes provided by each domain. The state diagram for the RSVP sender process is shown in Figure 5.15, and the corresponding diagram for the RSVP receiver process is shown in Figure 5.16.

A few changes have been made to enable e2e QoS verification using RSVP and to match our architecture. RSVP supports only the two IntServ service classes, Guaranteed Services (GS) and Controlled Load Services (CL). EF-based services are mapped to GS while AF-based services are mapped to CL while being carried by RSVP messages. In the PATH message, we use the ADSPEC object to carry the required nominal values for the application QoS carried from the QoS structure *QoSrequest* sent by the application as in Table 5.1. They are used to accumulate the path characteristics as defined in [131], but using DiffServ edge-to-edge services instead of hop-by-hop. Traffic profile parameters (*Tprof*) are carried by the TSPEC object to the receiver.

At the receiver, the accumulated e2e values of bandwidth, delay, jitter, and loss are copied into the FLOWSPEC object in the RESV message back to the sender as in Table 5.2. When the RESV message returns back to the RSVP sender, it compares the accumulated values with the original required QoS values and if they are not acceptable, an error is returned to the calling process indicating e2e QoS violation. As a result, the flow is not

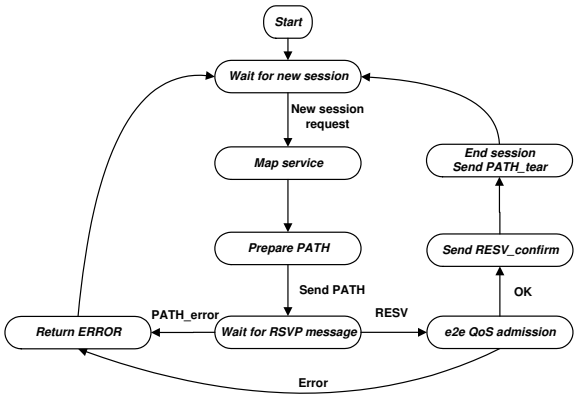


Figure 5.15: RSVP sender state diagram

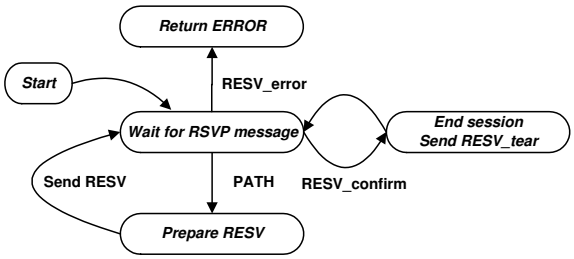


Figure 5.16: RSVP receiver state diagram

ADSPEC item	QoS parameter
Composed MTU field	bandwidth (bits/sec)
Minimum latency field	delay (μsec)
Hop count field	jitter (μsec)
Path BW field	loss (%age)

Table 5.1: Mapping in PATH message

FLOWSPEC item	e2e QoS parameter
Average rate (r)	e2e bandwidth
Minimum policed unit (m)	e2e delay
Maximum packet size (M)	e2e jitter
Burst size (b)	e2e loss

Table 5.2: Mapping in RESV message

admitted. Additionally, should an error occurs along the e2e path, a RESV-ERR message is generated and sent back to the sender and an error is returned indicating resource violation in any of the routers along the e2e path. As a result, the flow is not admitted as well. We use a third phase of messages to send a RESV-CONFIRM message from the sender to the receiver as a confirmation of the success of e2e signaling. Upon receiving a RESV-CONFIRM message, the receiver *QoS Manager* updates its service descriptions to accommodate the new established incoming flow.

Traffic Control and Marking

After admitting a flow, the manager installs traffic classifiers and traffic policers based on the flow identifier and traffic profile in the flow's *appSLA*, respectively. The data plane of the *QoS Manager* employs appropriate traffic controls and policing to do this job. In our prototype implementation, Token Bucket Filters in the Linux kernel traffic control subsystem [5] are used for traffic policing. After admitting a flow, the gateway installs a traffic classifier based on the flow identifier of the flow sent in the *QoS Agent's* request message. The traffic policer and appropriate markers are installed based on the traffic profile (*Tprof*)

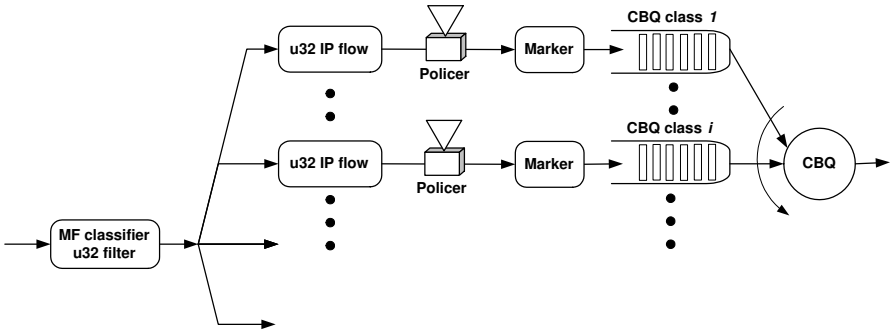


Figure 5.17: Adding markers for application flows

of the flow sent in the request. Traffic policers and markers are installed on a per-flow basis. This per-flow chain can be installed on the ingress or the egress interface of the *QoS Manager* device.

Packet markers are installed at the end of the traffic control chain mentioned in the previous section when the flow is admitted. The marker uses the DSCP assigned to the flow from the corresponding *netSLA*. Currently, we implement three marking algorithms for AF: two rate Three-Color Marker (trTCM) [60], Time Sliding Window (TSW) [39], and Equation Based Marking (EBM) [37]. For EF traffic, a single rate two color marker is used which is simply using a single token bucket filter to compare input packets against a rate and a burst size. It marks in-profile packets as EF and leaves out-of-profile packets unmarked.

Figure 5.17 shows how the *QoS Manager* arranges classifiers, policers, and packet markers. The manager installs a CBQ [12] queueing discipline at the root of the tree, and for every admitted IP flow, a new CBQ class or branch is created to support this flow. A series of u32 classifier, traffic policer, and packet marker are installed within a CBQ class as shown in the figure. CBQ classes are assigned bandwidth equal to their nominal rates with an allowance of burst that cover traffic burstiness. When the flow terminates, the manager deletes the corresponding traffic controls and markers and frees their resources. The QoSGW also uses IP Easy-pass [127, 128] in resource access control to protect the customer network against network resource theft and intrusion.

[12] To control a correct distribution of bandwidth among different traffic flows.

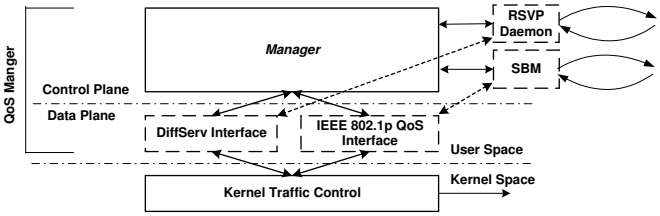


Figure 5.18: QoS Manager data and signaling planes can be exchanged to work with different underlying networks

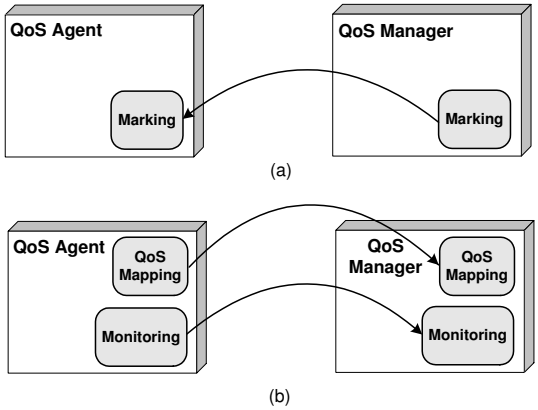


Figure 5.19: Functionality can be downloaded from the QoS Manager to QoS Agent and vice versa: (a) Marking can be done at the Agent; (b) QoS mapping and monitoring can be done at the Manager

Flexibility in QoS Manager Design

One of the QoSGW design considerations is flexibility to work with different underlying network frameworks. Figure 5.18 illustrates this flexibility in separating the control plane from the data plane in the *QoS Manager*. We can use multiple different data planes (e.g., DiffServ-specific, IntServ-specific, IEEE 802) that work with different underlying networks. However, the control plane remains unchanged. Moreover, the e2e signaling module in the *QoS Manager* can be replaced by a compatible module that uses network-specific signaling protocols (e.g., RSVP, Subnet Bandwidth Manager or SBM). If the manager is connected to multiple networks at the same time, then it can switch between different data planes.

In Figure 5.19, we illustrate how packet marking can be done by the *QoS Agent* instead of the Manager. This enables making the Manager smaller and less expensive. At the same time in case of small or embedded devices, the Manager can do QoS mapping and monitoring on behalf of the end-host. Just by intercepting the control and data traffic of the application/device, the Manager can support these functionalities.

5.5 Assumptions and Limitations of the QoSGW Design

Our QoSGW design is based on the following assumptions that enable the overall functionality:

- There is a complete deployment of DiffServ (or any other QoS infrastructure) in the external network (the Internet).
- The RSVP signaling protocol is supported by the border or edge routers in the external networks.
- There is a system for exchanging and setting up Service Level Agreements (SLAs) between the provider network and the customer network.
- Currently we assume that each application has only one flow. For applications with multiple flows per session, each flow will be handled individually.
- The network or links between the *QoS Agents* and the *QoS Managers* are secure, i.e., no intrusion. The use of IP Easy-pass [127, 128] can help removing this requirement.

5.6 Evaluation

To demonstrate the effectiveness of the QoSGW in providing QoS for applications, we conducted several experiments using the prototype we built. We use a simple network testbed shown in Figure 5.20. Linux-based PCs are used in the testbed, where all PCs are 600 MHz Pentium III with 256 MB RAM and running Linux kernel 2.4.2 with QoS and traffic control enabled. The links between all PCs are 100Mbps Ethernet point-to-point. Host 1 is running an application, which sends traffic to Host 2 passing through *QoS Manager*, Router 1, and Router 2. Both Routers 1 and 2 are DiffServ-enabled routers implementing a simple Expedited Forwarding (EF) PHB based on priority scheduling. Host 1 is running *QoS Agent* that has the application's profile in the QoS database. When the *QoS Manager* is active, the application's traffic is to be mapped to an EF-based service defined in the manager's SLA database; otherwise, the application's traffic will be treated

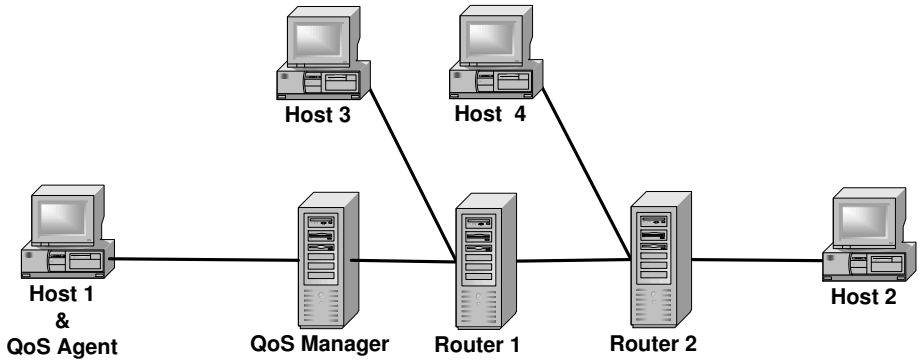


Figure 5.20: The evaluation network

as best-effort. In addition to Host 1, both Hosts 3 and 4 are sending best-effort traffic (as background) to Host 2 that enter through Routers 1 and 2, respectively, sharing the path and link bandwidth with the application's traffic. We use ON/OFF UDP background traffic with 20 sec ON and 20 sec OFF periods, and each experiment is executed for a duration of 50 sec.

5.6.1 Protection of Important Application Traffic

TCP Traffic Throughput Protection

In the first set of experiments, the application starts sending TCP traffic, and we compare the two cases of activating and deactivating the QoSGW. Figure 5.21 shows the performance of the application's traffic in terms of achieved throughput when the gateway is deactivated. As one can see, the application's traffic is not protected, and hence, suffers severely from the background traffic during the ON period. The loss here is 100%. In Figure 5.22, the gateway is activated and marks the application's traffic as EF traffic, and hence it does not suffer from any loss.[13]

[13]We can see small glitches at times 20 and 45 sec when the ON/OFF traffic changes mode, but this is negligible.

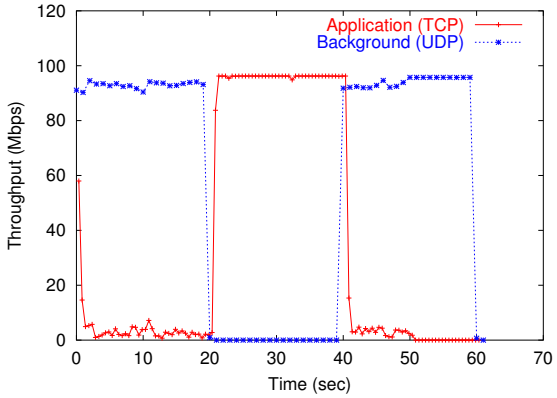


Figure 5.21: TCP throughput – QoSGW deactivated

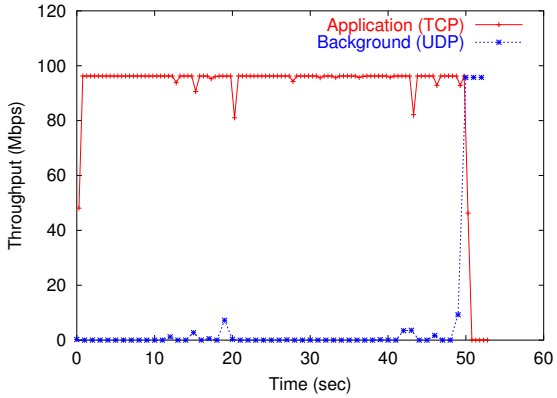


Figure 5.22: TCP throughput – QoSGW activated

UDP Traffic Throughput Protection

In the second set of experiments, the application sends real-time UDP traffic at a constant rate of 30 Mbps. In Figure 5.23, we plot the throughput of the application's traffic and the background traffic when the gateway is deactivated. It is clear that the application could not achieve more than 13 Mbps during the ON period of the background traffic as no protection is available. In Figure 5.24, we activate the gateway and as a result, the application can achieve the full 30 Mbps target rate without being affected by the background traffic.

Jitter and Loss Protection

Not only throughput but also jitter and loss are affected. In Figure 5.25, we compare the jitter with and without the gateway. Without the gateway, and during the ON period of the background traffic, the application's jitter is between 0.5 and 0.6 msec, while during the OFF period it is between 0.2 and 0.45 msec. This unpredictability in jitter is undesirable in real-time applications. When the gateway is activated, the application's jitter is almost constant at one level and unaffected by the background traffic. The application's loss is plotted in Figure 5.26 in the two cases. The application suffers no loss when the gateway is activated, in accordance with Figure 5.24.

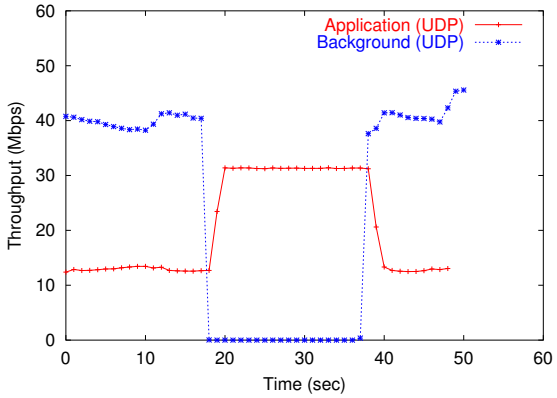


Figure 5.23: UDP throughput - QoSGW deactivated

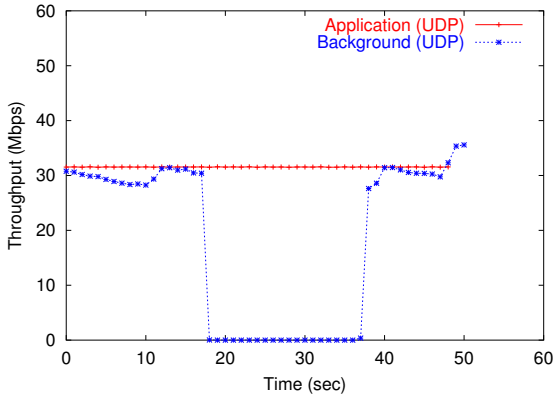


Figure 5.24: UDP throughput - QoSGW activated

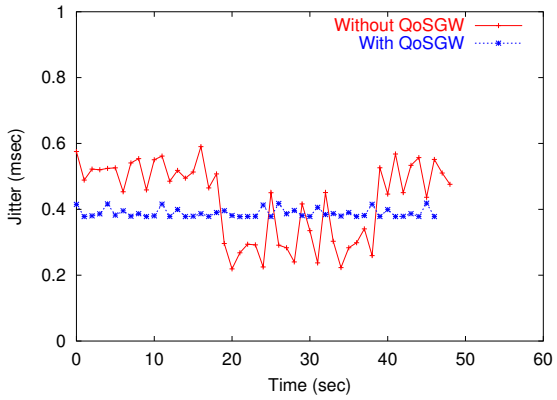


Figure 5.25: Comparing jitter with and without the QoSGW

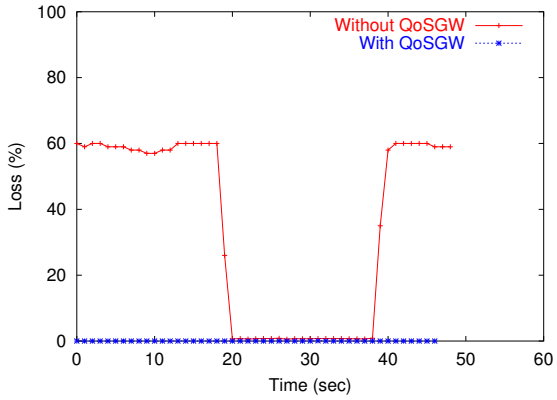


Figure 5.26: Comparing loss with and without the QoSGW

5.6.2 Overhead

We now consider the overhead in the gateway operation in supporting application's traffic. The first overhead is associated with application registration using the `RequestQoS()` API. This is usually done once for each application running on the host and can be performed as part of the application installation. Thus, no major overhead in this step. Second, at application startup time, the *QoS Agent* intercepts the application's connection and contacts the *QoS Manager* for QoS processing. This process involves a message from the kernel module to the daemon, followed by QoS profile lookup, then a network message from the agent to the manager, followed by a sequence of function calls to SLA and admis-

sion control. Without accounting for the e2e RSVP QoS verification, the manager replies back to the agent with an admission result message and installs the required traffic controls (this involves messages between the *Gateway* and the *Diff Agent*), then the agent unblocks the application and traffic starts transmission. Although this involves a number of messages to process the QoS request, only two of them are significant: the network messages between the agent and the manager.[14]

We measured the overhead caused by all these operations which was about 0.22 sec. Although the prototype is not optimized for performance, we find the setup time is not a major overhead, especially when it is executed only once at the application startup. Measuring the performance of RSVP signaling is outside the scope of this thesis. The only overhead left to account for is the overhead of marking the application's traffic on the egress interface of the manager. This process is done inside the Linux kernel's traffic control [5] code and it adds only a call to `ipv4_get_dsfield()` function which changes the DS-field in the IP headers of each packet. This adds an insignificant overhead in the packets' path. Overall, the overhead of the QoSGW is insignificant compared to its added value.

5.7 Related Work

Support for QoS at the application and host level has been taking a significant amount of efforts from both of the research and industrial communities. A close match to our approach, was given in [91], which was tailored toward ATM. The authors presented an end-point entity called the *QoS Broker* that orchestrates resources at the end-points, coordinates resource management across layer boundaries. Their architecture manages and translates three QoS parameters, application, network, and system parameters. Their design employs a broker-buyer end-point for buying resources, a broker-seller end-point for selling resources, as well as the negotiation protocol between them to establish an end-to-end QoS connection. It shares a common feature with our approach which is hiding network configuration and operational details from applications and higher layers. They use a module-structured design for ease of programming and extensibility which is also a

[14] Assuming both components of the manager are on the same machine.

common goal shared with our design. However, the main difference between our approach and the QoS Broker approach is that we propose a two-part architecture, an Agent and a Manager, with the advantages of more isolation, easier upgrade, heterogeneity, resource aggregation, more administration, and security.

The authors of [81] presented an architecture for a Residential Gateway (RG) that control the bandwidth allocation for different devices in Home Area Network (HAN). It supports automatic device discovery and dynamic bandwidth allocation. The design was based on the UPnP scheme and employs a UPnP control point and a traffic control (for QoS) modules at the RG besides a UPnP device host module at the home device or appliance. An interesting point in their work is classifying home generated traffic into seven categories and they apply bandwidth allocation between these categories according to different conditions. This categorization enables easy mapping to network offered services. Not enough details about the system architecture was given in the paper, and our overall approach follows a different track in terms of generality and flexibility.

Another architecture for QoS-enabled residential gateways is introduced recently by Motorola Labs [8]. This architecture works on layer 2 (MAC) and layer 3 (network) and is compliant with existing standards such as IEEE LAN bandwidth managers (SBM), RSVP, IEEE 802 LAN, and CableHome. A common point with our design is the use of a separate planes, a data plane and a control plane. In the data plane, they introduced a new entity called *Broadband Intelligent Bridge* (BIB) that controls traffic and QoS features among different flows. However, their architecture still lacks a lot of details which we provide in our design, such as QoS mapping, user-based differentiation, SLA management, complete admission control, and many others.

Bandwidth Managers such as PacketShaper from Packeteer [102], NetEnforcer from Allot Communications [27], and QoSWorks from Sitara Networks [94], as well as others are currently being introduced to the market. These devices are deployed to bridge customer networks and provide a toolbox of traffic management mechanisms such as traffic classification, monitoring, shaping, prioritization, bandwidth allocation, as well as other functions. A first look at them suggests that they are very close to our architecture, but they mainly do bandwidth management, not QoS support like our architecture. A second

difference is that they only employ one device that is in place of the *QoS Manager* in our design, and hence they do not have an application-sensitive agent like *QoS Agent* which interacts very closely with applications. Because they are commercial devices, not enough details about their design issues are available, and we believe that our approach is more flexible and more versatile than these devices.

Other gateway-based approaches, such as [113], exist. The authors introduced a distributed object gateway for supporting integrated QoS which is based on the Quality Objects (QuO) architecture. The approach was about a QoS-oriented Object Gateway for integrating a variety of QoS enforcement and implementation mechanisms. A QuO object gateway is a QoS-aware element inserted at the transport layer between clients and objects/servers to provide managed communication behavior for the particular QoS property being supported. Two goals were focused on: (i) providing a common platform for Integrated QoS Dimensions, and (ii) facilitating integration of new and alternative control mechanisms and policies. Our approach is different because it works at the application, transport, and network layers, and it is not based on any predefined object framework.

5.8 Concluding Remarks

In this chapter we presented a *QoS Gateway* (QoSGW) architecture for providing QoS support to QoS-dependent applications and devices in customer networks. The architecture is composed of two main components or tiers: an agent that either resides on the end-host or stand-alone and provides an adequate interface to QoS-dependent applications, and a QoS manager that provides an interface to network QoS capabilities for the agents. Our system acts as a QoS mediator between applications/devices and the underlying network QoS infrastructure. It also acts as an abstraction level that hides the complexity of the QoS network from devices and applications and provides easy management and deployment for QoS. Using the two-component approach provides better generality and scalability. The agent receives the application requirements either directly or indirectly, map them to an intermediate form, and consults the manager to map these requirements to an appropriate network service. The manager, on the other hand, serves a number of agents in providing

adequate network services that map to the application requirements in addition to other management responsibilities. Although we use a Differentiated Services (DiffServ) network architecture as the underlying QoS infrastructure, the approach we use is general enough to be used with other QoS infrastructures. The architecture presented in this chapter has some salient features such as:

- The ability to specify QoS profiles in the *QoS Agent* by user id, so that the same application can have different QoS specs (profiles) for different users. As an example, consider an IP-telephony or VoIP-based applications. Some users would be satisfied with a good voice quality in their calls. Other users may use the same application for making important calls that require higher voice quality.

- Putting applications QoS profiles in a separate QoS DB has the advantage of being able to be migrated to new systems and used in the same way. This way, a new QoS DB need not be initiated on each new system.

- Usually the *QoS Manager* is a trusted device provided by the ISP, but the *QoS Agent* can be from a third party as long as it can communicate with the installed manager. This allows the implementation of policy and other access control rules in the manager.

We have built a prototype on Linux OS implementing the main functionality of the overall design to test the operational correctness and to demonstrate the importance of our system in supporting real-time applications. We believe that a system like the QoSGW will promote QoS deployment and facilitate building QoS-aware customer networks.

CHAPTER 6

Concluding Remarks

6.1 Conclusions

We have presented in this thesis a set of versatile QoS control mechanisms, that are directed toward the dynamic Internet especially with the introduction of new time-sensitive applications. Our QoS controls draw from the basic idea of using feedback to provide an adaptive behavior required to handle network unpredictability and QoS variability. We use *model-based* feedback which provides more accurate and more reliable control in addition to other salient features such as reference tracking and robustness. The key point behind using a model of the process under control is that it gives more accurate prediction of its behavior when applying feedback control, and yields faster response and better handling of network dynamics than ad-hoc feedback methods. We demonstrate the use of the model-based QoS control approach in several case studies to prove its effectiveness as well as its simplicity. We also show that our QoS control approach can be used in today's "best-effort" Internet without the need for sophisticated QoS architectures or advance resource provisioning. This is an important feature that allows replacement of such complex and expensive-to-deploy QoS frameworks.

The thesis starts by using a previously-built TCP model, which represents TCP throughput, mainly, in terms of packet loss and round-trip time. The model is used at the heart of a feedback algorithm to control packet-marking probabilities in order to achieve a predictable throughput within the context of the DiffServ framework. The packet marking

algorithm (EBM) developed, addresses the problems found in other marking schemes regarding throughput guarantees and fairness among heterogeneous TCP flows. Through the use of this TCP model, EBM is able to better predict the behavior of application TCP traffic sources in response to network conditions, thus adjusting the marking engine performance accordingly. We compare EBM with other available marking schemes and prove analytically and through simulation experiments, that an iterative model-based control approach results in a better, predictable, and fair QoS performance than other ad-hoc packet marking approaches. We also present an accompanying packet separation mechanism that solves the fairness issue between responsive and non-responsive traffic, and we evaluate it along with EBM to achieve complete fairness in bandwidth distribution.

Different from packet marking, we use the model-based approach in controlling the *Per-Hop QoS* of a typical network node/router. In this case we develop the models used in control by employing formal statistical methods such as Analysis of Variance (ANOVA) and polynomial regression. We show that they yield simple, yet effective, functional relationships between the inputs of the node (in terms of input traffic and configuration parameters) and the output QoS, which include throughput, delay, jitter, and loss. This enables us to abstract per-hop QoS into a "black-box" model and never worry about the internal details. In building the models we use a full factorial design of experiments on a Linux-based testbed. We assess the accuracy of the models, then employ them in building a model-based controller for Per-hop QoS of a typical network node, and illustrate the correctness and effectiveness of the controller.

The success of the model-based control in these two cases led us to pursue further in using the same approach in creating Self-Controlled Network Services, or *CONNET*. This time we use more formal design and analysis methods based on control theory. Specifically, we use state-space design methods and chose optimal Linear Quadratic Gaussian (LQG) regulators for robust control. The emphasis of control in this case is on the delay and jitter of the traffic going through these network services. We extend the control from a single to multiple network nodes using the same technique. The two important features of CONNET are being "reservation-less" as well as being "self-controlled." The reservation-less feature is illustrated in controlling delay and jitter across FIFO network services without the need

of sophisticated scheduling algorithms. On the other hand, we demonstrate the ability of CONNET, as a self-controlled scheme, in tracking various QoS reference scenarios and handling variable network workloads with no significant management overhead. We show how to implement CONNET in several places in the Internet ranging from access links to domain-wide network services to active queue management. We also compare CONNET with other well-known scheduling algorithms such as CBQ, WFQ, priority queueing and RIO, and demonstrate, through experimental results, the effectiveness and superiority of our scheme.

To provide access support for QoS-dependent applications, enable the use of the proposed QoS control mechanisms, and create QoS-aware environment, we presented the design and implementation of a two-tier *QoS Gateway* architecture. Using a two tier architecture achieves both generality and scalability in providing QoS support for applications and end-devices. In this design, we introduce several novel aspects such as transparent support for legacy applications through socket interception, virtual device interface to support heterogeneous QoS-dependent devices, use of middleware to avoid operating system modifications, 2-level QoS mapping between applications and network services, and flexible design that adapts to the underlying physical network. These features have been realized through the modular design of the two entities of the gateway, namely, the *QoS Agent* and the *QoS Manager*. The functional modules of both entities can be realized independently, and their realizations are demonstrated through a Linux-based prototype. We evaluate the performance of the prototype in a simple network testbed, showing correct functionality and measuring the implementation overhead. This architecture is considered an essential step for effective deployment and utilization of the QoS techniques introduced.

6.2 Future Work

The materials covered in this thesis can be considered as the seed for further development of the same model-based control approach for various networking problems and applications. The emerging next-generation networking environment presents an IP-based core interconnecting many wired and wireless radio access networks (such as WiFi, Blue-

tooth, and 4G), providing ubiquitous access to end-users through a vast variety of hand-held devices. It is evident that the future network environment will be characterized mainly by heterogeneity of networks, especially the network access part. Although the IP protocol will be the common denominator, the new environment brings together many different interconnecting domains, each following different traffic and link models, complicating the overall end-to-end traffic control and engineering processes. A typical end-to-end network service will first traverse one access network that may be a high-speed wired segment (e.g., DSL) or a wireless LAN (e.g., 802.11), a wireless WAN (e.g., UMTS), or even a satellite network. These first-hop (and may be last-hop as well) segments will probably be supported by an IP-based core network, which will at least supply end-services with IP connectivity.

It would be interesting to analyze the new characteristics and additional requirements brought by wireless mobile environments on IP mechanisms and protocols such as link availability, capacity mismatch, location management, cell hand-offs, security, and others. One specific goal of this is to standardize the analysis and realization of well-defined network services across heterogeneous environments. Providing a unified interface to these network services is one of the expected outcomes. Building on the model-based control approach presented in this thesis, it would be interesting to investigate new techniques for building predictable, self-controlled, and self-healing end-to-end services in heterogeneous environments. In particular, it is important to study the ongoing evolution of IP telephony and Voice over IP (VoIP) running on top of both wireless and cellular mobile networks. In future Internet-based telecommunication, it is essential to have continuity of VoIP calls seamlessly of the underlying network support and connectivity. One can also analyze the end-to-end operation and performance of multimedia streaming applications (e.g., TV broadcast over hand-held devices) on the integrated wired and wireless networks, and characterize performance-critical factors.

On a more abstract level, virtualizing predictable network services to higher layers, such as applications footsteps, will result in effective use of these services. In this way, applications developers need not account for the heterogeneity of lower layers. This work will involve a cross-layer design that expands across link, network, and transport layers.

On a different dimension, it would be interesting to study "user-centric" approaches for building services that optimize network performance according to user profiles and different utilities. The goal of this research will be to enable the design and construction of smart network services that adapt to the underlying environment and type of usage.

Finally, it would be interesting to investigate the integration of the QoS Gateway design into home networking frameworks within the DSL forum or CableNetwork recommendations. Specifically, one can build a realization of the design onto network processors such as the Intel IXP [66], which can be embedded in various wireless and home routers and provides a ready-to-work QoS functionality to be used in today's home networking products.

We expect the model-based control approach to play an effective role in the design of future networks and provide means for building more self-controlled networks that require minimal management overhead.

APPENDICES

APPENDIX A

TCP Model

The TCP model developed in [104] is described as the following:

$$r_t = T(p, M, RTT, T_o, W_{max}) =$$

$$\begin{cases} M\frac{\frac{1-p}{p}+W(p)+\frac{Q(p,W(p))}{1-p}}{RTT(\frac{b}{2}W(p)+1)+\frac{Q(p,W(p))F(p)T_o}{1-p}} & \text{if } W(p) < W_{max} \\ M\frac{\frac{1-p}{p}+W_{max}+\frac{Q(p,W_{max})}{1-p}}{RTT(\frac{b}{8}W_{max}+\frac{1-p}{pW_{max}}+2)+\frac{Q(p,W_{max})F(p)T_o}{1-p}} & \text{otherwise} \end{cases} \tag{A.1}$$

where

$$W(p) = \frac{2+b}{3b} + \sqrt{\frac{8(1-p)}{3bp} + (\frac{2+b}{3b})^2}$$

$$Q(p,w) = min\left(1, \frac{(1-(1-p)^3)(1+(1-p)^3(1-(1-p)^{(w-3)}))}{1-(1-p)^w}\right)$$

$$F(p) = 1 + p + 2p^2 + 4p^3 + 8p^4 + 16p^5 + 32p^6$$

where

r_t: is the sending rate of the TCP flow in bits/sec.

M: is the average packet size in bits.

p: is the loss probability (i.e., probability of loss).

RTT: is the average round-trip time.

T_o: is the typical value of the retransmit timeout (typically $5RTT$).

W_{max}: is the maximum receiver window size enforced by the receiver in packets.

b: is the average number of packets acknowledged by an ACK, (usually 2).

APPENDIX B

Analysis of Variance (ANOVA) and Polynomial Regression

B.1 ANOVA

A brief description of the steps done in Analysis of Variance (ANOVA) [70] are as follows:

B.1.1 Models

The general model for k-factor full factorial design contains $2^k - 1$ effects. This includes k main effects, $\binom{k}{2}$ two-factor interactions, $\binom{k}{3}$ three-factor interactions, and so on.

For any three factors (i.e., $k = 3$) denoted as A, B, and C with levels a, b, and c, and with r repetitions of each experiment, the response variable y can be written as a linear combination of the main effects and their interactions:

$$y_{ijkl} = \mu + \alpha_i + \beta_j + \varepsilon_k + \gamma_{ABij} + \gamma_{ACik} + \gamma_{BCjk} + \gamma_{ABCijk} + e_{ijkl}$$
$$i = 1, \ldots, a; \qquad j = 1, \ldots, b; \qquad k = 1, \ldots, c; \qquad l = 1, \ldots, r \tag{B.1}$$

where

y_{ijkl} = response in the l-th repetition of experiment with factors A, B, and C at levels i, j, and k, respectively.

μ = mean response = $\bar{y}_{....}$

α_i = effect of factor A at level i = $\bar{y}_{i...} - \mu$

$\bar{y}_{i...}$ = average response at the i-th level of A over all levels of other factors and repetitions.

β_j = effect of factor B at level $j = \bar{y}_{.j..} - \mu$

γ_{ABij} = effect of the interaction between A and B at levels i and $j = \bar{y}_{ij..} - \alpha_i - \beta_j - \mu$

γ_{ABCijk} = effect of the interaction between A, B, and C at levels i, j, and k

$= \bar{y}_{ijk.} - \gamma_{ABij} - \gamma_{BCjk} - \gamma_{ACik} - \alpha_i - \beta_j - \varepsilon_k - \mu$

e_{ijkl} = error in the l-th repetition at levels i, j, and k, and so on.

The effects are computed so that their sum is zero:

$$\sum_{i=1}^{a} \alpha_i = \sum_{j=1}^{b} \beta_j = \sum_{i=1}^{a} \gamma_{ABij} = \sum_{j=1}^{b} \gamma_{ABij} = \cdots = \sum_{i=1}^{a} \gamma_{ABCijk} = \cdots = 0$$

The errors in each experiment add to zero too:

$$\sum_{l=1}^{r} e_{ijkl} = 0 \qquad \forall i, j, k$$

B.1.2 Allocation of Variation

One of the important steps in the statistical analysis conducted is to calculate the percentage of variation in the output response due to each factor and its levels as well as the interactions between them and the errors in the experiment runs. This step is called the *Allocation of Variation*.

Squaring both sides of the model in Eq. B.1, and adding across all values of responses (cross-product terms cancel out) we get:

$$\sum_{ijkl} y_{ijkl}^2 = abcr\mu^2 + bcr\sum_i \alpha_i^2 + acr\sum_j \beta_j^2 + abr\sum_k \varepsilon_k^2 + cr\sum_{ij} \gamma_{ij}^2 + br\sum_{ik} \gamma_{ik}^2 + ar\sum_{jk} \gamma_{jk}^2 + r\sum_{ijk} \gamma_{ijk}^2 + \sum_{ijkl} e_{ijkl}^2$$

This equation can be written as:

$$SSY = SS0 + SSA + SSB + SSC + SSAB + SSAC + SSBC + SSABC + SSE \qquad \text{(B.2)}$$

The total variation of y, denoted as the sum of square total or SST, is then:

$$SST = \sum_{ijkl} (y_{ijkl} - \mu)^2 = SSY - SS0.$$

The error in the k-th repetition is $e_{ijkl} = y_{ijkl} - \bar{y}_{ijk}$, and the sum of squared errors (SSE) is equal to:

$$SSE = \sum_{ijk} e_{ijkl}^2 = SST - SSA - SSB - SSC - SSAB - SSAC - SSBC - SSABC.$$

Once we got all the sum of squares, then the percentages of variation can be calculated as $100 \times (\frac{SSA}{SST})$ for the effect of factor A, $100 \times (\frac{SSAB}{SST})$ for the interaction between A and B, and so on, and finally $100 \times (\frac{SSE}{SST})$ for errors. From these percentages of variations, we can identify the most important factors. The factors with small or negligible contributions to the total variation of the output can be removed from the model.

B.1.3 Analysis of Variance

To statistically test the significance of a factor, interaction of factors, or errors, the sum of squares are divided on their corresponding degrees of freedom (DF)[1] to get the mean squares. For the previous example of three factors, we have the following degrees of freedom for Eq. B.2:

$$abcr = 1 + (a-1) + (b-1) + (c-1) + (a-1)(b-1) + (a-1)(c-1) + (b-1)(c-1) + (a-1)(b-1)(c-1) + abc(r-1)$$

For example, the mean square of factor A (MSA) equals to $\frac{SSA}{(a-1)}$ and the mean square error (MSE) equals to $\frac{SSE}{abc(r-1)}$.

To compare the significance of each effect with the experimental error, the ratios of mean squares are used. This is called the "F-test". For example the ratio $\frac{MSA}{MSE}$ corresponds to the effect of factor A. These ratios have F distribution [70] calculated at their corresponding degrees of freedom. If the computed F value is greater than the 95% quantile, $F_{[0.95;\alpha,\beta]}$, value from the tables of the F-variates, then the effect is significant and vice versa. Here α and β are the degrees of freedom of for SSA, and SSE, respectively. This usually leads to the same conclusion if we compare the percentage of variations of the factors with those of errors. Table B.1 summarizes the process.

[1]The degree of freedom is the number of independent values required to compute the corresponding sum of square.

Effect	SS	%age	DF	MS	F
y	$SSY = \sum y_{ijk}^2$		$abcr$		
$\bar{y}_{....}$	$SS0 = abcr\mu^2$		1		
$y - \bar{y}_{....}$	$SST = SSY - SS0$	100	$abcr - 1$		
A	$SSA = br\sum \alpha_j^2$	$100(\frac{SSA}{SST})$	$a-1$	$MSA = \frac{SSA}{a-1}$	$\frac{MSA}{MSE}$
. .					
AB	$SSAB = cr\sum \gamma_{ij}^2$	$100(\frac{SSAB}{SST})$	$(a-1)(b-1)$	$MSAB = \frac{SSAB}{(a-1)(b-1)}$	$\frac{MSAB}{MSE}$
. .					
ABC	$SSABC = r\sum \gamma_{ijk}^2$	$100(\frac{SSABC}{SST})$	$(a-1)(b-1)$ $(c-1)$	$MSABC = \frac{SSABC}{(a-1)(b-1)(c-1)}$	$\frac{MSABC}{MSE}$
e	$SSE = SST$ $-SSA - SSB - SSC$ $-SSAB - SSAC - SSBC$ $-SSABC$	$100(\frac{SSE}{SST})$	$abc(r-1)$	$MSE = \frac{SSE}{abc(r-1)}$	

Table B.1: ANOVA

B.2 Polynomial Regression

Polynomial regression is based on the *Multiple Linear Regression* model which finds a linear relation between one dependent and k independent variables. The linear model is given by [70]:

$$y_i = b_0 + b_1 x_{1i} + b_2 x_{2i} + \cdots + b_k x_{ki} + e_i \tag{B.3}$$

where $i = 1, \cdots, n$, and n is the sample of size.

An example of a polynomial regression model with two independent factors is as the following [93]:

$$y_i = b_0 + b_1 x_{1i} + b_2 x_{1i}^2 + b_3 x_{2i} + b_4 x_{2i}^2 + b_5 x_{1i} x_{2i} + e_i \tag{B.4}$$

The usual way of solving this polynomial model is to transform the right hand-side to a linear relation by the following substitution:

$$z_{1i} = x_{1i} \quad z_{2i} = x_{1i}^2 \quad z_{3i} = x_{2i} \quad z_{4i} = x_{2i}^2 \quad z_{5i} = x_{1i} x_{2i}$$

Non-linear relation	Transformed variable	Linear relation
$Y = X_1 X_2$	$Y' = logY$	$Y' = logX_1 + logX_2$
$Y = \frac{1}{X}$	$Y' = 1/Y$	$Y' = X$
$\sqrt{Y} = X$	$Y' = \sqrt{Y}$	$Y' = X$

Table B.2: Basic transformations

Once this transformation is done to the input factors, it can be solved using the same methods used for linear regression [70, 93]. One important measure for the goodness of the regression is called the coefficient of determination, R^2, which is the ratio between the variation explained by the regression (SSR) to the total variation (SST) caused by the regression and the error (SSE). The higher the value of R^2 the better the regression is.

APPENDIX C

QoS Gateway Prototype Implementation

This appendix presents details about important parts of the implementation. It include actual codes, functions, data structures, and messages formats that represent major functionality. Our prototype implementation is based on Linux 2.4.

C.1 Host-based Agent

The modules implemented in the Agent are the socket-capturing kernel module, application manager, QoS database support, part of QoS mapping, and the communication with the *QoS Manager*.

C.1.1 Socket Capturing Module

The module, when loaded, replaces the default kernel's socket system call handler `sys_socketcall` with our own modified system call handler that is called `diff_sys_socketcall`. This is done by accessing the kernel system calls table directly.

```
module_main.c:
    original_socketcall = sys_call_table[__NR_socketcall];
    sys_call_table[__NR_socketcall] = diff_sys_socketcall;
```

The new socket system call, when invoked, compiles a message to be sent to the daemon that contains information about the current application process.

```
socketcall.c:
    /* get task info here and send it to the daemon */
    sock_msg.call = <system call type>;
    sock_msg.current_pid = current->pid;
    sock_msg.current_gid = current->gid;
    sock_msg.current_tgid = current->tgid;
    sock_msg.current_uid = current->uid;
    sock_msg.current_euid = current->euid;
    sock_msg.current_gid = current->gid;
    sock_msg.current_egid = current->egid;
```

Then, the module blocks the application process waiting for a reply from the daemon.

```
socketcall.c:
    /* send msg to the user-space daemon */
    send_msg(SOCK_MSG, sock_Message_Ptr, SOCK_MSG_SIZE);
    /* don't move and wait for a reply from the user-space daemon */
    wait_for_reply();
```

The function `wait_for_reply()` is responsible for blocking the application process by putting it in the device wait queue as if it is waiting for a device read.

```
dev_file.c:
    current->state = TASK_INTERRUPTIBLE;
    add_wait_queue(&wait_for_reply_q_h, &wait_for_reply_queue);
    /* put another process on execution until I get a msg from the dae-
mon */
    schedule();
```

When a reply is received from the user-level daemon, the module wakes up the blocked process and continues work.

```
dev_file.c:
    /* wakeup that process who waited for reply only */
    wake_up_interruptible(&wait_for_reply_q_h);
```

The module communicates with the user-space daemon through a special character device file `/dev/diff0`. Messages sent throughout this communication link are exchanged through functions like `dev_read()` and `dev_write()` like a normal file. Data are moved from kernel space to user space and vice versa using the two functions `copy_to_user()` and `copy_from_user()`.

C.1.2 Application Management

The daemon (*QoSd*) uses the `/proc` filesystem to locate the application name using the process pid (`sock_msg_body->current_pid`). Then, the daemon searches in the applications QoS database to locate the current application.

```
qos_mapping.c:
    QoSDBresult = search_QoSDB_file(sock_msg_body, &app);
```

where `app` is the application structure we use throughout the application management.

```
daemon.h:
    typedef struct application {
       char name[CMDLINE_SIZE];
       uid_t uid;
       gid_t gid;
       int family;
       int type;
       int protocol;
       flowid_t fid;      /* 5-tuple IP header (core.h) */
       profile_t profile;
       QoSrequest_t qos;  /* QoS parameters needed */
    }application_t;
```

The Application Manager keeps a list of all active and registered QoS applications. Each entry is of type `application_t`. All application management functions are found in `app.c` where implementation of the list and its handlers are placed.

The daemon logs all messages and activities in a syslog file called `qosd.log` held in `/var/log/` directory. The daemon also uses a configuration file called `qosd.conf` to load the necessary configuration parameters such as the Gateway IP address and port number.

C.1.3 QoS Mapping and QoS Database

In the prototype, we use a simple QoS mapping based on a predefined values for application QoS stored in the text-based database called `QoSDB.dat`. The database uses the application name as the primary key, and each entry contains user information (`uid` and `gid`), flow identifier object (`fid`), traffic profile (*Tprof* or `profile`), and the QoS structure (`qos`). The flow identifier structure is given by:

```
gw_common.h:
    /* flowid_t: describes a flow using 5-tuple IP header */
    typedef struct _flowid {
       char src[IPLEN];  /* source address in IPv4      */
       char dst[IPLEN];  /* destination address in IPv4 */
       unsigned short sport;
       unsigned short dport;
       unsigned short metric;
    } flowid_t;
```

The traffic profile is given by:

```
gw_common.h:
    typedef struct t_profile {
       double rate;
       double burst;
       double pktsize;
       double peakrate;
       double maxpktsize;
    } profile_t;
```

The QoS structure is given by:

```
gw_common.h:
    typedef struct _QoSrequest {
     QoSmetric_t CIR;        /* Committed rate (bps) */
     QoSmetric_t PIR;        /* Peak rate (bps)      */
     QoSmetric_t delay;      /* mean delay (msec)    */
     QoSmetric_t maxdelay;   /* maximum delay (msec) */
     QoSmetric_t jitter;     /* jitter (msec)        */
     QoSmetric_t loss;       /* loss (%age)           */
     QoSmetric_t maxloss;    /* maximum loss (%age)   */
     QoSmetric_t trecov;     /* recovery time (sec)  */
    } QoSrequest_t;;
```

The function `qos_mapping()` in `qos_mapping.c` is the one responsible for performing application QoS mapping and contains all the QoS mapping algorithms. In the current prototype, this function only reads entries from the QoS DB indexed by application name.

C.1.4 Communication with QoS Manager

The function that is responsible for this communication is called:

```
qossignal.c:
    int qossignal(char *gwIPaddr, u_short gwport, flowid_t fid,
         profile_t *p, QoSrequest_t *q, __u32 flag, int *class)
```

It constructs a request packet to the *QoS Manager* that contains the flowid, traffic profile, and the QoS structure of the application flow. It calls:

```
qossignal.c:
    int sendqosreq(int sockfd, QoSmesg_t *msg, size_t n)
```

and waits for a reply (acknowledgment) through:

```
qossignal.c:
    int recvserviceack(int sockfd, QoSack_t *ack)
```

The underneath functions are all implemented in `rtrutil.c` which has functions for establishing a connection, waiting for a connection, sending and receiving messages through this connection.

C.2 QoS Manager

In the prototype, we implemented the SLA management, a simple QoS mapping, admission control, traffic classification, and traffic marking.

C.2.1 SLA Management

SLA is a simple manual database called `netSLA.dat` that is read by the gateway at the start of operation. It can also be modified at run-time and synchronized to the running gateway for any updates. The functions in `sla_db.c` are used to read the *netSLAs* from the database one line at a time and they are stored in an active list called `netSLAchain`. Each node in the `netSLAchain` is of type:

`sla.h:`

```
 typedef struct netSLA {
int id;   // primary key
unsigned short DiffServ_class;  // EF, AFx - for mapping and marking
unsigned short IntServ_class;  // GS, CL - for RSVP
unsigned short *dscp;  // carry the associated DSCP(s)
unsigned int dscp_length;  // for more than one DSCP
double total_bw;   // total bw allocated to this class
double available_bw;  // available bw left for this class
unsigned short bw_sharing;  // isolated, borrow, or restricted
struct netSLA *next;  // for linked list of netSLAs
 } netSLA_t;
```

The functions in `app_sla.c` handle the application SLA (*appSLA*) records in the gateway. Each *appSLA* entry is of type:

`sla.h:`

```
 typedef struct appSLA {
int id;  // primary key
flowid_t flowid;  // this appSLA belongs to this flow
profile_t tprof;  // profile or Tspec for this flow (TBF)
QoSrequest_t appQoS;  // application QoS
netSLA_t *net_class;  // handle to the associated netSLA
struct appSLA *next;  // for a linked list of appSLAs
 } appSLA_t;
```

C.2.2 QoS Mapping

The main QoS mapping function is:

```
qos_mapping.c:
    int maprequest(QoSrequest_t req, profile_t prof, flowid_t flow)
```

which is supposed to perform the mapping between application QoS requests and network classes. Currently, this function calls directly the SLA manager to handle the requests.

C.2.3 Admission Control

On of the important functions of the QoS Manager is Admission Control. The typical sequence of processing a QoS request from an end-host is depicted in Figure C.1 in terms of invoked modules inside the *QoS Manager*. The module `cqossig` is responsible for accepting host QoS requests, and it passes the request to the `qos_mapping::install_qos` function after performing validity checks. The `qos_mapping` module installs a set of packet classifiers, packet markers and the necessary traffic conditioning blocks for the admitted request. Choosing a suitable network service class for the request is done inside the `sla_manager` which periodically updates network and application SLA (*netSLA* and *appSLA*, respectively) databases. Before the requested flow(s) can be admitted, the e2e QoS verification is done via `rsvp_snd` – it communicates with a co-located RSVP daemon to start a RSVP signaling session. Once the RSVP session is complete, the original signaling process traces back and replies to host through a call-back function called `back_to_host`

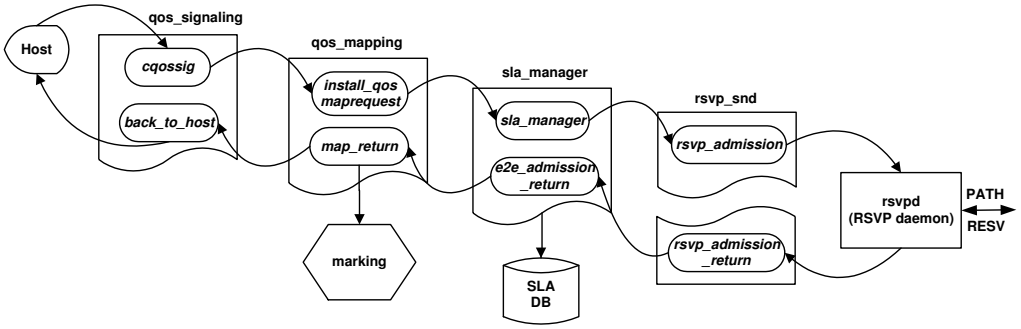


Figure C.1: Sequence of operations inside Gateway

with either an admission or a rejection decision .

C.2.4 Traffic Marking

Packet markers are installed at the end of the traffic control chain. The marker uses the DSCP assigned to the flow from the corresponding *netSLA*. Currently, we implement three marking algorithms for AF: two rate Three-Color Marker (trTCM), Time Sliding Window (TSW), and Equation Based Marking (EBM). For EF traffic, a single rate two color marker is used which is simply using a single token bucket filter to compare input packets against a rate and a burst size. It marks in-profile packets as EF and leaves out-of-profile packets unmarked.

Algorithmic packet markers are implemented as part of the Linux kernel traffic control. New queueing discipline (QDisc) called "dsmarker" implements several marking algorithms for both EF and AF. A specific marker type, and hence a specific algorithm, is chosen at the time of marker instantiation. The file `sch_dsmarker.c` contains the definition and functions of the dsmarker QDisc module. The functions:

```
static int dsmarker_enqueue(struct sk_buff *skb,struct Qdisc *sch)
static struct sk_buff *dsmarker_dequeue(struct Qdisc *sch)
```

add and remove packets from the QDisc. The latter function calls one of the following markers:

```
int TCM_MarkPacket(struct sk_buff *skb, struct Qdisc *sch);
int TSW_MarkPacket(struct sk_buff *skb, struct Qdisc *sch);
```

```
int EBM_MarkPacket(struct sk_buff *skb, struct Qdisc *sch);
int EFM_MarkPacket(struct sk_buff *skb, struct Qdisc *sch);
```

and upon return, it changes the DS-field of the IP packet's headers using the kernel function:

```
ipv4_get_dsfield(skb->nh.iph);
```

C.2.5 Traffic Classifications and Control

The data plane of the *QoS Manager* employs the required traffic policing and conditioning to do traffic control. Token Bucket Filters are used for traffic policing. After admitting a flow, the gateway installs a traffic classifier based on the flow identifier of the flow sent in the *QoS Agent's* request message. The traffic policer and appropriate markers are installed based on the traffic profile (*Tprof*) of the flow sent in the request. Traffic policers and markers are installed on a per-flow basis. This per-flow chain can be installed on the ingress or the egress interface of the *QoS Manager* device, if allowed. An optional functionality of the data plane is to implement PHBs to act as a first router for the application traffic. This is not required unless in a small-scale deployment scenarios where there is no DiffServ-enabled routers in the interconnecting network. PHBs are installed on the egress interface of the manager device.

When the *QoS Manager* starts, it installs the basic tree of traffic control which will hold branches for each new admitted flow. The Gateway creates the root of the tree which has a CBQ packet scheduler.

`core.c:`

```
send_req_qdisc_cbq()
```

Then for each flow, a new branch is created in the tree that defines a CBQ class which carries a u32 classifier, followed by a policer, then followed by a marker.

`qos_mapping.c:`

```
send_req_class_cbq()

send_req_filter_u32_IPflow()

send_req_qdisc_dsmarker()
```

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] The Linux Netfilter Project. http://www.netfilter.org.

[2] Windows 2000 Features in Support of Differentiated Services, July 2000. White paper, http://www.microsoft.com/TechNet/win2000/win2ksrv/diffserv.asp.

[3] T. Abdelzaher, K.G. Shin, and N. Bhatti. Performance Guarantees for Web Server End-Systems: A Control-Theoretical Approach. *IEEE Trans. on Parallel and Distributed Systems*, 13(1):80–96, 2002.

[4] T.F. Abdelzaher. QoS Adaptation in Real-Time Systems. Ph.D. Thesis, The University of Michigan, Ann Arbor, 1999.

[5] W. Almesberger. Linux Network Traffic Control - Implementation Overview. In *Proceedings of the 5th Annual Linux Expo*, May 1999.

[6] W. Almesberger, J. H. Salim, and A. Kuznetsov. Differentiated Services on Linux. Internet draft, work on progress, draft-almesberger-wajhak-diffserv-linux-01.txt, IETF, June 1999.

[7] Kendall E. Atkinson. *An Introduction to Numerical Analysis*. John Wiley and Sons, Inc., New York, 2nd ed., 1989.

[8] D. Bansal, J.Q. Bao, and W.C. Lee. QoS-Enabled Residential Gateway Architecture. *IEEE Communications Magazine*, April 2003.

[9] J. Bennett and Hui Zhang. Hierarchical Packet Fair Queueing Algorithms. In *Proc. of the ACM SIGCOMM'96*, August 1996.

[10] J. C. Bennett, K. Benson, A. Charny, W. F. Courtney, and J.-Y. Le Boudec. Delay jitter bounds and packet scale rate guarantee for expedited forwarding. *IEEE/ACM Transactions on Networking*, 10(4), August 2002.

[11] Y. Bernet et al. A Framework for Integrated Services Operation over DiffServ Networks. RFC 2998, IETF, November 2000.

[12] D. Bertsekas and R. Gallager. *Data Networks*. Prentice-Hall, 1992.

[13] S. Blake, D. Black, M. Carlson, E. Davis, Z. Wang, and W. Weiss. An Architecture for Differentiated Services. RFC 2475, IETF, December 1998.

[14] T. Bonald, M. May, and J. Bolot. Analytic Evaluation of RED Performance. In *Proceedings of IEEE INFOCOM'00, Tel-Aviv, Israel*, March 2000.

[15] D.P. Bovet and M. Cesati. *Understanding the Linux Kernel, 2nd ed.* O'Reilly, 2002.

[16] R. Braden, D. Clark, and S. Shenker. Integrated Services in the Internet Architecture: an Overview. RFC 1633, IETF, June 1994.

[17] R. Braden et al. Resource ReSerVation Protocol(RSVP) – Version 1 Functional Specification. RFC 2205, IETF, September 1997.

[18] R. Braden and D. Hoffman. RAPI – An RSVP Application Programming Interface (Version 5). Work in progress, internet-draft, draft-ietf-rsvp-rapi-01.ps, IETF, February 1999.

[19] B. Carpenter and K. Nichols. A Bulk Handling Per-Domain Behavior for Differentiated Services. Work in progress, internet-draft, draft-ietf-diffserv-pdb-bh-02.txt, IETF, January 2001.

[20] B.E. Carpenter and K. Nichols. Differentiated Services in the Internet. *Proceedings of the IEEE*, 90(9):1479–1494, September 2002.

[21] C Cetinkaya and E.W. Knightly. Egress Admission Control. In *Proceedings of IEEE INFOCOM'00, Tel-Aviv, Israel*, March 2000.

[22] A. Charny and J.-Y. Le Boudec. Delay bounds in a network with aggregate scheduling. In *Proc. 1st Int. Workshop Quality of Future Internet Services (QofIS'2000), Berlin, Germany*, September 2000.

[23] K. Cho. Managing Traffic with ALTQ. In *Proceedings of USENIX'99, Monterey, CA*, June 1999.

[24] D. Clark and W. Fang. Explicit Allocation of Best-Effort Packet Delivery Service. *IEEE/ACM Transactions on Networking*, 6(4):362–373, August 1998.

[25] D. Clark, S. Shenker, and L. Zhang. Supporting Real-Time Applications in an Integrated Services Packet Networks: Architecture and Mechanism. In *Proceedings of ACM SIGCOMM'92, Baltimore, MD*, August 1992.

[26] D. D. Clark. The Design Philosophy of the DARPA Internet Protocols. In *Proceedings of SIGCOMM'88, Stanford, CA*, August 1988.

[27] Allot Communications. NetEnforcer. http://www.allot.com.

[28] R.L. Cruz. A Calculus for Network Delay, Part I: Network Elements in Isolation. *IEEE Transactions on Information Theory*, 37(1):114–131, January 1991.

[29] B. Davie, A. Charny, J. Bennet, K. Benson, J. Bouded, W. Courtney, S. Davari, V. Firoiu, and D. Stiliadis. An Expedited Forwarding PHB (per-hop behavior). RFC 3246, IETF, March 2002.

[30] A. Demers, S. Keshav, and S. Shenker. Analysis and Simulation of a Fair Queueing Algorithm. In *Proceedings of the ACM SIGCOMM'89, Austin, Texas*, September 1989.

[31] M. El-Gendy and K.G. Shin. DQM: Delay-Controlled Active Queue Management. In *submitted to IEEE Infocom'06, Barcelona, Spain*, 2005.

[32] M. El-Gendy, K.G. Shin, and H. Fathy. CONNET: Self-Controlled Access Links for Delay and Jitter Requirements. In *Proceedings of International Conference on Network Protocols (ICNP'05), Boston, MA, to appear*, Nov. 2005.

[33] M.A. El-Gendy, A. Bose, S. Park, and K.G. Shin. Paving hte First Mile for QoS-dependent Applications and Appliances. In *Proc. of the 12th International Workshop on Quality of Service (IWQoS'04), Montreal, Canada*, June 2004.

[34] M.A. El-Gendy, A. Bose, and K.G. Shin. Evolution of the Internet QoS and Support of Soft Real-Time Applications. *Invited paper, Proceedings of the IEEE*, 91(7):1086–1104, July 2003.

[35] M.A. El-Gendy, A. Bose, H. Wang, and K.G. Shin. Statistical Characterization for Per-Hop QoS. In *Proc. of the 11th International Workshop on Quality of Service (IWQoS'03), Monterey, CA*, June 2003.

[36] M.A. El-Gendy, A. Bose, H. Wang, and K.G. Shin. Statistical Modeling and Control of Per-Hop QoS. *submitted to IEEE/ACM Transactions on Networking*, Dec 2004.

[37] M.A. El-Gendy and K.G. Shin. Equation-Based Packet Marking for Assured Forwarding Services. In *Proceedings of IEEE INFOCOM'02, New York, NY*, June 2002.

[38] M.A. El-Gendy and K.G. Shin. Assured Forwarding Fairness Using Equation-Based Packet Marking and Packet Separation. *Computer Networks, Elsevier Science*, 41(4), March 2003.

[39] W. Fang, N. Seddigh, and B. Nandy. A Time Sliding Window Three Colour Marker (TSWTCM). Internet-draft, work in progress, draft-fang-diffserv-tc-tswtcm-01.txt, IETF, March 2000.

[40] F. Le Faucheur et al. Multi-Protocol Label Switching (MPLS) Support of Differentiated Services. RFC 3270, IETF, May 2002.

[41] W. Feng, D. Kandlur, D. Saha, and K. Shin. Adaptive Packet Marking for Maintaining End-to-End Throughput in a Differentiated-Services Internet. *IEEE/ACM Transactions on Networking*, 7(5):685–697, October 1999.

[42] W.-C. Feng, K.G. Shin, D.D. Kandlur, and D. Saha. The BLUE Active Queue Management Algorithms. *IEEE/ACM Transactions on Networking*, 10(4), August 2002.

[43] A. Feroz, A. Rao, and S. Kalyanaraman. A TCP-Friendly Traffic Marker for IP Differentiated Services. In *Proceedings of the IWQoS'00, Pittsburgh*, 2000.

[44] G. Ferrari. Applying Feedback Control to QoS Management. In *Proceedings of the 7th CaberNet Radicals Workshop, Bertinoro, Italy*, October 2002.

[45] T. Ferrari. End-to-End Performance Analysis with Traffic Aggregation. *Computer Network Journal, Elsevier*, 34(6):905–914, December 2000.

[46] T. Ferrari and P. Chimento. A Measurement-based Analysis of Expedited Forwarding PHB Mechanisms. In *Proceedings of IWQoS'00, Pittsburgh, IEEE 00EXL00*, pages 127–137, June 2000.

[47] V. Firoiu and M. Borden. A Study of Active Queue Management for Congestion Control. In *Proceedings of IEEE INFOCOM'00, Tel-Aviv, Israel*, March 2000.

[48] V. Firoiu, J.-Y. Le Boudec, D. Towsley, and Z.L. Zhang. Theories and Models for Internet Quality of Service. *Proceedings of the IEEE*, 90(9):1565–1591, September 2002.

[49] S. Floyd and V. Jacobson. Random Early Detection Gateways for Congestion Avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, August 1993.

[50] S. Floyd and V. Jacobson. Link-Sharing and Resource Models for Packet Networks. *IEEE/ACM Transactions on Networking*, 3(4):365–386, August 1995.

[51] S. Floyd and E. Kohler. Internet Research Needs Better Models. In *The First Workshop on Hot Topics in Networks, Princeton, New Jersey*, October 2002.

[52] S. Floyd, J. Padhye, and J. Widmer. Equation-Based Congestion Control for Unicast Applications. In *Proceedings of the ACM SIGCOMM'00*, 2000.

[53] G.F. Franklin, J.D. Powell, and A. Emami-Naeini. *Feedback Control of Dynamic Systems*. Addison-Wesley, 1995.

[54] G.F. Franklin, J.D. Powell, and M.L. Workman. *Digital Control of Dynamic Systems*. Addison-Wesley, 1998.

[55] N. Gandhi, J.L. Hellerstein, S.S. Parekh, D. Tilbury, and Y. Diao. MIMO Control of an Apache Web Server: Modeling and Controller Design. In *Proceedings of American Control Conference*, May 2002.

[56] N. Giroux and S. Ganti. *Quality of Service in ATM networks: state-of-the-art traffic management*. Prentice Hall, 1999.

[57] M. Goyal, A. Durresi, R. Jain, and C. Liu. Performance Analysis of Assured Forwarding. Internet-draft, work in progress, draft-goyal-diffserv-afstdy-00.txt, IETF, February 2000.

[58] R. Guerin. Quality of Service in IP Networks, Tutorial. In *Proc. of the 6th IEEE Real-Time Technology and Applications Symposium, Washington, D.C.*, May 2000.

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

[60] J. Heinanen and R. Guerin. A Two Rate Three Color Marker. RFC 2698, IETF, September 1999.

[61] U. Hengartner, J. Bolliger, and T. Gross. TCP Vegas revisited. In *Proceedings of IEEE INFOCOM'00, Tel-Aviv, Israel*, March 2000.

[62] Z. Heying, L. Baohong, and D. Wenhua. Design of a Robust Active Queue Management Algorithm Based on Feedback Compensation. In *Proceedings of ACM SIGCOMM'03, Karlsruhe, Germany*, Aug. 2003.

[63] C.V. Hollot, V. Misra, D. Towsley, and W.B. Gong. On Designing Improved Controllers for AQM Routers Supporting TCP Flows. In *Proceedings of IEEE INFOCOM'01, Anchorage, Alaska*, April 2001.

[64] C.V. Hollot, V. Misra, D. Towsley, and W.B. Gong. A Control Theoretic Analysis of RED. In *Proceedings of IEEE INFOCOM'02, New York, NY*, June 2002.

[65] J. Ibanez and K. Nichols. Preliminary Simulation Evaluation of an Assured Service. Internet-draft, work in progress, draft-ibanez-diffserv-assured-eval-00.txt, IETF, August 1998.

[66] Intel. Network Processors, IXP architecture. http://www.intel.com/design/network/products/npfamily/index.htm.

[67] V. Jacobson. Congestion Avoidance and Control. In *Proceedings of ACM SIGCOMM'88, Stanford, CA*, August 1988.

[68] V. Jacobson. Congestion Avoidance and Control. In *Proceedings of the ACM SIGCOMM'88*, pages 314–332, August 1988.

[69] V. Jacobson, K. Nichols, and K. Poduri. The "Virtual Wire" Per-Domain Behavior. Work in progress, internet-draft, draft-ietf-diffserv-pdb-vw-00.txt, IETF, July 2000.

[70] R. Jain. *The Art of Computer Systems Performance Analysis*. John Wiley and Sons, Inc., 1991.

[71] S. Keshav. A Control-Theoretic Approach to Flow Control. In *Proceedings of ACM SIGCOMM'91, Zurich, Switzerland*, September 1991.

[72] S. Keshav. *An Engineering Approach to Computer Networking*. Addison Wesley Professional Computing Series, 1997.

[73] H. Kim. A Fair Marker. Internet-draft, work in progress, draft-kim-fairmarker-diffserv-00.txt, IETF, April 1999.

[74] A. Kolarov and G. Ramamurthy. A control-theoretic approach to the design of an explicit rate controller for ABR service. *IEEE/ACM Transactions on Networking*, 7(5), October 1999.

[75] C.M. Krishna and K.G. Shin. *Real-Time Systems*. McGraw-Hill, 1997.

[76] A. Kumar. Traffic Sensitive Active Queue Management, Dec. 2003. Masters thesis, Worcester Polytechnic Institute.

[77] E. Kusmierek and R. Kooldi. Random Packet Marking for Differentiated Services. UMN Technical Report TR-00-020, Dept. of Comp. Science & Eng., University of Minnesota, 2000.

[78] M. Kuznetsov et al. A Next-Generation Optical Regional Access Networks. *IEEE Communications Magazine*, 38:66–72, January 2000.

[79] LBL. Packet filter library (Libpcap). ftp://ftp.ee.lbl.gov/.

[80] D. Lee. *Enhanced IP Services for Cisco Networks*. Cisco Press, 1999.

[81] B. Lei and T.S. Teck A.L. Ananda. QoS-aware Residential Gateway. In *Proceedings of the 27th Annual IEEE Conference on Local Computer Networks (LCN'02)*, 2002.

[82] B. Li and K. Nahrstedt. A Control-Based Middleware Framework for Quality-of-Service Adaptations. *IEEE Journal on Selected Areas in Communications*, 17(9), September 1999.

[83] W. Lin, R. Zheng, and J. Hou. How to Make Assured Services More Assured. In *Proceedings of the ICNP'99*, 1999.

[84] L. Ljung. *System Identification: theory for the user*. Prentice Hall, 2nd ed., 1999.

[85] R. Makkar et al. Empirical Study of Buffer Management Schemes for Diffserv Assured Forwarding PHB. Technical report, Nortel Networks, May 2000.

[86] Y. Mansour and B. Patt-Shamir. Jitter Control in QoS Networks. *IEEE/ACM Transactions on Networking*, 9(4):492–502, August 2001.

[87] M. Mathis, J. Semke, J. Mahdavi, and T. Ott. The Macroscopic Behavior of the TCP Congestion Avoidance Algorithm. *ACM Computer Communication Review*, 27(3), July 1997.

[88] A. Mehra, D. Verma, and R. Tewari. Policy-Based DiffServ on Internet Servers: The AIX Approach. *IEEE Internet Computing*, September-October 2000.

[89] V. Misra, W.B. Gong, and D. Towsley. Fluid-Based Abalysis of a Network of AQM Routers Supporting TCP Flows with an Applications to RED. In *Proceedings of ACM SIGCOMM'00*, 2000.

[90] A. Mohammed et al. DiffServ Experiments: Analysis of the Premium Service Over the Alcatel-NCSU Internet2 Testbed. In *Proceedings of the 2nd European Conference on Universal Multiservice Networks ECUMN'2002, CREF, Colmar, France*, pages 124–130, April 2002.

[91] K. Nahrstedt and J.M. Smith. The QoS Broker. *IEEE Multimedia Magazine*, 2(1), Spring 1996.

[92] B. Nandy, N. Seddigh, P. Pieda, and J. Ethridge. Intelligent Traffic Conditioners for Assured Forwarding Based Differentiated Services Networks. In *Proceedings of High Performance Networking 2000 Conference, Paris, France*, May 2000.

[93] J. Neter, W. Wasserman, and M. Kutner. *Applied Linear Statistical Models: Regression, Analysis of Variance, and Experimental Designs*. Homewood, R.D. Irwin, Inc., 1985.

[94] Sitara Networks. QoSWorks. http://www.sitaranetworks.com.

[95] H.T. Ngin and C.K. Tham. A Control-Theoretical Approach for Achieving Fair Bandwidth Allocations in Core-Stateless Networks. *Computer Networks, Elsevier Science*, 40(6), December 2002.

[96] 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, December 1998.

[97] K. Nichols and B. Carpenter. Definition of Differentiated Services Per-Domain Behaviors and Rules for their Specifications. RFC 3086, IETF, April 2001.

[98] K. Nichols, V. Jacobson, and L. Zhang. A Two-bit Differentiated Services Architecture for the Internet. RFC 2638, IETF, July 1999.

[99] NLANR. Iperf 1.1.1, February 2000. http://dast.nlanr.net/Projects/Iperf/.

[100] K. Ogata. *Modern Control Engineering*. Prentice-Hall, 3rd edition, 1997.

[101] P.V. Overschee and B.D. Moor. *Subspace Identification For Linear Systems: Theory, Implementation, Applications*. Kluwer Academic Publishers, 1996.

[102] Packeteer. PacketShaper. http://www.packeteer.com.

[103] J. Padhye, V. Firoiu, and D. Towsley. A Stochastic Model of TCP Reno Congestion Avoidance and Control. Tech. Rep. 99-02, Department of Computer Science, University of Massachusetts, Amherst, 1999.

[104] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose. Modeling TCP Throughput: A Simple Model and its Empirical Validation. In *Proceedings of ACM SIGCOMM'98*, October 1998.

[105] J. Padhye, J. Kurose, D. Towsley, and R. Koodli. A Model Based TCP-Friendly Rate Control Protocol. In *Proceedings of the NOSSDAV'99*, June 1999.

[106] K. Papagiannaki et al. Preferential Treatment of Acknowledgment Packets in a Differentiated Services Network. In *Proceedings of the IWQoS'01*, June 2001.

[107] S. Parekh et al. Using Control Theory to Achieve Service Level Objectives in Performance Management. *Journal of Real-Time Systems, special issue on Control-Theoretical Approaches to Real-Time Computing*, 23(1/2), July/September 2002. Also in IFIP/IEEE Int'l Symposium on Integrated Network Management, 2001.

[108] V. Phirke, M. Claypool, and R. Kinicki. RED-Worcester - Traffic Sensitive Active Queue Management. In *Poster at the 10th IEEE International Conference on Network Protocols (ICNP'02), Paris, France*, Nov. 2002.

[109] J. Rezende. Assured Service Evaluation. In *Proceedings of the IEEE Globecom'99, Rio De Janiero, Brazil*, December 1999.

[110] E. Rosen, A. Viswanathan, and R. Callon. Multiprotocol Label Switching Architecture. RFC 3031, IETF, January 2001.

[111] A. Rubini and J. Corbet. *Linux Device Drivers*. O'Reilly, 2001. 2nd ed.

[112] S. Sahu, P. Nain, D. Towsley, C. Diot, and V. Firiou. On Achievable Service Differentiation with token marking for TCP. In *Proceedings of the ACM SIGMETRICS'00, Santa Clara, CA*, June 2000.

[113] R. Schantz et al. An Object-level Gateway Supporting Integrated-Property Quality of Service. In *Proceedings of the 2nd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, May 1999.

[114] N. Seddigh, B. Nandy, and J. Heinanen. An Assured Rate Per-Domain Behavior for Differentiated Services. Work in progress, internet-draft, draft-ietf-diffserv-pdb-ar-01.txt, IETF, July 2001.

[115] N. Seddigh, B. Nandy, and P. Pieda. Bandwidth Assurance Issues for TCP flows in a Differentiated Services Network. *Proceedings of IEEE Globecom'99*, March 1999.

[116] N. Seddigh, B. Nandy, and P. Pieda. Study of TCP and UDP Interaction for the AF PHB. Internet-draft, work in progress, draft-nsbnpp-diffserv-tcpudpaf-01.txt, IETF, August 1999.

[117] M. Shreedhar and G. Varghese. Efficient Fair Queueing using Deficit Round Robin. In *Proc. of the ACM SIGCOMM'95*, 1995.

[118] Network simulator. Ns-2, University of California at Berkeley, CA. Version ns-2.1b6, http://www.isi.edu/nsnam/ns/, Jan. 2000, and Diffserv additions to ns-2 by S. Murphy, available from http://www.teltec.dcu.ie/ murphys/ns-work/diffserv/, May 2000.

[119] P. Siripongwutikorn, S. Banerjee, and D. Tipper. A Survey of Adaptive Bandwidth Control Algorithms. *IEEE Communications Surveys & Tutorials*, 5(1):14–26, Third Quarter 2003.

[120] N. Spring et al. Receiver Based Management of Low Bandwidth Access Links. In *Proceedings of IEEE INFOCOM'00, Tel-Aviv, Israel*, March 2000.

[121] D.C. Steere et al. A Feedback-driven Proportion Allocation for Real-Rate Scheduling. In *Proceedings of Operating Systems Design and Implementation, OSDI'99*, 1999.

[122] D. Stiliadis and A. Varma. Rate-Proportional Servers: A Design Methodology for Fair Queueing Algorithms. *IEEE/ACM Transactions on Networking*, April 1998.

[123] D. Verma. *Supporting Service Level Agreement on IP Networks*. Macmillan Technology Series, 1999.

[124] D.C. Verma, H. Zhang, and D. Ferrari. Delay Jitter Control for Real-Time Communication in a Packet Switching Network. In *Proceedings of IEEE TriCom'91*, 1991.

[125] VoIP Troubleshooter. http://www.voiptroubleshooter.com/problems/access.html.

[126] M. Vojnovic and J.-Y Le Boudec. Stochastic Analysis of Some Expedited Forwarding Networks. In *proceedings of IEEE INFOCOM'02, New York, NY*, June 2002.

[127] H. Wang, A. Bose, M. El-Gendy, and K.G. Shin. IP Easy-pass: Edge Resource Access Control. In *Proceedings of IEEE Infocom'04, Hong Kong*, Mar. 2004.

[128] H. Wang, A. Bose, M. El-Gendy, and K.G. Shin. IP Easy-pass: Edge Resource Access Control. *IEEE/ACM Transactions on Networking, to appear*, Feb. 2006.

[129] Z. Wang. *Internet QoS: Architecture and Mechanisms for Quality of Service*. Morgan Kaufmann publishers, 2001.

[130] H.-Y. Wei and Y.-D. Lin. A Survey and Measurement-Based Comparison of Bandwidth Management Techniques. *IEEE Communications Surveys & Tutorials*, 5(2):10–21, Fourth Quarter 2003.

[131] J. Wroclawski. The Use of RSVP with IETF Integrated Services. RFC 2210, IETF, September 1997.

[132] J. Wroclawski and A. Charny. Integrated Service Mappings for Differentiated Services Networks. Work in progress, internet-draft, draft-ietf-issll-ds-map-01.txt, IETF, August 2001.

[133] I. Yeom and A. Reddy. Modeling TCP Behavior in a Differentiated Services Network. Technical report, Dept. of Electrical Engineering, Texas A & M University (TAMU ECE), May 1999.

[134] H. Zhang and S. Keshav. Comparison of Rate-Based Service Disciplines. In *Proceedings of ACM SIGCOMM'91, Zurich, Switzerland*, September 1991.

[135] L. Zhang et al. RSVP: A New Resource ReSerVation Protocol. *IEEE Networks*, September 1993.