DIFFERENTIATED SERVICES: ARCHITECTURE, MECHANISMS AND AN EVALUATION

WENJIA FANG

A DISSERTATION
PRESENTED TO THE FACULTY
OF PRINCETON UNIVERSITY
IN CANDIDACY FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY

RECOMMENDED FOR ACCEPTANCE
BY THE DEPARTMENT OF
COMPUTER SCIENCE

NOVEMBER, 2000

Abstract

The current Internet assumes the *best-effort* service model. In this model, the network allocates bandwidth among all the instantaneous users as best it can, and attempts to serve all of them without making any explicit commitments as to bandwidth or delay. Routers keep no state about end host connections, and when congestion occurs, all connections are expected to slow down and achieve a collective sending rate equal to the capacity of the congestion point.

As the Internet has transitioned from a research network to a commercial, heterogeneous network, three problems arise. First, an increasing number of real-time applications require some kind of quality of service (QoS) guarantees from the Internet rather than the simple *best-effort* service. Second, a heterogeneous user base has a variety of different requirements from the network and some users are willing to pay to have their requirements satisfied, and the current Internet service model cannot offer a range of flexible services. Third, in a commercial network, Internet Service Providers (ISPs) have to find ways to charge for the service rendered and recuperate the cost of provision the network, and the current Internet is missing mechanisms to account for network usage.

This thesis describes *Differentiated Services*, a scalable architecture that can provide flexible services that address the above three issues. In the *Differentiated Services* architecture (DiffServ), a network classifies packets into different classes, and gives differentiated service to different class of traffic. Network users can choose from the service level best suited for their applications. They subscribe and pay for Service Level Agreements (SLAs) from their ISPs. An SLA specifies the expected service a user will receive during network congestion. If the network is not congested, then the user can send traffic beyond its SLA. The DiffServ architecture augments the current Internet architecture. It consists of mechanisms to be implemented in existing Internet devices—network routers and end hosts—but pushes the complexity of the system towards the edge of the network, which makes it more scalable. A variety of services can be constructed using the simple primitives provided by the DiffServ architecture, therefore, DiffServ offers very flexible services to users with dif-

ferent requirements. Pricing based on SLAs, instead of the actual usage, reflects the nature of the Internet provisioning: mostly the Internet connection is a fixed-cost and the marginal cost of delivery only occurs when there is congestion. Thus, this kind of pricing structure can manage congestion, encourage network growth and recuperate cost without a complex implementation.

This thesis proposes a set of DiffServ mechanisms that offer robust and precise allocation of bandwidth. It proposes *RED with In/Out* (RIO), a differentiated dropping algorithm to be used in interior routers. It proposes *Time Sliding Window* (TSW), a probabilistic tagging algorithm for monitoring and tagging packets in edge routers. Finally, it proposes three modifications to TCP's congestion control algorithm, collectively called TCP-DiffServ mechanisms. The mechanisms include 1) a change of TCP's window increase algorithm; 2) an adjustment to TCP state variable *ssthresh* to reflect the contracted SLAs; 3) a usage of TCP ECN mechanism and DiffServ codepoints to provide accurate feedback of network conditions.

We use elaborate simulation experiments to evaluate the proposed mechanisms. We observe that when using router mechanisms (RIO and TSW) only, a DiffServ domain is able to allocate differentiated bandwidths according to the specified service profiles. However, TCP's window increase algorithm has an intrinsic bias against long-RTT connections, which cannot be overcome by the proposed router mechanisms. We then proceed to apply TCP-DiffServ mechanisms to end hosts in a DiffServ domain. We find that while the enhanced TCP is robust and fair, in times of congestion or in presence of non-responsive connections, TCP connections with service profiles are not protected from those without. Since current Internet allocates its resources using a congestion control loop completed with mechanisms in both routers and end hosts, changing one set of devices without changing the other will not achieve an effective allocation scheme. Finally, we apply both router mechanisms and TCP-DiffServ mechanisms and conclude a DiffServ system could allocate resources in a robust and precise manner when both groups of mechanisms are used.

Acknowledgments

I have been very fortunate to work with my advisors in both Princeton University and M.I.T. Four people are instrumental in making this thesis a reality, Dr. David D. Clark of M.I.T, Professors Larry Peterson, Edward Felten and Randy Wang at Princeton University. I would like to thank Dr. David D. Clark for his wisdom, intuition, patience and guidance and Prof. Larry Peterson for his insights, knowledge, methodical reshaping and refining of my ideas and writing. I have been very fortunate to work with two top researchers in the networks field whose styles and skills are complement of each other. I owe many thanks to Prof. Edward Felten for his discussions and support for completing the thesis, however physically remote it has been. I thank Prof. Randy Wang for his discussions, critiques and advice during the course of thesis writing.

The process of education takes many forms throughout my graduate career. In this sense, I am deeply indebted to many professors at Princeton University, who, by example, showed me what a good researcher and a good teacher can be. I pay my respect and thanks especially to Prof. Kai Li, Prof. Doug Clark, Prof. Ken Steiglitz, and Prof. Andrea LaPaugh. I am also extremely fortunate to have met Prof. Edwin Zschau here in Princeton, who has shown me that a good teacher should be teaching from experiences and not just from knowledge. Thanks also go to Prof. Vincent Poor of the Electrical Engineering Dept. for his enthusiasm about the wireless space, his encouragement and discussions, technologies and beyond.

I am indebted to many researchers in the Internet research area for creating a conducive environment for this work: Prof. Dave Farber of University of Pennsylvania, Prof. Lixia Zhang of U.C.L.A., John Wroclawski and Dr. Karen Sollins of M.I.T., and Hans-Werner Braun of NLANR. I especially thank Hans-werner for his endless supply of Internet traces and helpful discussions. I also like to thank Alan Berenbaum and Dr. Brian Kernighan of Bell Labs for their friendship, mentorship, and guidance throughout my undergraduate and graduate years.

This work would not be possible without Sun Microsystems' generous support for my

two-year tenure in M.I.T.. Thanks go to Jos Marlowe and Dr. Bob Sullivan of Sun Microsystems for creating a Sun Microsystems Graduate Fellowship for me. This work is also supported by DARPA Contract N66001-96-8518.

Thanks also go to Melissa Lawson, Sandy Barbu, Trish Killian, whose able administrative skills make my life easier in the department. Certainly, my two-year absentia to M.I.T. wouldn't be possible without their assistance.

I am grateful to have made my *Arabic Connections* in Cambridge, MA: Dina Katabi, Kamal Khuri-Makdisi, Robert S. Cheng[1], and always present in spirit: Gibran Kahail. Whether digging ancient coins in Palmya, catching kamens in the Amazon, negotiating for hebiskis teas on the streets of Carol, or simply discussing TCP's funny windows in Cambridge, Rob and Dina are incredible companions (MA'A Houbbi). They made my time in M.I.T. and beyond an intellectually enriching experience.

I am fortunate to have met many of my fellow students in Princeton. In particular, I'd like to thank Dirk "Balfi" Balfanz and Lujo Bauer for their friendship. Many things wouldn't have been possible without their artisic, engineering and perfectionist touch. I also like to thank Rudro Samanta, Allison Klein, Stefanos Damianaki, Pritpal Ahuja, Dongming Jiang, Cheng Liao, Lena Petrovic and Yaoyun Shi for their friendship and support throughout the seemingly endless graduate career. I'd like to thank members of the Network Systems Group (NSG) —Scott Karlin, Xiaohu Qie, Limin Wang, Aki Nakao, Tammo Spalink—and the remote but always visible Bjorn Knutsson for their discussions and friendship. I thank Ralf Wittenberg, Tadashi Tokieda, Georg Essl, Matthias Blume, Sanjeev Kumar, Patrick Min for making my Princeton experience a wonderfully international one.

My profuse apologies to those who have suffered from being my officemates at some point. I seem to occupy an increasing amount of space with my photographic materials, a collection of sizable drinking cups, rather incoherent research papers, and a growing stack of broken wooden boards. I will, eventually, retreat.

[1]Rob, with his aptitude for the Arabic language, convinced me that he either was an Arab in his previous life, or will incarnate as one in his next life.

Last but not the least, I’d like to thank my family.

I dedicate this thesis to Haipeng, my “editor friend”, who has been and will always correct my mistakes in writing, Ping Ying and Life. In you, I find the seed of Perfection.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Historical Context

The context of this thesis is set in the early to mid-1990s, when the Internet emerged from an obscure research network connecting mostly educational and research institutes into the public light. The Internet can trace its roots to the ARPANET, a collaborative research network funded by the Advanced Research Projects Agency (ARPA) aimed to connect "existing interconnected networks"[6]. The TCP/IP protocols — the Transmission Control Protocol[52] and the Internet Protocol[51] — were developed for the ARPANET. In the mid-1980s, the National Science Foundation (NSF) created the NSFNET in order to provide connectivity to its supercomputer centers and other general services. The NSFNET adopted the TCP/IP protocols and provided a high-speed backbone for the developing Internet. By the early 1990s, primary users of the Internet were researchers, professors, and students in educational and research institutes. The primary applications on the Internet were text-based interactive (*telnet*, *gopher*) or bulk-data transfer (*ftp*) applications.

The power of the Internet was truly unleashed when the World Wide Web (WWW), or the Web, was developed in the early 1990s. WWW and its hypertext mark-up language HTML provide a very intuitive and convenient way to distribute and access information from anywhere on the Internet. WWW is the killer application for the Internet. It didn't

take long before many web sites were sprouting all over the Internet, putting a vast amount of content on-line for Internet users to browse and exchange. Commercial users as well as typical households began to connect to the Internet. The more users are connected to the Internet, the more positive network externality Internet can provide to its users. In other words, the benefits of having an additional user connected to a network is proportional to the number of users that have already connected to the Internet. By 1994, the Internet was connecting over 45,000 networks and more than four million hosts [42].

The success of the Internet turns out to be a mixed blessing. On one hand, the Internet architecture based on TCP/IP has been very robust in its expansion to incorporate new networks and hosts. On the other hand, some of the architectural components of the current Internet are no longer adequate to keep up with the growth of the Internet. In particular, there are three problems.

First, new applications have been developed to run over the Internet, many of which are real-time video and audio applications that require some kind of quality of service (QoS) guarantees from the network. The current Internet assumes the *best-effort* service model. In this model, the network allocates bandwidth among all the instantaneous users as best it can, and does not make any explicit commitment as to bandwidth or delay. Routers keep no state about end host connections, and when congestion occurs, routers drop packets. All sending connections are expected to slow down and achieve a collective sending rate equal to the capacity of the congestion point. Therefore, from the end host perspective, the service from the network is not always predictable, as it depends on how many other connections are simultaneously sending.

Second, the Internet has transitioned from a government-funded project in a closed, cooperative environment to a commercial network with a heterogeneous user base. That transition has happened gradually and was completed when NSF shut down its NSFNET backbone on April 30th, 1995 and ended its funding [42]. Though the NSF is continuing to fund some regional nets, its role in the Internet has been greatly reduced, and the Internet is therefore a commercial network. In a commercial and heterogeneous network environ-

ment, individual users have different requirements for network resources they would like to receive, and some are willing to pay to to receive those services. However, the current Internet architecture does not offer flexible services to meet different user requirements.

Third, since the Internet is a commercial network, the Internet Service Providers (ISPs) would like to recuperate their costs in some fashion and use the proceeds for further network engineering and expansion. In its design, the current Internet architecture does not have any kind of mechanisms to account for network usage. This is simply the legacy of a government-funded research network [6]. Though accounting was considered as an important feature in the original ARPANET design, it did not receive a high priority in a research network and was missing from the design[1]. However, in a commercial network, if there are no accounting mechanisms to serve as the basis for pricing and billing, the network is likely to be heavily used and congested.

The above three problems set the context for two bodies of research work that eventually led to Differentiated Services, a scalable, technical solution to the above three problems. The two bodies of research work are Integrated Services and Internet Pricing, coming from Internet community and the economics community, respectively. The Integrated Services research proposes solutions to resolve the first two problems and the Internet Pricing research addresses the last problem. We will describe these two bodies of work in Section 1.2.1 and Section 1.2.2 before describing the general idea of Differentiated Services in Section 1.3.

[1]By accounting, we refer to the measurement of traffic profiles of the Internet and the attribution of such profiles to the corresponding users. It is not to be confused with billing, nor pricing. Billing refers to the process of compiling the accounting information and charging the users for such usage, whereas pricing refers to the formation of prices for different types of services, usually through elaborate economic models. See [16] for a more detailed discussion.

1.2 Previous Research

1.2.1 Integrated Services

Integrated Services (IntServ) is an Internet service model that includes both traditional best-effort service and real-time services. IntServ research started in early 1990s, and has generated much interest and discussions by 1994[4, 10, 26, 39, 49, 67, 59]. The problem IntServ research work addresses is how to provide network support for real-time applications as well as for non-real-time applications. The researchers make the observation that many emerging real-time applications — multimedia teleconferencing, remote video, computer-based telephony, remote visualization, etc — will have very different Quality of Service (QOS) requirements than the traditional text-based non-real-time applications like ftp or telnet. A network architecture that can tailor itself to service real-time applications is a major departure from the Internet architecture. One could conceivably build two separate networks, one offering real-time services and the other offering non-real-time services; however, an integrated network offering both sets of services seems more attractive because it would be cheaper to build the network and easier for the application developers.

IntServ researchers realized that they were designing a service model that is based on conjectures about future applications, institutional requirements, and technical feasibility[59]. In [10, 59], Clark, Shenker and Zhang divide applications into those are *elastic*, and those are *real-time*. Elastic applications adjust easily and flexibly to delay in delivery; that is, a packet arriving earlier helps performance and a packet arriving later hurts performance, but there is no set need for a packet to arrive at a certain time. Typical Internet applications are elastic in nature, and the Internet service model has performed well for them. Therefore, a service model consisting of several classes of best-effort services will be sufficient for them. In contrast, real-time applications have more stringent requirements. As observed by Shenker[59], "the performance of elastic applications is more closely related to the average delay of the packets, whereas the performance of real-time applications is more closely related to the maximum delay of the packets". Real-time applications can be further clas-

sified as *tolerant* or *intolerant* applications. Intolerant applications need a service that has a firm worse-case bound on delay; tolerant applications, on the other hand, only need a service that can offer a loose bound on delay.

Services for both tolerant and intolerant real-time applications involve admission control; before commencing transmission, applications must request service from the network. This request consists of a traffic QoS descriptor *flowspec*, in which applications specify their traffic load, and a filter specification (*filter spec*), which describes the subset of the traffic from this application that is to receive the resources. In contrast, there is no admission control for the best-effort service classes (for elastic applications). Thus, for real-time service, the prominent failure mode is that requests can be rejected, and for best-effort service, the failure mode is that best-effort packets can be dropped.

The Integrated Service model is an *extension* to the original Internet best-effort architecture, and it includes two services targeted towards real-time applications [4]: *guaranteed* service and *predicted* service. *Guaranteed* service involves pre-computed worse-case delay bounds and *predicted* service uses the measured performance of the network in computing delay bounds.

Based on the above considerations, the researchers believe that the Integrated Service model would 1) keep additional flow state in routers; 2) require an explicit setup mechanism to install and eliminate flow state in routers. The proposed solution, then, is to have end hosts initiate a quality of service (QoS) request prior to the transmission of traffic to networks. Such request will be carried by a reservation protocol called RSVP. If such request is accepted by networks, then the network will create per-flow state to guarantee such QoS request during the transmission. If the network does not have enough resources, then the QoS request is denied.

Integrated Service effort has done a very good job in 1) analyzing the requirements of real-time applications and 2) arguing for the efficiency of an integrated network offering different kind of services instead of disjoint networks each with a distinct service model. However, the implementation framework IntServ has raised serious concerns. Is it feasible

to implement and maintain per-flow state in routers, especially in those which handle a lot of aggregated traffic? The RSVP protocol is complex, and it may be unrealistic to expect routers to devote much resources to interpreting RSVP requests and handling admission control for established state. Last, there is the partial deployment problem for RSVP itself. If there are routers on a path which do not understand RSVP and can't make reservations, then it would be impossible to make any guarantees on end-to-end delay bounds.

1.2.2 Pricing for the Internet

Network researchers looked into accounting mechanisms and pricing schemes as early as in the beginning of the 1990s. In [11, 12], Cocchi et al. studied the intertwining of pricing policies and multiclass service disciplines. Pricing policies provide monetary incentives, whereas multiclass service disciplines produce performance incentives. In [11], Cocchi, et al. developed a multiclass network discipline which uses a FIFO queue and is able to drop packets depending on whether the packets have priority flags set. The authors used a pricing model that is a graduated set of prices with the lower quality of service class being cheaper. In order to evaluate the different pricing and performance schemes, they used a simple user utility function. Using simulations, they demonstrated that it is possible to set the prices so that every user is more satisfied with the combined cost and performance of a network with graduated prices and a multiclass service discipline than that without a pricing model and a multiclass service discipline.

In [12], Cocchi et al. presented a more elaborate formulation of service disciplines and pricing policies, and argued that for *any* multiclass service discipline to have the desired effect of maximizing network performance, some form of service-class sensitive pricing is required. The authors concluded that effective multiclass service disciplines allow networks to focus resources on the performance sensitive applications, while effective pricing policies allow service providers to spread the benefits of multiple service classes around to all users, rather than just having these benefits remain exclusively with the users of applications that are performance sensitive. Using simulations, they find that it is possible to set

the prices so that users of every application type are more satisfied with the combined cost and performance of a network with service-class sensitive prices than without. For some application types, the performance penalty received for requesting a less-than-optimal service class is offset by the reduced price of the service. For the other application types, the monetary penalty incurred by using the more expensive, higher quality service classes is offset by the improved performance they receive.

Though their conclusions are hardly surprising, their work was the first to combine many disparate issues, such as service disciplines, application performance, user behaviors, congestion externalities, and incentives. The limitation of the work is that it uses simple models in each issue in order to make the problem more tractable.

In 1995, two economists, Jeffrey MacKie-Mason and Hal Varian, reflecting on the end of government funding for the Internet, asked the question of how to price a commercial Internet[42]. They observed that if the Internet is to be utilized as a public good and free for all, then it will soon suffer from a well-known economic phenomenon called the "tragedy of the commons". That is, without instituting new mechanisms for charging for the usage of the Internet, the Internet is likely to be over-grazed. Therefore, some kind of pricing structure has to be in place to manage congestion, encourage network growth, and guide resources to their most valuable usage.

As a general rule, users should face prices that reflect the resource costs that they generate so that they can make informed decisions about resource utilization. MacKie-Mason and Varian observe that most of the Internet connection cost is fixed, and the incremental cost of sending extra packets if the network is not congested is zero. However, when the network is congested, the cost of sending a packet is not zero, therefore, the pricing should also be positive. To reflect the congestion cost, they propose the "smart market" mechanism, in which each packet carries a bid in its header to indicate how much its sender is willing to pay to send it. The network admits all packets with bid prices that exceed the current cutoff amount, determined by the marginal congestion cost imposed by the next additional packet. Users do not pay for price they actually bid, but rather, they pay for the

market-clearing price, which is always lower than the bids of all admitted packets.

MacKie-Mason and Varian made an important contribution in analyzing the interesting problem of pricing the Internet, and how to charge for the cost of transmitting a packet when the network is congested. However, their proposed smart-market scheme is not feasible with the mechanisms available in the current Internet.

In [59], Shenker observes that access-based pricing—charging only for the size of the access link—is both technically easy and predictable. However, access-based pricing cannot provide incentives for users to specify the appropriate service class, nor prevent reselling. Therefore, access-based pricing is not economically efficient. He argues for usage-based pricing, which is economically efficient. However, the concern for usage-based pricing is the level of accounting granularity. Depending on the granularity chosen, the cost of usage-based accounting can be prohibitive. Additionally, usage-based accounting implies a major shift in operating systems as well as in networks, which might be not practical.

In [8], Clark analyzes the advantages and disadvantages of flat-rate pricing and usage-based pricing. Flat-rate pricing, he argues, is simple to implement and encourages usage (if the marginal cost is zero, it is not a problem). The disadvantages of flat-rate pricing is that it does not reflect the congestion cost. A usage-based pricing scheme, on the other hand, can drive away big users and lead the providers to increase prices to recuperate costs. The current Internet architecture lacks the necessary mechanisms needed to do accurate usage-based pricing. He proposed a concept called the "expected capacity" to capture the advantages of both schemes. Expected capacity specifies the service users are *expected* to receive from the network when the system is congested. Users are charged only on the expected capacity they have contracted from their ISPs. If the system is not congested, then the users can send beyond their expected capacity and the extra amount of traffic they send is not charged.

Expected capacity has a direct relationship to the facility costs of the provider. The provider must provision enough to carry the expected capacity from all its subscribers, and thus its provisioning costs directly relate to the total of the expected capacity it has sold.

This would help the provider to provision for the fixed cost of network. Expected capacity is not a measure of actual usage; rather it is the expectation that the user has of potential usage. Therefore, it encourages network usage while sets users' expectations. Expected capacity also leads to a very simple implementation because it is not based on accurate accounting of packets transferred, but on an expected usage, which is known before transmission.

1.3 Differentiated Services

Differentiated Services (DiffServ) has benefited from the previous two areas of research work. It shares the same goal as IntServ—to provide QoS to applications—but emphasizes a *scalable* solution. The central idea is simple. DiffServ defines a small set of packet classes, and creates mechanisms in the network to treat different classes of packets differently during congestion. When there is no congestion, all packets are forwarded just the same. Users contract Service Level Agreements (SLAs) from their respective ISPs. Each SLA defines the expected service profiles the user pays for and expects to receive. Such SLAs and policy elements are stored only at the edge of the network. Traffic from users is classified and mapped into one of a few classes of packets at the edge of the network, and is forwarded within the network core receiving consistent treatment based on the class it belongs to. The mechanisms within the core remain simple.

DiffServ benefits from the research work on pricing and accounting and insights from the expected capacity work. It defines service profiles—contracts between a service provider and a customer that describe the service the customer expects to receive from the network—which are essentially the same concept as *expected capacity*. One could imagine an SLA to be either a simple profile or a sophisticated profile. If we apply Cocchi's work, then an SLA can be constructed using a few primitive services associated with each class of packets, and the price of such a sophisticated SLA can be the user's utility function based on the pricing for each of the primitive services. This way, more sophisticated pricing schemes can be supported by the underlying DiffServ mechanisms.

Since 1996, DiffServ has generated a tremendous amount of research interest. Early work like [9] focuses on architecture and mechanisms. Later work [14, 61, 62, 19, 20, 17, 32] focuses on different types of services that can be supported by DiffServ architecture and the necessary mechanisms to achieve those types of services. The different proposals are testimonies that DiffServ architecture is flexible enough to support different types of services.

1.4 Thesis Organization

This thesis contributes to the Differentiated Service research. It describes and discusses the architecture of DiffServ, then proposes an integrated set of mechanisms for allocating different bandwidths to TCP traffic and offers an evaluation on those mechanisms. This thesis is organized as follows.

In Chapter 2, we first describe the current Internet service model, as well as the Integrated Services, and its respective mechanisms. In essence, the DiffServ architecture tries to combine the best of both worlds by offering service assurance to applications with scalable, efficient network mechanisms. We then introduce the DiffServ architecture and compare it with previous approaches. We proceed to discuss a few aspects that DiffServ offers, e.g., a variety of services, sender-based control and receiver-based control and how DiffServ can be seen as a preliminary network defense against Denial-of-Service (DOS) attacks. Within the same flexible architecture, there have been a number of proposals for an integrated set of mechanisms, which we will discuss. These proposals, along with what we will propose in Chapter 3, can be seen as different schemes to satisfy different kinds of DiffServ services.

In Chapter 3, we describe a set of mechanisms for robust and precise allocation of network bandwidth for TCP traffic. Though the specific mechanisms can be seen as modifications to the existing congestion control mechanisms in both routers and end hosts, we do provide a framework to think about other possible designs of bandwidth allocation

schemes. The set of mechanisms include RIO, a preferential dropping algorithm; TSW, a probabilistic tagging algorithm, and three mechanisms for end hosts, collectively called TCP-DiffServ. Those mechanisms are simple modifications to the existing Internet mechanisms, and are practical and deployable.

In Chapter 4, we evaluate the proposed mechanisms using elaborate simulations and a limited set of testbed experiments. We first consider applying only the router mechanisms: RIO and TSW, and we explore different aspects of the DiffServ architecture in this setup. Then we consider applying both router mechanisms and end host TCP mechanisms. We conclude that with the set of integrated mechanisms we propose, a DiffServ domain could allocate network bandwidth in a fair, robust, and precise manner.

Finally, in Chapter 5, we offer some perspectives on how DiffServ can be connected to other part of the Internet architecture, as well as speculating on how to achieve end-to-end DiffServ.

Chapter 2

Architecture

In this chapter, we describe the *Differentiated Services* architecture. We first describe two related architectures: the current Internet architecture characterized by the *best-effort* service model and the *fate-sharing* principle, and the proposed Integrated Services (IntServ) architecture. Then, we introduce the Differentiated Services architecture, which is an attempt to combine the best of the two previous architectures: efficiency from the current Internet and provisioning for QoS from IntServ.

We proceed to discuss a few aspects of DiffServ: its flexible support for a variety of services, provisioning issues, and its support for sender-controlled as well as receiver-controlled schemes. Unlike the current Internet architecture, DiffServ provides a preliminary protection against denial-of-service attacks. Finally, we discuss related work in the DiffServ research and standardization efforts.

2.1 The Best-effort Service Model

Topologically, the current Internet can be modeled as an arbitrary interconnection of Autonomous Systems (ASes). Each autonomous system mirrors a real-world entity, e.g., an Internet Service Provider (ISP), or a university. There is a two-tier routing structure in today's Internet. Within one AS, routers exchange routing information using an interior gate-

way protocol (IGP). There is a consistent routing metric being used by all routers within an AS. ASes further exchange routing information using an exterior gateway protocol (EGP). The most recent version of EGP is called Border Gateway Protocol, or BGP [53]. BGP can handle different routing policies. Therefore, the Internet is a heterogeneous environment of many inter-connected ASes, each with its individual routing policies and belonging to different constituents.

Within each AS, there are two types of devices: routers and end hosts. Routers exchange routing information with each other, multiplex IP packets streams from different incoming links, and forward IP packets to their neighbors. Routers provide a very simple unreliable service to IP packets called the best-effort service. Such service is usually implemented with a queuing mechanism called *drop-tail* queuing, in which a router drops all arriving packets if its queue is full. The end hosts implement the TCP/IP protocol suite, and provide a reliable, sequential transport-layer service to high-level applications. Figure 2.1 depicts the components in the current Internet architecture.

The reason that the Internet chooses the best-effort model can be traced to its military research network roots. The Internet was conceived in late 1960s as an extension to the ARPANET, a defense research project funded by the Advanced Research Project Agency (ARPA). The primary design goal of the Internet was "survivability in the face of failure" [6]. That is, if two entities are communicating over the Internet and some failure occurs and temporarily disrupts the Internet, the two entities should still be able to communicate without re-establishing their conversation. The approach chosen to implement this is called *fate-sharing* [9], which puts the state information at the endpoint of the network. This way, the only way that state information is completely lost is when the communicating party itself has failed. The building block chosen then was IP datagram (packetized data), and consequently, routers are stateless packet switches. It is up to the end hosts to maintain the state information about connections.

In the current Internet, the network resource allocation scheme is built on a set of congestion control mechanisms in both routers and end hosts. In routers, there are drop-tail

queues that drop packets when the buffers are full. Lost packets in networks are detected by TCP in the end hosts, and are taken as congestion signals. TCP slows down its sending rate upon detecting congestion. This alleviates network congestion, and the end host TCP then gradually increases its sending rate. The bandwidth a TCP connection receives from the network is dependent on the network congestion state of the routing path and how many other simultaneous connections there are, and is not always predictable.

The current Internet uses a simple architecture that is characterized by the following three attributes:

- Distinction between two types of devices: routers and host devices

  The routers are network-level (IP) devices that exchange routing information and collaboratively deliver a packet from its source to its destination. They form a consistent substrate to deliver packets generated by end hosts. The end hosts are transport-layer (TCP or UDP) devices that send and receive IP packets to be delivered by the routers.

- No explicit contract on service provided by network to end hosts

  End hosts can send traffic into the network without explicit connection setup. The routers use a simple mechanism to deliver IP packets, in which routers drop packets if they are congested or have failed. It is up to the transport-layer protocols at the end hosts to ensure that a packet eventually arrives at its final destination. Therefore, there is no explicit commitment from the network to end hosts as to whether a packet will be delivered or when a packet will be delivered.

- One type of service provided by a consistent network substrate

  At the network level, all packets are subjected to the same treatment from the network, i.e., forwarded according to the destination IP address and dropped if the routers are congested.

The best-effort service model is simple and allows a great deal of statistical multiplexing, which leads to efficient use of the network resources. The pitfall of this approach is

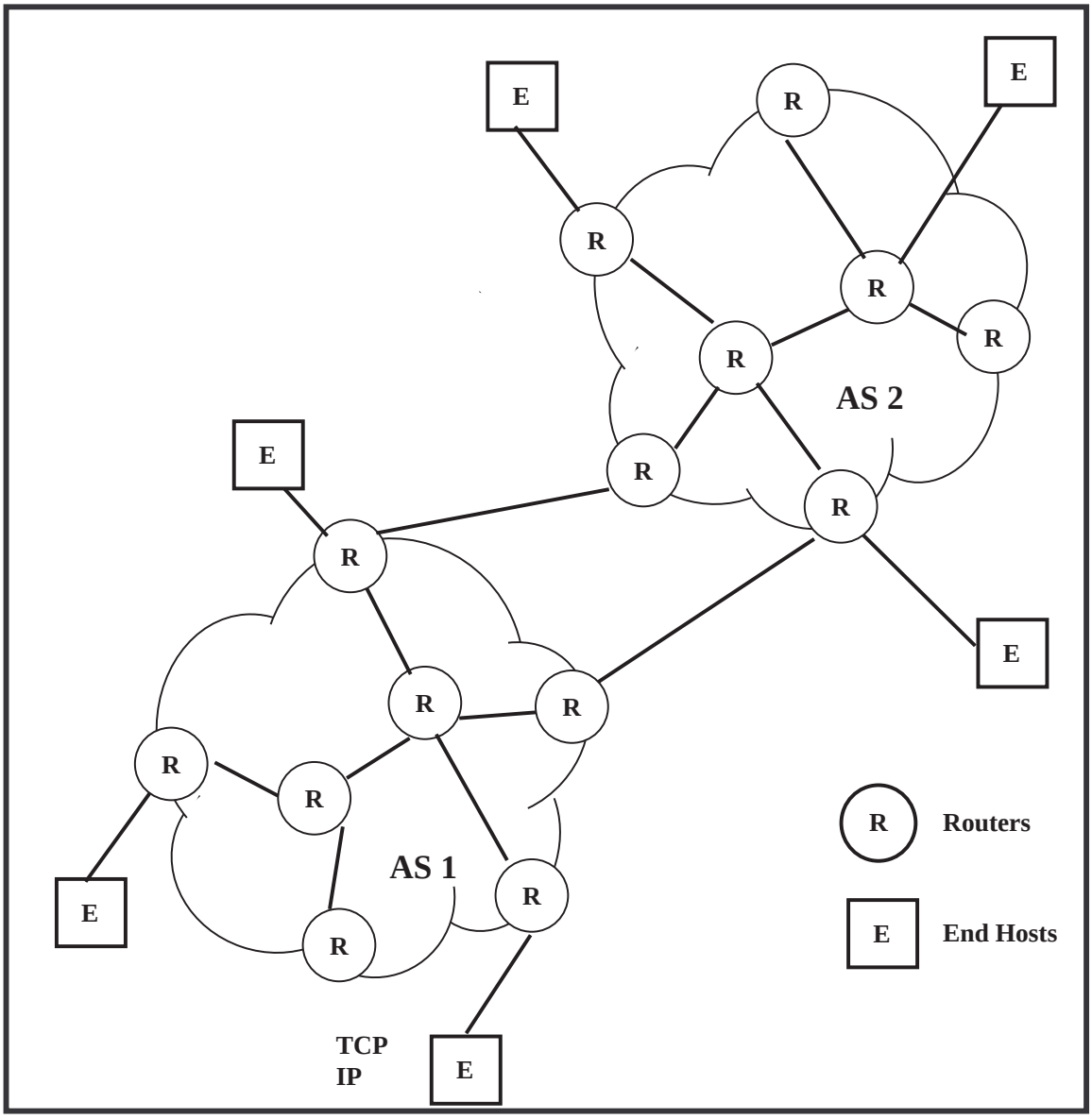


Figure 2.1: Key Elements in the Current Internet

that without explicit support from the network, it is difficult to provide any kind of service assurance or guarantees to end host traffic.

2.2 Integrated Services

As briefly discussed in Chapter 1, the Integrated Services model is an extension to the original Internet best-effort architecture[4]. It includes two services targeted towards real-time applications: *guaranteed* and *predicted* service. *Guaranteed* service involves pre-computed worse-case delay bounds and *predicted* service uses the measured performance of the network in computing delay bounds.

Based on these considerations, network researchers believe that the Integrated Services model will 1) keep additional flow state in routers; 2) require an explicit setup mechanism to install and eliminate flow state in routers. Therefore, in terms of implementations, an Integrated Services framework has four components:

- Packet Scheduler

  The packet scheduler manages the forwarding of different packet streams using a set of queues or perhaps other mechanisms like timers.

- Classifier

  When a packet arrives at a router, it is first *classified* by the classifier, i.e., mapped into some classes. Packets in the same class get the same treatment from the packet scheduler. The classifications are decided by the upper layer policies and can be used for accounting purposes.

- Admission Control

  Admission control implements the decision algorithm that a router or a host uses to determine whether a new flow can be granted the requested QoS without impacting earlier guarantees. Admission control is invoked at each router to make a local accept/reject decision.

- Reservation Protocol

  A reservation protocol is the signaling protocol that carries QoS requests from end host applications to routers on the routing path. QoS requests are necessary to create and maintain flow-specific state along the path of a packet stream. Towards this end, Zhang, Deering, Estrin and Shenker in [67] proposed a setup protocol called *RSVP*. RSVP is a receiver-initiated reservation protocol that can accommodate the needs of both unicast and multicast traffic. The general idea is that RSVP carries QoS requests—a list of parameters called a *flowspec*[49]—along the reverse path of the data stream, and make reservations in each router. If any router does not have sufficient resources to accept such request, its admission control module rejects the request, and the rejection is carried by RSVP to the initiating end host. If all routers along the data path accept the flowspec, then RSVP is responsible for maintaining and refreshing the QoS state that has been established by the routers. The state kept in the routers is *soft state*, that is, it expires if not refreshed by periodic RSVP requests. This provides graceful support for dynamic membership changes and automatic adaptation to routing changes.

IntServ's proposed mechanisms are a major departure from those in the best-effort model in that IntServ requires both explicit setup prior to traffic transmission and keeping per-flow state in the network. Architecturally, IntServ is similar to the best-effort model in that there are two types of devices: routers and host devices. However, in IntServ, the routers are much more complicated. They have to implement a reservation protocol to establish QoS state and maintain mechanisms to keep, delete or refresh QoS state. We list the attributes of an IntServ architecture below.

- Distinction between two types of devices: routers and host devices

  Just like that in the best-effort model, the routers are network-level (IP) devices that exchange routing information and collaboratively deliver a packet from its source to its destination. The routers form a consistent substrate to deliver packets generated

by end hosts. In addition, they have to implement protocols to establish, delete and refresh QoS state at the requests of end hosts. The end hosts are transport-layer (TCP or UDP) devices which generate and receive IP packets to be delivered by the routers. Additionally, they initiate RSVP protocols to establish QoS state.

- Explicit contracts on service between the network and the end hosts

  In order to receive a certain QoS, the end hosts have to specify explicit QoS requests (*flowspec*) to the network. The traffic generated by the end hosts to receive such QoS requests is also subjected to explicit checking and policing by the network.

- Fine-grained, per-flow levels of services by a consistent network substrate

  Inside the network, packets that belong to flows having QoS specifications and having been admitted into the network receive per-flow, QoS treatment by all routers on the routing path. Since QoS requests by end hosts are fine-grained, the services provided by IntServ network to end hosts are also fine-grained.

It is clear that if IntServ mechanisms can be implemented in the network, then the network can support explicit QoS requests from the end hosts. However, this particular feature comes at the cost of losing some network efficiency. Explicit reservations for flows mean that there is less or no statistical sharing of network resources among flows that have requested explicit QoS. Since statistical multiplexing is what made the best-effort service model efficient, IntServ's approach reduces the network efficiency. Another serious concern is about the complexity involved in establishing, maintaining, and deleting QoS state at per-flow granularity in IntServ's approach.

2.3 Differentiated Services

Differentiated Services (DiffServ) tries to combine the benefits of the best-effort model and the IntServ model. The primary goal is to preserve the statistical multiplexing nature of the

current Internet, while using scalable, flexible mechanisms to provide a wide range of QoS services. A secondary goal is to make network resource usage accountable.

The general idea of the DiffServ architecture is to mark packets into a small number of classes at the edge of the network and to create mechanisms inside the network to differentially treat different classes of packets. Each user is associated with a *Service Level Agreement* (SLA), which is a contract between a customer and an Internet Service Provider (ISP) that specifies the expected forwarding service a customer should receive. An SLA also specifies a profile of what a customer's traffic will look like. The customer pays for the SLA, under the condition that the ISP delivers the specified forwarding service to traffic within the profile. Any traffic that is in excess of the profile is considered opportunistic traffic and is not provided with any kind of service assurance.

At the edge of the network, the ISP's edge routers classify packets and map traffic to their respective SLAs. Traffic sent within the profiles in SLAs are marked by edge routers into different classes of packets. Traffic sent outside the profiles in SLAs are left unmarked by the edge routers and is considered opportunistic traffic. In this thesis, we illustrate the different classes of packets with only two types of packets, IN packets and OUT packets. IN packets represent packets within a profile and OUT packets represent packets beyond a profile. The edge routers mark packets as IN packets if the traffic is within a customer's SLA; anything *in excess* of the SLA is tagged as OUT packets. Inside the network, core routers only need to distinguish between two types of packets and give IN packets preferential treatment in terms of bandwidth or delay, or both.

In DiffServ, only edge routers need to keep per-flow state and the core routers keep no per-flow state. This way, the complexity of the system is pushed to the edge of a network. Putting per-flow state in only edge routers makes DiffServ architecture more scalable than IntServ. Additionally, SLAs serve as a basis for ISPs to account for the network resource usage.

Figure 2.2 depicts a Differentiated Services network with two DiffServ domains. The core routers adopt simple mechanisms on its forwarding path and give preferential treat-

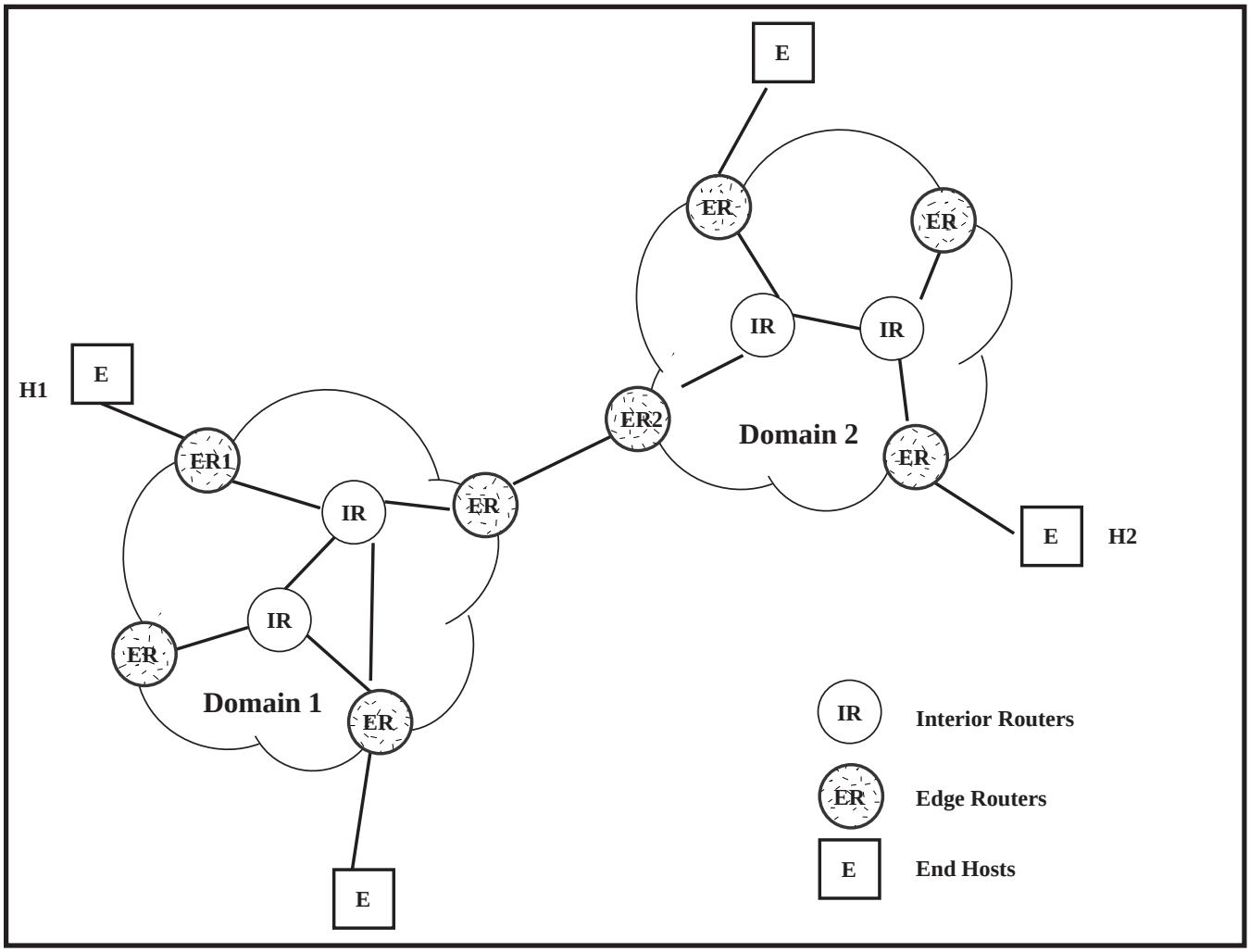


Figure 2.2: Key Elements in a DiffServ Network

ments to different classes of packets. The edge routers deploy mechanisms—called *Traffic Conditioners*—that classify, monitor, and tag packets. Finally, DiffServ might also require simple mechanisms to be deployed in end hosts to achieve precise resource allocation. Details on mechanisms can be found in Chapter 3.

A connection that spans multiple DiffServ domains goes through several edge routers; its packets are marked and remarked at the edge routers of each domain. For example, in Figure 2.2, a connection from H1 to H2 spans two adjacent DiffServ domains. Packets from the connection are marked for the first time at edge router ER1 of Domain 1. The marking is according to an SLA between customer H1 and Domain 1. When packets exit Domain 1 and enter Domain 2, they are marked again by edge router ER2. This time, the packets, together with packets from other connections traversing from Domain 1 to Domain 2, are marked according to an SLA between Domain 1 and Domain 2. In other words, edge router ER2 marks *aggregated* traffic, since ER2 does not keep SLAs for individual connections.

Compared to both the current best-effort Internet and the IntServ, the DiffServ architecture has the following four attributes:

- Distinctions between edge routers, interior routers and end hosts

  The DiffServ architecture distinguishes between end hosts, which implement transport-layer protocols, and routers, which implement network-layer protocols, just as the current Internet does. Additionally, it further distinguishes between two types of routers: edge routers and interior routers. Edge routers keep state information about SLAs, classify packets, mark packets into different classes and then police the arriving packets according to the service profiles. Interior routers do not need to keep per-flow state, they only need to distinguish among a relatively few classes of packets and give preferential treatments to different classes of packets.

- Explicit contracts on services between the network and the end hosts

  The service provided by the network to the end hosts is described in SLAs, which are long-term, static service profiles. An SLA describes the expected traffic speci-

fications from an end host to the network. Traffic from an end host can exceed the specified SLA as long as it doesn't congest the network. If the network is congested, the end host is expected to slow down to the specified profile. The difference between an SLA and a *flowspec* in IntServ is that the former is long-term, *expected* service profile and the latter is an exact description of traffic specification corresponding to an explicit QoS request.

- A few differentiated levels of services by a consistent network of interior routers

  In the DiffServ architecture, once packets are inside the core of the network, they are treated as aggregates. Interior routers distinguish between a few classes of packets by examining the Type of Services (TOS) field of IP headers and treat different classes of packets differently. The interior routers form a network substrate that provide consistent treatment of packets of different classes. Packets inside the network are aggregated, and enjoy a high degree of statistical multiplexing, as in the best-effort service model. This design preserves the network efficiency that comes with statistical multiplexing.

- A variety of services provided to end hosts

  DiffServ architecture hopes to provide a variety of services, in terms of bandwidth and delay requirements, to either aggregated traffic or per-flow traffic. This is done by having different traffic conditioners at the edge of the network.

Architecturally, DiffServ takes an approach that combines the best of both worlds: a high degree of aggregation from the best-effort model and QoS assurance from the IntServ. The edge routers keep per-flow state, and monitor and mark traffic using traffic conditioners. The interior routers are still stateless, as in the current Internet. This approach is more scalable in terms of mechanisms than that of the IntServ.

Like IntServ, DiffServ takes advantage of the same observation that many real-time applications are adaptive and do not require stringent network guarantees. However, it differs from IntServ in a number of ways. Instead of trying to support a very fine level of

QoS specifications in networks itself, DiffServ only supports that level of specifications at the edge of the network, where it maps these QoS specifications to a few classes of packets. In the interior of the network, DiffServ treats different classes of packets differently. This design immediately simplifies the design of the interior network and makes the mechanisms scalable.

DiffServ separates the *implementable services* from the actual *implementing mechanisms*. DiffServ aims to support many different flexible services, whether they are for fine-grained, per-flow traffic or aggregated traffic, and whether they are sender-based or receiver-based. In terms of the actual implementations, however, these services can be supported by one or many integrated sets of mechanisms. For example, there can be different approaches to providing differentiated levels of bandwidths to applications and each approach proposes an integrated set of mechanisms for implementation. From an ISP's point of view, this provides a number of implementation alternatives to choose from. From the users' perspective, applications can run transparently on top of different mechanisms as long as the mechanisms can achieve what's specified in the profile.

In terms of time scale of service profiles, IntServ expects to support walk-in, dynamic traffic that establishes a reservation prior to sending. In contrast, DiffServ service profiles are long-term and static.

In terms of mechanisms used, IntServ uses a hop-by-hop, receiver-initiated reservation protocol RSVP, and shaping and policing mechanisms to admit traffic into the network. In DiffServ, there are quite a few proposals on what the actual mechanisms could look like. Some require admission, shaping and dropping traffic at the edge of the network and scheduling mechanisms in the interior of the network; others require only marking packets at the edge of the network and a dropping mechanism in interior routers. Since there are only a few different classes of packets inside the network, interior routers do not need to support a complex protocol like RSVP. Table 2.1 summarizes the differences between IntServ and DiffServ.

Table 2.1: Differences between IntServ and DiffServ

	IntServ	DiffServ
Service profiles	fine-grained, per-flow receiver-based	per-flow or aggregate sender-based or receiver-based
Time scale of service profile	walk-in dynamic	long-term static
Mechanisms involved	per-hop RSVP	separate edge and interior mechanisms, flexible

2.4 Issues in DiffServ

In the next four sections, we discuss a few selected aspects of DiffServ architecture[1].

2.4.1 A Variety of Services

In designing the DiffServ framework, we are serving two potentially conflicting goals. First, we would like to implement a set of simple services that are useful and easy to understand and adopt; second, we do not want to embed the above services into the mechanisms so that the framework cannot adapt to new applications with new service requirements in the future. The decoupling of the SLAs at the edge of the network from the differential treatment of packets in the center of the network allows this flexibility. To over simplify, the preferential dropping scheme adopted in routers will not change over time and can be standardized; whereas the characteristics of a service are defined and captured by its corresponding traffic conditioners and it is only necessary to create new traffic conditioners at the edge of the network to adopt a new service.

The services provided by this framework are diverse. As a simple example, it could be the equivalent of a dedicated link of some specified bandwidth from a source to a destination. Such a model is easy for users to understand. A more elaborate model can be an aggregated commitment to a range of destinations, or anywhere within an ISP, sometimes

[1]Section 2.4.1, section 2.4.2, and section 2.4.3 are taken in large part from a joint work by Dave Clark and the author

called a Virtual Private Network (VPN). A virtual network is by nature more difficult to offer with high assurance since offering commitments to "anywhere within a Virtual Network" implies that the ISP has provisioned its resources adequately to support all users sending IN traffic simultaneously to any destination.

Not all Internet traffic is continuous in its requirement for bandwidth. In fact, most Internet traffic is very bursty. It may thus be that a "virtual link" service model is not what users really want. It is possible to support bursty traffic by changing the traffic conditioners to implement this new sort of service. The key issue is to ensure, in the center of the network, that there is enough capacity to carry this bursty traffic, and thus actually meet the commitments implied by the outstanding profiles. This requires a more sophisticated provisioning strategy than the simple "add 'em up" needed for constant bit-rate virtual links. However, in the center of the existing Internet, especially at the backbone routers of major ISPs, there is a sufficiently high degree of aggregation that the bursty nature of individual traffic flows is no longer visible. This suggests that providing bursty SLAs to individual users will not create a substantial provisioning issue in the center of the network, while possibly adding significant value to the service as perceived by the users.

A more sophisticated SLA would be one that attempts to provide a specified and predictable throughput to a TCP stream. This is more complex than a profile that emulates a fixed capacity link, since TCP hunts for the correct operating rate by increasing and decreasing its window size, which causes rate fluctuations to which the profile must conform. The service allocation profile is easy for a user to test by simply running a TCP-based application and observing the throughput. This is an example of a higher level profile, because it is less closely related to some existing network components and more closely related to the users' actual demands. This kind of service is the focus of this thesis. The mechanisms and evaluations we offer are targeted throughput for a TCP connection.

In summary, three things must be considered when describing an SLA:

- Traffic specification

  What exactly is provided to the customer in terms of bandwidth, delay or both? For

example, a SLA can specify a 5Mbps average bandwidth, or an end-to-end delay of no more than 100ms, or both.

- Geographic scope

  To where this service is provided. Examples might be a specific destination, a group of destinations, all nodes on the local provider, or "everywhere". The specifics of this condition reflect the geographic scope of the domain with which DiffServ services are provided, as well as the set of upstream and downstream domains (where the possible destination machines are) this domain is connected to.

- Probability of assurance

  With what level of assurance is the service provided. Since DiffServ does not provide a hard guarantee, the probability of assurance is a metric a service provider should also include in the SLAs.

These things are coupled; it is much easier to provide "a guaranteed one megabit per second" to a specific destination than to anywhere in the Internet.

2.4.2 Network Resource Allocation and Provisioning

The statistical multiplexing nature of the Internet makes efficient use of bandwidth and supports an increasing number of users and new applications. However, it does lead to some uncertainty as to how much of the bandwidth is available at any instant. The approach we take to allocating network resources is to follow this philosophy to the degree that the user can tolerate the uncertainty. In other words, a capacity allocation scheme should provide a range of service assurance. At one extreme, the user may demand an absolute service assurance, even in the face of some network failures. Less demanding users may wish to purchase a SLA that is *usually available*, but may still fail with low probability. The presumption is that a higher assurance service will cost substantially more to implement.

Thus, DiffServ takes a different approach than the previous Integrated Services effort. In Integrated Services, applications that require a higher level of commitment than the best-effort service take explicit actions to make reservations along the traversing route, using protocols like RSVP [67]. In DiffServ, SLAs are *expected* services from the network. The term *expected* suggests that the SLAs do not describe a strict guarantee, but rather an expectation that the user can have about the service he will receive during times of congestion. This sort of service somewhat resembles the Internet of today in that users have some expectation of what network performance they will receive; the key change is that our mechanism permits different users to have different expectations.

It should be noted that traffic requiring this higher level of assurance can still be aggregated with other similar traffic. It is not necessary to separate out each individual flow to ensure that it receives its promised service. For example, there could be two queues in the router, one for traffic that has received a statistical assurance, and one for this higher or *guaranteed assurance*. Within each queue, IN and OUT tags would be used to distinguish the subset of the traffic that is to received the preferred treatment.

Fundamentally, statistical assurance is a matter of provisioning. Theoretically, provisioning a network with an arbitrary set of links and required capacity is an NP-complete problem [40]. There are only solutions to reduce it to a NP problem using heuristics. In practice, however, an ISP monitors and measures the amount of traffic crossing various links over time, and provides enough capacity to carry this subset of the traffic, even at times of congestion. This is how the Internet is managed today. Using DiffServ, the additional classification of packets will give an ISP a better idea of how much of the traffic at any instant is *valued* traffic, and how much is discretionary or opportunistic traffic for which a more relaxed attitude can be tolerated.

2.4.3 Sender-controlled and Receiver-controlled Schemes

Up to this point, we have assumed that it is the sender that is concerned with getting preferential treatment of its packets and is willing to pay for the profiles. However, in today's

Internet, the receiver of the traffic, not the sender, is often the more appropriate entity to make such decisions. For example, some video-on-demand subscribers are willing to pay for a better delivery of traffic to their homes. We will show that the DiffServ architecture is flexible to allow receiver-controlled schemes, hence, receiver-based pricing.

The receiver-based scheme in the DiffServ framework is the complement of the sender-based scheme. It relies on a newly proposed change to TCP called the Explicit Congestion Notification (ECN) bit [21]. In ECN semantics, congested routers turn on the ECN bit in a packet instead of dropping the packet. The TCP receiver copies the ECN bit into the acknowledgment (ack) packet, and the sender TCP gracefully slows down upon receiving an ack with the ECN bit on.

In the receiver-based expected capacity scheme, routers are ECN-compatible routers; they turn on the ECN bit in a packet when there is congestion, instead of dropping them. A traffic conditioner, installed at the receiver checks whether a stream of received packets is inside of the profile. Each arriving packet debits the receiver's service allocation profile. If there is enough profile to cover all arriving packets, the traffic conditioner will turn off the ECN bits in those packets which had encountered congestion since the receiver is entitled to receive at this rate. If the receiver's profile is exceeded, packets with their ECN bits on will be left unchanged at the traffic conditioner. If packets arrive at the TCP receiver with ECN bits still on, it means that the receiver has not contracted for sufficient capacity to cover all the packets that encountered congestion, and the sender will be notified to slow down.

There are a number of interesting asymmetries between the sender-based and the receiver-based schemes, which arise from the fact that the data packets flow from the sender to the receiver. In the sender-based scheme, the packet first passes through the traffic conditioner, where it is tagged, and then through any point of congestion. In contrast, in the receiver-based scheme the packet first passes through any points of congestion, where it is tagged, and then through the receiver's traffic conditioner. The receiver scheme can convey to the end point dynamic information about the current congestion levels, since routers only set

the ECN bit if congestion is actually detected. In the sender scheme, in contrast, traffic conditioners must tag the packets as IN or OUT without knowing if congestion is actually present. Thus, we could construct a service, based on the receiver scheme, to bill user for actual usage during congestion.

On the other hand, the receiver scheme is more indirect in its ability to respond to congestion. Since in the sender scheme, a packet carries the explicit assertion of whether it is IN or OUT of profile, the treatment of the packet is evident when it reaches a point of congestion. In the receiver scheme, the data packet itself carries no such profile indication, so at the point of congestion, the router must set the ECN bit, and still attempt to forward the packet, trusting the sender will correctly adjust its transmission rate. Of course, if the traffic conditioner at the receiver's side employs a dropping algorithm that drops any packets that exceeds the profile, the sender will slow down if it is a properly behaved TCP.

Another difference between the two schemes is that in the sender scheme, the sending application can set the IN/OUT bit selectively to control which packets are favored during the congestion. In the receiver scheme, all packets sent to the receiver pass through and debit the traffic conditioner before the receiver host gets them. Thus, in order for the receiver host to distinguish those packets that should receive preferred service, it would be necessary to install some sort of packet filter in the edge router that keeps the SLA.

2.4.4 Denial-of-Service Attacks

As the Internet has transitioned from a closed research network to an open, heterogeneous network, it has become vulnerable to malicious Denial-of-Service (DOS) attacks. As some events in early 2000 [54] demonstrate, Denial-of-Service attacks come in various forms. One common attack is to use spoofed but valid IP addresses as source addresses of packets and inject packets into the targeted network. The amount of bogus packets injected into a targeted domain is so great as to overwhelm the routers and stall the normal operations of the network. The difficulty in detecting and preventing DOS attacks is that the network has no way of telling a *good* packet—packet that is from a valid host and should be carried

through the network—from a *bad* packet, a packet with a spoofed IP source address. The lack of detection mechanisms prevents a network from quickly filtering out the bad packets and recovering from the attacks.

The DiffServ architecture, if deployed, is not immune to such attacks. In this section, we discuss problems and solutions when a DiffServ domain is under a DOS attack.

2.4.4.1 Sender-based Scheme

In the sender-based scheme, traffic from a customer is allowed into a DiffServ domain if it already has an SLA with the service provider. Traffic from customers who do not have SLAs with the service provider is at the whim of the service provider. A discrete service provider may choose to implement explicit admission control at the edge of the network and drop all traffic that does not have an SLA contract. In this case, the network is less vulnerable to DOS attacks because it must have provisioned enough to handle all "inside profile" traffic. In the case where a service provider does not implement explicit admission control, but only classify and mark packets to different DiffServ classes, the mechanisms in core routers should be able to handle the extreme condition when the network is under heavy attacks.

In DiffServ, the criteria of good and bad packets is very clear: packets within the SLAs are considered good packets and should be delivered by the network; packets outside the SLAs are considered bad, or opportunistic traffic, and can be dropped. Since DiffServ introduces a criteria to distinguish among packets and installs admission control mechanisms at the edge of the network, it presents a preliminary shield against DOS attacks.

There are three scenarios when a DiffServ domain is under DOS attacks.

- All spoofed sources do not have valid SLAs

  This is the best scenario. In this case, all traffic generated to mount DOS attacks are OUT packets. Since DiffServ networks are already capable of shielding IN packets from OUT packets under various congested state, the network can easily filter out the bad packets (they happen to be OUT) and can resume normal operations.

- All spoofed sources do have valid SLAs

  This is the worse scenario. In this case, all traffic generated to mount DOS attacks are IN packets. Since a DiffServ domain is supposed to carry all IN packets, all this traffic is allowed into the network, and the DiffServ domain has no additional criteria to distinguish between the good and bad packets and therefore, the domain is vulnerable to disruptions. In this case, a DiffServ domain is no more vulnerable than the current Internet because it still has admission control at the edge to determine the amount of IN packets allowed into the network. Since it should have provisioned enough to carry IN packets, the network should be able to carry spoofed IN packets.

- Some of the spoofed sources have SLAs and some do not

  Between the two above scenarios, there could be a spectrum of scenarios, each with an arbitrary mix of the above two scenarios. In any case, the ability of a DiffServ domain to shield IN packets from OUT packets will prevent itself from being overwhelmed by DOS packets which are OUT packets; the overall provisioning of a DiffServ domain will prevent itself from being overwhelmed by DOS IN packets. Therefore, this scenario is no worse than case 2.

Therefore, we have shown that a domain implementing sender-pay DiffServ mechanisms is less vulnerable to DOS attacks than it is without those mechanisms. This is because the DiffServ mechanisms define a criteria for distinguishing between good packets and bad packets and have sufficient provisioning for the good packets. Under DOS attacks, the DiffServ mechanisms for distinguishing good and bad packets can provide a shielding effect by dropping the bad packets, thus dampening the possible effect of DOS attacks.

2.4.4.2 Receiver-based Scheme

The receiver-based scheme, however, poses a bigger security threat and requires additional robust and scalable network mechanisms in order to work. In the receiver-based scheme, a receiver purchases an SLA from a DiffServ domain, and any sender sending to that receiver

can turn on a bit—the receiver-pay bit—indicating it is the receiver, not the sender, who is paying for the traffic. A DiffServ domain allows the traffic with the receiver-pay bit on into the network. The traffic is delivered to the edge of the network and the traffic conditioners check the traffic against the receiver's profile. If the receiver's SLA covers the amount of traffic, the packets are further directed to the receiver; otherwise, the network can drop the packets at the edge of the network.

In this scheme, the information of whether the receiver has enough profile to cover the traffic is not known at the point when traffic *enters* the network but when the traffic *exits* the network. If the receiver does not have enough profile, these packets are dropped only *after* they have unnecessarily consumed network resources. The network is prone to a new type of Denial-of-Service attack, in which, a malicious host can simply mark all its traffic as receiver-pay with a spoofed destination and inject into the network. This is the converse scenario to the normal DOS attacks in the current Internet. In other words, if receiver-based scheme is adopted, a DiffServ domain is more open to DOS attacks.

There are two possible approaches to deal with this. One approach is a preventive measure, in which a receiver's SLA is installed at *all* edge routers (other than just the edge router closest to the receiver), along with an access list of hosts which are authorized to use this SLA. All traffic, whether it is sender-pay or receiver-pay, is subjected to admission control by traffic conditioners when they enter the network. This approach is robust, however, not scalable because all edge routers have to maintain an access list for every receiver-pay SLA.

An alternative approach in dealing with potential DOS attacks are to "detect and penalize". We expect that even in an increasingly heterogeneous Internet, end host misbehavior will be the exception and not the rule. Detect-and-penalize is a cheaper method than a preventive measure if DOS attacks is an infrequent event, but it is less robust because it takes time for the network to identify and react to misbehaving hosts. However, we believe that as long as the penalty imposed is sufficiently harsh, this approach should also provide the proper incentives for users and end hosts to behave appropriately.

In the context of receiver-pay schemes, a misbehaving end host is one that sends unauthorized receiver-pay traffic. The challenge here is not only in detecting it but rather detecting it early enough to minimize the damage it might have caused. Since the egress router closest to the receiver knows the receiver's SLA, detecting an unauthorized receiver-pay traffic stream at this egress point is easy. The problem is how to quickly communicate this information back to the ingress point of the DiffServ domain so the offending traffic can be dropped as early as the ingress point. There are a number of solutions. First, we could use explicit control messages, such as IP Source Quench [50] messages from egress routers to ingress routers, who ultimately will take responsibilities for penalizing the unauthorized source. Another option is to adopt a push-back mechanism, such as a protocol described in [65]. In this protocol, the desired information is pushed back hop-by-hop by the routers along the reverse direction of the packet stream.

Once a DiffServ domain has detected a misbehaving host, the traffic conditioner in the edge router that is the closest to the misbehaving host must apply an appropriate penalty. The penalty serves two purposes: 1) limit the damage to the network that misbehaving hosts can cause; and 2) provide the proper (dis)incentives to discourage abusive behavior. One method of penalty is to simply drop all packets of the offending flow. Alternatively, edge router could downgrade the traffic to OUT packets, or turn into sender-pay packets and count against the sender's profile.

2.4.4.3 Summary of DOS Attacks in DiffServ Domains

In summary, DiffServ architecture, with its flexibility to support both sender-pay and receiver-pay schemes, opens new challenges in the face of Denial-of-Service attack. If DiffServ only supports sender-pay schemes, then the resulting environment is better than the current Internet in its defense against DOS attacks. This is because DiffServ architecture has instrumented a criteria for distinguishing between *good* and *bad* packets, and good packets always have priority over bad packets to network resources. However, the receiver-pay scheme opens up another loophole for DOS attacks and additional network mechanisms

are needed before DiffServ architecture can support them. We discussed two approaches: a preventive measure that is robust but not scalable; and a "detect and penalize" measure that can potentially be both scalable and robust. We propose a few mechanisms in detecting and penalizing offending flows.

2.4.5 Framework for Designing DiffServ Mechanisms

As described previously, a DiffServ network can allocate different network resources to different entities by putting scalable tagging and dropping mechanisms in routers. These mechanisms, as we shall see in Chapter 3, will be integrated into existing Internet mechanisms. At a high-level, the idea is simple: edge routers maintain policy information (SLAs), classify and tag packets; interior routers create differentiations among different classes of packets. The actual implementation of this idea, however, may choose from any of the three designs depicted in Figure 2.3.

Each design occupies a row, depicted by an end host, an edge router and an interior router. The edge router and the interior router represent the network (shaded area). Unlike the current Internet, in DiffServ there is a trust boundary between end host and the network. Everything inside the network is owned and trusted by an ISP, and everything outside the network is not. When traffic crosses the trust boundary from end hosts to the network, it is subjected to classification, tagging, shaping, policing and even dropping. Collectively, these are the functions of a *traffic conditioner*. There are two types of traffic conditioners: active and passive. Active traffic conditioners can shape and drop packets, essentially, affecting traffic patterns, whereas passive traffic conditioners only classify and tag packets but do not affect traffic patterns. Tagging algorithms, or taggers, are passive traffic conditioners. We focus on taggers only.

In design 1, the tagger is placed *inside* the end host. It communicates with the edge router using a signaling protocol that can inform itself about the SLA. This design has the advantage that the tagger can access the internal state variables of TCP, especially those used in TCP's congestion control algorithm, e.g., round trip time (RTT) estimate and the

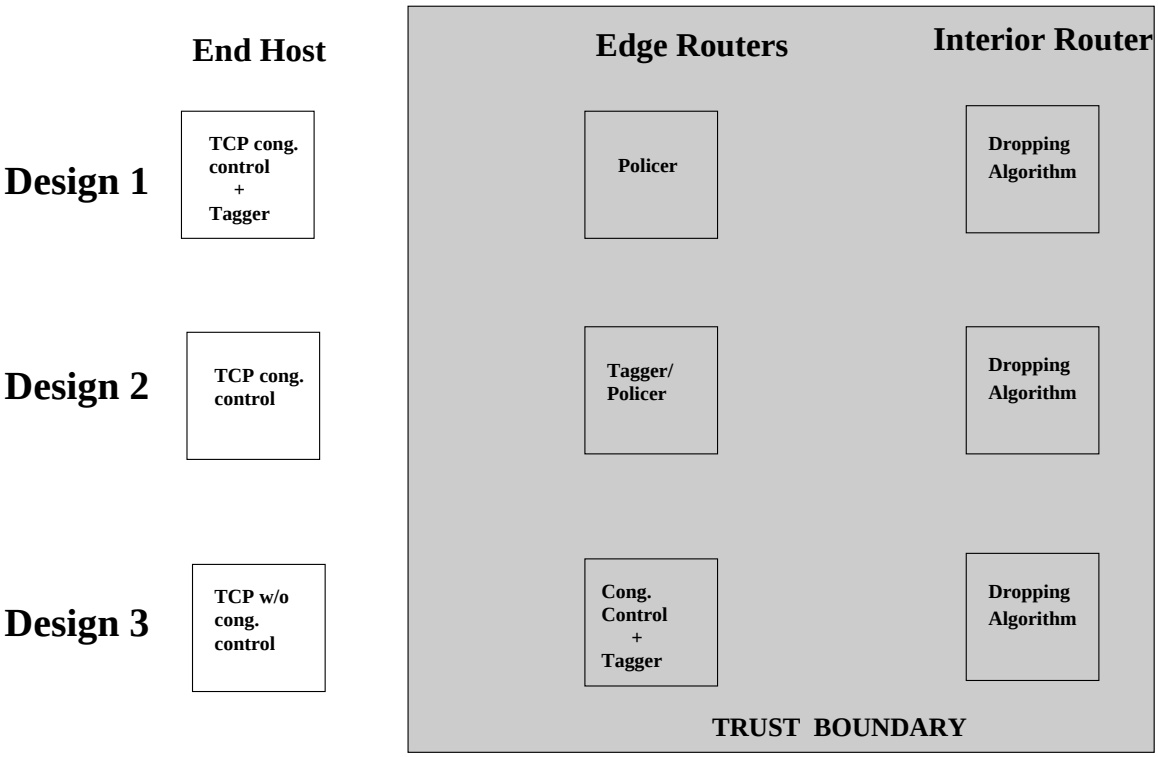


Figure 2.3: Three Possible Designs for Mechanisms

congestion window *cwnd*. Since these TCP state variables are dynamic, knowing them will help the tagger to tag packets precisely and accurately. Moreover, the tagger can be integrated with TCP's congestion control algorithms in controlling the rate that the end host's TCP is sending. Since the tagger communicates with the edge router to obtain the latest SLA, the pace at which end hosts send is well-controlled.

The problem with this design is that the tagger is placed *outside* the trust boundary of the network. In this case, there has to be some kind of authentication and verification mechanism to verify that the tagger has not been tampered with inside the end host, and is tagging/pacing the traffic according to SLAs. Minimally, the network has to do additional checking, or policing, at the edge of the network to check that the end host is not cheating.

In design 2, the tagger is placed at the edge router, *inside* the trust boundary. The end host is not modified. The tagger is configured with the latest SLA stored in edge routers, but they do not have access to the internal state of TCP. This particular design is the most practical approach because the router mechanisms can be relatively easily adopted and changed by ISPs. On the other hand, since the tagger does not have knowledge of the dynamic state in the end host, this design limits the precision and the accuracy of the tagger. A tagger designed this way will have to be versatile enough to work with different traffic mixes, with different round trip times.

In design 3, the tagger is placed at the edge router, inside the trust boundary, we propose a more radical change to the current Internet by taking TCP congestion control mechanisms out of the end hosts and placing them in the edge router. What's left between end hosts and the edge routers is a flow control mechanism that tells the end host TCP the amount of packets to be sent into the network[2]. We place the congestion control mechanism into the edge router and keep per-flow state in the edge routers. The tagger then can be integrated with congestion control mechanism in edge routers. In essence, this design keeps the entire congestion control loop within the network itself. At the edge router, the congestion control mechanism is very much the same as that of the TCP, but it is now combined with a tagging

[2]For example, the control can be in the form of setting TCP's receiver's window.

algorithm for each flow. The interior routers implement congestion avoidance and control schemes. Since a tagger is integrated with congestion control mechanism in an edge router, it can provide very accurate indication about the flow of the traffic to internal networks.

This design does not require authentication and verification mechanisms between end hosts and edge routers as in design 1, and is more accurate than design 2. The disadvantage of this design is that it is a more radical approach and might not be practical in actual deployment [3].

In designing our mechanisms, we start with design 2 because this is the most practical approach, we end up proposing mechanisms which can also be used in the other two designs.

2.5 Related Work

The inception of the idea that eventually led to DiffServ research can be traced to early work by David Clark. In [8], Clark proposed an idea in which accounting information is installed at the edge of the network for pricing and cost allocation purposes, and packets are classified and marked differently according to this accounting information. Inside the network, routers distinguish packets with one bit and treat packets differently. In [7], Clark also mentioned the implications of combining sender-pay and receiver-pay schemes in considering Internet pricing schemes. [9] was the first research paper that demonstrates these ideas can be implemented in the real TCP/IP network and have promising results. Soon after that, there were quite a few different proposals on mechanisms to implement DiffServ, which eventually led to a standardization effort by IETF. Currently, there are two proposed interior router forwarding services proposed by IETF: Assured Forwarding and Expedited Forwarding.

There has been quite a flurry of research on DiffServ since 1997. The rest of this section describes these efforts. Though they conform to the same DiffServ architectural

[3]The idea of separating congestion control mechanism from TCP and treating it as a very general mechanism can also be found in "Congestion Manager" [3].

approach, they differ primarily in two ways: the high-level user perceivable services and the mechanisms required in order to achieve such services.

2.5.1 Proportional DiffServ

The *Proportional DiffServ* [14] offers a different perspective on differentiated services. Instead of offering an *absolute* bandwidth or delay to a class of packets, Proportional DiffServ offers *relative* differentiated service under the premise that the service received by higher classes will be better, or at least no worse, than that of lower classes. In this context, applications and users do not get an absolute service level assurance, such as an end-to-end delay bound or bandwidth, since there is no admission control and resource reservations. Instead, the network assures them that higher classes will be proportionally better than lower classes, so it is up to to the applications and users to select the class that best meets their cost and policy constraints.

In [14], Dovrolis et al. identify and evaluate two packet schedulers, WTP and BPR, that approximate the proportional delay differentiation model in heavy-load conditions, even in short time scales. They conclude that although both schedulers are appropriate for relative delay differentiation, their studies illustrate that WTP is significantly better than BPR in the context of proportional delay differentiation. These schedulers give network operators "tuning knobs" to adjust the quality spacing between classes. They demonstrate that such per-hop and class-based mechanisms can provide consistent end-to-end differentiations to individual flows from different classes, independent of the network path and flow characteristics.

In the proportional DiffServ model, the network does not implement admission control at the edge of the network and relies on its queuing mechanisms to create differentiations among classes of packets. When the network is moderately loaded, the two proposed queuing schemes BPR and WTP [14] may not create sufficient differentiation because the class queues are not sufficiently long for these schedulers to distribute the class delays. When the network is heavily loaded, if the queues are sufficiently long, the schedulers perform

well. However, if there is only limited buffers in routers, the network can be overwhelmed by the heavy load and incur losses, and there is not sufficient class queues for the schedules to distribute class delays either.

2.5.2 Minimum Rate Guarantees in Networks

Another series of work by Feng, et al. [19] explores mechanisms in routers and TCPs that enable the network to guarantee minimal levels of throughput to different TCP sessions. The service realized by the proposed mechanisms can be seen as a possible implementation of the "controlled-load" service in IntServ, in that those modifications allow the network to guarantee a minimal level of end-to-end throughput to different network sessions.

In [19], Feng et al. proposes policing traffic at the source and marking packets using a token bucket. Inside the network, routers use Enhanced Random Early Detection (ERED) algorithm, which is a minor modification of the Random Early Drop (RED) algorithm[4]. ERED uses thresholds for dropping unmarked (non-conformant) packets and does not drop marked (conformant) packets unless the queues are full. In ERED, the thresholds have to be set appropriately to ensure no marked packets are dropped.

At the end host, each TCP is configured with a token bucket policing and marking unit. There are a few mechanisms proposed to TCP to utilize the marking schemes available. First, TCP's sending pace is no longer controlled by the acknowledgment packets from the receiver, but controlled by a *delayed* timer and a *periodic* timer. These timers help TCP to utilize the available tokens in the token bucket more effectively: if there are no tokens available, TCP withholds its sending by the delayed timer until new tokens becomes available; if there are tokens in the bucket, the TCP sender is eligible to inject new data packets into the network even when no acknowledgment packets have arrived. Second, TCP's congestion control algorithm is modified. The essential idea is to divide TCP's congestion control window *cwnd* into two parts: reserved window, which reflects the minimal guaranteed rate from the network, and the rest of the available bandwidth to the connection. The conges-

[4]We will describe RED in detail in Chapter 3.

tion control algorithm is modified so that when TCP detects congestion signals from the network (typically packet drops), it does not reduce its congestion window below the minimal guaranteed rate. The author warns that these modified congestion control algorithms in TCP should be exercised if and only if the network supports minimum rate guarantees through end-to-end signaling, admission control, and resource reservation. Without such mechanisms in place, the use of this modified TCP may cause congestion collapse.

In a similar piece of work[20], Feng et al. propose an adaptive marking engine that can either be integrated with TCP or be transparent to the end hosts. In either case, the marking engine maintains local state that includes the target rate requested for a connection or a group of connections. It passively monitors the throughput of a connection (or the aggregated throughput of connections) and adjusts packet marking in order to achieve the target rate by the user. The overall framework can provide simple service differentiation, without the risk of congestion collapse.

In terms of approach, Feng et al.'s work is the closest to this thesis. We both focus on mechanisms in routers and end host TCPs, even though the services we try to achieve are different. Feng's work focuses on providing minimum network bandwidth guarantees, and we focus on providing an average throughput for TCP traffic. Feng's proposed changes to TCP are complex and it is not clear whether they can actually be deployed.

2.5.3 IETF Standardization Efforts

Research work on DiffServ started to appear in late 1997. By early 1998, a working group for Differentiated Services (DiffServ WG) was chartered in IETF and active standardization process was in progress. The DiffServ WG attempts to standardize the use of Type of Service (TOS) byte in both IPv4 and IPv6 headers.

There are two Per-Hop Behavior (PHB)—behaviors defined for interior routers—groups currently defined in DiffServ: the Assured Forwarding PHB group [28] and the Expedited Forwarding PHB group [32]. Each PHB group is allocated three bits of the DiffServ field, or six out of eight bits in TOS byte. The other two bits in the TOS byte are currently being

considered for Explicit Congestion Notification (ECN) mechanism [22].

2.5.3.1 Assured Forwarding PHB

Assured Forwarding (AF) PHB is a means for a DiffServ domain to offer different levels of forwarding assurances for IP packets received from a customer of a DiffServ domain. Four AF classes are defined, where each AF class in each DiffServ node is allocated a certain amount of forwarding resources (buffer space and bandwidth). IP packets that wish to use the services provided by the AF PHB group are assigned by the customer or the provider of a DiffServ domain into one or more of these AF classes according to the services that the customer has subscribed to.

Within each AF class, IP packets are marked with one of three possible drop precedence values. In case of congestion, the drop precedence of a packet determines the relative importance of the packet within the AF class. A congested DiffServ node tries to protect packets with a lower drop precedence value from being lost by preferably discarding packets with a higher drop precedence value.

In a DiffServ node, the level of forwarding assurance of an IP packet thus depends on (1) how much forwarding resources has been allocated to the AF class that the packet belongs to, (2) what the current load of the AF class is, and in case of congestion within the class, (3) what the drop precedence of the packet is. The recommended interior router mechanism to implement AF PHB is an algorithm similar to RIO, presented in Chapter 3.

2.5.3.2 Expedited Forwarding PHB

In contrast, the Expedited Forwarding (EF) PHB group is designed to build a low loss, low latency, low jitter, assured bandwidth, end-to-end service through a DiffServ domain. Such a service appears to the endpoints like a point-to-point connection or a “virtual leased line”.

In [32], Jacobson, et. al state that a service that ensures no queues for some aggregate is equivalent to bounding rates such that, at every transit node, the aggregate’s maximum arrival rate is less than that aggregate’s minimum departure rate. Thus, in the interior routers,

the queuing scheduling should be such that it guarantees that the aggregate has a well-defined minimum departure rate, independent of the dynamic state of the router. Often, the interior router implements such guarantees by a priority queue, which allows unlimited preemption of other traffic if necessary. At the edge routers, the traffic conditioners should ensure that arrival rate of an aggregate at any interior router is always less than that router's configured minimum departure rate.

2.5.4 LIRA and SCORE Network

In [61], Stoica and Zhang propose an alternative Assured Forwarding service, LIRA (Location Independent Resource Accounting). LIRA does not provide absolute bandwidth profiles, but rather, it defines service profiles in units of resources tokens. The number of resource tokens charged for each "in profile" packet is a dynamic function of the path it traverses and the congestion level. The assessment of congestion level of a link is through a utility function. The authors leverage the existing routing infrastructure to distribute the path costs to edge routers. Since such costs are dynamically generated, reflecting the congestion level along the path, the costs can also be used to design dynamic routing and load balancing. Defining service profiles in terms of resource tokens allows more dynamic and flexible network control algorithms that can simultaneously achieve high utilization and ensure high probability delivery for *in profile* packets. In LIRA, Stoica and Zhang demonstrate how the routing subsystem can be integrated with packet delivery to achieve both high network utilization and service assurances.

In [62], Stoica and Zhang propose the Scalable Core (SCORE) architecture, in which only edge routers perform per-flow management while core routers do not. They attempt to use SCORE to provide end-to-end per flow delay and bandwidth guarantees as defined in IntServ, but without per flow management. Thus, they can have the best of both worlds, i.e., providing services as powerful as those defined in IntServ, while utilizing algorithms as scalable and robust as those used in stateless network as DiffServ.

Current IntServ solutions assume a stateful network in which two types of per flow

state are needed: *forwarding state*, which is used by the forwarding engine to ensure fixed path forwarding, and *QoS state*, which is used by both the admission control module in the control plane and the classifier and scheduler in the data plane. In [62], Stoica and Zhang propose two algorithms for providing QoS state, one to schedule packets, and the other to perform admission control. The primary technique is called Dynamic Packet State (DPS). Each packet carries in its header some state that is initialized by the ingress router. Core routers process each incoming packet based on the state carried in the packet's header, updating both its internal state and the state in the packet's header before forwarding to the next hop. Therefore, DPS is essentially a *synchronizing* and *coordinating* mechanism piggybacked on the packet itself. Since such mechanism must traverse the same path as data payload, distributed algorithms can be designed to approximate the behavior of a broad class of stateful networks using networks in which core routers do not maintain per flow state.

In terms of actual implementation, Stoica and Zhang propose to use the fragmentation field of IPv4 header (13 bits) as well as four bits from the TOS byte to encode the state information. They have implemented an testbed to demonstrate such algorithms are possible.

SCORE pushes the idea of DiffServ further in that the packet itself carries out the task of a traditional signaling protocol by synchronizing and coordinating with routers.

Chapter 3

Mechanisms

This chapter presents mechanisms that implement the Assured Forwarding model. Although the specific mechanisms can be seen as modifications to the existing congestion control mechanisms in both routers and end hosts, we also provide a framework to think about other possible designs of bandwidth allocation schemes. We first describe the existing congestion control mechanisms in routers and end host TCP, in section 3.1. In the next three sections, we describe three mechanisms: 1) for interior routers, we propose RIO, a probabilistic early dropping algorithm that can create preferential treatment of packets in different classes; 2) for edge routers, we describe TSW, a rate estimator and a probabilistic tagging algorithm for marking packets; and 3) for end host TCP, we propose TCP-DiffServ, a collection of three mechanisms to make TCP robust, fair, and meet the specified SLA. Finally, in Section 3.5, we discuss our design choices.

3.1 Congestion Control in the Current Internet

3.1.1 Network Congestion

Network congestion has long been an active research topic. Fundamentally, network congestion exists because of a mismatch in either memory sizes, processing speed, or link

bandwidth of network devices [34], and it becomes more pronounced and frequent as networks become more heterogeneous. Early debate in the 1980s centered around a few issues, namely, whether congestion control should take the form of a prior-reservation or a walk-in; whether it should be rate-based [63, 2, 64, 60, 66, 5] or window-based; whether the control should be done in the routers [43, 13] or at the end hosts [31, 33, 38, 55]; and whether an open-loop or a close-loop mechanism would suffice. In [35], Jain presents an objective comparison of the alternatives and argues that a complete congestion management strategy should include several congestion controls and avoidance schemes that work at different levels of protocols and handle congestion of varying duration.

The Internet suffered a congestion collapse in the late 1980s because of a lack of congestion control mechanisms. In 1988, Jacobson [31] proposed a collection of practical congestion control mechanisms for TCP. The mechanisms significantly improved TCP's performance and alleviated network congestion. The particular mechanisms have since been widely adopted and implemented. Later research work in congestion control involves mechanisms on the router side [27, 43, 41], as well as improvements to TCP's congestion control [30, 15, 23, 5], but the adoption and deployment of these is slow.

We are interested in congestion control and avoidance mechanisms because these are the mechanisms by which Internet bandwidth is allocated. In the current Internet, congestion control and avoidance mechanisms include those in end host TCP and those in routers. In the following two subsections, we describe router mechanisms and TCP mechanisms, respectively.

3.1.2 Congestion Avoidance Mechanisms in Routers

Most of today's Internet routers use a drop-tail algorithm to manage incoming their queues. A drop-tail queue uses FIFO scheduling and drops packets when it runs out of buffer space. The problem with a drop-tail queue is that it doesn't do any congestion avoidance: it only drops packets when the congestion is already severe. This tends to create a phenomenon called *global synchronization*, in which many TCP connections tend to synchronize in their

increase and decrease phases when a drop-tail queue is used in routers[1]. This is because when the queue is full, packets from all connections are dropped together. The connections all try to recover packets, and after a period of silence, start again, so the queues at congested routers will move between full and empty. Global synchronization can lead to low network utilization. One proposal to solve this problem is to use a randomized drop queuing discipline[43], instead of the drop-tail queuing.

The most effective detection of congestion occurs in routers because routers can reliably distinguish between propagation delay and queuing delay, as well as between transient congestion and persistent congestion. There have been a few proposals on putting congestion detection and avoidance mechanisms into routers. The DECbit congestion avoidance scheme [37] is an early example of congestion detection in the router. DECbit routers give explicit feedback when the average queue size exceeds a certain threshold. Another example, which combines the idea of DECbit and randomized drop, is RED, or Random Early Drop gateway, proposed by Sally Floyd and Van Jacobson. The RED algorithm can detect incipient congestion, avoid global synchronization, and keep the overall throughput high while maintaining a small average queue size.

The RED algorithm operates as follows. First, it computes the average queue size (avg). The average queue size is calculated using a low-pass filter of instantaneous queue size ($inst_{queue}$), once upon every packet arrival (Formula 3.1).

$$avg = avg * w_q + inst_{queue} * (1 - w_q); \tag{3.1}$$

where w_q is a configurable parameter.

Then, it uses the calculated avg to determine whether an arriving packet should be dropped. If the average queue size is below a minimum threshold (min_{th}), the arriving packet is not dropped. When the average queue size exceeds min_{th} but is less than a maximum threshold (max_{th}), RED drops the arriving packet with a certain probability. This probability is calculated as a function of the average queue size (avg) and a parameter

[1]Though this shouldn't be a problem if the TCP connections have very different round trip times.

P_{max}, as in Formula 3.2.

$$P_{drop} = (avg - min_{th})/(max_{th} - min_{th}) * P_{max}; \tag{3.2}$$

The closer avg is to max_{th}, the higher the dropping probability. When the average queue size exceeds the maximum threshold, max_{th}, RED drops all arriving packets with probability 1.

RED allows routers to tolerate transient bursts but to detect incipient and persistent congestion. By using a low-pass filter to calculate avg, RED can filter out transient bursts and temporary congestion. Persistent congestion in the router is reflected by a high average queue size, which results in a high dropping probability. Thus, RED can detect persistent congestion.

RED also allows routers to detect congestion early. This is done by starting to drop packets when the average queue size exceeds a minimum threshold, not when the router is out of buffer space. This allows a grace period for congestion detection.

Finally, RED drops packets randomly so end host connections (especially TCP connections) can back off at different times. This mechanism avoids the global synchronization effect we mentioned earlier. Therefore, RED is able to keep the overall throughput high while maintaining a small average queue length, and tolerate transient congestion.

3.1.3 Congestion Phases in RED

A RED router is configured with the following parameters: min_{th}, max_{th}, w_q, and P_{max}. It works as illustrated in Figure 3.1. The X axis is avg, the average queue size. The Y axis is the probability of dropping an arriving packet. There are three phases in RED: normal operation, congestion avoidance, and congestion control. The three phases are defined by the average queue size avg in the range of [0, min_{th}), [min_{th}, max_{th}), and [max_{th}, ∞), respectively.

- Normal operation (phase 1): avg is between [0, min_{th})

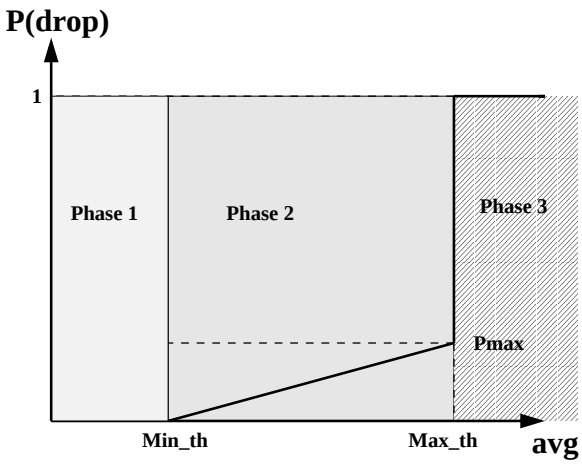


Figure 3.1: RED Algorithm

In this phase, the router is operating with no congestion; the amount of arriving packets is well below the router's capacity. The router sees very short instantaneous queues and a very small average queue. No packets are dropped.

- Congestion avoidance (phase 2): avg is between [min_{th}, max_{th})

  In this phase, the router observes that the queue is gradually building up. It starts dropping packets as a congestion signal to end hosts. The packets are dropped early and randomly to avoid global synchronization. If the end hosts implement the appropriate congestion control algorithm[31], they will back off and congestion at the router will dissipate. This phase is a buffering phase with a relatively steady dropping rate.

- Congestion control (phase 3): avg is between [max_{th}, ∞)

  In this phase, the router is congested. The router drops all arriving packets in the hope to control congestion. The router degrades into a drop-tail router, which has a few undesirable consequences. A drop-tail router is more likely to drop multiple packets from the same TCP connection, and if the TCP connection fails to recover those packets using its retransmission mechanism, it will go into a timeout period

and suffer a long period of silence. A drop-tail queuing mechanisms can also lead to global synchronization effect. This is the operating phase that a router should avoid.

3.1.4 Congestion Control and Avoidance Mechanisms in TCP

TCP is the only transport protocol that implements congestion control and avoidance mechanisms. The mechanisms were introduced in the late 1980s by Van Jacobson [31]. Immediately preceding this time, the Internet was suffering from a congestion collapse: end hosts would send their packets into the Internet and quickly congest the network. The routers would drop packets once they run out of buffers. The end hosts would time out and retransmit packets again, resulting in even more severe congestion.

The mechanism proposed by Jacobson was to have TCP probe the available bandwidth available in the network and pace its injection of new packets into the network by the arrival of acknowledgment packets. However, determining the available capacity in the network is not an easy task. TCP does this by increasing the number of packets it injects into the network until a packet is dropped—the underlying assumption is that a packet is always dropped because network congestion has occurred—at which point, TCP determines that correct operating point should be one half of the number of outstanding packets[2], and reduce its sending rate by one half. This is often called the "linear increase, multiplicative decrease" algorithm, first proposed by Raj Jain, et al. in [38].

In Jacobson's mechanism, each sending TCP maintains a new state variable called *congestion window*, or *cwnd*. *cwnd* is dynamically adjusted to reflect the number of packets a given TCP can send into the network. There are two phases in TCP's congestion window adjustment algorithm: exponential increase phase (accomplished by a mechanism called "Slow-Start"), and linear increase phase (accomplished by a mechanism called "Congestion Avoidance"). During the exponential increase phase, *cwnd* is doubled every round trip time (RTT). During the linear increase phase, *cwnd* is increased linearly, or by one packet

[2]Assuming the buffer space in the congested router equals the number of packets currently in the "pipe", then cutting down the TCP window size by one half will match TCP's operating point to the number of packets in the pipe, or reduce the queue in the router to zero.

every round trip time.

TCP maintains another new state variable called "Slow-Start threshold" or *ssthresh*, which marks the turning point when TCP switches from the exponentially increase phase to the linear increase phase. When the *cwnd* is less than *ssthresh*, TCP uses the Slow-Start mechanism to exponentially increase its *cwnd*; once the *cwnd* goes beyond *ssthresh*, TCP switches to Congestion-Avoidance, linearly increasing its *cwnd*. *ssthresh* is typically pre-configured to be 64Kbps and is set to be one half of *cwnd*, after a packet drop. Intuitively, *ssthresh* reflects an estimation of the equilibrium operating point of the TCP connection.

The ideal operating region of TCP uses Congestion Avoidance mechanism only and stays in a linear-increase phase until a packet drop, at which point TCP readjusts both *cwnd* and *ssthresh* to be one half of what *cwnd* was prior to the packet drop. Since the new value of *cwnd* is equal to the new *ssthresh*, TCP stays in the linear-increase phase and uses the congestion-avoidance mechanism again. Slow-start is only evoked when a TCP initially starts and does not know its ideal operating point, or after TCP has a timeout. However, it is often the case that a TCP loses a number of packets in the same congestion window when congestion occurs, in which case, TCP has to recover through a timeout mechanism. If a timeout occurs, TCP is usually silent for a while before it can send packets again. Since TCP doesn't have a very accurate timer, this silent period can be long. If this occurs, TCP's achieved throughput can be unpredictable.

Figure 3.2 illustrates four TCP epochs of linear increase and packet drops. Each "x" in the figure indicates a packet drop, after which, the *cwnd* and *ssthresh* is reduced to one half of the value of *cwnd*.

3.2 RIO Dropping Algorithm

As discussed in Section 2.4.5, there are a few possible approaches in designing DiffServ mechanisms and we start with design 2. In the next few sections, we will describe our experiences in designing an interior-router mechanism (RIO), an edge-router mechanism (TSW)

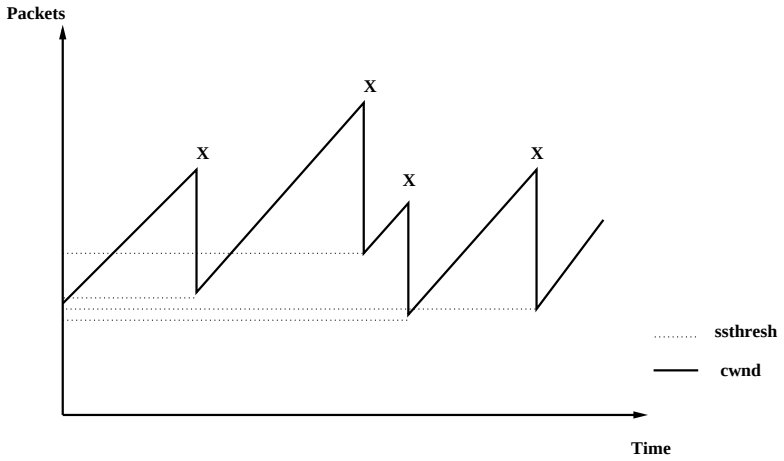


Figure 3.2: TCP Operating Epochs Using Congestion Avoidance Mechanism

and TCP mechanisms, then we conclude by revisiting the three designs in Section 3.5.

In addition to detecting congestion and manage congestion, the algorithm in interior routers has to have two additional attributes: *versatility* and *differentiation*. In other words, the algorithm should create differentiated treatment of packets regardless the traffic mix and network conditions. For example, the network might be very well-provisioned, and there is a low utilization of network at any instant. Alternatively, the network might be heavily congested but most of the packets are OUT packets, or the network is under-provisioned, and a major portion of the packets causing long queues are in fact IN packets. In all cases, the router algorithm has to perform well.

We designed a preferential dropping algorithm, RIO, based on Random Early Drop (RED) algorithm. RIO stands for Random Early Drop (RED) with IN/OUT. We start our design with RED algorithm because RED can detect congestion early, manage congestion, and avoid global synchronization.

RIO uses twin RED algorithms, one for IN packets and the other for OUT packets, and gives preferential treatment to IN packets. RIO retains all attributes of RED algorithm: detecting incipient congestion, avoiding global synchronization and keeping the overall throughput high while maintaining a small average queue size. Additionally, RIO creates service discrimination to different classes of packets.

3.2.1 Twin Algorithms in RIO

RIO uses the same early drop mechanism as RED, but is configured with two sets of parameters, one for calculating the probability of dropping IN packets, and the other for calculating the probability of dropping OUT packets. RIO works as follows. When a packet arrives, RIO first exams whether it is an IN packet or an OUT packet. If it is an IN packet, RIO calculates avg_in, the average queue size for IN packets. Then RIO uses the same algorithm RED uses to determine whether to drop this IN packet. There are two configurable thresholds: the minimum threshold min_in for IN packets and the maximum threshold max_in for IN packets. If avg_in is less than min_in, then the packet is not dropped; if avg_in is between two thresholds min_in and max_in, then the packet is dropped with a probability P_{drop_in} calculated based on Formula 3.3; if avg_in is beyond max_in, then the packet is dropped with a probability of 1.

$$P_{drop\_in} = (avg\_in - min\_in)/(max\_in - min\_in) * P_{max\_in} \tag{3.3}$$

Similarly, if the arriving packet is an OUT packet, RIO determines whether to drop this packet with the following algorithm. It first calculates avg_total, the average *total* queue size for *all* arriving packets (both IN and OUT)[3]. There are two configurable thresholds: the minimum threshold min_out for OUT packets and the maximum threshold max_out for OUT packets. If avg_total is less than min_out, then the packet is not dropped; if avg_total is between two thresholds min_out and max_out, then the packet is dropped with a probability P_{drop_out} calculated based on Formula 3.4; if avg_total is beyond max_out, then the packet is dropped with a probability of 1.

$$P_{drop\_out} = (avg\_total - min\_out)/(max\_out - min\_out) * P_{max\_out} \tag{3.4}$$

Figure 3.3 contains the pseudo code for RIO algorithm.

[3]We consider both IN and OUT packets when calculating the dropping probability of OUT packets. This is a subtlety which we discuss at the end of this section.

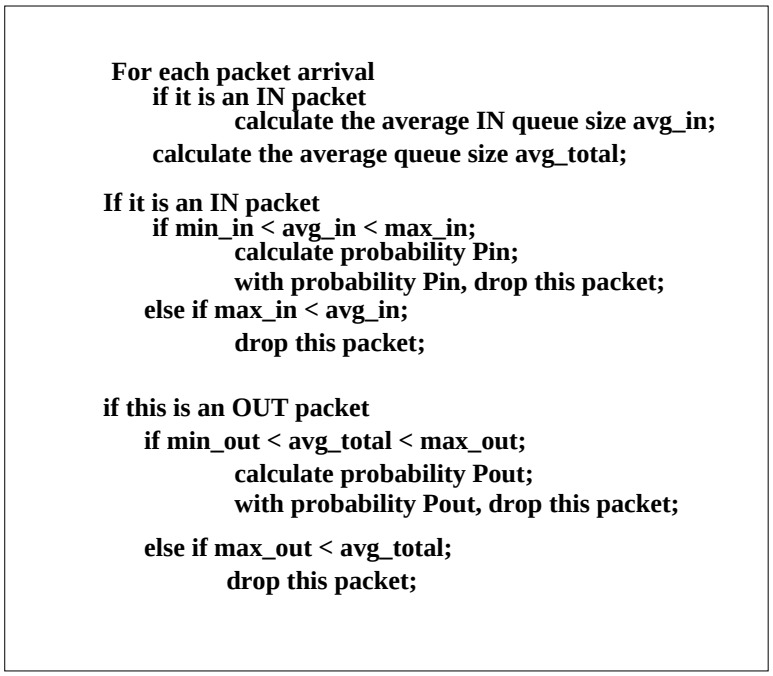


Figure 3.3: RIO Algorithm

Figure 3.4 shows the twin algorithms in RIO.

3.2.2 Designing RIO

We didn't arrive at this particular design easily. There were a few other attempts, mostly focusing on what metric (the X axes in the twin algorithms) we should use to decide the congestion state of the router. In RED, the value of avg_q is particularly important because this estimate should accurately reflect the congestion state of the router, and in turn, determine the dropping probability. In RIO, we have to determine the proper metrics to use for both IN packets and OUT packets. Determining the metric for IN packets is relatively straightforward because IN packets represent the provisioned traffic, so the network service providers should know the amount of IN packets to expect. Therefore, the metric for IN packets is the averaged queue size for IN packets, i.e., the weighted average of instantaneous queue sizes of IN packets *only*. However, determining that for the OUT packets is

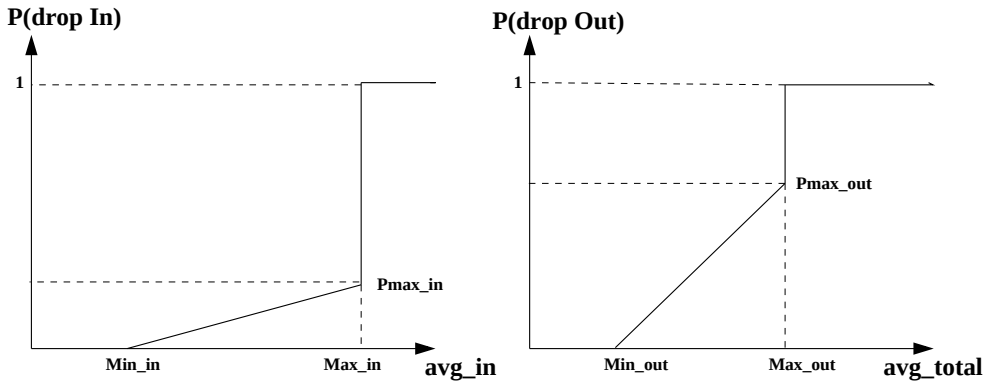


Figure 3.4: Twin Algorithms in RIO

not that easy. There are three possible choices:

- Using avg_out_q

  We could use a similar metric for OUT packets, namely, the weighted average of instantaneous queue sizes of OUT packets *only*. This design has a problem because the OUT packets represent the amount of opportunistic packets not provisioned for, so the ISP has no idea about the appropriate amount of OUT packets to expect. For example, a lot of arriving OUT packets may be due to the fact that there are very few IN packets in the network so the sending flows can send beyond their service profiles, thus generating a lot of OUT packets; or this is because some non-conforming flows are sending a lot of OUT packets to congest the network. In these two cases, the congestion state of the network are very different. However, by examining avg_out_q alone, the routers do not have enough information to determine the proper congestion state.

- Using avg_in_q

  We could use avg_in_q to determine the proper rate to drop OUT packets. At first, this design seems counter-intuitive. However, it makes sense with the following rationale. The amount of IN packets is what the network has provisioned for. If the avg_in_q is small, then it means that the routers should have *more* buffers available

for OUT packets. Conversely, if avg_in_q is large, then it means that the routers should have *less* buffers for OUT packets. In other words, the average queue size for IN packets can determine how much room the router has for OUT packets. However, the problem with this particular design is that it works well when avg_in_q is large; when there are very few IN packets in the network, using this metric can lead to a very lenient treatment of OUT packets, resulting a long delay for arriving packets and eventually congestion.

- Using avg_total

  Finally, we come up with a design that has the benefits of the above two designs. We use avg_total, or the average queue size for *both* IN and OUT packets as the metric. For IN packets, we still use avg_in_q. This way, the dropping probability of IN packets is determined *only* by the amount of IN packets in the queue, but that for OUT packets is determined by all arriving packets, regardless what kind of traffic mix. This way, the dropping algorithm is versatile, and can maintain short queue length and high throughput no matter what kind of traffic mix the arriving packets have.

3.2.3 Congestion Phases in RIO

Figure 3.5 illustrates RIO graphically. RIO divides up the router's congestion state into five phases[4]:

- Congestion free phase (phase 1)

  The router is operating with no congestion: the amount of IN and OUT packets are well below its capacity. It sees very short instantaneous queues and a very small average queue. No packets are dropped.

[4]We merged the two pictures in the previous graph into one. However, please note that the X-axes of the two previous pictures are avg_in and avg_total respectively. In this graph, the X-axis is avg_total, the average queue size for both IN and OUT packets.

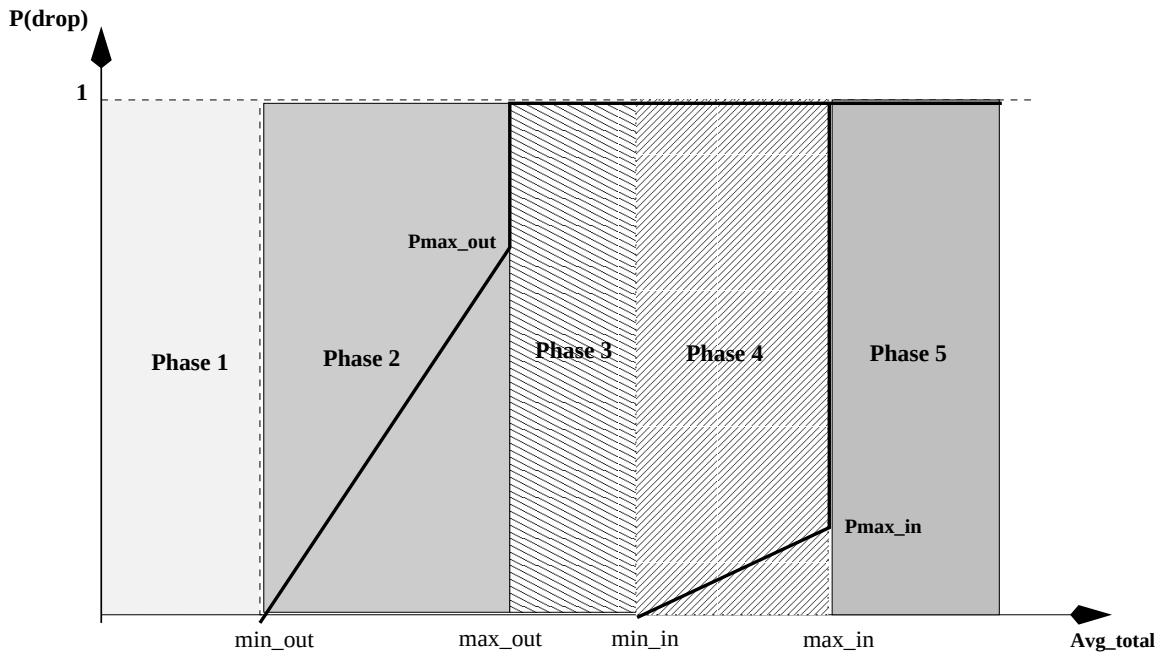


Figure 3.5: Phases in RIO Algorithms

- Congestion sensitive phase (phase 2)

  The router suspects that the queue might be building up so it starts dropping packets as congestion signals, however, it drops OUT packets only. During this phase, the IN packets only see short instantaneous queues and they are never dropped.

- Congestion tolerance phase (phase 3)

  All OUT packets are dropped, but no IN packets are dropped. This is the buffering phase for the IN packets before routers start dropping any IN packets. During this phase, the average queue size is building up with the arriving IN packets. In practice, max_out and min_in can be the same, then, this phase is eliminated.

- Congestion alarm phase (phase 4)

  All OUT packets are dropped. In addition, the router starts to drop IN packets as a means to keep the queue from overflowing. This is an undesirable phase for ISP because it compromises the ISP's SLAs by dropping IN packets.

- Congestion control phase (phase 5)

  The system is congested. The router drops both IN and OUT packets with probability 1. In this phase, the router has switched its primary goal from creating differentiations among two types of packets to congestion control. The router degrades into a drop-tail router. If the router constantly operates in this phase, it is a sure sign that either the system is under-provisioned or the parameters of traffic conditioners/RIO are not set correctly.

Phases 2 and 3 are the ideal operating phases for a router because both the instantaneous and average queue sizes are short, but the network link is highly utilized. Only OUT packets are dropped, which doesn't compromise an ISP's service profiles to its customers. When operating in phase 1, the router sees little congestion but the link capacity is not well utilized. When the input traffic is predictable, an ISP should try to configure its system to avoid phases 4 and 5, and operate mostly in phases 1, 2 and 3.

3.2.4 Creating Differentiation with RIO

The discrimination against OUT packets in RIO is created by carefully choosing the parameters (min_in, max_in, P_{max_in}), and (min_out, max_out, P_{max_out}). A RIO router is more aggressive in dropping OUT packets on three accounts: first, it drops OUT packets much earlier than it drops IN packets, this is done by choosing min_out smaller than min_in. Second, in the congestion avoidance phase, RIO drops OUT packets with a higher probability, by setting P_{max_out} higher than P_{max_in}. Third, by choosing max_out much smaller than max_in, RIO goes into congestion control phase for the OUT packets much earlier than for the IN packet. In essence, RIO drops OUT packets first when it detects incipient congestion, and drops all OUT packets if the congestion persists. Only as a last resort, occurring when the router is flooded with IN packets, does it drop IN packets in the hope of controlling congestion. In a well-provisioned network, this should never happen. When a router is consistently operating in a congestion control phase by dropping IN

packets, this is a clear indication that the network is under provisioned.

We had some heuristics in choosing the parameters for RIO for our simulation experiments. Usually, we decide the maximum tolerable delay for arriving packets before the router starts to drop OUT packets, and this would set the min_out. For example, if we decide that when arriving packets have an average queuing delay of 5ms, it is time to start dropping OUT packets, then the min_out (in bytes) is set to be the multiple of 5ms and the link bandwidth (bytes/sec). Similarly, we determine the maximum delay tolerable for all arriving packets before the router drops all arriving packets. For example, if we decide that when an arriving packet sees a 20ms of queuing delay this router is in a heavily congested state and all arriving packets have to be dropped, then, max_in is set to be the multiple of 20ms and the link bandwidth. Similar heuristics are used for max_out and min_in. We usually use min_in to be at least one half of max_in and min_out to be at least one half of max_out. The respective P_{max} for IN and OUT are chosen to be different enough to create strong discrimination between the two types of packets. In our simulations, we set P_{max_in} to be 0.02 and P_{max_out} to be 0.5.

A conservative choice of buffer size of a router is at least the number of packets in the *pipe*, i.e., the multiple of link speed and the longest round trip time of connections going through this router. For example, if the link speed is 1.5Mbps, and the RTT of the longest connection going through this router is 100ms, then the buffer size is 0.15M bits.

3.3 Tagging Algorithms

In this section, we discuss the design of tagging algorithms (or taggers). First, we describe the desired attributes of an ideal tagging algorithm. Instead of evaluating a tagging algorithm for all kinds of traffic, we focus on long-lived TCP traffic and describe our experiences in designing a tagger for such traffic. We describe a few previous attempts and the difficulties we run into in each case. Finally, we describe a tagging algorithm called Time Sliding Window (TSW), which has overcome those difficulties. We use TSW tagger in our

subsequent simulation experiments.

3.3.1 Ideal Algorithms

In our design, taggers reside in edge routers and should be able to work with *any* type of traffic coming from the end hosts. A tagger does not *actively* control the speed and spacing of end host traffic, but only *passively* monitors and tags the end host traffic. The flow's behavior is therefore determined by the combined effect of its tagger and the dropping algorithms in the network. We believe an ideal tagger should have the following attributes:

- Versatility

  The ideal tagger should be able to work with all types of traffic, whether it is TCP, UDP, bursty, long-lived or interactive traffic.

- Long-term Perspective

  In designing different taggers, we are less concerned about the particular behavior of a flow over a short period of time, for example, during one round trip time. Rather, we are concerned about eliciting the correct behavior of a flow—meeting the service profile—over a long period of time, e.g., minutes, because this is the time granularity that ISPs are concerned about. That is, if we have a number of taggers to compare and evaluate, the determining criteria is that whether this tagging algorithm has affected the behaviors of a sending flow to achieve the target rate in SLA over a long period of time.

  This particular observation has two implications. The tagger should be able to 1) induce a reduction in a flow's speed by tagging if the flow has been sending above its target rate. The more a flow exceeds the target rate, the more this flow should scale back during the next period of time; 2) tolerate a burst of packets after the flow has been sending below its target rate. These two tagging behaviors are complementary to each other, and they both require the ideal tagger to have some memory of the past

history. We define the first kind of memory as "negative memory", i.e., the tagger remembers that the flow has sent beyond its target rate and should be clamped down. Similarly, we define the second type of memory "positive memory", i.e., a memory that tolerates a bursts of packets from the flow.

3.3.2 Tagging TCP Traffic

In the following, we focus on a particular type of traffic: long-lived TCP traffic. We describe the idiosyncrasies of a tagger for such kind of TCP traffic. We chose TCP traffic because it represents about 97% of entire Internet traffic. Most of file transfers on the Internet are long-lived TCP traffic. However, while TCP's behaviors are well understood, the effect of a tagger on such traffic is not. Additionally, evaluating the effect of a tagger on long-lived TCP traffic is relatively straightforward. Since a long-lived TCP traffic flow always has packets to send, a connection's behavior depends only on the combined effect of tagging and dropping algorithms, and not on upper-layer applications. If a flow doesn't reach its specified targeted rate, we can safely conclude that it is because the tagging and dropping algorithms do not work well, not because the upper-layer applications do not have packets to send. Therefore, the evaluating criteria is simple: we look at the time average of a long-lived TCP traffic after it has reached a stable state, and see how close this particular flow comes to its target rate.

The pitfall in choosing on long-lived TCP traffic is that this might not be representative of TCP traffic on the Internet. We think majority of the TCP traffic are in fact short-lived, transactional TCP transfers, given the popularity of the Web.

In previous sections, we have described how TCP adjusts its sending rate using its congestion control and avoidance algorithms, and it usually keeps a sawtooth behavior. In a DiffServ system, after having both taggers and droppers in place, the ideal behavior of a long-lived TCP connection trying to achieve an average target rate R_t could look like what's depicted in Figure 3.6.

In Figure 3.6, the X axis is time, and the Y axis is the sending rate of a TCP connection.

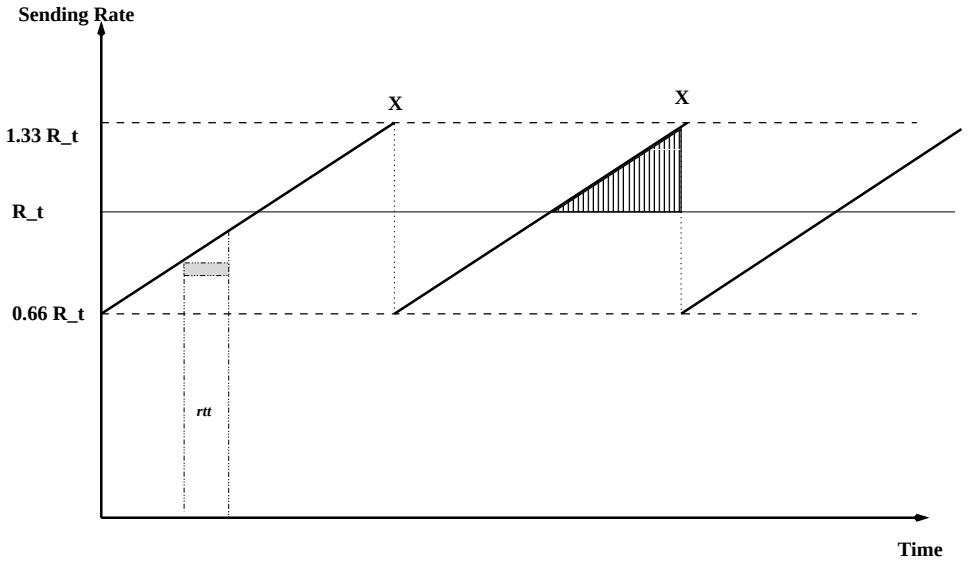


Figure 3.6: Ideal TCP Operating Epochs

At any point, the thick, dark line respects the sending rate of a TCP connection. Each x in the figure symbolizes a packet drop. After a packet drop, the TCP connection cuts its rate down by one half and starts increasing its rate again. The time between two packet drops is an epoch. In Figure 3.6, there are three epochs.

There are three things worth noting:

1. TCP keeps consistent sawtooth swings between $1.33R_t$ and $0.66R_t$.

2. Once TCP has gone beyond $1.33R_t$, part of its traffic is marked as OUT.

3. The OUT packets in TCP will eventually trigger packet drops or congestion indications from the network, which will lead TCP to cut it sending rate by half, and back down to $0.66R_t$.

The choice of $1.33R_t$ and $0.66R_t$, or the high sending rate when TCP traffic is marked as OUT and the low sending rate TCP is reduced to, is not incidental. Let H denote the high sending rate, and L denote the low sending rate, and they have to meet the following two equations:

$$(H + L)/2 = R_t \tag{3.5}$$

$$H/2 = L \tag{3.6}$$

Equation 3.5 is because a TCP connection swings between the two sending rates should achieve an average of R_t over time; Equation 3.6 is because TCP's *multiplicative decrease* window reduction algorithm always cuts the sending rate by one half[5]. Solving the above two equations, we will get the high sending rate to be $1.33R_t$, and the slow sending rate to be $0.66R_t$.

Ideally, packet drops happen at the point when the TCP connection has just passed the $1.33R_t$ threshold, and the tagging algorithm has started tagging some of the packets as OUT. In actuality, this is hardly the case. For one, there will be a feedback delay between the time when a packet is tagged and when a congestion signal is received, which is at least a round trip time. Secondly, the sending TCP might suffer from packet drops (congestion signals) before it reaches the $1.33R_t$ point.

Let's calculate the time it would take for a TCP connection to increase from $0.66R_t$ to $1.33R_t$, or the *period* of the TCP epochs. We assume that TCP stays in the linear increase phase, then each round trip time (RTT), the connection opens up its window by one packet, or increases its sending rate by $1pkt/rtt$ (bytes/sec). In Figure 3.6, the lightly shaded box depicts the one packet increase in TCP's window. The width of box is one RTT time because TCP increases its congestion window by this packet during one RTT time. Therefore, the height of the box is the rate increase for the connection within this RTT time, which is $1pkt/RTT$.

Therefore, to increase its sending rate from $0.66R_t$ to $1.33R_t$, a connection would have to take

$$\frac{(1.33R_t - 0.66R_t)}{1pkt/RTT} = \frac{0.67R_t * RTT}{pkt}$$

[5]We are simplifying things here a little and ignoring the Fast Retransmit and Fast Recovery here, which would create a short gap between two consecutive epochs.

round trip times, or

$$\frac{0.67R_t * RTT^2}{pkt}$$

seconds. In other words, the period of ideal TCP epochs is a function of both the target rate R_t and the RTT of a TCP connection. In order to give a precise indication of when a TCP connection has exceeded its target rate and by how much, the tagger has to have knowledge of both the RTT and the target rate R_t, so it can accurately tag the traffic.

3.3.3 Token Bucket Tagging Algorithm

We have designed and tried a number of tagging algorithms. One possibility is a simple token bucket, which works as follows. There is a bucket of depth D (measure in bytes) and it is constantly being filled by tokens at a replenish rate of R (bytes/sec). However, the number of tokens in the bucket never exceeds D. Packets passing through the token bucket tagger will be tagged as IN if there are enough tokens in the bucket for the size of packets (bytes); if not, the packet will be tagged as OUT. For example, if the sending rate of the flow is $2R$, where R is the replenish rate of the token bucket; then it would first take D/R time for the flow to drain all the tokens already in the bucket, during which, all packets will be IN. After this point, every other packet will be an OUT packet.

One is tempted to use a simple token bucket tagging algorithm, since the replenish rate R matches the target rate R_t in an SLA well. However, when using a token bucket tagger for general traffic, we run into two problems. The first is how to set the depth of the a token bucket. Without knowing anything about the traffic, the depth D is difficult to configure. Second, the idea of a *bucket* is to have *positive memory*, i.e., if the flow has been sending below its target rate, then tokens will be accumulated in the bucket, so it will allow a burst of packets up to D after a while. However, missing in the design of a token bucket is the *negative memory*. If a flow has been sending above its target rate for a while, then the bucket will be exhausted, at this point, $\frac{R-R_t}{R_t}$ percent of the packets are tagged as OUT. However, whether the flow was twice exceeding the target rate or three times exceeding the

target rate is not recorded, or eventually, reflected in future tagging. This does not meet the second attribute of an ideal tagger.

One could argue that the design of a token bucket tagger is to specify the minimum guaranteed rate, and therefore, having a positive memory is sufficient. There are desirable attributes associated with token bucket controlled traffic that meets the requirement of having minimum guaranteed. In [47, 48], Parekh and Gallager show that if all end host sources use token buckets to pace their respective traffic, then the overall network can offer a delay bound, no matter what topological configurations the network has. However, in this case, the token bucket algorithm is used to *actively* shape and control the sending rate of end hosts, i.e., if the end host cannot receive a token, it won't be able to send. However, in our case, the tagging algorithm cannot alternate the pace or speed of a traffic flow and can only tag packets *passively*. Therefore, a flow can still send beyond the burst of D and the overshoot part should eventually be compensated for in the long run. That's why we think in designing taggers, *negative memory* is important as well.

When using a token bucket tagger for a more specific long-lived TCP traffic we are experimenting with, we could have a better assessment of the depth of a token bucket. To see this, we have to go back to Figure 3.6. When TCP is sending below its target rate R_t, it does not exhaust any of the tokens in a token bucket, so the bucket is full. Only when TCP is sending beyond R_t, is it going to use both the tokens being replenished and tokens already in the bucket. At some point, the tokens in the bucket are exhausted and then the tagger starts to tag packets as OUT. In order to keep TCP in the ideal, perfect epochs depicted in Figure 3.6, the token bucket should run out of tokens precisely at the time when TCP exceeds $1.33R_t$. This implies the depth of the token bucket should be the amount of tokens that will supply the TCP connection from R_t to $1.33R_t$ *in excess* of the R_t. This is the shaded triangle area depicted in the figure. The height of the triangle is $0.33R_t$, and the width of triangle is $0.33 * R_t * RTT^2/pktsize$, so the area of $1/2 * height * width$, measured in bytes. It should not be surprising that the depth of the bucket has to be dependent on the RTT of the connection.

However, we find that even if we have the right configuration of D, a token bucket tagger still does not work well with long-lived traffic. This is because the time when a token bucket tagger runs out of tokens does not coincide with the the time when TCP runs at $1.33R_t$ (or, when TCP is running at R_t, the token bucket is full with depth D). It is in fact very difficult to synchronize the state of the token bucket with the quite unpredictable behaviors of a TCP connection. If the TCP connection starts with a burst of packets, as normally it would during the exponential increase phase, then this burst would drain the token bucket initially, and then the token bucket would start tagging packets as OUT before TCP is sending at $1.33R_t$. Conversely, if the TCP connection has stopped sending for a while, and then starts again, it is very difficult to make sure the token bucket is full when the sending rate of TCP has reached R_t.

Our experience with a token bucket tagger has convinced us that we need to have a tagger that has both *positive memory* and *negative memory*. In other words, this is a tagger that can remember the rate that a flow has been sending in some past time frame, whether it is beyond the target rate or below the target rate, and can gradually forget the past history to reflect the new sending rate. What we also learned from using a token bucket tagger is that there are in fact two separate parts to a tagger: first part is the rate estimator, which has to keep certain amount of past history, and second is the tagging algorithm, which tags packets based on the estimated rate. The second part is relatively easy. We know the estimated sending rate, the tagging can be done probabilistically with a probability $P = \frac{R-R_t}{R_t}$, when the estimated sending rate R has exceeded the target rate R_t. So we set out to design a good rate estimator.

At first, we used a low-pass filter rate estimator, similar to that in RIO. We could use a weighted average of past sending rate and an instantaneous rate. Upon each packet arrival, we estimate an instantaneous rate, R_{inst}, and calculate the average rate using the following formula:

$$R = (1 - w) * R + w * R_{inst} \tag{3.7}$$

where R is the estimated rate. Instantaneous rate R_{inst} can be easily calculated by dividing packet size over the inter-packet arrival time.

There are two problems with this way of calculating the average sending rate. First, packets from end hosts can be bursty and arrive at the tagger back to back, so the inter-packet arrival time can be zero, which makes calculating instantaneous arrival rate difficult. Second, this way of discounting past history is biased towards fast-sending flows. Since each discounting event happens when a packet arrives, the faster a flow sends, the quicker this flow forgets the past history. Conversely, if a flow has not sent for a while, the rate estimate still uses the rate that was recorded before the silent period, no matter how long the silent period is.

After a few more experiments, we eventually settled on a simple tagging algorithm that meets the above criteria. This algorithm, called Time Sliding Window, is described in detail in the next section.

3.3.4 TSW Tagging Algorithm

The Time Sliding Window (TSW) tagging algorithm runs on the edge routers that tag packets as IN or OUT according to specific service profiles. It has two independent components: a *rate estimator* that estimates the sending rate over a certain period of time, and a *tagger* that tags packets based on the rate reported by the rate estimator. The rate estimator accommodates both the burstiness[6] and silence often observed in TCP traffic, and smoothes out its estimate to approximate the actual sending rate at the source TCP. With the estimated rate, the TSW tagger determines whether the sending host has exceeded its target rate. If the source is sending below the target rate, the tagger will tag all packets as IN; if the source is sending above the target rate, the tagger will tag those packets *in excess* of the target rate as OUT. The tagger also tags OUT packets probabilistically to reduce the likelihood of packets within a TCP window being dropped together in interior routers.

[6]The burstiness of TCP traffic can be caused by a phenomenon called "compressed acks"[68], in which acknowledgment packets arrive at the sender back to back, triggering data packets to be sent in reverse direction as a burst.

The logical relationship of the rate estimator and the tagger is shown in the following block diagram 3.7.

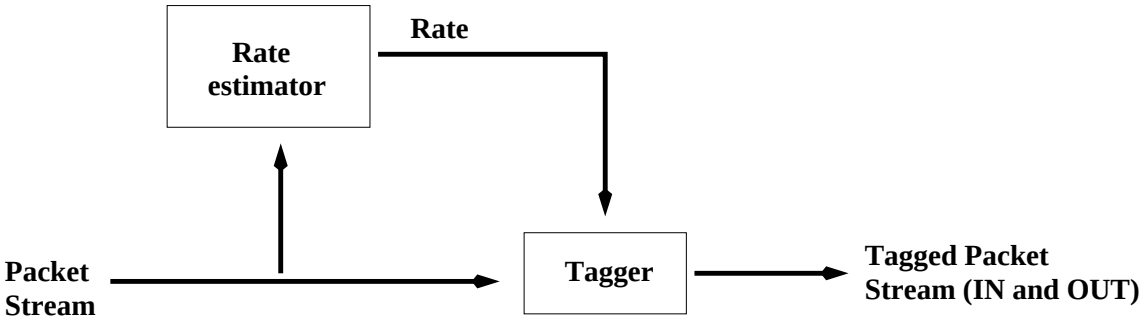


Figure 3.7: TSW Block Diagram

3.3.5 TSW Rate Estimator

As discussed in the previous section, ideally, the decaying function embodied in the rate estimator should be independent of connections' sending rates, but according to time. This is how the TSW estimator works. It estimates the sending rate upon each packet arrival and decays the past history over time. Since the decaying is according to time, the decay factor is the same regardless how fast the source is sending.

The algorithm in TSW is simple, as shown in Figure 3.8. TSW maintains three local variables: Win_length, which is measured in units of time, Avg_rate, the rate estimate upon each packet arrival, and T_front, the time of the last packet arrival. Win_length is the only parameter that needs to be configured; Avg_rate and T_front are local variables that are updated each time a packet arrives. TSW only needs to execute four lines of code for each packet. In essence, TSW remembers the amount of "history" in Win_length and decays it over time. TSW also incorporates the instantaneous arrival rate of the arriving packets.

Figure 3.9 illustrates how TSW calculates Avg_rate. There are three packet arrivals, depicted as crosses on the time line, at time t1, t2 and t3, respectively. In the figure, the lightly shaded box depicts the TSW rate estimator. The width of the box is the Win_length,

Initially:

Win_length = a constant;
Avg_rate = connection's target rate, R_t;
T_front = 0;

Upon each packet arrival, TSW updates it state variable follows:

Bytes_inTSW = Avg_rate * Win_length; (1)
New_bytes = Bytes_in_TSW + packet_size; (2)
Avg_rate = New_bytes / (now - T_front + Win_length); (3)
T_front = now; (4)

whereas, now is the time of current packet arrival, packet_size is the packet size.

Figure 3.8: TSW Rate Estimator Algorithm

along the Time axis. The height of the box is the Avg_rate; the higher the box, the higher the estimated Avg_rate. The right edge of the box is the T_front, or the time of last packet arrival. The bytes contained in the TSW is the area covered by the shaded box, or the amount of *memory* that TSW keeps about a particular flow.

TSW works as follows:

- At time t1, TSW calculates a new Avg_rate after a packet arrival. TSW is depicted by a shaded box and the height of the box is the newly recorded Avg_rate.
- At time t2, a new packet (in dark) arrives. The area of the box depicts the size of the new packet in bytes.
- The TSW rate estimator spreads both the bytes in its memory and the bytes in the new packet across the sum of its Win_length and the inter-packet arrival time. This is depicted in Figure 3.9 as both the shaded box and the dark box are spread across the time span of (Win_length + (t_2 - t_1)).
- This is the decaying phase of the TSW, where TSW remembers only the Avg_rate

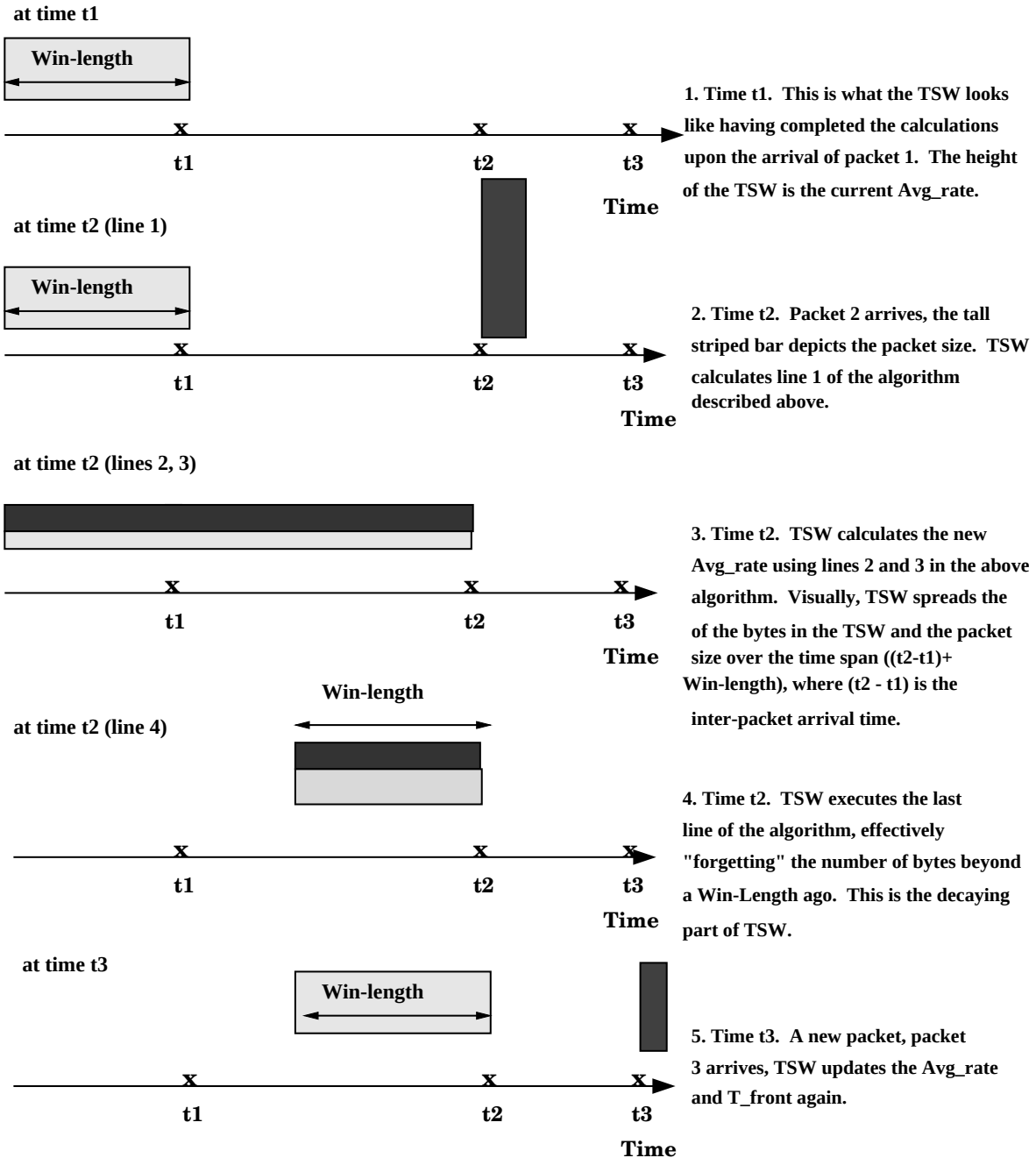


Figure 3.9: The Operations of TSW Rate Estimator

of its newly calculated rate, and forgets everything beyond a Win_length period of time. This happens when TSW resets its T_front to be the current time (line 4 in Figure 3.8). A proof of this decaying function is given below.

- A new packet arrives at time t3. TSW restarts its cycle of incorporating the new arrival rate and decaying the past history.

3.3.5.1 Decaying Function

We now present a proof for the decaying function embedded in TSW rate estimator. Similar proofs can be constructed for non-constant arrival rates as well.

Let w denote the size of each packet, let δ denote the arriving interval between packets, therefore, the sending rate is w/δ. Let L denote the Win_length of a Time Sliding Window rate estimator. Let R_0 denote the estimated rate by TSW at time t_0, and subsequently, R_i denote the estimated rate by TSW upon receiving ith packet. Then, after receiving the first packet;

$$R_1 = R_0(\frac{L}{L+\delta}) + \frac{w}{L+\delta}$$

and, after receiving the second packet;

$$R_2 = R_1(\frac{L}{L+\delta}) + \frac{w}{L+\delta}$$

or,

$$\begin{aligned} &= \left(R_0(\frac{L}{L+\delta}) + (\frac{w}{L+\delta})\right)(\frac{L}{L+\delta}) + \frac{w}{L+\delta} \\ &= R_0(\frac{L}{L+\delta})^2 + \frac{w}{L+\delta}(\frac{L}{L+\delta}) + \frac{w}{L+\delta} \end{aligned}$$

....

after receiving the nth packet;

$$R_n = R_0 \left(\frac{L}{L+\delta}\right)^n + \frac{w}{L+\delta}\left(\frac{L}{L+\delta}\right)^{n-1} + \frac{w}{L+\delta}\left(\frac{L}{L+\delta}\right)^{n-2} + ... + \frac{w}{L+\delta} \quad (3.8)$$

The last $n-1$ terms are of exponential series, so Equation 3.8 can be reduced:

$$R_n = R_0 \left(\frac{L}{L+\delta}\right)^n + \frac{w}{L+\delta} \bullet \sum_{i=0}^{n-1} \left(\frac{L}{L+\delta}\right)^i \quad (3.9)$$

Recall that $\sum_{i=0}^{n} q^i$, where $q < 1$, is $\frac{1-q^n}{1-q}$, or $\frac{1}{1-q}$, where n goes to ∞. Additionally, since $n = \frac{L}{\delta}$, the first term in Equation 3.9 becomes

$$\left(\frac{L}{L+\frac{L}{n}}\right)^n = \left(\frac{1}{1+\frac{1}{n}}\right)^n$$

Therefore, Equation 3.9 is

$$R_n = R_0 \left(\frac{1}{1+\frac{1}{n}}\right)^n + \frac{w}{L+\delta} \bullet \left(\frac{1-\left(\frac{L}{L+\delta}\right)^n}{1-\left(\frac{L}{L+\delta}\right)}\right) \quad (3.10)$$

when $n \rightarrow \infty$, or when the Win_length is usually much greater than the inter-packet arrival rate, δ, Equation 3.10 can be reduced:

$$\lim_{n\to\infty} R_n = R_0 \bullet \frac{1}{e} + \frac{w}{L+\delta} \bullet \frac{1}{\left(\frac{L+\delta-L}{L+\delta}\right)}.$$

or

$$\lim_{n\to\infty} R_n = R_0 \bullet \frac{1}{e} + \frac{w}{\delta}.$$

In summary, if Win_length is much greater than inter-packet arrival rate δ, the TSW rate estimator decays the sending rate R_0 by a factor of e for every Win_length time period, and reflects the new sending rate. In other words, the TSW rate estimator decays past history by a constant factor over a certain period of time, independent of the connection's sending rate.

It should be fairly obvious then that the TSW rate estimator has both *positive memory* and *negative memory*, because it discounts both high sending rate (bursts) and low sending rate (silence) equally over time.

3.3.5.2 Configuring the Rate Estimator

Configuring the variable Win_length depends on two factors. On one hand, since TSW keeps a long-term perspective on estimating sending rates, we want Win_length to be much bigger than the typical RTTs of a TCP flow. On the other hand, we want a value of Win_length that is small enough so that the rate estimator is sensitive to the changing rate of the connection. In our simulation experiments, we have TCP connections whose RTT range from 30ms to 150ms, and we use Win_length to be between 0.6 seconds and 1 seconds. As a rule of thumb, choose a Win_length that is an order of magnitude larger than the RTT. Subsequent experimental studies by Seddigh, Nandy, and Pieda[45] have used a Win_length value of 1 second for scenarios where the contracted traffic consisted of multiple TCP flows with different RTT values.

3.3.6 Probabilistic Tagger

We use a probabilistic tagging algorithm to mark an arriving packet as either IN or OUT based on the rate that the TSW rate estimator reports. The probabilistic tagger is configured with a *target rate*, or R_t. It marks traffic within the target rate as IN packets, and marks traffic *in excessive of* the target rate as OUT packets. Equivalently, if the estimated rate is R, the probabilistic tagger marks all arriving packets with a probability $P = (R - R_t)/R_t$, if $R > R_t$. The algorithm for the probabilistic tagging algorithm is described in Figure 3.10.

Although we present a tagging algorithm to mark two types of packets, the tagging algorithm can be easily extended to mark a few types of packets, corresponding to different drop preference within a class in Assured Forwarding (AF) PHB. In [18], we extend the tagging algorithm to mark three types of packets: red, yellow and green. The colors red, yellow and green translate into DiffServ codepoints representing drop precedence 2, 1, and

```
Estimate sending rate R;

    if R < R_t;
        mark packet as IN;
    else
        calculate P = (R - R_t) / R_t;
        with P, mark the packet as OUT;
```

Figure 3.10: TSW Probabilistic Tagging Algorithm

0 of a single AF class, respectively.

3.3.7 Discussion

Both RIO and TSW use a probabilistic function to spread out the dropping and tagging of OUT packets. This is to keep TCP operating in the more controllable Fast-Retransmit phase of its congestion control algorithm. Current TCP implementations have a mechanism called "Fast Retransmit and Fast Recovery" which can recover a couple of packets lost within a TCP's sending window. When this mechanism is evoked, TCP's behavior is robust. However, if multiple packets within the same TCP window are lost, current TCP implementations rely on a timeout mechanism to recover lost packets. A timeout can be a long period of silence. When this happens, TCP's performance is neither robust nor predictable.

If the network is heavily congested, then the preferential dropping algorithm has to drop every single OUT packets in order to control congestion (phases 2 & 3 in Figure 3.5). In this scenario, we would like TCP to stay in the Fast-Retransmit phase, therefore, the tagger should avoid tagging multiple packets within a TCP window as OUT as to reduce the probability of all of them being dropped by a congested router. Thus, in designing the two router algorithms, we incorporate a probabilistic function in both: when dropping packets, interior routers will drop randomly, with fairly regular intervals; when tagging

packets, the tagger tries to space OUT packets evenly across time. The combined effect is to decrease the likelihood of consecutive OUT packets from the same connection being dropped by a router.

It should be noted that although TSW is presented here strictly as a tagging algorithm, it can incorporate an early drop function and a discrimination mechanism as in RIO to serve as a policer. In this case, the mechanisms in RIO—early, preferential and probabilistic dropping—are duplicated in TSW; the interior routers might simply use a drop-tail algorithm or a RED algorithm to achieve the combined effect of RIO+TSW presented above.

3.4 DiffServ Mechanisms for TCP

The previous two sections have primarily focused on router mechanisms, however, the effectiveness of such schemes is limited by the impreciseness and biases in the window-based congestion control algorithms of TCP. More specifically, the rate adjustment scheme in the current Internet depends on a feedback loop completed by both TCP's congestion control algorithm and the router's congestion avoidance algorithm. Thus, by changing mechanisms in routers alone, the rate adjustment schemes are not very effective or precise in achieving the targeted SLAs.

The congestion control mechanisms in TCP have been very successful in dealing with the growth of the Internet. However, the mechanisms in the current TCP are not ideally suited for a DiffServ network in that they are not fair or robust enough to meet the specifications of SLAs very well. This section focuses on mechanisms that can be incorporated into TCP's congestion control algorithm in meeting requirements of SLAs. We first describe the problems with the current congestion control algorithm and then propose these mechanisms.

3.4.1 Problems

3.4.1.1 Bias in Window Open-Up Algorithm

During the slow start phase, TCP doubles its window each round trip time (RTT), and during the congestion avoidance phase, TCP increases its window by one packet per round trip time. To see why this particular window open-up algorithm is biased against long-RTT connections, we can simply look at TCP's algorithm during the congestion avoidance phase. In congestion control phase, TCP opens up its window by one packet (measured in bytes) per RTT (measured in seconds). Let r_i denote source i's average round trip time, including queuing delays. After each RTT seconds, TCP increases its sending rate from $cwnd/rtt$ to $(cwnd + 1)/rtt$ (bytes/sec). The longer the RTT is, the slower the TCP connection is to increase its sending rate. If two TCP connections are sending at the same rate prior to their respective drops, it would take the long-rtt connection a significantly longer time to its previous throughput than it does for a short-rtt connection. The same thing can be said for TCP during its slow-start phase.

3.4.1.2 Not SLA-aware

As discussed in section 3.1, the value of *ssthresh* reflects the perceived network available bandwidth to a TCP. In the current TCP, *ssthresh* is set using a set of mechanisms. *ssthresh* is initially set to a default value and is readjusted after each packet drop to be one half of the *cwnd* before the packet drop. A packet drop is recovered either through Fast Retransmit and Fast Recovery, or through a timeout mechanism. When a single packet is lost, the Fast Recovery and Fast Retransmit mechanism recovers the lost packet successfully and both *cwnd* and *ssthresh* are reduced to one half of *cwnd* prior to the packet drop. At that point, TCP continues to operate in the linear window increase phase with a reduced *ssthresh*. When multiple packets are dropped within a window, current implementations of TCP (Reno) usually fail to recover the lost packets and have to rely on a timeout mechanism. This is because when packets are lost, TCP (Reno) receiver generates duplicated acknowledgment

packets (duplicate acks), and the TCP sender *infers* that a packet has been lost. When multiple packets within the same congestion window are lost, the sender won't be able to put enough packets into the network to generate sufficient duplicate acks to trigger additional Fast Recovery and Fast Retransmit. Therefore, ultimately, TCP uses the timeout mechanism to recover. However, for each successive packet loss, TCP reduces its *ssthresh* by one half, so when TCP eventually recovers from packet loss via a timeout mechanism, TCP operates with a much reduced *ssthresh*.

In the DiffServ architecture, bandwidth allocation is based on SLAs. The underlying premise is that each entity is assured of its target throughput specified in its SLA when congestion is experienced, and can exceed such profiles when there is no congestion. With the knowledge of these target throughputs, the ISP is supposed to provision the network so that all service profiles are satisfied. However, since DiffServ relies on statistical multiplexing of shared resources and not strict admission control, there will still be cases when either the ISP fails to provision properly or certain routers experience incipient congestion. In the DiffServ domain, when a TCP connection loses a packet, how should *ssthresh* and *cwnd* be set? The underlying DiffServ premise implies that the ideal behavior of TCP is to reduce its sending rate when congestion is experienced, but can recover to its target throughput robustly. In other words, TCP should be SLA-aware.

3.4.1.3 A Lack of Robustness

As described in Section 3.4.1.2, the mechanisms TCP has in recovering packets—using either Fast Retransmit or Fast Recovery or a timeout mechanism—are not particularly robust. Some recently proposed changes to TCP include the use of Explicit Congestion Notification (ECN) mechanisms to improve TCP's ability to recover lost packets. ECN de-couples congestion control notification from packet drops and can be implemented in both end host TCPs and RED gateways [22]. In this proposed scheme, RED routers mark an ECN bit in a packet's header instead of dropping the packet, and TCP responds to the explicit congestion notifications instead of inferring congestion from duplicated acknowledgments. This

mechanism has the advantage of avoiding unnecessary packet drops and unnecessary delay for packets from low-bandwidth delay-sensitive TCP connections. A second advantage of the ECN mechanism is that TCP doesn't have to rely on coarse granularity of its clock to retransmit and recover packet losses. TCP-SACK, on the other hand, uses a completely different acknowledgment mechanism from the current TCP. TCP-SACK receiver generates a bitmap of packets it has received so the TCP-SACK sender can selectively retransmit packets. TCP-SACK also improves TCP's robustness.

There is a similarity in the ECN mechanism and the DiffServ mechanism. Both allow routers to mark one bit in packet headers to convey information on the data path. In ECN, this information is between routers and the end host TCP; in DiffServ, this information is between edge routers and interior routers. If we decide that the end host TCP can be modified to incorporate some additional mechanisms, then this change will complete the feedback loop between routers and end host TCP in a DiffServ architecture, just as that in the ECN mechanism. The question is whether the end host TCP can take advantage of additional DiffServ information from the routers.

3.4.2 Mechanism 1: Fair Window Open-Up Algorithm

In the DiffServ architecture, each entity (potentially at the finest granularity of a single TCP connection) is associated with an SLA that defines a target throughput rate. Although the SLA definition has not been finalized by the IETF DiffServ working group, there are two potential definitions to choose from. Each definition will in turn determine the underlying window open-up algorithm. We use the fairness index proposed in [36],

$$F = \frac{(\sum_{i=1}^{n} x_i)^2}{n(\sum_{i=1}^{n} x_i^2)}$$

where x_i is the resource allocation to the ith user. This fairness index ranges from 0 to 1, and is maximized when all users receive the same allocation. This index is k/n when k users equally share the resource, and the other $n-k$ users receive zero allocation. Examples

of possible definition of resource allocation include response time, throughput, throughput times hops, and so on [36].

In the first definition, an SLA includes both a target throughput as well as a range of RTT values within which the target throughput can be met (BW_{target}, (min_RTT, max_RTT)). The bigger the RTT value, the smaller the corresponding target throughput. For example, an SLA could be (1Mbps, (50ms, 100ms)), which says that if a connection's round trip time is between 50ms and 100ms, it should expect to receive 1Mbps of bandwidth. A network domain with similar provisioning can offer other SLAs like (3Mbps, (20ms, 50ms)) and (0.5Mbps, (100ms, 200ms)). Corresponding to this definition, the underlying TCP window increase algorithm is that TCP increases $c * rtt$ packets per round trip time, where c is a constant, chosen as a scaling factor, and RTT is TCP's round trip time estimate variable. Using this algorithm, a connection that goes through k bottleneck gateways will share $1/k$ of a bottleneck link bandwidth as a connection which goes through one bottleneck gateway. This definition will meet the criteria of fairness index when the resource allocation is defined as throughput times the number of gateways.

In the second definition, an SLA simply includes a target throughput (BW_{target}), which implies that the ISP is to assure the target throughput regardless of the round trip time of the connection. The underlying algorithm for this definition is that TCP increases $c * rtt^2$ packets per round trip time, where RTT is the TCP round trip time estimate variable. Using this algorithm, when n connections are sharing a single bottleneck gateway, the window open-up algorithm allows all connections to receive $1/n$ of the bottleneck bandwidth, regardless of their RTT. This will maximize the fairness index when the resource allocation is defined as throughput of individual connections.

The related SLA definitions and their corresponding window open-up policy and fairness criteria are tabulated in Table 3.1.

It should be noted that both alternatives to the current window algorithm of TCP still fall under the *linear increase* rule. Only that the linear increase is by c packets per RTT (the first policy), or by c packets per second (the second policy).

Table 3.1: SLAs and the corresponding TCP mechanisms to achieve fairness

	Policy 1	Policy 2
SLA	[BW_{target}, (min_RTT, max_RTT)]	BW_{target}
Definition	Throughput * # of routers	Throughput
Window Algorithm	$c * rtt$	$c * rtt^2$

3.4.2.1 Choice of c

Another way of viewing the change in TCP's linear window increase algorithm is the following: instead of increasing TCP's congestion window by one packet each round trip time, the proposed mechanism opens up *cwnd* by one packet during a certain standard unit of time. If all TCP implementations adopt such algorithm, then they will all increase their windows at the same rate regardless of their RTTs. Thus, the choice of c, which determines the value of such standard unit of time is a crucial one. For example, if c is chosen to be 100, then, the standard unit of time is implicitly set to be 100ms ($C * RTT^2$ = $100 * (0.1)^2 = 1pkt$). In other words, all TCP implementing the above proposed mechanism will be increasing their congestion windows at the same rate as a current TCP implementation with an RTT of 100ms. Essentially, this algorithm make those TCP connections with RTT less than 100ms less aggressive than the current implementations, and those with RTT greater than 100ms more aggressive than the current implementations.

Two potential problems arise from this. The first is how to choose a value that can be universally agreed upon. The technical merits of the proposed mechanism have been argued, but the ultimate choice lies in the policies by which the choice of c makes sense. One problem of choosing a relatively small c (less than 100ms, for example) is that for long-RTT connections, the new algorithm results in an effective rate increase even greater than during the slow start phase. For example, if a 1sec TCP connection uses the proposed algorithm, it means it will open up its window at the rate of one packet each 100ms, which is 10 packets each RTT. Depending on the current number of packets outstanding, this rate can be greater than that during the slow start phase, which is already very fast. This can lead

to a congestion collapse. The problem with choosing a relatively large c is that this makes TCP window increase algorithm very slow and if this algorithm is universally adopted, it might result in low utilization of link bandwidth immediately after a congestion epoch.

One possible solution is to define a set of inclusive RTT ranges, and within which, modified TCP connections will open up their window at the same rate, but each range has a different window open rate. A reasonable heuristic is that the longer RTT, the slower the window increase rate is because the longer the connection, the more resources (buffer space, or packets in the pipe) it would take. Such ranges of RTTs can be easily specified in the SLAs as the ISP will set a lower expected throughput for longer connections. The range of RTTs can be chosen to reflect actual market concerns. For example, we could define four ranges of RTTs, inclusively: (0, 50ms) for LANs and WAN range of connections; (50ms, 100ms) for intra-continental connections; (100ms, 200ms) for inter-continental connections; and (200ms, ∞) for non-tether connections. Of course, such policies have to be universally agreed upon and standardized. These ranges define the particular algorithm and the corresponding values for c. Table 3.2 lists one possible way of choosing constant c.

Table 3.2: Choice of c in TCP fair window algorithm

RTT range	Constant C	Equivalent RTT
(0, 50ms)	1024	31.2ms
(50,100ms)	256	62.5ms
(100,200ms)	64	125ms
(200ms, ∞)	4	500ms

Another problem with the choice of c lies in incremental deployment of such algorithm. When TCPs with different implementations operate in a heterogeneous environment, TCPs observing the fair algorithms might be at a disadvantage. Fortunately, DiffServ router mechanisms offer a solution for migrating TCPs to the fair algorithms. See Section 4.8.6 in Chapter 4 for a detailed discussion of this point.

3.4.3 Mechanism 2: Setting *ssthresh* for TCP

We propose the following changes to reflect the change in the underlying premise from a purely best-effort service model to a DiffServ model. Each TCP is made aware of the upper-layer policy SLA, and the targeted rate, R_{target}, that it is supposed to be running at. This can be done via a signaling protocol between end host TCP and an edge router or a policy server. Then TCP sets its *ssthresh* (in bytes) as the multiple of its known R_{target} (bytes/sec) and its estimated RTT (sec). There are two instances where TCP sets its *ssthresh*. Initially, before TCP starts transmitting packets, it estimates its round trip time during its three-way handshake. With this initial estimate of RTT, it can set its initial *ssthresh*. This helps to "gauge" the operating point of TCP. Additionally, we propose that TCP sets its *ssthresh* similarly when congestion is detected. When TCP detects congestion via either implicit or explicit mechanisms, it resets its *ssthresh* to be the multiple of its known R_{target} and the estimated RTT then. Since RTT might change during the course of a TCP connection, the *ssthresh* will change as well. TCP reduces *cwnd* to be one-half of the previous value before the packet drop, as it would in current implementations. This has the effect of reducing instantaneous sending rate of TCP connections to alleviate temporary congestion, but allows each TCP connection to quickly throttle back to its target operating point.

3.4.4 Mechanism 3: ECN-enabled TCP in a DiffServ Domain

Mechanisms similar to ECN could be deployed in the DiffServ architecture. Instead of dropping packets, the RIO gateway can also take advantage of the ECN mechanism by marking them. A RIO gateway can apply its preferential algorithm in which it marks an OUT packet as experiencing congestion with a much higher probability than an IN packet [9]. Both the ECN bit and the IN/OUT bit will be copied by the transport-layer receiver and relayed back to the sender. The TCP sender has to be able to recognize the two types of packets (IN and OUT), and respond to ECN bits in them differently.

When an OUT packet arrives back at the TCP sender with the ECN bit marked, it in-

dicates that the RIO gateway is operating in the congestion sensitive phase (phase 2 in Section 2.1). When a RIO gateway deploys the ECN mechanism as the congestion notification mechanism, TCP window reduction should be no more aggressive than when a packet is dropped in the current TCP implementations. We recommend that TCP reduces its *cwnd* to be one half of the current *cwnd* value, and resets *ssthresh* to be the the multiple of R_{target} and RTT. Depending on the value of *cwnd* and *ssthresh* prior to receiving the ECN signal, TCP can be operating in either the linear increase mode or the exponential increase mode again. In either case, the reduction in the window size will induce a temporary reduction in TCP's sending rate to alleviate congestion, but still keep TCP operating close in the targeted operating point.

When an IN packet arrives back at the TCP sender with the ECN bit marked, it indicates that the RIO gateway operates in the congestion control phase (phases 4 & 5 in Section 3.2.3), meaning that the gateway has seen persistent long queues and is forced to mark both IN and OUT packets with probability 1. When such packet is received, the TCP sender should react to the congestion signal more drastically. We recommend that TCP reduces its *cwnd* to be one packet, and reset *ssthresh* to be the multiple of R_{target} and RTT. This is the same window reaction as in the current implementation when a packet has been dropped but TCP starts in its slow start phase with a configured *ssthresh*. This will cause a more drastic reduction in TCP's sending rate, but since the new *ssthresh* will be greater than the new *cwnd*, TCP will quickly recover to the target operating point using exponential increase window increase algorithm. Table 3.3 tabulates the combined mechanisms—in both TCP and routers—proposed in this chapter.

3.5 Revisit Designs and Discussion

As mentioned in Section 2.4.5, we started with our second design, i.e., modifying router mechanisms only. We developed RIO and TSW algorithms and did simulation experiments. Our simulation experiments demonstrate that with RIO and TSW in place, a DiffServ net-

Table 3.3: Summary of mechanisms in routers and endhost TCP

	TCP sender	*Tagger*	*RIO*	*TCP receiver*

ECN-capable	Turn ECN bit off
	
	if (ECN) {
	  if (IN)
	  cwnd = 1;
	  else
	  cwnd = cwnd/2;
	  ssthresh = byte equi. of (R_t * RTT);
	}
	else {
	  increase *cwnd* by $c * RTT^2$;
	}
		Mark IN/OUT bit according to profile	if (ECN bit off) {
			  mark packets differentially;
			}
			else {
			  drop packets differentially;
			}
				Copy ECN and TOS bits to ack pkts

ECN-incapable	Turn ECN bit ON
	
	if (packet dropped) {
	  ssthresh = byte equi. of (R_t * RTT);
	  cwnd = cwnd/2;
	}
	else {
	  increase *cwnd* by $c * RTT^2$;
	}
		Mark IN/OUT bit according to profile	Drop packets differentially	Copy ECN to ack pkts

work can create discriminations between two types of packets. The design of TSW and RIO is also versatile enough to deal with different mixture of traffic. The parameters of RIO can be chosen as to create strong discriminations under different network conditions. The simulation results show that there are still discrepancies between the SLAs and the achieved targeted rate.

We then realized that much of the difficulties in meeting the specific SLAs lies in TCP itself. As discussed in the previous section, the current version of TCP (Reno) is neither robust nor fair. We have experimented other versions of TCP: TCP-SACK[15, 23], TCP-newreno[30], or TCP with ECN mechanism [22]. These three versions improve TCP's robustness in recovering packets, but do not change TCP's window increase algorithm, therefore, they are also biased against long-RTT connections. Then, we proposed mechanisms that can both improve TCP's robustness, fairness, as well as make TCP SLA-aware—the three TCP-DiffServ mechanisms. This DiffServ-enhanced TCP can work very well with both RIO and TSW. Figure 3.11 illustrates how different versions of TCP fall in the Robustness and Fairness space.

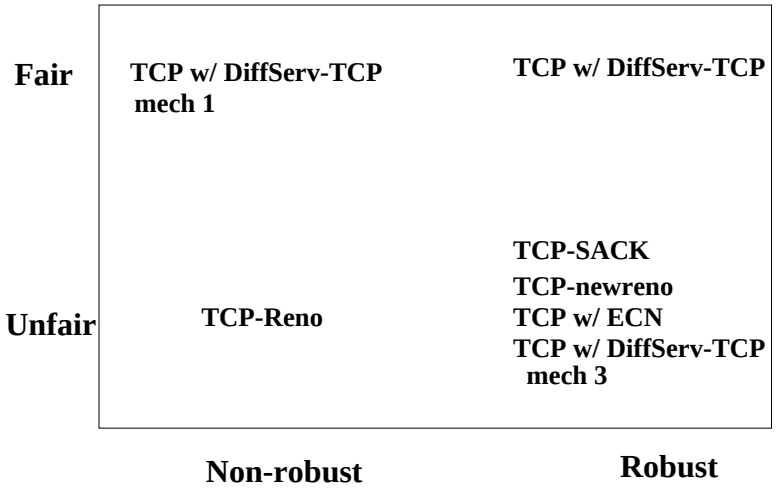


Figure 3.11: Different Versions of TCP on Robustness and Fairness Scale

In hindsight, the realization that we have to change TCP should not be surprising. Since the bandwidth allocation scheme in the current Internet relies on a congestion feedback loop

and mechanisms in both routers and TCP, and the essence of Assured Forwarding is to redesign the bandwidth allocation model, then, only changing mechanisms in routers will not be particularly effective. Changes have to affect end host congestion control mechanisms as well.

If this is the case, shouldn't we have used alternative designs? For example, in design 1 proposed in Section 2.4.5, we incorporate tagging algorithms with end hosts and modify both end hosts and routers. In fact, once we realize we have to modify the end hosts TCP anyway in design 2, the difference between designs 1 and 2 is small. Basically, the difference is where to do the proper authentication and verification. In design 1, we incorporate TSW in end hosts, and have a signaling protocol between end hosts and edge routers. There is also an additional checking and verification function in edge routers. In design 2, we incorporate TCP-DiffServ in end hosts, TSW in edge routers, but we also need a signaling protocol between end hosts and edge routers to inform the enhanced TCP the SLA. The checking function is done by TSW.

Design 3 avoids the authentication and verification problem altogether by moving the new bandwidth allocation scheme to the networks entirely, and leaving end hosts TCP/UDP to do flow control only. This is a more radical change. In essence, this proposal believes that congestion control should be left in the network itself and not in the end hosts. This meets the requirements of a commercial network really well because the end host transport protocol is left *passively* sending and receiving packets through network, so there is no chance that end hosts can flood the network, as they could in today's Internet.

It should be noted that the mechanisms proposed in this chapter are not limited to design 1. They are a group of mechanisms that can provide fair and robust allocation of bandwidth meeting specific requirements of SLAs, and they are versatile enough to deal with different type of traffic mix. They can be applied in all three designs, regardless what the ultimate DiffServ networks looks like.

Chapter 4

Evaluation

We use extensive simulations to evaluate our proposed mechanisms. This chapter presents the evaluation results. At a high level, we organize this chapter in the following manner. We categorize the proposed mechanisms into two groups—mechanisms in routers (RIO and TSW), and mechanisms in end hosts (TCP-DiffServ). In the first set of simulation experiments, we apply the router mechanisms (RIO and TSW) to a Diff-Serv domain, and evaluate various aspects of the DiffServ architecture.

For this set of simulations, we go into depth for each simulation scenario. We study the sender-controlled scheme, in which TSW taggers are installed in the ingress routers of the domain and RIO droppers are installed in the interior routers of a domain (Section 2). Using a similar setup, we evaluate how router mechanisms work with a more robust version of TCP, TCP-sack (Section 3). We also consider the receiver-controlled scheme, in which TSW taggers are installed in the *egress* routers of a domain, and RIO droppers are installed in the interior routers (Section 4). We then consider the effect of cascaded taggers on individual connections (Section 5), and the impact of taggers on an aggregation of TCP connections (Section 6). Finally, we study how TCP connections co-operate with non-responsive connections after we apply the RIO and TSW algorithms to a domain (Section 7).

We conclude from the above scenarios that though a DiffServ domain with router mech-

anisms can differentially allocate network bandwidth according to the specified service profiles, the extent to which such differentiation can be achieved is not always predictable. The limitations, as we discussed in Chapter 3, are in TCP itself.

Our subsequent experiments are studies on the combined effects of applying both the TCP-DiffServ mechanisms and the router mechanisms. We compact the simulations somewhat with one simulation run going through four network congestion states: start-up, under-provision, recovery and over-provision. We conclude that neither group of mechanisms, by itself, is sufficient to achieve robust, precise allocation of bandwidth. However, when both groups of mechanisms are applied to a DiffServ domain, the domain can allocate bandwidth resources in a robust and precise manner under a variety of traffic conditions.

4.1 Simulation Methodology

We use the *ns*[1] network simulator to evaluate the proposed router mechanisms. *Ns* is a discrete event simulator for network research. It provides substantial support for TCP, router queuing mechanisms, and various topologies, making it ideal for evaluating the DiffServ architecture. We developed modules for RIO, TSW, and TCP-DiffServ mechanisms, and compiled those into *ns*. Simulations scripts are written in Tcl and based on some early research work in [24]. Each script defines a topology, sets attributes of network devices, defines parameters, and describes network events during a period of time frame. This section describes the common simulation setup we used; each of the following sections (Sections 4.2 - 4.7) uses additional topologies as well. The complete simulation scripts and modules can be found at http://www.cs.princeton.edu/ wfang/papers.html.

- TCP Connections

  Unless otherwise specified, all TCP connections are configured as follows: they are TCP-Reno with support for Slow-Start, Fast-Retransmit and Fast-Recovery. The packet size is 1000 bytes and the TCP timer granularity ($tcptick$ variable in the default.tcl file) is 0.1 second. The receiver's advertised window is usually configured

large enough so that it is never a limit on the TCP sender's window. The TCP connections start at a slightly different time from each other, but all within the first 5 seconds of the simulations. Each simulation simulates 20 seconds of network condition unless otherwise specified.

- Interior Routers and RIO Algorithm

  We implement the RIO algorithm in interior routers. We present simulation results in tables to save space, with the parameters of RIO described in table headers. The parameters are presented in the format of (min_in, max_in, P_{drop_in}) for IN Packets, and (min_out, max_out, P_{drop_out}) for OUT packets. The thresholds are measured in number of packets, and the dropping probability is a fraction. We use heuristics described in Section 3.2.4 to choose the RED and RIO parameters.

- Edge Routers and TSW Algorithm

  We implement the TSW algorithm as a module on the access link to edge routers. In the simulator, a tagger is implemented as a subclass of a drop-tail link (drop-tail.cc). Data traffic from sending hosts will reach a TSW tagger before arriving at a router. In an actual router implementation, a TSW tagger should be a module sitting after packet classification and before the forwarding function. Logically, our simulator implementation is equivalent to a router implementation as long as there is only one connection on every single access link, which is the case for all our simulations.

- SLAs

  We define a very simple SLA, which is "average TCP throughput of R_t from this host to anywhere", where R_t is a target rate. The Round Trip Times (RTTs) used in the simulations range from 20ms to 150ms. In our simulations, we try to push the envelope by choosing very different R_ts and very different RTTs.

- Evaluating Metric

As the evaluating metric, we use the average throughput each host's TCP achieves. Average throughput is calculated at the receiver's side after a TCP connection has reached a steady state. We consider the simulations to have reached a steady state after 5 seconds. Average throughput is defined as the total amount of data received over total transfer time. Total amount of data is the data packets TCP layer presents to upper layer applications, excluding all unnecessary retransmissions. The closer the average throughput is to the respective target-rate, R_t, the better a scheme works.

- Presentations

  Simulation results are presented in a number of ways. Most throughput results are presented in tables to save space. In addition, we use two types of graphs. They are:

  - Throughput graphs

    These are graphs of achieved throughput vs. TCP's round trip time. Each data point on the graph is a throughput for a particular TCP connection with a certain RTT. Each data point is calculated by averaging the results from 5 simulation runs. These graphs are to illustrate the effectiveness of the router mechanisms under different network scenarios.

  - TCP window oscillation graphs

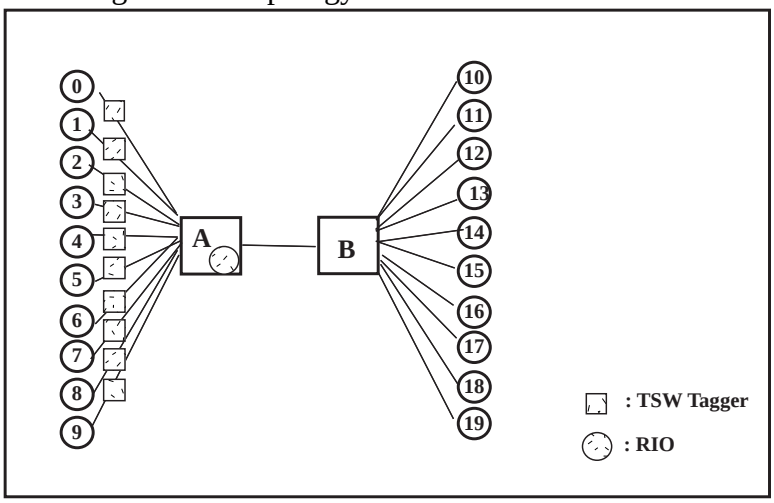
    These are graphs of TCP congestion window (*cwnd*) vs. time in a typical simulation run. They are to illustrate the robustness of different versions of TCP.

- Topology

  We use the topology depicted in Figure 4.1, in which, ten hosts are connected to ten other hosts, sharing a common link between routers A and B. Each host (i) has a TCP connection to its peer host ($i + 10$). Altogether, there are 10 TCP connections. The ten connections are of different RTTs. They can be divided into five groups. The two connections in each group have the same RTTs, but different target rates, R_t. For example, connections 0 and 1 both have 40ms round trip time, but have 5Mbps and

1Mbps as target rates, respectively. Each of the five groups has a different RTT from the other groups. The RTTs for the five groups are 20ms, 40ms, 50ms, 70ms and 100ms, respectively.

Figure 4.1: Topology for Sender-based Scheme



4.2 Sender-based Scheme

This section presents two sets of simulation experiments. In Section 4.2.1, we use simulations to illustrate the network bias against long-RTT TCP connections. In Section 4.2.2, we presents the results of experiments using sender-based scheme to allocate network resources in a DiffServ domain.

4.2.1 Network Bias Against Long-RTT Connections

As discussed in Section 3.4.2, current Internet, when allocating resources, has a bias against long-RTT connections. This bias is due to TCP's window increase algorithm. During each RTT, TCP opens up its window size by exactly one packet (in bytes). The longer the RTT (seconds), the slower TCP increases its sending rate (bytes/seconds). After a TCP connection reduces its sending rate following a packet loss, it takes a long-RTT connection a

longer time to recover to its original sending rate than a short-RTT connection. If routers drop the same number of packets from a long-RTT connection as from a short-RTT connection, the short-RTT connection will receive more bandwidth than the long-RTT connection over time.

We simulate this with the following network configurations using the topology in Figure 4.1: bottleneck link A-B has a link speed of 6Mbps, and there are ten TCP connections, each with a different RTT. All connections start sending within the first five seconds of simulation. The simulation results are listed in column 3 of Table 4.1. The respective RTTs for the connections are listed along with the average throughputs they achieve. This simulation reflects current situation in the Internet, where most hosts implement TCP-Reno. We make two observations. First, the network bandwidth is distributed according to the RTTs of TCP connection and there is a strong bias against long-RTT connections. Second, in TCP-Reno, the congestion window oscillations are usually drastic and unpredictable. As a result, the throughput TCP-Reno can achieve is usually unpredictable, e.g., connections 2 & 3 have the same RTT, but differ significantly in their throughput (22%).

Table 4.1: TCP's bias against long-RTT connections. Link A-B capacity = 6Mbps. RED gateway

Conn #	RTT (ms)	Achieved Bandwidth by Reno-TCP(Mbps)	Achieved Bandwidth by Modified TCP (Mbps)
0	20	0.829482	0.623805
1	20	1.0932	0.620590
2	40	0.541590	0.600449
3	40	0.694729	0.655939
4	50	0.690377	0.674704
5	50	0.538202	0.527582
6	70	0.430481	0.503961
7	70	0.384579	0.620259
8	100	0.389476	0.671800
9	100	0.419572	0.511367
Total		6.011688	6.010456

Further, we want to verify that this bias is caused by TCP's window increase algo-

rithm. The current TCP window increase algorithm works as follows: upon receiving each acknowledgment packet, TCP sender calculates its congestion window *cwnd* as

$$cwnd+ = 1/cwnd;$$

Therefore, after receiving *cwnd* number of acknowledgment packets, TCP's *cwnd* is increased by one packet. Since within each RTT, there are always *cwnd* number of packets outstanding, this algorithm essentially increases *cwnd* by one packet each RTT.

If the algorithm were to increase TCP window by a constant factor regardless of the RTT (algorithmically, this is to increase RTT^2 for each RTT period of time), then all TCP connections would equally share the link bandwidth. This hypothesis can be verified using the *ns* simulator. We change TCP's window increase algorithm to be

$$cwnd = cwnd + c * RTT^2$$

for every acknowledgment packet received, and re-run the same simulation. The result of this simulation are listed in column 4 of Table 4.1. It is clear that the bias against long RTT is eliminated, as the 10 connections sharing the 6Mbps link have roughly 0.6Mbps each.

4.2.2 Sender-based Scheme

In this section, we study the case of a sender-based allocation scheme. We implement the RIO algorithm for interior routers and the TSW algorithm for edge routers, and incorporate them into *ns*, one TSW tagger for a TCP connection. We use the topology depicted in Figure 4.1. Each of the 10 sending hosts has a contracted SLA and a respective target rate, R_t, of either 1Mbps or 5Mbps. We choose very different R_ts to experiment how well the over system deals with varying target rates. The 10 TCP connections also have very different round trip times. The sum of all R_ts is 30Mbps, and the bottleneck link speed is set to be 33Mbps. This allows 10% buffering during the congestion. The respective R_ts and RTTs are listed in columns 2 and 4 in the table.

We run two simulations: one simulates the current Internet, where none of the connections has an SLA or a tagger, and the other simulates a Diff-Serv network, where 10 connections each has its respective tagger and SLA. RIO is used in the bottleneck router A.

A note on how to choose RED and RIO parameters. As discussed in Section 3.2.4, we use heuristics in choosing the RED and RIO parameters. For example, we set RED parameters to be (10, 30, 0.02), or 10 packets as min (the minimum threshold), and 30 packets as max (the maximum threshold). This is equivalent to saying that when the average queue size exceeds 2.4ms, the RED queue starts to drop packets randomly [1]. Similarly, by the time the average queue size exceeds 7.2ms, RED is in *congestion avoidance* phase and drops all arriving packets. RIO parameters are configured similarly.

Table 4.2: Comparison of current Internet scenario and a DiffServ network with router mechanisms. Link BW=33Mbps. Parameters for RED router: (10, 30, 0.02); parameters for RIO:(40,70, 0.02) for *IN*s and (10, 30, 0.2) for *OUT*s. Used TCP-Reno

Conn #	RTT (ms)	Current Internet(Mbps)	R_t	DiffServ (Mbps)
0	20	7.04873	1	2.27289
1	20	6.22214	5	5.7619
2	40	2.83662	1	1.28011
3	40	2.28316	5	5.26757
4	50	2.62307	1	1.21957
5	50	2.81556	5	5.18823
6	70	1.61073	1	1.34831
7	70	1.57837	5	4.12794
8	100	1.64488	1	0.99633
9	100	1.85132	5	4.12563
total		30.51458		31.588476

Table 4.2 compares the results from these two simulations. Column 3 lists the achieved rate for all TCP connections in the current Internet. The network bias against long RTT connections is pronounced. Similar to simulations in Section 4.2.1, TCP window oscillations are drastic and unpredictable. Column 5 lists the throughputs achieved by the connections in a Diff-Serv environment. The total link throughputs in both simulations are comparable:

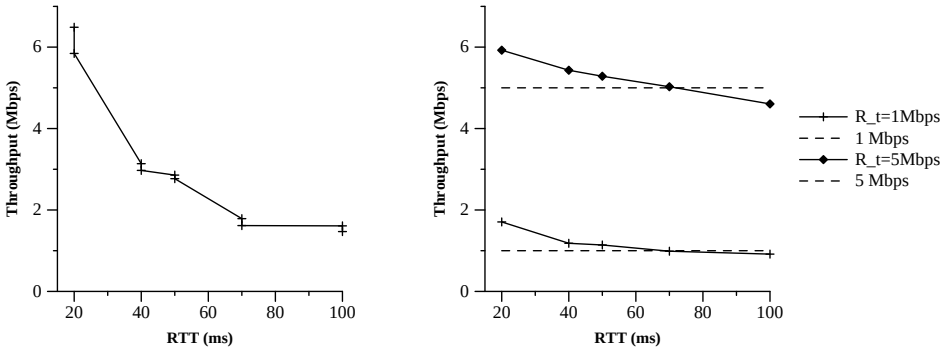
[1] 10 packets of 1000 bytes is 80,000 bits, divided by 33Mbps, gives close to 2.4ms queuing delay.

30.51Mbps vs. 31.6Mbps, or 92.5% vs. 96% link usage.

We observe that all being equal, the network bandwidth in the current Internet can be distributed according to the RTTs of the connection, and strongly in favor of connections with short RTTs. The throughput achieved by TCP is subject to congestion in routers and can be very unpredictable. In contrast, a DiffServ network with router mechanisms can allocate network capacity according to the respective service profiles the users have contracted for. For TCP connections with the same service profiles but different RTTs, (e.g. connections 1 and 9), there is a bias against long-RTT connection, but the bias has been significantly mitigated (though not eliminated) by having the router mechanisms in place. Most importantly, now the system can provide quite different throughputs to different connections with reasonable assurance.

Figure 4.2 presents the throughput graphs for these two scenarios. For each graph, we run five simulations of the exact same configuration; each simulation uses a randomly generated seed, and we get a slightly different result each time. Each data point on the throughput graph is the average of 5 simulations runs. In both graphs, the X axis is the TCP's round trip time, measured in ms; and the Y axis is the achieved throughput of a connection, measured in Mbps. The left graph plots the throughput in the current Internet scenario. Since every two TCP connections have the same RTT, there are two data points for a given RTT. The achieved bandwidth decreases as the RTT increases. There is a varying difference in the two throughput data points for a given RTT, showing the unpredictability in the current Internet environment. The right graph plots the achieved throughput with their respective target rates. The target rates are 1 Mbps and 5 Mbps, in dotted lines. A visual interpretation of how well a scheme works is how close the achieved throughputs are to the dotted lines. It is clear that the network bias is still visible, as shown by the slowly decreasing throughput line as RTT increases. However, such bias has been much mitigated since the slop is not as deep as that in the left graph.

Figure 4.2: Throughput Graphs for Current Internet and DiffServ Scenarios



Bandwidth Distribution in the Current Internet Bandwidth Distribution in a DiffServ Network

4.3 DiffServ with TCP-sack

TCP-Sack[15, 23], or TCP with Selective Acknowledgment, has a very different approach for handling acknowledgment packets from the previous versions of TCP, TCP-Reno or TCP-Tahoe. In previous versions of TCP, the receiver acknowledges the highest sequence number of a data packet it has received. When a packet is lost, either due to a congested gateway or a lossy link, the receiver sends duplicate acknowledgments (duplicate acks) upon receiving successive packets after the lost packet. For example, if packet #4 is lost, the TCP receiver receives packets #3, #5 and #6. The respective acknowledgment packets the receiver sends would be #3, #3 and #3. The first ack #3 acknowledges the receipt of packet #3 but the last two acks indicate that packet #4 has not been received.

The sender infers from the duplicate acks that a packet had been lost, and retransmits the presumably lost packet, usually, the packet immediately after the one that the duplicated acks acknowledge. In the above example, the sender would retransmit packet #4. Each such retransmission also results in a reduction in congestion window *cwnd*, and subsequently, a reduction in sending rate.

Since in TCP-reno the information as to which packet was lost is implicit conveyed to the sender, the sender can only *deduce* which packet has been lost. This leads to either

unnecessary retransmissions of packets that did not get lost, or successive reduction of congestion window *cwnd*, and eventually, a *cwnd* so small that the sender can no longer recover the lost packets. The sending TCP will have to wait for a timeout to recover the lost packets, and start from Slow-Start again, with a much reduced *ssthresh*. This has a much more drastic effect on TCP performance, and a DiffServ domain tries to avoid it by adding probabilistic functions to both the tagging and dropping schemes.

TCP-SACK uses a completely different approach than that of TCP-reno: the acknowledgments now contain complete information as to which packets have been received and which have not. This complete information allows the TCP sender to retransmit packets selectively, and avoid unnecessary retransmits. The TCP sender can also recover multiple packet losses within a window by only reducing the congestion window once. Therefore, TCP-sack is more robust than TCP-reno, and can stay in the Fast-Retransmit phase and avoid using timeouts to recover packets. Because the end host is better at handling network congestion signals, RIO parameters can be chosen to create *stronger* discrimination against OUT packets, without fearing consecutive OUT packet drops will drive end host TCP to timeouts. This way, the overall system can create stronger differentiations among connections if end hosts use TCP-SACK.

4.3.1 Results

We use the same topology as in the previous section. We replace TCP-reno with TCP-Sack and repeat those simulations. There are two scenarios as well, the first scenario is to simulate the current Internet; the second case is to simulate a DiffServ domain. The taggers are installed on the access links. Table 4.3 lists the simulation results.

The two primary observations from Section 4.2.2 (sender-based, TCP-Reno) still hold here: in the current Internet, network bias against long-RTT TCP connections is evident, and in the DiffServ network, allocation of bandwidth is roughly according to the target rates R_t. However, TCP-sack is more *robust* in handling congestion signals, so we use (10, 30, 0.5) for OUT packets, compared to (10, 30, 0.2) in the previous section. When using

Table 4.3: Using TCP-Sack in Diff-Serv. Parameters for RED routers are (10, 30, 0.02), and those for RIO routers are (40, 70, 0.02) for INs and (10, 30, 0.5) for OUTs. BW=33Mbps

Conn #	RTT(ms)	Current Internet(Mbps)	R_t (Mbps)	DiffServ(Mbps)
0	20	6.74918	1	1.33071
1	20	6.47331	5	6.13875
2	40	2.64113	1	1.2283
3	40	3.09084	5	5.38146
4	50	2.17542	1	1.13145
5	50	2.65581	5	5.20269
6	70	1.75715	1	0.98892
7	70	1.90942	5	4.92265
8	100	1.12921	1	0.89991
9	100	1.49762	5	4.68653
total		30.0791	30	31.9114

TCP-SACK, the overall system can achieve better throughputs, i.e., throughputs that are closer to the targeted rates.

In the following sections, whenever possible, we will use TCP-SACK as the end host TCP.

4.4 Receiver-based Scheme

In DiffServ, the receiver-based scheme complements the sender-based scheme. A receiver-based scheme can be implemented if 1) end host TCP implementations understand Explicit Congestion Notification (ECN) semantics; and 2) RED[2] gateways also understand ECN mechanism, and they *mark*, instead of drop, packets when congestion occurs. Taggers for the receiver-based scheme are installed on the access link from the receiving hosts to the network, or the egress routers of the network.

In the receiver-based scheme, a TSW tagger monitors the arriving traffic stream from the network to the receiver, and resets the ECN bit of a packet if the arrival rate is less

[2]Note: the interior routers are RED gateways, not RIO gateways, because when packets arrive at an interior gateway, they are haven't gone through any taggers and are not marked. There is no differentiation among them.

than the subscribed service profile (SLA). This means that even if a packet had caused congestion at the network, as long as this packet is still within the receiver's service profile, the network should have provisioned for it, and the receiver's TSW tagger will turn off the ECN bit so no congestion signals are sent to the end host TCP. If the arrival rate is higher than the subscribed service profile, then the tagger will only turn off the ECN bit for packets which are within service profile, and leave those *in excess* of the service profile intact. This way, packets arriving at the TCP receiver with their ECN bits still on are those that are beyond the receiver's profile *and* have caused congestion in the network. These packets will cause the TCP sender to slow down by the ECN mechanism, and thus, control the sender to conform to a sending rate that is within the receiver's service profile. Because there are no explicit packet drops, and the congestion control signals from network is clearly conveyed to end hosts, the end hosts typically operate in the *congestion avoidance* phase. Therefore, the overall system is responsive.

4.4.1 TSW Tagger for Receiver-based Scheme

Figure 4.3 contains the pseudo code for the TSW tagger in the receiver-controlled scheme. A receiver-controlled tagger turns off ECN bit in a packet if the traffic has not exceeded the corresponding service profile (SLA).

We use a similar network topology as that in Section 4.1. The only difference is that the profile meters are installed at the receivers' side, instead of at the senders' side. Figure 4.4 illustrates the receiver-controlled scheme. Configurations for this topology are the same as those in Figure 4.1.

4.4.2 Results

The simulation setup for end hosts is the same as in the sender case. We simulate two scenarios: one is the current Internet and the other is a DiffServ domain with RIO routers and TSW taggers for the receiver-based scheme.

```
Estimate sending rate R;
causedCong = whether ECN bit in pkt is set;
tagged = mark_packet (pkt, R);
if  (causedCong) {
      if (! tagged)
            turn ENC bit off;
      else
            /* leave ECN bit on, do nothing */
}
/* else, if it didn't cause congestion, let the pkt through */

subroutine:
      mark_packet (pkt, R)
      {
            if R < Rt, return 0;
            if R > Rt, with Pmark = (R - Rt)/Rt, return 1;
      }
```

Figure 4.3: Receiver-based TSW Algorithm

Figure 4.4: Topology for Receiver-controlled Scheme

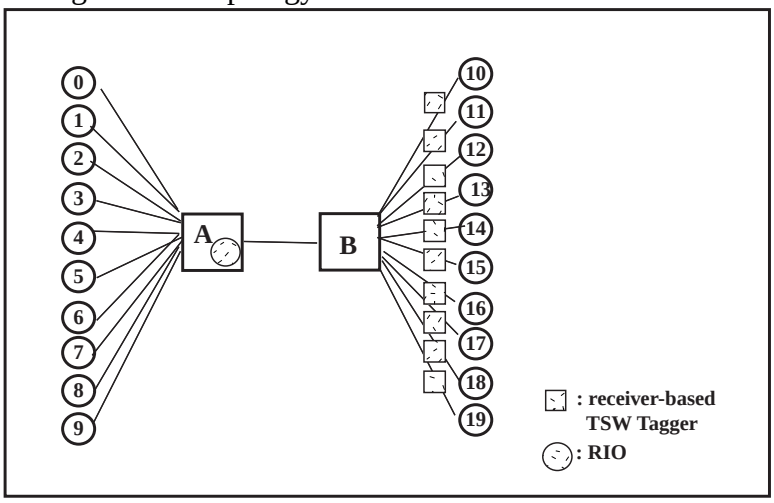
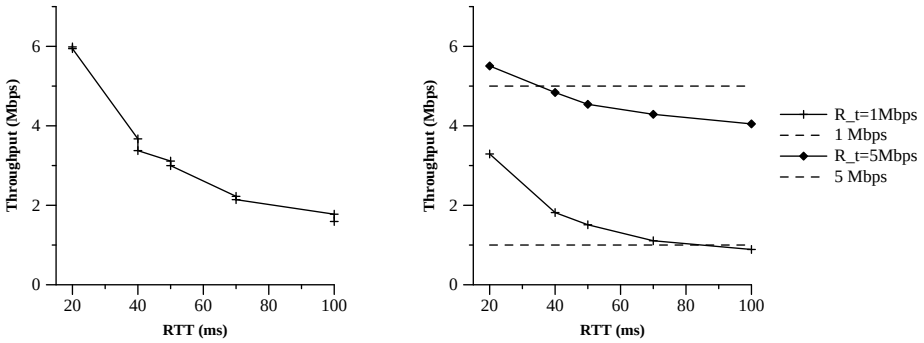


Table 4.4: Receiver-based scheme in a DiffServ domain. BW = 33Mbps. RED router is configured with parameters (15, 40, 0.02). Used TCP-Reno with ECN semantics

Conn #	RTT (ms)	Current Internet (Mbps)	R_t (Mbps)	DiffServ with RIO+TSW(Mbps)
0	20	6.18994	1	2.71758
1	20	5.69154	5	5.38844
2	40	3.30375	1	1.65922
3	40	4.06525	5	5.02699
4	50	2.67894	1	1.3496
5	50	3.19256	5	4.83154
6	70	2.16772	1	1.04867
7	70	2.28335	5	4.7517
8	100	1.84308	1	0.868532
9	100	1.46514	5	4.41489
Total		32.88127	33	32.057

Table 4.4 lists the results from two simulation runs. Column 2 is the round trip times (RTTs) of TCP connections. Column 3 is the results using the current Internet mechanisms when both TCP and routers use ECN mechanisms. Column 4 is the respective target rates, or R_t, for the ten connections, and Column 5 is the achieved bandwidths in a DiffServ environment using the receiver-based scheme.



Bandwidth Distribution in the Current Internet Bandwidth Distribution in a DiffServ Network

Figure 4.5: Bandwidth Allocation Using Receiver-based Scheme

The two primary observations from the previous two sections still hold here: in today's network (column 3), short-RTT connections have a clear advantage over long-RTT connections. In the DiffServ architecture (column 5), the network allocates bandwidth according to the target rates, R_t, in times of network congestion, and there is a bias against long-RTT connections. The overall throughput is slightly better than that of the sender-based scheme: 32.88Mbps vs. 30.51Mbps (Table 4.2). The reason is that the receiver-controlled scheme is essentially a scheme using ECN mechanisms, which is a more robust way for TCP to deal with congestion. Like TCP-SACK, TCP's ECN mechanisms can adjust TCP's sending rate responsively but not drastically, so the overall system utilization is higher.

However, the receiver-based scheme is different from a sender-based scheme which also uses the ECN mechanism. If TCP sending hosts use ECN, and the RIO algorithm is changed to be marking instead of dropping packets, we would have seen a result that is quite similar to that of TCP-SACK. In this case, sending end hosts are robust against packet drops, and RIO can therefore be configured to discriminate against OUT packets without driving end host TCPs to timeout and eventually Slow-start. In the receiver-based scheme, however, whether the packet is an IN packet or an OUT packet is not known at the time when network congestion happens. This piece of information is only known when the packet is *exiting* from the network, therefore, there is no way for the network to create stronger discrimination against receiver-pay packets. Therefore, a receive-based scheme

doesn't provide better service differentiations, but only better network utilization.

Figure 4.5 gives the throughput graphs for the receiver-based scheme. For each graph, we run five simulations of exactly the same configuration. Each simulation uses a randomly generated seed, and we get a slightly different result each time. Each data point on the throughput graph is the average of 5 simulations runs. In both graphs, the X axis is a TCP's round trip time, measured in ms; and the Y axis is the achieved throughput of a connection, measured in Mbps. The left graph plots the throughput in the current Internet scenario. Since every two TCP connections have the same RTT, there are two data points for a given RTT. The achieved bandwidth decreases as the RTT increases. There is a varying difference in the two throughput data points for a given RTT, showing the unpredictability in the current Internet environment. The right graph plots the achieved throughput with their respective target rates. The target rates are 1 Mbps and 5 Mbps, in dotted lines. A visual interpretation of how well a scheme works is how close the achieved throughputs are to the dotted lines. It is clear that the network bias is still visible, as shown by the slowly decreasing throughput line as RTT increases. However, such bias has been much mitigated since the slop is not as deep as that in the left graph.

4.5 Cascaded DiffServ Domains

If the DiffServ architecture and mechanisms are fully deployed in the Internet, a packet might traverse multiple DiffServ domains to reach its destination. Each DiffServ domain has ingress routers implementing tagging algorithms and interior routers implementing dropping algorithms. Therefore, a packet from a sending host will traverse through a series of taggers and droppers before reaching its final destination. For example, Host 1 in Figure 2.2 will go through two taggers, one in AS 1 and another in AS 2 to reach its final destination, Host 2.

The taggers in each DiffServ domain interpret service level agreements (SLAs). Each SLA specifies a bilateral agreement between two neighboring domains or between a do-

main and an end host. The closer the tagger is to the sending source, the more applicable the service level agreement is to a particular source. At the edge of the network, a SLA can be specified for a particular host or source. However, in the middle of the network, a SLA between two domains is only applicable to the aggregated traffic between two domains, and therefore, less effective on any particular connection.

In a well-provisioned network, an ISP could contract a profile from its downstream ISP (data flows from source to destination, or from upstream to downstream) equal to or greater than the sum of all upstream profiles it has contracted out. This would ensure that all IN packets from upstream will remain IN packets throughout. However, this is often not the case, because not all IN packets from upstream will *simultaneously* go through the same downstream egress router to the next domain. The outgoing traffic can be just as dispersed as the incoming traffic. Therefore, ISPs often contract a service profile *less* than the total sum of all upstream profiles, and hope the multiplexed traffic from all upstream sources will still be less than the downstream service profile. If this is the case, then IN packets could be turned into OUT packets at the ingress router of a downstream domain. In this section, we study how cascaded taggers affect bandwidth allocation in a DiffServ environment.

4.5.1 Setup

The simulation setup is listed as following:

- Topology

  There are two topologies used for this set of simulations. Figure 4.6 is what we call the "controlled" case. There are five taggers, and one for each of the five connections. Figure 4.7 is what we call the "compared" case, where five connections go through their respective taggers as well as an aggregated tagger in a subsequent domain. In the compared case, each connection first traverses its individual tagger, a dropper in router A, an aggregated tagger in router B, and finally a dropper and a congested bottleneck in router C.

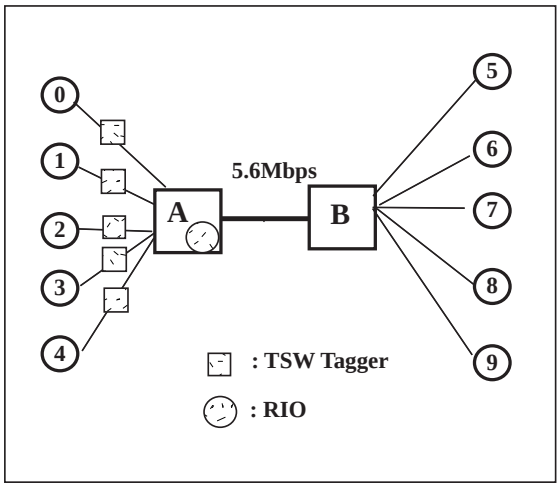


Figure 4.6: Cascaded DiffServ Domain: the Controlled Case

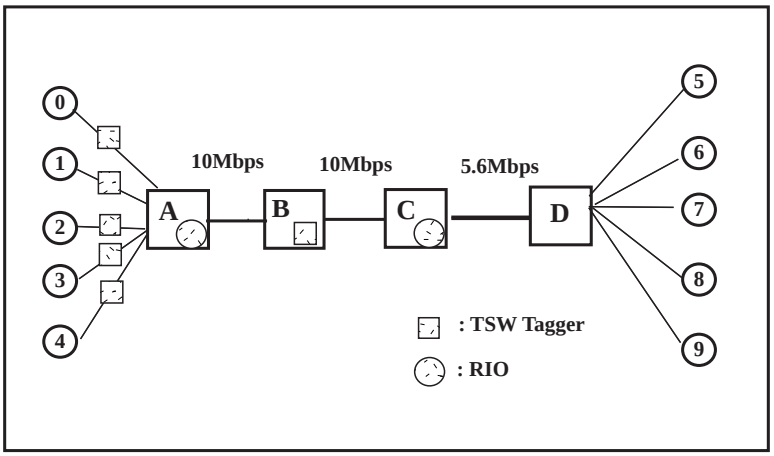


Figure 4.7: Cascaded DiffServ Domain: the Compared Case

In both cases, the individual access link from a host to the first router is 10Mbps. Each individual tagger is set to have a 1Mbps target rate (R_t). The RTTs of different connections range from 30ms to 70ms. In the compared case, links from router A to router B, and from B to C, are 10Mbps each. Link C-D has a speed of 5.6Mbps, which is a 12% mark-up over the sum of allocated R_ts (5Mbps). Link C-D is the only bottleneck. For the aggregated tagger in router B, we try different target rates. The corresponding connections in the two cases have the same RTTs.

- RIO Parameters

  TCP packet size is set to be 500 bytes. In the controlled case, the configurations for RIO router (router A) are: (7,14, 0.02) for IN packets and (5, 10, 0.5) for OUT packets. This is equivalent of setting 5ms[3] and 10ms average queuing delays for the respective min_in and max_in; and setting 3.5ms and 7ms average queuing delays for the respective min_out and max_out. In the compared case, the configurations for RIO router (router C) are: (7, 14, 0.02) for IN packets, and (5, 10, 0.5) for OUT packets.

- Cascaded Tagger

  The cascaded tagger at router B uses a TSW algorithm. The algorithm differs slightly from the one presented in Chapter 3: it estimates the average rate of incoming IN packets only, and when this estimated rate exceeds the contracted target rate R_t, it tags IN packets into OUT packets. In other words, the cascaded tagger does not re-mark incoming OUT packets, but may turn IN packets into OUT packets if the aggregated traffic exceeds the SLA. The OUT packets from upstream represent *out of profile* traffic, which the network is not provisioned for. Therefore, there is no reason for a cascaded tagger to re-mark an OUT packets into an IN packet. However, an IN packet could be marked to be OUT if the upstream domain has contracted too little service profile from the downstream domain, and the downstream domain can change

[3] 7 packets of 500 bytes each is 28,000 bits, divided by 5.6Mbps, is equivalent of 5ms queuing delay.

some IN packets to OUT packets as to meet the specifications of a bilateral SLA. The cascaded tagger marks packets probabilistically, just as TSW. In our simulation, the cascaded tagger has a target rate of 5Mbps.

4.5.2 Results

The throughput results from two simulations are listed in Table 4.5. The respective RTTs and target rates R_t for the five connections are listed in columns 2 and 3. The results for the controlled case are listed in column 4 and those for the compared case are in column 5. The simulation results show that if the cascade tagger has a service profile that is sufficient for all upstream traffic, then its impact on end-to-end TCP throughput is rather negligible.

To further confirm this, we traced the number of IN and OUT packets tagged at each individual taggers (columns 6 and 7 respectively). We then counted the number of IN packets which are re-marked by the cascaded tagger into OUT packets. We observe that out of the total 11857[4] IN packets that go through the cascaded tagger, only 158 are turned into OUT packets. The 158 IN-turned-into-OUT packets are randomly distributed among the five connections. Since the number of re-marked packets is significantly fewer than the number of OUT packets marked originally by the individual taggers in the upstream domain (column 7), the impact of a cascaded tagger is small.

We also simulated the above scenarios when the cascaded tagger has a target rate of 4Mbps, which is significantly less than the sum of all upstream individual profiles. The results of that simulation are listed in column 4 of Table 4.6. In this case, the limiting factor is the bottleneck link speed of 5.6Mbps and not the cascade tagger. Each connection still achieves a throughput similar to that listed in Table 4.5. Having an additional downstream tagger has no effect on the throughputs if the network is well-provisioned. This case is similar to a well-provisioned network where the contracted profile is less than the bottleneck link speed, in which case, a connection can go beyond its profile.

[4]The sum of all IN packets from upstream taggers, or 11858 packets, minus one, which was dropped at Link A-B.

Table 4.5: Results for cascaded domains: the controlled case and the compared case. Bottleneck link speed = 5.6Mpbs. The cascaded tagger has a target rate of 5Mbps. Used TCP-SACK

Conn #	RTT(ms)	R_t(Mbps)	Controlled Case (Mbps)	Compared Case (Mbps)	# of IN pkts	# of OUT pkts
0	30	1	1.30024	1.22604	2549	437
1	40	1	1.09194	1.17093	2450	323
2	50	1	1.09401	1.01913	2413	480
3	60	1	1.06269	1.107	2362	255
4	70	1	0.95335	0.932333	2084	193
Total			5.502227	5.455433	11858	

However, one should realize that if there is additional congestion in the downstream, then the under-contracted profile will have an impact on individual connection's end-to-end performance. We simulate this scenario by changing the bottleneck link speed (Link C-D) from 5.6Mbps to 4Mbps; the results are listed in column 5 of Table 4.6. In this case, the link speed becomes a limiting factor, and the case is similar to that of an under-provisioned network where neither connection can achieve its target profile.

We should note here that when downstream congestion happens, other competing traffic sharing the bottleneck bandwidth of 5.6Mbps, then the re-marking by the cascaded tagger will have an effect on upstream traffic. We study this case in Section 4.6.2.

Table 4.6: Cascaded tagger case, when the network is under-provisioned

Conn #	RTT(ms)	R_t (Mbps)	The link C-D is 5.6Mbps but the cascaded tagger is 4Mbps	When the network is under-provisioned. Link C-D = 4Mbps
0	30	1	1.20459	0.874846
1	40	1	1.17409	0.794778
2	50	1	1.00739	0.740870
3	60	1	1.0069	0.835148
4	70	1	0.954569	0.734181
Total			5.347539	3.979823

4.6 Aggregated Profiles

In this section, we study how a Diff-Serv domain allocates bandwidth when traffic is aggregated from a number of TCP sources. Aggregated traffic has different characteristics from that of an individual TCP connection. Usually, aggregated traffic is more stable, and does not have the pronounced sawtooth swings typical in a TCP connection. Aggregated taggers are taggers located at routers where traffic has already been mixed. In particular, we are interested in studying the distribution of bandwidth among connections, and how aggregated traffic taggers can control the individual traffic rate. We study these two questions in two separate sender-based simulations.

4.6.1 Taggers in the Center of the Network: Aggregated Traffic

We use the topology presented in Figure 4.8. The simulation setup is similar to that in section 4.2.2 except now the taggers are for a number of hosts instead of an individual host (Figure 4.1). There are two taggers, one tagger is for aggregated traffic from hosts 0, 1, 2, 3 and 4, and the other is for packets from hosts 5, 6, 7, 8 and 9. The topology is used to study behaviors of individual connections when there is an aggregated tagger for a number of TCP connections.

The bottleneck link A-B has a link speed of 5.6Mbps. The two aggregated taggers are configured with target rates of 4Mbps and 1.5Mbps, respectively. Hosts 0 - 4 share the aggregated tagger 1 (4Mbps), and hosts 5-9 share the aggregated tagger 2 (1.5Mbps). We compare two scenarios. In the first scenario, hosts 0-4 each has a TCP connection, sharing aggregated tagger 1; simultaneously, only host 6's TCP is active and using tagger 2. The results are listed in column 4 of Table 4.7. In the second scenario, all hosts have active TCP connections. The first five TCPs (from hosts 0-4) share tagger 1, and the second five TCPs (from hosts 5-9) share tagger 2. The results are listed in column 5 of Table 4.7.

When we apply a service profile and a tagger for aggregated traffic, the marked packets are distributed over the n flows which are currently sharing the same service profile. Since

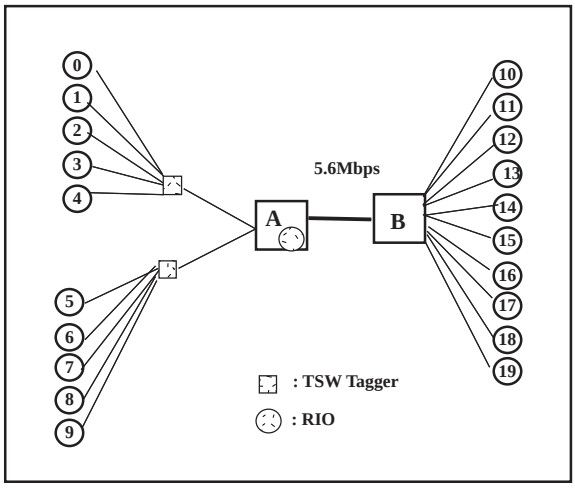


Figure 4.8: DiffServ Domain with Taggers for Aggregated Traffic

Table 4.7: Taggers for aggregated traffic. Topology 4.1. Bottleneck link A-B = 5.6Mbps. Parameters for RIO routers are (12, 30, 0.02) for INs and (2, 15, 0.5) for OUTs

Conn #	RTT (ms)	R_t (Mbps)	Scenario 1 individual rate (Mbps)	Scenario 2 individual rate (Mbps)
0	20		1.14091	1.19158
1	40		1.0318	0.891620
2	50		0.960004	0.787156
3	70		0.643260	0.575159
4	100		0.392509	0.564507
Subtotal		4	4.168483	4.01
5	20		N/A	0.443117
6	40		1.41626	0.347406
7	50		N/A	0.320943
8	70		N/A	0.235642
9	100		N/A	0.232686
Subtotal		1.5	1.41626	1.57979
Link Total		5.6	5.584743	5.58979

aggregation allows more statistical multiplexing and better link utilization, the service profile at edge routers can be better utilized than when there is only one flow using the profile. On the other hand, since the tagged packets are distributed over n flows, the aggregated tagger loses the fine control it has when tagging only one connection, therefore, bandwidth

allocation among all flows will be quite similar that in the current Internet.

This is what we have observed in our simulation. First, among the connections that share the same tagger and service profile, the distribution of the bandwidth is primarily according to the RTT of each connection, just as in the current Internet. For example, in scenario 1 (column 4), as the RTT increases, the achieved throughput decreases. Connection 4, with an RTT of 100ms, achieves 0.392Mbps, whereas connection 0 (RTT=20ms) achieves 1.14Mbps. Similarly, in scenario 2, within each service profiles, the bandwidth is shared primarily according to the RTT of each connection.

Second, the effect of having a tagger and a service profile is to ensure the connections achieve the contracted share of bandwidth when other *out-of-profile* traffic is present. For example, in scenario 1, connection 6 has a 1.5Mbps service profile, and achieves 1.416Mbps. Had connection 6 not have such a profile, it would have achieved a throughput comparable to that of connection 2 (1.01318Mbps), since they have the same RTT of 40ms. Combining these two effects together, we observe that in scenario 2 (column 5), tagger 1 ensures all its five connections achieve an aggregated rate of 4.01Mbps, and tagger 2 ensures that all its five connections achieve an aggregated rate of 1.5Mbps. Within each service profile, however, the bandwidth allocation is according to the RTTs of individual connection.

Third, the aggregated service profile is better utilized by an aggregation of flows. For example, in Scenario 2, when the second aggregated profile is used by five simultaneous TCP connections, the overall throughput (1.579Mbps) is higher than it is used by one individual TCP (1.41Mbps, a little less than the target rate). This is because aggregation leads to greater statistical multiplexing, which leads to better service profile utilization.

The aggregated case here is very simple: we only consider aggregation of five TCP flows. By no means, this is representative of Internet aggregation traffic. However, we think the simulation has demonstrated the essential points for the aggregated profile case. We think as the aggregation becomes larger, the real question is what would be the right profile for an aggregated traffic of n flows, each with some targeted rate. A simple "add'em

up" algorithm to calculate the aggregated profile will not be very efficient.

4.6.2 Combined Effect of Aggregate Taggers and Cascade Taggers

In this set of simulations, we study how different taggers—individual, aggregated, and cascaded—affect the performance of individual TCP connections. We compare the performance of five connections sharing one aggregated tagger with that of five connections, each with an individual tagger.

4.6.2.1 Setup

- Topology

  We did two simulations, one controlled case, and one compared case. The two topologies are depicted in Figure 4.9 and Figure 4.10, respectively. In the controlled case (Figure 4.9), connections 0-4 traverse through two adjacent domains, and therefore, two taggers, one individual tagger (on the access link) and one aggregated tagger (in router B). Connections 5-9 share an aggregated tagger (in router C) in this scenario. In the compared case (Figure 4.10), the set up for connections 0-4 is the same. However, in this case, connections 5-9 each has an individual tagger.

  In both cases, each access link (from host to the first router, either A or C) is an Ethernet link, with 10Mbps. The speed of links between routers are shown in the figures. Connections 0-5 each has an individual profile of 1Mbps, and the aggregated profile for them is 5Mbps, enforced by the aggregated tagger at router B. The bottleneck in both cases is the link between routers D and E, which is 10Mbps.

  In the controlled case, router C has an aggregated tagger of 5Mbps, shared among five connections 5-9. In the compared case, connections 5-9 each has an individual tagger of 1Mbps, same as that of connections 0-4.

- RIO Parameters

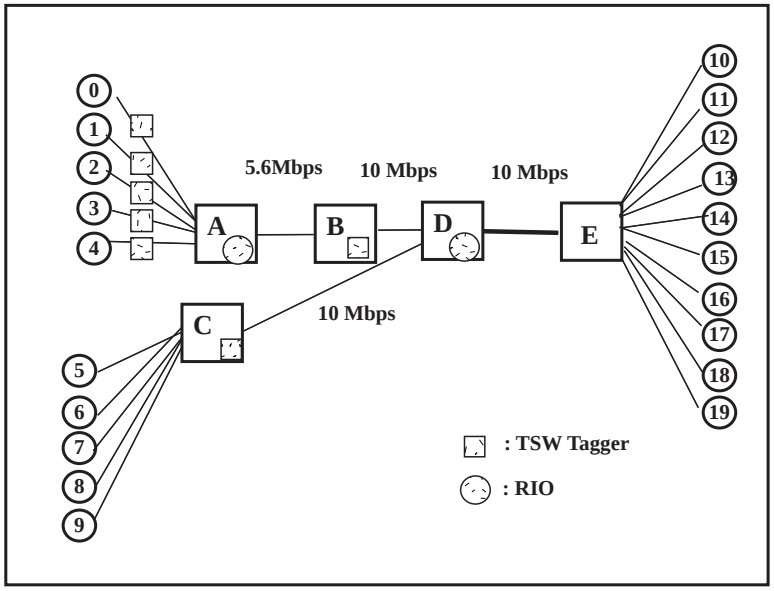


Figure 4.9: Compare Aggregated Taggers With Individual Taggers: Controlled Case

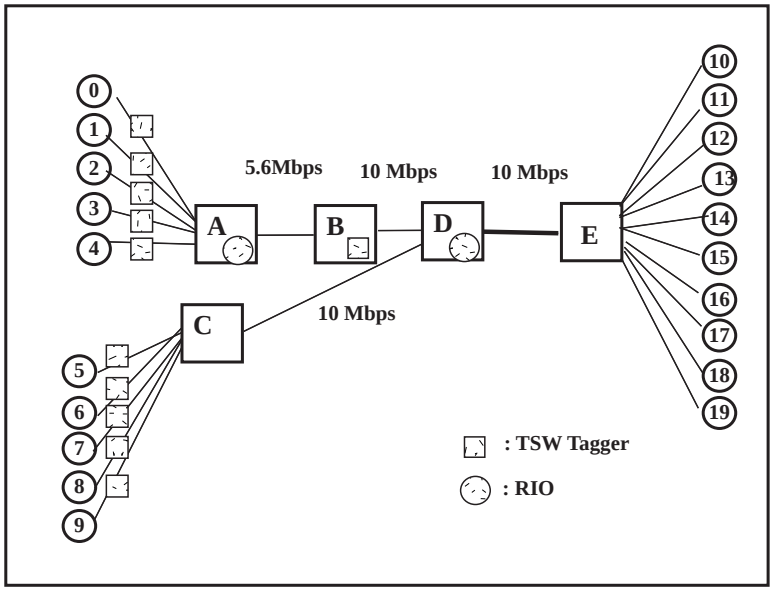


Figure 4.10: Compare Aggregated Taggers With Individual Taggers: Compare Case

In both the controlled and compare cases, parameters for the RIO algorithm in router A are (7, 14, 0.02) for INs—7 packets for min_in, 14 packets for max_in and 0.02 for P_{drop_in}—and (5, 10, 0.5) for OUTs. On a 5.6Mbps link, an average queue of 7 packets is equivalent of 5ms of average queuing delay. In both cases, the parameters for the RIO algorithm in router D are (7,18,0.02) for INs, and (5, 15, 0.5) for OUTs. TCP packet size is 500 bytes. We use TCP-Sack.

- Cascaded Tagger

  The cascaded tagger at router B uses a TSW algorithm. The algorithm differs slightly from the one presented in Chapter 3: it estimates the average rate from IN packets only, and when this estimated rate exceeds the contracted target rate R_t, it tags IN packets into OUT packets. In other words, the cascaded tagger does not mark upstream OUT packets, but may turn IN packets into OUT packets if the aggregated traffic exceeds the SLA. The cascaded tagger marks packets probabilistically. In the simulation, the cascaded tagger has a target rate of 5Mbps.

4.6.3 Results

The results from the above two cases are tabulated in Table 4.8. Results from the controlled case, where connections 5-9 share an aggregated service profile, are listed in column 4. Results from the compared case, where connections 5-9 each has an individual service profile, are listed in column 5.

Performance of connections 0-4 is fairly consistent in both the control and the compared cases. Each achieves a rate close to the 1Mbps target rate subscribed in their service profile. The more interesting results come from the bottom half of the table, where throughputs of connections 5-9 differ considerably when they have different taggers. When the tagger is for aggregated traffic (the controlled case, column 4), the network bias against long RTT connections is very visible. In contrast, when the taggers are for each individual connection, such bias is ameliorated.

Table 4.8: Aggregated taggers and cascaded taggers. Bottleneck link speed= 10Mpbs. Used TCP-SACK

Conn #	RTT (ms)	R_t (Mbps)	Aggregated Tagger Achieved Bw (Mbps)	Individual Tagger Achieved Bw (Mbps)
0	30	1	1.02679	1.09818
1	40	1	0.943344	1.02829
2	50	1	0.949632	0.979580
3	60	1	0.929313	1.00836
4	70	1	0.901795	0.950515
subtotal			4.750874	5.064925
5	30	1	1.46652	1.06329
6	40	1	1.1965	1.00333
7	50	1	0.936831	0.980113
8	60	1	0.831452	0.876147
9	70	1	0.752801	0.855813
subtotal			5.184104	4.778693
Total			9.934978	9.843618

There are two things worth noting. First, a careful trace of the IN and OUT packets for both cases lead us to draw the same conclusion as in the previous section (Section 4.5): an additional cascaded tagger does not have a significant impact on the traffic performance, as long as the cascaded tagger has sufficient profile for all upstream traffic. Second, in the controlled case, connections 5-9 are sharing one aggregated service profile and the aggregated tagger can not regulate individual TCP flows very well. TCP window graphs show less frequent, but more drastic window swings. In contrast, connections 0-4 each has an individual tagger, drops at regular intervals, and the individual taggers regulate TCP window swings very well. This can be explained as follows. When a connection has an individual tagger, the rate estimation is sensitive to the rate change in this individual connection, hence, the tagging mechanism is sensitive as well. For example, when the connection exceeds its target rate, the tagger is likely to tag packets as OUT immediately. This way, the overall feedback mechanism—individual TSW tagger and RIO—is responsive. In contrast, when a connection is aggregated with other traffic before reaching a tagger, statistical mul-

tiplexing of all flows will *dampen* bursts from individual flows, so the aggregated tagger is not sensitive to any particular individual flow. Therefore, the overall feedback mechanism will not be as accurate as that when there are individual taggers.

4.7 Non-responsive Connections

4.7.1 Non-responsive Connections in a DiffServ Network

Non-responsive connections refer those that do not have any congestion avoidance mechanisms and do not slow down when their packets are dropped at the routers[25]. In the current Internet, when non-responsive connections are present, TCP, or any transport layer protocol that implements congestion avoidance mechanisms, is at a disadvantage. While TCP backs off upon detecting congestion, non-responsive connections will get their packets through while continuing to cause congestion.

There has been research work done on 1) limiting the effect of non-responsive connections on TCP connections (or congest-control compatible transport-layer protocols); 2) detection mechanisms of non-responsive connection so network can penalize them, so as to provide proper incentives to a fair network resource utilization. Towards the first goal, John Nagle in [46] proposed a *Fair Queuing* mechanism. *Fair Queuing*, and the later, more elaborated work by Demers et. al.[13], propose to queue the connections (or flows) separately, thus, any given packet sees only the queuing delay created by packets in the same connection and is shielded from potential damaging effect from other connections. Fair queuing and its variations are mechanisms to *isolate* non-responsive connections. Towards the second goal, Floyd[25] proposed algorithms to study the drop statistics of the RED algorithm, thus identifying and detecting non-responsive connections.

In this section, we study the effect of non-responsive connections in a DiffServ environment. Our proposed mechanisms utilize the congestion feedback loop between routers and end host TCP. When a connection exceeds its contracted service profile, its traffic is marked by TSW taggers as a mixture of IN and OUT packets. When the network is congested, RIO

routers either send implicit signals to end hosts by dropping packets, or they send explicit signals to end hosts by marking packets as *having experienced congestion* (ECN bit). If the transport layer protocol in end host implements congestion control algorithms like TCP, the end host slows down, and eventually to a point below its contracted service profile. Then its traffic is marked only as IN. However, if the transport layer protocol in the end host is not congestion control compatible, then it will not respond to the congestion signals from the network. Therefore, the traffic is still a mixture of IN and OUT packets. This means that the drop statistics from RIO, especially those of the OUT packets, are indications of the flows which are not responsive to congestion signals from the network. Thus, in a DiffServ environment, RIO drop statistics provide a mechanism to identify and detect non-responsive connections. Conceivably, one could make use of this mechanism as a basis for penalizing or policing the non-responsive flows.

4.7.2 Setup

We use a similar topology that is depicted in Figure 4.1 (the sender-based scenario). As shown in Figure 4.11, ten hosts are connected to ten peer hosts. They share a common link between routers A and B. Each host (i) has a TCP connection to its peer host ($i+10$). There are altogether 10 TCP connections. The ten connections are of different Round Trip Time (RTTs). They can be divided into five groups. The two connections in each group have the same RTTs, but different target rates (R_t). Each of the five groups has a different RTT from another other groups. The RTTs for them are 20ms, 40ms, 50ms, 70ms and 100ms, respectively. Parameter configuration is very similar to that in the sender-based scenario as well. The bottleneck link speed is 33Mbps and the RIO parameters are set to be (40, 70, 0.02) for IN packets, and (10, 30, 0.5) for OUT packets. This is equivalent of setting (2.4ms, 7.2ms) as the respective min and max thresholds for OUT packets on a 33Mbps link, and setting (10ms, 17ms) as the respective thresholds for IN packets. TCP packet size is 1000 bytes and we use TCP-SACK.

We use a constant bit rate (CBR) source to model non-responsive sources since a CBR

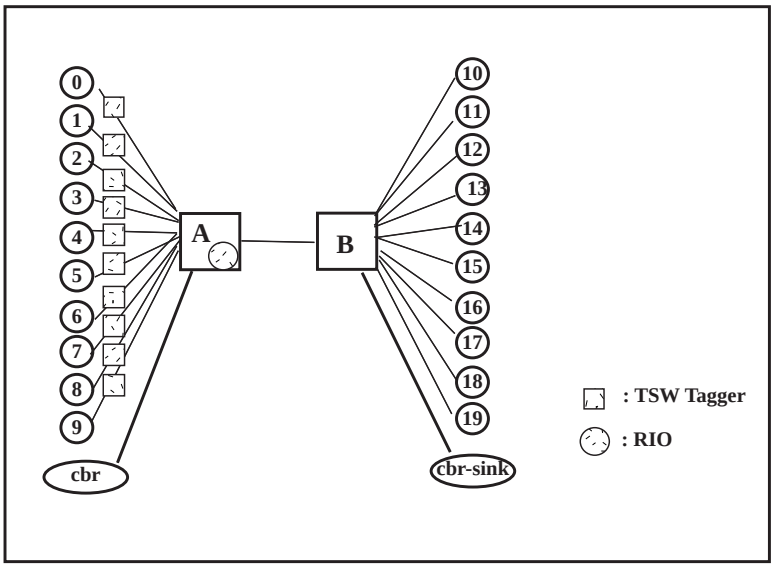


Figure 4.11: In Presence of Non-responsive Connections

source does not have a congestion control mechanism. We add a CBR connection to the above scenario, with a sending rate of 6Mpbs, or roughly 20% of the total bandwidth. The achieved throughput of the CBR connection is calculated by creating a receiver sink (CBR-sink) and calculated the received packets over time.

4.7.3 Results

We run two separate simulations. In the first, we simulate the current Internet environment. We have 11 connections (10 TCP connections and 1 CBR connection). We use RED gateways and there are no DiffServ router mechanisms. The results are listed in column 3 of Table 4.9. In the second simulation, we simulate a DiffServ domain. We use the same 11 connections. In addition, we implement RIO and TSW algorithms in interior and edge routers, respectively. The results are listed in column 5 of Table 4.9.

In today's Internet, a non-responsive connection causes persistent congestion in routers, and TCP connections with congestion control algorithm will back off as a result. Column 3 in Table 4.9 lists this effect: the CBR connection gets almost all its packets through, and

all TCP connections have a drastic performance degradation, compared to what they could have achieved when connection 11 is not present (column 3 in Table 4.2). In contrast, in a DiffServ network (column 5), connections with service profiles are protected from non-responsive connections: the link bandwidth is allocated according to the contracted service profile while packets from CBR are severely dropped. The CBR connection receives 1.78Mbps or 29.6% of its 6Mbps sending rate in our framework, vs. 5.85Mbps or 97% of its sending in today's Internet.

Table 4.9: 10-connection case with a non-responsive connection (CBR). BW= 33Mpbs, CBR is sending at 6Mbps. RIO parameters: (40, 70, 0.02) for INs and (10, 30, 0.5) for OUTs. Used TCP-SACK

Conn #	RTT(ms)	Current Internet (Mbps)	R_t (Mbps)	with RIO-TSW (Mbps)
0	20	5.40752	1	1.21924
1	20	5.36329	5	5.87978
2	40	1.98478	1	1.1372
3	40	2.56938	5	5.15273
4	50	2.443	1	0.989687
5	50	2.54567	5	4.99475
6	70	1.28912	1	0.837817
7	70	1.53377	5	4.86143
8	100	1.1074	1	0.720575
9	100	1.50127	5	4.72234
CBR	50	5.85338	0	1.78195
Total		31.59858	30	32.297499

4.8 TCP-DiffServ Mechanisms

The previous section concludes our simulations using router mechanisms— RIO and TSW— only. We explore different aspects of a Diff-Serv architecture: sender-based vs. receiver-based, aggregated profiles and cascaded profiles. We also explore using different versions of TCP with RIO-TSW: TCP-reno (sender-based scheme), and TCP-SACK. We conclude that while RIO/TSW alone can create service differentiations to end host TCP connections,

such differentiations are neither precise nor fair. The reason lies in TCP itself. TCP's window increase algorithm is not fair, TCP's congestion control mechanisms are not robust enough and are not aware of the upper-layer policy of SLA.

In this section, we explore the effect of applying both the TCP-DiffServ mechanisms and RIO+TSW. Unlike previous sections, where we go in depth for each individual scenario, in this section, we choose to compact our simulations somewhat and focus our discussion on TCP-DiffServ mechanisms.

4.8.1 Setup

We use a simulation topology similar to that in the previous sections, in which six TCP connections are sending to six respective receivers. We also use long-lived FTP/TCP connections, with two different RTTs: 80ms and 30ms. We feel that we have understood the effect of RTTs on TCP performance so we will not explore this dimension any more. The 80ms connections represent long-RTT connections and the 30ms connections represent short-RTT connections. Between the first pair of sender and receiver (host 0 and host 6), there is also a CBR connection that sends for a period of time. The only bottleneck is the link between R3 and R4, which is set at 8Mbps. All the other links are 10Mbps and are not bottlenecks.

Each simulation run has four different phases. The first phase is the *start-up* phase in which all six FTP/TCP connections reach their respective operating points. The second phase is a *congested* phase, in which, a constant bit rate (CBR) connection starts, running at $1/4$ of the bottleneck bandwidth, or 2Mbps. This causes heavy congestion in the router and TCP connections back off during this phase. The third phase is the *recovery* phase, in which the CBR source stops and all FTP/TCP connections recover to their respective operating points. The fourth phase is the *over-provisioned* case, during which, one of the FTP/TCP connections (TCP1) stops sending and the available bandwidth is shared among the rest five FTP/TCP sources. Each individual phase lasts for 25 seconds. All packet sizes are set to 1000 bytes. We use TCP-Reno, and receiver windows are large enough to not be

a constrain on the congestion windows.

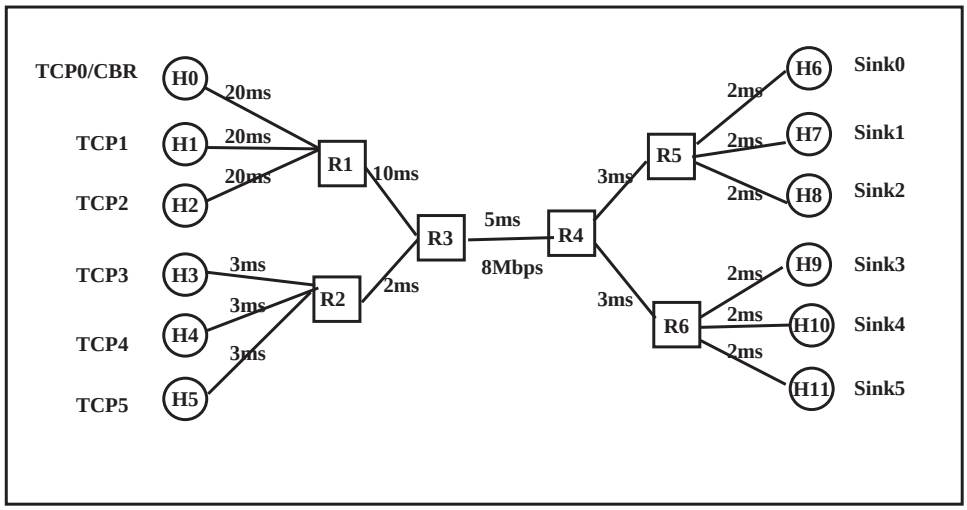


Figure 4.12: Topology for Applying DiffServ TCP Mechanisms

Table 4.10: Configurations of TCP connections

	RTT (ms)	R_t (Mbps)
TCP0	80	2
TCP1	80	2
TCP2	80	1
TCP3	30	1
TCP4	30	0.6
TCP5	30	0.6

The parameters for RED and RIO gateways are set comparably. The bottleneck speed is 8Mbps. The low threshold (min_th) for RED is the byte-equivalent of 5ms of queue delay, the high threshold (max_th) is the byte-equivalent of 10ms of queuing delay, and the dropping probability P_{max} is 0.1. Since the packet size is 1000 bytes, 5ms of queue delay on a bottleneck link of 8Mbps is equivalent of 5 packets.

We set parameters for RIO comparably: (5, 10, 0.5) for OUT packets, and (10, 20, 0.02) for IN packets. This is equivalent to (5ms, 10ms) queuing delays for the respective thresholds for OUT packets, and (10ms, 20ms) for the IN packets. We use tables to represent the

time-averaged throughput of three representative connections during different phases. Each setup is run three times with a different random seed, and the data presented in the tables are averages of the three runs. For each scenario, we show the throughput of 1) a long-rtt FTP/TCP (with and without a target throughput of 2Mbps); 2) a short-rtt FTP/TCP (with and without a target throughput of 0.6Mbps); and 3) a CBR connection with sending rate at 2Mbps during the congested phase. The constant c in TCP's window open-up algorithm is chosen to be 100, which is equivalent of increase one packet each 100ms.

The total allocated throughput to TCP connections is 7.2Mbps, or 90% of the bottleneck link. The details of the simulation set up are listed in Table 4.10.

4.8.2 Impact of TCP-DiffServ Mechanisms

Table 4.11: Comparison of Diff-Serv mechanisms applied to routers and endhost TCP; Modified TCP = standard TCP + three TCP-DiffServ mechanisms. All measured in Mbps

		Start-Up Phase	Congested Phase	Recovery Phase	Over-provision Phase
Standard TCP+RED (Scenario1)	TCP0 (80ms, no target)	0.676768	0.491638	0.723149	0.832894
	TCP3 (30ms, no target)	1.622382	1.126404	1.585279	1.804911
	CBR		1.978168		
Modified TCP+RED (Scenario2)	TCP0 (80ms R_t=2Mbps)	1.86133	1.31369	1.81553	2.30319
	TCP3 (30ms R_t=1Mbps)	1.11268	0.84987	1.12360	1.42987
	CBR		1.92003		
Standard TCP +RIO+TSW (Scenario3)	TCP0 (80ms R_t=2Mbps)	1.43707	1.32511	1.40382	1.49129
	TCP3 (30ms R_t=1Mbps)	1.05836	0.90249	1.11443	1.37187
	CBR		1.78891		
Modified TCP RIO+TSW (Scenario4)	TCP0 (80ms R_t=2Mbps)	2.02678	1.89689	2.02658	2.36111
	TCP3 (30ms R_t=1Mbps)	1.04109	0.91049	1.04853	1.33992
	CBR		1.00350		

We separate the mechanisms into two groups: Diff-Serv mechanisms to be applied in the end hosts (combinations of all the mechanisms proposed in Section 3.4) and Diff-Serv mechanisms to be applied in the router (RIO and TSW algorithms). We consider four different scenarios: 1) standard TCP-reno algorithm with RED gateways; 2) Diff-Serv

enhanced TCP (incorporating TCP-DiffServ mechanisms) with RED gateways; 3) standard TCP with RIO and TSW gateways; and 4) Diff-Serv enhanced TCP with RIO and TSW gateways. Table 4.11 lists the results from four different scenarios.

Scenario 1 is our basis for comparison, representing the current *best-effort* model. It illustrates two well-known phenomena: 1) short-RTT TCP connections have advantage over long-RTT connections when sharing the same bottleneck (first body row vs. second body row); and 2) a non-congestion controlled source has a detrimental effect on TCP connections (second body column), where TCP0 and TCP3 throughput dropped by 30% when CBR starts. In this case, the CBR source gets almost all its packets through a RED gateway at the expenses of other TCP connections' throughput.

Scenario 2 illustrates the effect of the mechanisms incorporated into TCP. With configured knowledge of target throughputs, TCP could robustly recover to its target rate after packet losses. The proposed window open-up algorithm also corrects the bias against long-RTT connections, e.g., in the Start-up and Recovery phases, TCP0, with an *rtt* of 80ms, doesn't suffer from network bias and gets close to its allocate target rate (1.86Mbps or 93%). However, in the presence of a non-congestion controlled source, all TCP sources suffer, e.g., a drop in TCP0 and TCP3's throughput (30%) when CBR starts. The RED gateway is not capable in discriminating against an *out-of-profile* source.

Scenario 3 shows the results of applying only the router mechanisms. Compared to scenario 2, the RIO algorithm discriminates against *out-of-profile* sources to limit the detrimental effect OUT packets have on IN packets during congestion. In this case, the CBR source is getting 89% of its packets through vs. 96% of its packets in scenario 2. (The bottleneck link has enough available bandwidth to accommodate 50% of the CBR packets.) The service differentiation among TCP connections with varying RTTs is the most pronounced during congestion (body column 2). When the network is well-provisioned, the service discrimination effect of RIO is dampened by the TCP window algorithm. Short-RTT connections obtain most of the available bandwidth in the over-provisioned situation. In other words, when free of congestion, the innate TCP biases can override the targeted

bandwidth allocation created by the Diff-Serv mechanisms in routers.

Scenario 4 illustrates the effects of the mechanisms in both the end host TCP and routers. Compared to scenario 2, the improvement lies in the congested phase, in which the RIO algorithm is able to shield IN packets from the interference of OUT packets. In this case, the CBR source is able to get 50% of its packets through (body column 2), which is roughly what the router can accommodate besides all its pre-allocated resources. Compared to scenario 3, the improvement lies in allocating bandwidth according to each connection's profile regardless of its *rtt* and the network conditions. When the network is congested, each TCP receives close to its targeted throughput; when the network is well-provisioned, the allocation of extra available bandwidth is fair among all TCP connections.

In summary, we observe that by incorporating Diff-Serv mechanisms in endhosts, the combined scheme can allocate resources fairly, precisely and differentially among connections, regardless of network conditions. In fact, if the endhost TCP has incorporated the Diff-Serv mechanisms, the RIO algorithm in routers can be configured to create strong differentiation among classes of packets, therefore, more effectively shield traffic that within SLAs from those that are outside SLAs.

4.8.3 Impact of Individual TCP-DiffServ Mechanisms

In this section, we isolate the effect of each of the host mechanisms proposed. We start with scenario 3 of Table 4.11, which has standard TCP-reno implementations and has applied Diff-Serv mechanisms to routers (tagging and RIO algorithm), and we add each proposed mechanism to TCP. We denote the three proposed mechanisms with the following abbreviations: *WinAdj* for changing the window open-up algorithm with $c * rtt^2$; *Ssthresh* for configuring TCP's *ssthresh* with the target throughput; and *ECN* for incorporating differential ECN mechanisms into TCP. Table 4.12 lists the four stages of the progressive changes, by applying ECN, Ssthresh and WinAdj to TCP. The first stage corresponds to scenario 3 in Table 4.11, and the last stage corresponds to scenario 4 in Table 4.11.

The second stage shows an improvement over stage 1: all TCP connections gain more

Table 4.12: Comparison of individual endhost mechanisms applied to TCP

		Start-Up Phase	Congested Phase	Recovery Phase	Over-provision Phase
Standard TCP +Tagging+RIO(3)	TCP0 (80ms R_t=2Mbps)	1.43707	1.32511	1.40382	1.49129
	TCP3 (30ms R_t=1Mbps)	1.05836	0.90249	1.11443	1.37187
	CBR		1.78891		
TCP+ECN +Tagging+RIO	TCP0 (80ms R_t=2Mbps)	1.6858	1.6902	1.7006	1.8257
	TCP3 (30ms R_t=1Mbps)	1.2576	0.9696	1.2144	1.6694
	CBR		1.4154		
TCP+ECN + Ssthresh +Tagging+RIO	TCP0 (80ms R_t=2Mbps)	1.95429	1.81875	1.95571	2.08191
	TCP3 (30ms R_t=1Mbps)	1.08955	0.95229	1.08742	1.50265
	CBR		0.93761		
TCP+ECN +Ssthresh+WinAdj +Tagging+RIO(4)	TCP0 (80ms R_t=2Mbps)	2.02678	1.89689	2.02658	2.36111
	TCP3 (30ms R_t=1Mbps)	1.04109	0.91049	1.04853	1.33992
	CBR		1.00350		

bandwidth in all phases than they did in stage 1 and the CBR source gets less than it did in stage 1. This is because after incorporating ECN, TCP only reacts to packet losses at most once per round trip time, thus it is robust in reacting to congestion signals. However, notice that the short RTT TCP (tcp3) has an advantage over the longer RTT connection (tcp1) due to a bias in the window open-up algorithm.

The third stage shows an improvement over stage 2 as well. The effect of setting $ssthresh$ to gauge TCP's operating point is obvious: both TCPs, with different targeted rates, operate close to their respective targeted rate. Compared to stage 2, the higher target-rate TCP (tcp1) has improved bandwidths in all phases and operates close to its 2Mbps targeted rate; the lower target-rate TCP (tcp3) has lower bandwidths in all phases, and also operates close to its 1Mbps target rate. Thus, setting $ssthresh$ makes TCPs operate more precisely to its target rate. The effect of TCP's bias against long RTT connections is dampened in all phases where there is congestion (phase 2). However, such bias is quite visible when there is sufficient bandwidth for all TCPs (phases 1, 3 and 4). CBR, in this stage, can get only 47% of its bandwidth. Thus, when the ECN and Ssthresh mechanisms are applied, TCP can effectively operate at its targeted rate and is shielded from unresponsive

connections.

The final stage shows results after TCP has incorporated all three mechanisms. Compared to stage 3, the bias between the two TCP connections has disappeared during all phases. Especially in phase 4, where there is extra available bandwidth for all TCP connections, and there is no particular bias of one TCP against another. In this stage, the CBR source actually gets *more* packets through the gateway than it did in stage 3. This is due to the following subtle reason. We configure all TCP connections with a new window open-up algorithm using a constant of 100, which is equivalent to the window open-up rate of 1 packet per 100ms during the *linear increase* phase. Since 100ms is *longer* than the RTT of both TCP connections (30ms and 80ms), the new TCPs are *less aggressive* in opening up their windows than their counterparts before incorporating the mechanism. As a result, the CBR connection gains more bandwidth through the gateway. This is similar to the problem of deploying such fairness mechanisms in a heterogeneous environment [29]. We will come back to this point in Section 4.8.6.

4.8.4 Robust Recovery from Losses

This section focuses on the details of TCP's window behaviors before and after incorporating the Diff-Serv mechanisms. We illustrate the effects in Figure 4.13. The left graph shows TCP0's *cwnd* and *ssthresh* throughout the entire 100 seconds of simulation (scenario 1 setup in Table 4.11, in which TCP uses the standard Reno algorithm). The right graph shows TCP0's *cwnd* and *ssthresh* throughout time (scenario 4 in Table 4.11, in which TCP incorporates all three Diff-Serv mechanisms). The most pronounced and visible difference lies in how *ssthresh* is adjusted in the two graphs: in the left graph, the *ssthresh* adjustment is according to the perceived network conditions and can be drastic and unpredictable. For example, from time 25 to 50 seconds, when there is a CBR source keeping the networks in a congested state, the TCP sources usually detect this and run at a much reduced operating point. There are several cases in which *ssthresh* is adjusted multiple times, each for a packet drop within the same congestion window. (Not visible given the granularity of the

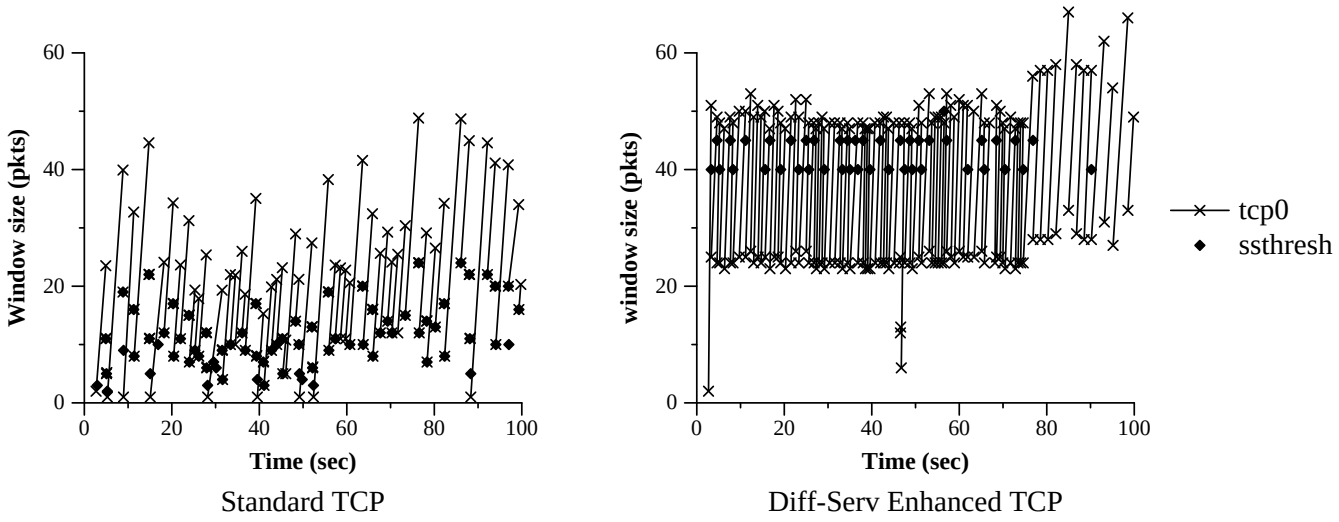


Figure 4.13: TCP Window Algorithm Before and After Incorporating Diff-Serv Mechanism

graph.) From time 80 second and onwards, the network is in a over-provisioned state, and the rate adjustments are infrequent and the *ssthresh* is high. In contrast, in the right graph, *ssthresh* is set by the targeted throughput, so after a packet drop, TCP's *cwnd* is reduced but not its *ssthresh*. Note that *ssthresh* is adjusted if the estimated RTT changes, because it is set to be byte-equivalent to the target-rate delay product. This is shown in the graph as a few discreet values of *ssthresh*: 40, 45 and 50 packets, etc. By keeping *ssthresh* near its target operating point, TCP can quickly recover from its packet losses and not be affected by worsening network conditions caused by non-congest control sources.

Another difference between the two graphs lies in the rate at which TCP adjusts its window, or the slope of each discrete segment of the TCP window adjustments. During the congestion control phase, the enhanced TCP uses a constant c of 100 in calculating its window increase rate and opens its window *slower* than its counterpart before incorporating the Diff-Serv mechanisms. The right graph appears to have a *faster* rate of increase than the left graph because most of the time. TCP operates in the *Slow Start* phase after a packet drop. This is because the *ssthresh* (set by multiplying RTT and target rate) is greater

than *cwnd*. The fact that TCP is operating in the *Slow Start* phase is also a sign that the TCP connection is not meeting its targeted throughput. In the left graph, TCP operates in the congestion avoidance phase after a packet drop because in the current TCP congestion control algorithm, both *ssthresh* and *cwnd* are re-adjusted after a packet drop.

4.8.5 Backward Compatibility

Among the above three proposed mechanisms, the first and second mechanisms require only the TCP sender to change its window adjustment algorithm, and does not require the receiver's cooperation.

The second mechanism needs some policy servers that keep information about SLAs, and additionally, a signaling protocol for communicating between the transport layer at the end host and the edge router if the profile is changing in real time. The information kept in policy servers is used to configure TCP with its initial *ssthresh* value and the *ssthresh* value after each packet drop.

The third mechanism requires TCP to be aware of the IN/OUT bit (TOS field) of the IP header. This mechanism can be deployed at the same time as the ECN field. The mechanism works as follows: a TCP sender always sends out packets with IN/OUT bit as *OFF*. A packet goes through a traffic conditioner, which in turn tags the packet's TOS field as either *ON* or *OFF*. A RIO and ECN capable gateway will mark packets differentially, and turn on ECN field for those packets if necessary. The transport layer at the receiver side has to copy both the ECN field and the TOS field of the IP header in the due acknowledgment packet. The sending TCP will react to a packet with both ECN and TOS bits (an IN packet) set differently from that with only ECN bit set (an OUT packet).

4.8.6 Heterogeneous Environments

Among the mechanisms we propose, the first mechanism has been studied in a context of improving fairness for TCP connections with varying RTTs [29]. One important problem

Table 4.13: Heterogeneous deployment of TCP mechanisms, measured in Mbps. **Mech1** is the fair window open up algorithm, **new** include all three mechanisms

		Start-Up Phase	Congested Phase	Recovery Phase	Over-provision Phase
Standard TCP+RED (Scenario1)	TCP0 (80ms, Reno)	0.676768	0.491638	0.723149	0.832894
	TCP3 (30ms, Reno)	1.622382	1.126404	1.585279	1.804911
	TCP5 (30ms, Reno)	1.541346	1.122553	1.610749	1.850088
	CBR		1.978168		
Mixed TCP algorithms +RED (Scenario2)	TCP0 (80ms, w/ mech1)	0.851694	0.499243	0.898498	0.90222
	TCP3 (30ms, w/ mech1)	0.950140	0.584215	0.792326	1.3719
	TCP5 (30ms, Reno)	1.893473	1.462454	1.845942	2.018788
	CBR		1.986283		
Uniform TCP algorithms +TSW+RIO (Scenario3)	TCP0 (80ms, R_t=2, new)	2.02678	1.89689	2.02658	2.36111
	TCP3 (30ms, R_t=1, new)	1.04109	0.91049	1.04853	1.33992
	TCP5 (30ms, R_t=0.6, new)	0.659625	0.533941	0.629245	0.969653
	CBR		1.00350		
Mixed TCP algorithms +TSW+RIO (Scenario4)	TCP0 (80ms,R_t=2, new)	1.984425	1.917876	1.991106	2.18545
	TCP3 (30ms,R_t=1, new)	0.993548	0.924187	0.991756	1.182006
	TCP5 (30ms, R_t=0.6, Reno)	0.602984	0.424578	0.591179	0.940206
	CBR		1.151985		

pointed out by [29] lies not in the algorithm itself, but its interaction with the standard TCP algorithm when they both exist in a heterogeneous network environment. As discussed before, the fair algorithm makes all TCP connections open up their windows at the same rate. With a chosen constant c corresponding to some standard unit of time, this algorithms makes any TCP connections with RTT shorter than the standard unit *less aggressive* than their current implementation, and any TCP connections with RTT longer than the standard unit *more aggressive* than their current implementations. As a result, if two TCP implementations co-exist in a heterogeneous network environment and their RTTs are both shorter than the standard unit of RTT, the connection with the current implementation will be more aggressive than the connection with the fair algorithm implementation. This takes away any incentives for people to deploy the fair algorithm. Of course, connections with RTT longer than the standard unit RTT will be more aggressive than their current implementation, and there would be an incentive for people to deploy such an algorithm.

The first half of the Table 4.13 illustrates this case. We include the results of another 30ms TCP connection (TCP5). Scenario 1 is the case when all TCPs use the standard algorithm and RED is used by routers as the queuing discipline. The two 30ms TCP connections have a clear advantage over the 80ms TCP connection, as expected from the current TCP window algorithm. Scenario 2 illustrates the case when TCP0 and TCP3 have upgraded to use the new and fair window algorithm, while TCP5 remains the same. The constant c is chosen to be 100, which makes both TCP0 and TCP3 *less* aggressive than their counterparts in scenario 1. We see that TCP0 and TCP3 achieve comparable results, (0.85Mbps and 0.95Mbps), whereas TCP5 has gained an advantage over both (1.89Mbps). TCP0 performs slightly better than its counterpart in scenario 1 (0.67Mbps), but TCP3 performs much worse (1.62Mbps).

Fortunately, we find that the Diff-Serv mechanisms in routers can be used to assist in such migration. We find that when the Diff-Serv router mechanisms are deployed first and TCPs incorporate all three proposed mechanisms, the allocation of bandwidth is according their respective SLAs (for those TCPs which have respective SLAs), and there is no clear

advantage for standard TCP over enhanced TCP. Scenarios 3 and 4 in Table 4.13 illustrate this. In scenario 3, all TCPs have upgraded to incorporate the Diff-Serv mechanisms, and the allocation of resources is according to their respective service profiles regardless the state of the network. When the network is over-provisioned, the available bandwidth is equally distributed among all connections. In scenario 4, TCP4 (not shown) and TCP5 both use the standard TCP window open-up algorithm. The results show current TCP algorithm has no clear advantage over the fair TCP algorithm in the Diff-Serv environment. This preserves the incentives for customers to update their TCP to incorporate the fair algorithm.

4.9 Testbed Implementations

The algorithms described in this thesis—RIO, TSW and TCP-DiffServ—have been implemented and evaluated in a testbed environment. In [58], Seddigh et al. reported implementing both RIO and TSW algorithms in an experimental testbed. They used a RIO algorithm configured with three classes of packets and three levels of assurances and use topology setups similar to those in our sender-based simulations. They verify our simulation experiments that service differentiation can be achieved when router algorithms are applied, but they are not certain to what extent the differentiations can be predicted.

As a follow-up work to the above [57], Seddigh et al. studied the five factors that impact throughput assurances for TCP and UDP flows in an experimental network. The five factors are RTT, number of flows in an aggregation, size of the target rate, packet size and presence of non-responsive flows. They show that in an over-provisioned network, all target rates are achieved regardless of the five factors, but in an under-provisioned network, none of the target rates may be achieved. The role of the RTT, target rate size and packet size play in determining throughput rates can be explained via the following equation [44]:

$$BW < \frac{packetsize}{RTT * \sqrt{target - rate}}$$

The effect of non-responsive flows (UDP) is similar to what we have experimented in sim-

ulation. They recommend using intelligent marking schemes that take in account packet sizes, target rate and RTTs.

Table 4.14: Effect of C in a testbed environment (throughput measured as Mbps)

	TCP0 (30ms)	TCP1 (80ms)
Both using Reno	2.7	2.0
C=50	2.7	2.0
C=100	2.5	2.3
C=200	2.4	2.4
C=500	2.3	2.6
C=1000	2.2	2.7

We have implemented the first two TCP mechanisms (window open-up algorithm and setting *ssthresh*) in a testbed. The testbed currently has edge routers that implement the TSW tagging algorithms, and a RIO algorithm with three dropping preferences, conforming to the Diff-Serv WG standard. The end hosts use Linux RedHat 2.3.39 distribution, which has the standard TCP-Reno algorithms. We incorporated the first two mechanisms in an end host kernel, and ran some initial test experiments. By the time of thesis submission, we have only conducted a few simple test experiments. This is on-going work.

In a simple test case to study the effect of constant *c*, we have two TCP connections, with the new mechanisms and the other without. Both share a 5Mbps bottleneck connection. The standard TCP (TCP0) connection has an RTT of 30ms, and the TCP connection with the new mechanisms (TCP1) has an RTT of 80ms. When both TCPs use TCP-reno, we observe a network bias against long-rtt connections (2.7Mbps for TCP0 and 2.2Mbps for TCP1). Then we configure TCP1 with increasingly large values of *c*, and therefore, an increasingly aggressive window open-up algorithm. We observe the effect of *c*. TCP1, with the new window algorithm, can gradually overcome the network bias. Eventually, the effect of an aggressive window open-up algorithm ($c = 1000$) is limited because the actual sending window is limited by the receiver's window, instead of the congestion window. The results are summarized in Table 4.14.

4.10 Conclusions

In this chapter, we apply two groups of mechanisms—router mechanisms TSW and RIO, and TCP-DiffServ mechanisms—to a DiffServ domain and evaluate the effectiveness of the mechanisms.

We observe that when we apply only the router mechanisms—RIO and TSW algorithms—, a Diff-Serv domain can create service differentiations among connections with different target rates. We have explored different aspects of this setup. We consider sender-based as well as receiver-based schemes. We find in a DiffServ domain with RIO and TSW, both the sender-based and the receiver-based schemes can allocate bandwidth according to the service profiles. We use the most widely deployed TCP, TCP-reno, as well as a more robust version, TCP-SACK. We find TCP-SACK is more robust in dealing with congestion signals from the network and can cope with the RIO/TSW algorithms better. We simulate and experiment with the effect of having cascaded service profiles on a single connection. We observe that as long as an additional tagger has sufficient service profile, it does not affect the end-to-end performance of TCP connections. If the additional cascaded tagger does not have enough service profile for all upstream traffic, then it turns some *in profile* traffic to *out of profile*, and consequently affects the end-to-end performance of the TCP connections. We also consider the case where a tagger is for an aggregation of connections instead of a single TCP connections. We find that a tagger for an aggregate of upstream traffic does not regulate the traffic as precisely as one for an individual connection. An aggregate tagger is effective in regulating traffic within its profile against other traffic regulated by other profiles. Finally, we experiment with the case where there are non-responsive flows together with TCP connections that implement congestion control mechanisms and have service profiles. We find that the router mechanisms in DiffServ can protect connections from being severely affected when non-responsive flows are present. The combination of TSW and RIO can also provide a means of identifying non-responsive flows by analyzing the dropping statistics of OUT packets, thus, making it possible to isolate a non-responsive flow in the middle of a congested network.

However, in all above cases, we find that even though the routers mechanisms can allocate services according to service profiles, the overall system is not fair or robust. The problem lies in TCP's congestion control mechanism itself, and cannot be corrected by changing router mechanisms alone.

We therefore study the effect of applying our three proposed TCP mechanisms (called TCP-DiffServ) to end hosts. We experimented with a few scenarios. We start with the current Internet scenario as the comparison case. We then apply either of the two groups of mechanisms individually—RIO/TSW and TCP-DiffServ. We find that when applying TCP-DiffServ mechanisms only, TCP connections will not be able to enjoy any service differentiation when the network is congested. In other words, although each TCP has a configured target rate, it does not get special treatment from the network. In the presence of non-responsive connections, TCP will not achieve its target rate. When only RIO/TSW mechanisms are applied, the overall system is not robust or fair enough. When we apply both groups of mechanisms, the overall system can allocate bandwidth in a robust, precise manner.

Finally, we describe testbed experiments by us and our collaborators on verifying those simulation results in an experimental testbed network.

Chapter 5

Conclusions and Future Work

5.1 Thesis Summary and Conclusions

The current Internet assumes the *best-effort* service model. In this model, the network allocates bandwidth among all the instantaneous users as best it can, and attempts to serve all of them without making any explicit commitment as to bandwidth or delay. Routers keep no state about end host connections, and when congestion occurs, routers drop packets. End host connections are expected to slow down and achieve a collective sending rate that is equal to the bottleneck speed. Therefore, what each connection achieves in terms of network bandwidth is determined by the network congestion state and the number of simultaneous connections sharing the same path, and is not always predictable.

As the Internet has transitioned from a research network to a commercial, heterogeneous network, three problems arise. First, an increasing number of real-time applications require some kind of quality of service (QoS) guarantees from the Internet than the simple *best-effort* service. Second, a heterogeneous user base has a variety of different requirements from the network and some are willing to pay to have their requirements satisfied, and the current Internet service model cannot offer a range of flexible services. Third, in a commercial network, Internet Service Providers (ISPs) have to find ways to charge for the service rendered and recuperate the cost of provision the network, and the current Internet

is missing mechanisms to account for network usage.

This thesis describes the *Differentiated Services*, a scalable architecture that can provide flexible services which address the above three issues. In the *Differentiated Services* architecture, or DiffServ, a network classifies packets into different classes, and gives differentiated service to different class of traffic. Network users can choose from the available service levels that best suited for their applications. They subscribe and pay for Service Level Agreements (SLAs) with their ISPs, and receive different services. What specified in an SLA is the expected service a user will receive and pay for. If the network is not congested, then the user can send traffic beyond its SLA. The architecture augments the current Internet devices—network routers and end hosts—and pushes the complexity of the system towards the edge of the network, therefore, is scalable. A variety of services can be constructed using the simple primitives provided by the DiffServ architecture, therefore, DiffServ offers very flexible services to users with different requirements. Pricing based on SLAs, instead of the actual usage, reflects the nature of Internet provisioning: mostly the Internet connections incur a fixed cost and the marginal cost of delivery only occurs when there is congestion. Thus, this kind of pricing structure can manage congestion, encourage network growth and recuperate cost without complex implementation.

After describing the architectural components of Differentiated Services, we proceed with a design framework in which different mechanisms can be implemented. We focus on the Assured Forwarding service model within DiffServ architecture and propose mechanisms to allocate bandwidths. This is a particular implementation of a general DiffServ architecture.

The current Internet accomplishes its bandwidth allocation by mechanisms in a congestion control feedback loop between network routers and end host TCPs. We propose a set of modifications to this congestion control feedback loop. We propose RIO, a differentiated dropping algorithm for interior routers of a DiffServ domain. We propose TSW (Time Sliding Window), a probabilistic tagging algorithm for monitoring and tagging packets at the edge routers of a DiffServ domain. Finally, we propose three modifications to TCP's con-

gestion control algorithm, collectively called TCP-DiffServ mechanisms. The mechanisms include 1) a change of TCP's window increase algorithm; 2) adjusting TCP state variable *ssthresh* to reflect the contracted SLAs; and 3) a combined use of TCP ECN mechanism and DiffServ codepoints to give accurate feedback of network conditions.

We use elaborate simulation experiments to evaluate the above proposed mechanisms. We observe that when applied router mechanisms (RIO and TSW) only, a Diff-Serv domain is able to allocate differentiated bandwidths according to the specified service profiles. However, routers mechanisms are not sufficient to overcome bias against long-RTT connections, which is a result of TCP's window increase algorithm. We then proceed to apply only TCP-DiffServ mechanisms to end hosts in a DiffServ domain. We find that while the enhanced TCP is robust and fair, in times of congestion or in presence of non-responsive connections, TCP connections with service profiles are not protected from those without. Since the current Internet allocates its resources using a congestion control loop completed with mechanisms in both routers and end hosts, and changing one without changing the other will not achieve an effective allocation scheme. Finally, we apply both router mechanisms and TCP-DiffServ mechanisms and conclude a DiffServ system could allocate resources in a robust and precise manner when both groups of mechanisms are applied.

5.2 Discussion and Future Work

This thesis is just the beginning of some interesting research directions which can be further pursued. We list them below.

5.2.1 End-to-end DiffServ

In DiffServ architecture, we see a clear divergence in functionalities of routers. There are two types of routers: edge routers and interior routers. In addition to providing traditional router functionalities like routing and forwarding, edge routers are now maintaining necessary state information (SLAs) in order to classify, monitor, tag, and police packets.

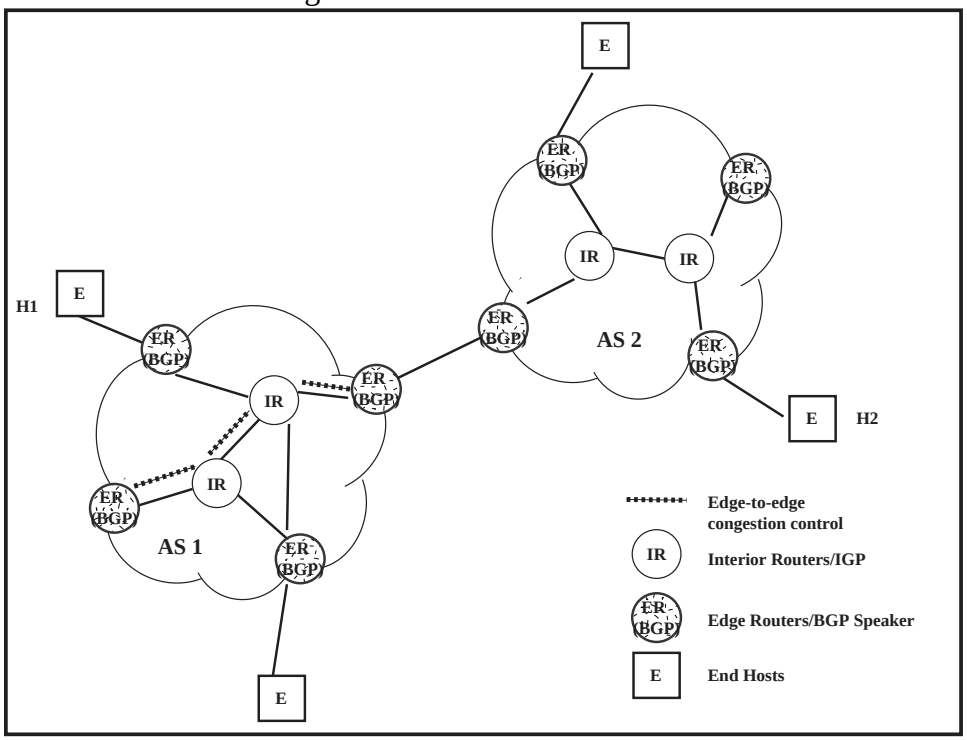
Similarly, interior routers have the additional functionality of creating differentiated service to different class of packets. However, the design of DiffServ architecture is to keep interior routers as simple as possible while keeping all necessary state information in edge routers. This is because the number of edge routers is relatively few compared to the possible interior routers in a network. (In contrast, the Int-Serv mechanisms need to affect all routers.) This design significantly simplifies DiffServ architecture and makes it scalable.

This kind of divergence in router functionality can also be founded in routing. The current commercial Internet is an arbitrary interconnection of Autonomous Systems (ASes). Within one AS, an interior gateway protocol (IGP) with a single routing metric is used. Between ASes, an exterior gateway protocol (EGP) [56] is used. The most widely accepted and deployed exterior gateway protocol is the Border Gateway Protocol (BGP) [53]. At the boundary of each AS, there are routers which are BGP speakers, and they exchange routing information with both BGP speakers in neighboring domains and interior routers in its own domain. Inside each domain, interior routers exchange routing information with its peers using some kind of interior gateway protocol (IGP), e.g., OSPF, RIP. The introduction of two-level of routing protocols—BGP and IGP—came when the Internet was growing to connect thousands of networks and millions of hosts, and a simple routing protocol is not sufficient to keep up with the growth. Separating routing functionality into two levels allows both autonomy and isolation among Autonomous Systems (ASes): each AS can choose an interior routing protocol and a routing metric that is best suited for itself without affecting others, so long as they can use BGP to exchange routing information. BGP is also designed with mechanisms to allow each AS to apply its own policies as whether to allow certain traffic traverse it or not.

One question we never explicitly pointed out in the thesis is where to place the edge routers and interior routers. The terms “edge routers” or “interior routers” do not necessarily refer to the actual placement in the network, they are simply terms describing the functions routers implement. However, putting DiffServ in perspective with other components of the Internet architecture, we suggest implementing edge routers in the BGP speakers

and implementing interior routers in the IGP routers within a single Autonomous System (AS)[1]. BGP speakers, in this case, will keep not only routing policies of a domain, but also the SLA specifications with its neighboring domains as well. IGP routers can adopt the mechanisms of DiffServ interior routers, creating differentiations among different classes. Figure 5.2.1 depicts a two-tier structure of DiffServ domains coincide with ASes.

Figure 5.1: End-to-end DiffServ



This begs the question of how to achieve end-to-end differentiated services? The set of mechanisms we proposed in this thesis—RIO, TSW and TCP-DiffServ—are just one possible set of mechanisms that can provide precise and robust bandwidth allocation for DiffServ. Just as there could be many IGP protocols, there could also be many combinations of mechanisms that can meet the requirements specified in SLAs. The elegance of having a two-tier architecture is to be able to de-couple intra-AS mechanisms from inter-

[1]AS, though originated as a purely routing concept, has become a synonym of Administrative Domain, which mirrors a real-world entity with administrative policies, e.g., a corporation, a university, etc.

AS mechanisms. One could conceive that a domain can achieve the same kind of SLAs using any of the alternative DiffServ approaches described in Chapter 2, or even using IntServ mechanisms since they maybe scalable within a domain. However, as long as the mechanisms within a domain can meet the specifications of SLAs, end-to-end differentiated services can be constructed by concatenating a series of SLAs. This matches routing very well.

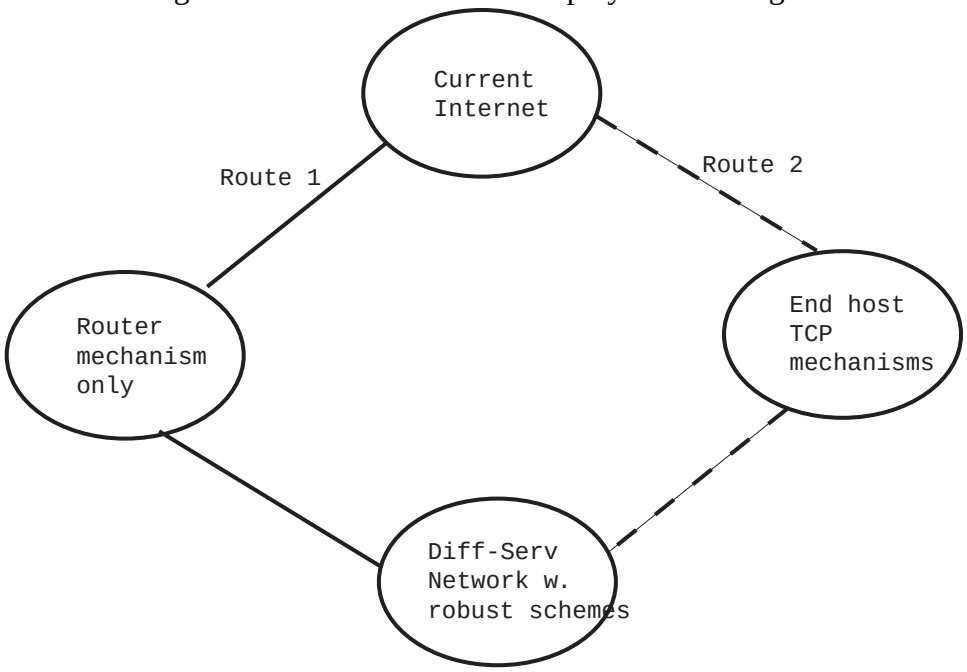
Just as end-to-end routing is done by concatenating a series of intra-AS routes, end-to-end differentiated services can be done by concatenating SLAs, each SLA specifies the edge-to-edge requirements between two neighboring ASes. Therefore, a series of AS domains, each meeting the specific SLAs specified, will be able to provide end-to-end DiffServ services.

Since there are already a number of proposals for intra-AS mechanisms like the ones we proposed in this thesis, the question of providing end-to-end differentiated services is then reduced to designing an inter-AS QoS protocol that is equivalent of BGP in routing. This is an interesting future research direction. At this point, we can only speculate what this protocol would look like. This protocol is very much inter-wined with the routing protocols (IGP and BGP) because any route changes will definitely affect the ability for a domain to meet its SLA. We could imagine an internal protocol to be used to convey congestion information within a domain among the edge routers. The protocol provides feedback between edge routers to adjust either 1) routes, or 2) traffic directed on any particular route in order to fulfill the requirements of SLA, or both. In essence, this protocol embodies what we call "edge-to-edge" congestion control mechanism, i.e., a congestion control loop operates between two edge routers of a DiffServ domain, and relay feedback from interiors routers to edge routers and back forth. At the inter-AS level, another protocol is needed to adjust the long-term committed SLAs between ASes.

5.2.2 Deployment Strategy

In this section, we offer some perspectives on deploying the mechanisms we proposed in a DiffServ domain. There are two possible routes to deploy both the router mechanisms (RIO+TSW) and end host mechanisms (TCP-DiffServ) that can migrate a domain from *best-effort* to a DiffServ domain. Figure 5.2.2 illustrates the two routes. In the first route (solid lines), ISPs deploy the router mechanisms first; then each individual users will upgrade their end host TCPs to adopt their respective TCP-DiffServ mechanisms. In the second route (dashed line), each individual users will first upgrade their end host TCPs to adopt the respective TCP-DiffServ mechanisms, and then, ISPs will deploy router mechanisms.

Figure 5.2: Possible DiffServ Deployment Strategies



There are practical as well as technical arguments for either route. Typically, in order for a mechanism to be adopted by industry, it has to go through IETF standardization process and be adopted by vendors. In the case of adopting router mechanisms, it is a relative simple matter because there are only a handful of router vendors, namely, Cisco, Nortel and

Lucent. It is conceivable that an ISP can choose to adopt a new version of router software which has incorporated the respective DiffServ mechanisms and upgrade its network in a matter of days. In terms of adopting end host TCP mechanisms, however, it is a bit more complicated because there are many variations of TCP implementations, offered by many vendors of end hosts. Even if an IETF working group has standardized some modifications to TCP, one shouldn't expect those modifications to be adopted by all implementations of TCP, nor should one expect those modifications to be deployed completely. Rather, one should expect that there will be a long time when the Internet has a great mix of old TCPs (TCP-Reno) and TCPs with TCP-DiffServ. This is the *partial deployment* problem.

We explore the *partial deployment* problem in Chapter 4, in which, we make the observation that if the DiffServ router mechanisms are deployed first, they would facilitate deployment of TCP-DiffServ because they offer incentives in performance for those users who want to adopt TCP-DiffServ mechanisms. On the other hand, if TCP-DiffServ mechanisms were deployed first, the end users might find them in a situation where the performance of their TCPs is *worse* than what they had before, depending on the value of the constant factor c in TCP's window increase algorithm. This can happen when the new TCP's window increase algorithm is *less* aggressive than that of the older, existing TCPs when the two versions of TCPs have the same round-trip-time connections. This strategy of deployment would create business *dis*-incentives to adopt DiffServ. Therefore, based on the simulation results in this thesis, we would recommend route 1, i.e., adopting router mechanisms first and then adopting end host TCP mechanisms. With this route, we can gradually migrate the current Internet to a DiffServ network.

5.2.3 Interactions with Applications

While the IntServ research started by analyzing the requirements of applications and then proposed mechanisms in networks to support such requirements, DiffServ went through a different route. DiffServ started by designing simple, scalable network mechanisms that can provide different levels of services. While DiffServ has been quite successful in its

research and standardization effort, there is no research work to demonstrate that applications can actually take advantage of the services provided by DiffServ networks and end users can see a visible difference. Seamless integration of the network QoS services and applications will be the "Holy Grail" of QoS research effort.

There are a number of issues to consider. First of all, there has to be an interface between applications and network protocols like TCP/IP by which, the applications can convey the necessary QoS requirements to the network protocols. Second, the end host operating system will check whether such service requirement fits within its contracted SLA with the ISP. If such request is beyond the SLA, then such request will likely not be satisfied and the application should be informed of such. (There is still a chance that such request *can* be satisfied if the network is not congested.) If the request is within the SLA, then the applications can be assured at this point that such request can be satisfied. It is up to the end host operating system to adjust the amount of system resources (memory, CPU cycles, and the necessary target-rate that would configure this TCP connection, for example) dedicated to this application. Third, the end host must have a signaling protocol between itself and the immediate edge router. This protocol is to automate the process of updating SLA information between edge routers and end hosts. An update can happen because the customer explicitly requests a change of upper-layer contract or the ISP adjusts the SLA due to some unexpected network conditions, or the end host's operating system requests a short-term change of the SLA. If the application's request is beyond the long-term SLA between the customer and the ISP, the end host still has the option to send a request to temporarily increase the SLA for the duration of this application which might be granted by the edge router. Such handshakes can be accomplished by such signaling protocol.

Bibliography

[1] *Ns network simulator*. Available via http://www-nrg.ee.lbl.gov/ns/.

[2] BALA, K., CIDON, I., AND SCHRABY, K. Congestion control for high-speed packet switched networks. In *Proceedings of Infocom* (1990), pp. 520–526.

[3] BALAKRISHNAN, H., RAHUL, H., AND SESHAN, S. An integrated congestion management architecture for internet hosts. In *Proceedings of SIGCOMM '99* (1999), vol. 29.

[4] BRADEN, B., CLARK, D., AND SHENKER, S. Integrated services in the internet architecture: an overview. In *IETF RFC 1633*. IETF, 1994.

[5] BRAKMO, L., AND PETERSON, L. Tcp vegas: End-to-end congestion avoidance on a global internet. *IEEE Journal of Selected Areas in Communications (JSAC) 13*, 8 (October 1995), 1465–1480.

[6] CLARK, D. The design philosophy of the darpa internet protocols. In *Proceedings of ACM SIGCOMM* (1988), pp. 16–19.

[7] CLARK, D. D. Combining sender and receiver payment schemes in the internet. In *Interconnection and the Internet: Selected Papers from the 1996 Telecom Policy Research Conference* (Mahwah, NJ, 1996), D. W. Gregory L. Rosston, Ed., Lawrence Elrbaum Associates Publisher, pp. 95–112.

[8] CLARK, D. D. Internet cost allocation and pricing. In *Internet Economics* (1997), J. B. L. McKnight, Ed., MIT Press, pp. 215–253.

[9] CLARK, D. D., AND FANG, W. Explicit allocation of best effort packet delivery service. *IEEE/ACM Transactions on Networking 6*, 4 (1998).

[10] CLARK, D. D., SHENKER, S., AND ZHANG, L. Supporting real-time applications in an integrated services packet network: Architecture and mechanisms. In *Proceedings of ACM SIGCOMM* (Baltimore, 1992), pp. pp 14–26.

[11] COCCHI, R., ESTRIN, D., SHENKER, S., AND ZHANG, L. A study of priority pricing in multiple service class networks. In *SIGCOMM Proceedings* (Zurich, Switzerland, September 1991).

[12] COCCHI, R., SHENKER, S., ESTRIN, D., AND ZHANG, L. Pricing in computer networks: Motivation, formulation and example. *IEEE/ACM Transactions on Networking 1*, 6 (1993), 614–627.

[13] DEMERS, A., KESHAVE, S., AND SHENKER, S. Analysis and simulations of a fair queueing algorithm. In *Proceedings of ACM SIGCOMM '89* (Austin, Texas, 1989), pp. 1–12.

[14] DOVROLIS, C., STILIADIS, D., AND RAMANATHAN, P. Proportional differentiated services. In *Proceedings of SIGCOMM* (October 1999), vol. 29.

[15] FALL, K., AND FLOYD, S. Simulation-based comparisons of tahoe, reno, and sack tcp. In *Computer Communications Review* (July 1996), vol. 26, pp. 5–21.

[16] FANG, W. Building an accounting infrastructure for the internet. In *Proceedings of Globecom Internet '96 workshop* (London, England, November 1996).

[17] FANG, W., AND PETERSON, L. Tcp mechanisms for a diff-serv architecture. In *Submitted to INFOCOM 2001* (2000).

[18] FANG, W., SEDDIGH, N., AND NANDY, B. A time sliding window three color marker (tswtcm). In *Internet Drafts*. IETF, March 2000.

[19] FENG, W., KANDLUR, K., SAHA, D., AND SHIN, K. Understanding tcp dynamics in an integrated services internet. In *NOSSDAV '97* (May 1997).

[20] FENG, W., KANDLUR, K., SAHA, D., AND SHIN, K. Adaptive packet marking for providing differentiated services on the internet. In *Proceedings of 1998 International Conference on Network Protocols (ICNP '98)* (October 1998).

[21] FLOYD, S. Tcp and explicit congestion notification. *ACM Computer Communication Review 24*, 5 (October 1994), 10–23.

[22] FLOYD, S. Tcp and explicit congestion notification. In *Computer Communication Review* (October 1995), vol. 24.

[23] FLOYD, S. Issues of tcp with sack. Tech. rep., LBL, March 1996. ftp://ftp.ee.lbl.gov/papers/issues.sa.ps.Z.

[24] FLOYD, S. Ns simulator tests for random early detection (red) gateways. Tech. rep., http://www-nrg.ee.lbl.gov/nrg-papers.html, October 1996.

[25] FLOYD, S., AND FALL, K. Router mechanisms to support end-to-end congestion control. Tech. rep., LBL, 1997. available via http://www-nrg.ee.lbl.gov/nrg-papers.html.

[26] FLOYD, S., AND JACOBSON, V. On traffic phase effects in packet-switched routers. *Internetworking: Research and Experience 3*, 3 (1992), 115–156.

[27] FLOYD, S., AND JACOBSON, V. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking* (August 1993), 297–413.

[28] HEINANEN, J., BAKER, F., WEISS, W., AND WROCLAWSKI, J. Assured forwarding phb group. In *IETF RFC 2597*. IETF, June 1999.

[29] HENDERSON, T. R., SAHOURIA, E., MCCANNE, S., AND KATZ, R. On improving the fairness of tcp congestion avoidance. In *Proceedings of IEEE Globecom '98* (Sydney, 1998).

[30] HOE, J. Improving the start-up behaviors of a congestion control scheme for tcp. In *Proceedings of ACM SIGCOMM* (Stanford, CA, 1996).

[31] JACOBSON, V. Congestion avoidance and control. In *Proceedings of ACM SIGCOMM* (Stanford, CA, 1988).

[32] JACOBSON, V., NICHOLS, K., AND PODURI, K. An expedited forwarding phb. In *IETF RFC2598*. IETF, June 1999.

[33] JAIN, R. A timeout-based congestion control scheme for window flow-controlled networks. *IEEE Journal on Selected Areas in Communications SAC-4*, 7 (1986), 1162–1167.

[34] JAIN, R. Congestion control in computer networks: Issues and trends. *IEEE Network Magazine* (May 1990), 24–30.

[35] JAIN, R. Myths about congestion management in high speed networks. *Internetworking: Research and Experience 3* (1992), 101–113.

[36] JAIN, R., CHIU, D., AND HAWE, W. A quantitative measure of fairness and discrimination for resource allocation in shared systems. Tech. rep., Digital Equipment Corporation, 1984.

[37] JAIN, R., AND RAMAKRISHNAN, K. Congestion avoidance in computer networks with a connectionless network layer: Concepts, goals and methodology. In *Proceedings of IEEE Computer Network Symposium* (Washington D.C., April 1988), pp. 134–143.

[38] JAIN, R., RAMAKRISHNAN, K., AND CHIU, D.-M. Congestion avoidance in computer networks with a connectionless network layer. Tech. Rep. DEC-TR-506, Digital Equipment Corporation, August 1987.

[39] JAMIN, S., SHENKER, S., ZHANG, L., AND CLARK, D. An admission control algorithm for predictive real-time service. In *Proceeding of Third International Workshop on Network and Operating System Support for Digital Audio and Video*. San Diego, CA, 1992, pp. 73–91.

[40] JIANG, Y., KAUTZ, H., AND SELMAN, B. Solving problems with hard and soft constraints using a stochastic algorithm for max-sat. In *Proceedings of the 1st Workshop on Artificial Intelligence and Operations Research* (1995).

[41] LIN, D., AND MORRIS, R. Dynamics of random early detection. In *Proceedings of SIGCOMM'97* (1997).

[42] MACKIE-MASON, J., AND VARIAN, H. Economic faqs about the internet. In *Internet Ecnomics* (Cambridge, MA, 1997), J. B. L. McKnight, Ed., MIT press, pp. 27–63.

[43] MANKIN, A. Random drop congestion control. In *Proceedings of SIGCOMM '90* (Philadelphia, PA, September 1990), pp. 1–7.

[44] MATHIS, M., SEMSKE, J., MAHDAVI, J., AND OTT, J. The macroscopic behavior of the tcp congestion avoidance algorithm. *Computer Communication Review 27*, 3 (July 1997).

[45] NABIL SEDDIGH, B. N., AND PIEDA, P. Bandwidth assurance issues for tcp flows in a differentiated services network. In *Proceedings of Global Internet Symposium, Globecom '99* (Rio De Jeaneiro, Brazil, December 1999).

[46] NAGLE, J. On packet switches with infinite storage. *IEEE Transactions on Communications COM-35*, 4 (April 1987).

[47] PAREKH, A., AND GALLAGER, R. A generalized processor sharing approach to flow control in integrated services networks - the single node case. *IEEE/ACM Transactions on Networking 1*, 3 (June 1993), 344–357.

[48] PAREKH, A., AND GALLAGER, R. A generalized processor sharing approach to flow control in integrated services networks - the multiple node case. *IEEE/ACM Transactions on Networking 2*, 1 (April 1994), 137–150.

[49] PARTRIDGE, C. A proposed flow specification. In *IETF RFC 1363*, I. R. Editor, Ed. 1992.

[50] POSTEL, J. Internet control message protocol. In *RFC792*, I. RFC-Editor, Ed. 1981.

[51] POSTEL, J. Internet protocol. In *IETF RFC 791*, I. R. Editor, Ed. Information Sciences Institute, University of Southern California, 1981.

[52] POSTEL, J. Transmission control protocol. In *IETF RFC 793*, I. R. Editor, Ed. Information Sciences Institute, University of Southern California, 1981.

[53] REKHTER, Y., AND LI, T. A border gateway protocol 4 (bgp-4). In *IETF RFC*. IETF, 1995.

[54] RICHTEL, M., AND BRINKLEY, J. Spread of attacks on web sites is slowing traffic on the internet. In *New York Times*. February 2000.

[55] ROSE, O. The q-bit scheme: Congestion avoidance using rate-adaption. *Computer Communication Review* (April 1992), 29–42.

[56] ROSEN, E. Exterior gateway protocol (egp). In *IETF RFC*. IETF, 1982.

[57] SEDDIGH, N., NANDY, B., AND PIEDA, P. Bandwidth assurance issues for tcp flows in a differentiated services network. In *Proceedings of the Global Internet Symposium, GLOBECOM'99* (Rio de Janeiro, December, 1999).

[58] SEDDIGH, N., NANDY, B., PIEDA, P., SALIM, J. H., AND CHAPMAN, A. An experimental study of assured services in a differentiated services network. In *SPIE symposium on QoS Issues Related to the Internet* (Boston, November 1998).

[59] SHENKER, S. *Public Access to the Internet*. Pretice Hall, NJ, 1995, ch. Service Models and Pricing Policies for an Integrated Service Internet.

[60] SIDI, M., LIU, W., CIDON, I., AND GOPAL, I. Congestion control through input rate regulation. In *Proceedings of Globecom '89* (Dallas, TX, November 1989), vol. 3, pp. 1764–1768.

[61] STOICA, I., AND ZHANG, H. Lira: A model for service differentiation in the internet. In *Proceedings of NOSSDAV'98* (1998).

[62] STOICA, I., AND ZHANG, H. Providing guaranteed services without per flow management. In *SIGCOMM Proceeding* (1999), vol. 29, pp. 81–94.

[63] TURNER, J. New directions in communications (or which way to the information age?). *IEEE Communications Magazine 24*, 10 (1986), 8–15.

[64] WONG, L., AND SCHWARTZ, M. Access control in metropolitan area networks. In *Proceedings of ICC '90* (Atlanta, GA, April 1990).

[65] YANG, R. Scalable distributed router mechanisms to encourage network congestion avoidance. Master's thesis, Massachusetts Institute of Technology, 1998.

[66] ZHANG, L. Virtual clock: A new traffic control algorithm for packet switching networks. In *Proceedings of SIGCOMM '90* (Philadelphia, PA, September 1990), pp. 19–29.

[67] ZHANG, L., DEERING, S., ESTRIN, D., SHENKER, S., AND ZAPPALA, D. Rsvp: A new resource reservation protocol. *IEEE Network 7*, 5 (1993), 8–18.

[68] ZHANG, L., SHENKER, S., AND CLARK, D. D. Observations on the dynamics of a congestion control algorithm: The effects of two-way traffic. In *Proceedings of ACM SIGCOMM* (Zurich, Switzerland, September 1991).