EDGE-TO-EDGE CONTROL: A CONGESTION AVOIDANCE AND SERVICE DIFFERENTIATION ARCHITECTURE FOR THE INTERNET

By

David Harrison

A Thesis Submitted to the Graduate

Faculty of Rensselaer Polytechnic Institute

in Partial Fulfillment of the

Requirements for the Degree of

DOCTOR OF PHILOSOPHY

Major Subject: Computer Science

Approved by the
Examining Committee:

____
Shivkumar Kalyanaraman, Thesis Adviser

____
Christopher Carothers, Member

____
Biplab Sikdar, Member

____
Boleslaw K. Szymanski, Member

Rensselaer Polytechnic Institute
Troy, New York

December 2001
(For Graduation May 2002)

EDGE-TO-EDGE CONTROL: A CONGESTION AVOIDANCE AND SERVICE DIFFERENTIATION ARCHITECTURE FOR THE INTERNET

By

David Harrison

An Abstract of a Thesis Submitted to the Graduate

Faculty of Rensselaer Polytechnic Institute

in Partial Fulfillment of the

Requirements for the Degree of

DOCTOR OF PHILOSOPHY

Major Subject: Computer Science

The original of the complete thesis is on file
in the Rensselaer Polytechnic Institute Library

Examining Committee:

Shivkumar Kalyanaraman, Thesis Adviser
Christopher Carothers, Member
Biplab Sikdar, Member
Boleslaw K. Szymanski, Member

Rensselaer Polytechnic Institute
Troy, New York

December 2001
(For Graduation May 2002)

CONTENTS

LIST OF TABLES . . . viii

LIST OF FIGURES . . . ix

ACKNOWLEDGMENT . . . xiii

ABSTRACT . . . xiv

1. Introduction . . . 1
   1.1 Switched Network Architectures . . . 5
      1.1.1 PSTN . . . 5
      1.1.2 Internet . . . 6
      1.1.3 ATM . . . 9
         1.1.3.1 ATM QoS Building Blocks . . . 10
      1.1.4 IETF IntServ and DiffServ . . . 11
      1.1.5 MPLS . . . 14
   1.2 Edge-to-edge Control . . . 15
      1.2.1 Implementing Edge-to-edge Control . . . 17
      1.2.2 Virtual Circuits versus EC Trunks . . . 19
      1.2.3 Edge-to-edge Control versus Differentiated Services . . . 20

2. Survey of QoS in Packet Networks . . . 23
   2.1 Complexity . . . 24
   2.2 Fairness . . . 25
   2.3 Responsiveness . . . 27
   2.4 Scheduling . . . 28
      2.4.1 First-In, First-Out . . . 28
      2.4.2 Priority Scheduling . . . 28
      2.4.3 Generalized Processor Sharing . . . 29
      2.4.4 Weighted Fair Queuing . . . 30
         2.4.4.1 Implementation of WFQ . . . 31
      2.4.5 Other Schedulers and Their Relation to Edge-to-edge Control . . . 32
   2.5 Traffic Control . . . 33
      2.5.1 DECbit . . . 37
         2.5.1.1 Advantage of DECbit . . . 40
         2.5.1.2 Disadvantages of DECbit . . . 40

2.5.2 TCP Tahoe and Reno 40
2.5.2.1 Advantages of TCP Reno 42
2.5.2.2 Disadvantages of TCP Reno 42
2.5.3 Extensions to TCP 44
2.5.4 TCP Vegas 44
2.5.5 Other Implicit, Window-based, End-to-end, Congestion Avoidance Schemes 47
2.5.6 ATM ABR 47
2.5.7 TCP Trunks 50
2.5.8 Congestion Manager 51
2.5.9 End-to-end Closed-loop Differentiated Service 52
2.5.10 Other Traffic Control Schemes 53
2.6 Buffer Management 53
2.6.1 RED 54
2.6.2 FRED 55
2.6.3 ECN 56
2.6.4 Buffer Management Summary 56
2.7 Differentiated Services 57
2.7.1 AF and RIO 57
2.7.2 Virtual Wire and Expedited Forwarding 58
2.7.3 Core Stateless Fair Queuing 59
2.8 Performance Enhancing Proxies 61
2.8.1 TCP Rate Control 61
3. Congestion Avoidance 63
3.1 Differences between End-to-end and Edge-to-edge Control 64
3.2 Riviera 65
3.2.1 Accumulation: Riviera Backlog Estimation 65
3.2.2 Riviera Control Policy 69
3.2.3 Rate-Clocking versus Ack-Clocking 69
3.2.4 Round-trip Propagation Delay Estimation 73
3.2.5 Riviera Control Packets 75
3.2.6 Additional Riviera Details 76
3.3 Properties of Riviera 79
3.3.1 Terminology 79
3.3.2 Stability 79
3.3.3 Fairness 81

3.3.4 Steady-State Approximate Queue Behavior . . . 83
3.3.5 Riviera Slow Start Queue Length Bound . . . 87

4. End-system Control Building Blocks . . . 90
4.1 Weighted TCP Service Building Block . . . 90
4.2 Distributed Buffer Management . . . 93
4.3 Near Zero Congestive Loss TCP End-System Control Building Block . . . 94
4.4 UDP Congestion Control and Isolation of Denial-of-Service Attacks . . . 96
4.5 Other Potential End-system Control Building Blocks . . . 97

5. Per-trunk Service Building Blocks . . . 99
5.1 Bandwidth Allocation Basics . . . 99
5.2 Weighted Proportional Fair Service Building Block . . . 102
5.3 Guaranteed Bandwidth Building Block . . . 105
5.3.1 Guaranteed Building Block Implementation . . . 106
5.3.2 QLL Sample Simulation Results . . . 109
5.3.3 Quasi-Leased Line Startup Queue Length Bound . . . 110
5.4 Assured Bandwidth Building Block . . . 114
5.4.1 Assured Building Block Implementation . . . 116
5.5 Smallest First Assured Bandwidth Allocation . . . 119
5.6 Heaviest Weight First Assured Bandwidth Allocation . . . 119
5.6.1 All-or-none Assured Bandwidth Allocation . . . 121

6. Conclusion . . . 123

LITERATURE CITED . . . 125

A. Simulations . . . 136
A.1 Simulation Models . . . 136
A.2 Steady-State over CoS . . . 137
A.3 Single Bottleneck . . . 139
A.4 Steady-State over Multiple Bottlenecks . . . 141
A.5 Heterogeneous Delay . . . 145
A.6 Fairness under Variance . . . 149
A.7 Loss . . . 152
A.8 Staggered Starts . . . 156
A.9 Reverse Path Congestion . . . 157

A.10 Changing Round-Trip Propagation Delay 158
A.11 Weighted TCP Per-Flow Service Building Block 160
A.12 Distributed Buffer Management 161
A.13 Weighted Proportionally Fair Service Building Block 165
A.14 Guaranteed Bandwidth Building Block 169
A.15 Assured Bandwidth Building Block 170

B. Riviera Pseudocode 174
B.1 Ingress 174
B.1.1 Initialization 174
B.1.2 Receive Edge Control 176
B.1.2.1 Receive Edge Control: Round-trip Estimation 176
B.1.2.2 Receive Edge Control: Departure Rate Computation 177
B.1.2.3 Receive Edge Control: Loss 178
B.1.2.4 Receive Edge Control: Decrease 179
B.1.2.5 Receive Edge Control: Discard Old Control Packets 179
B.1.2.6 Receive Edge Control: Increase 180
B.1.2.7 Receive Edge Control: Update Counters 181
B.1.3 Send Packet 181
B.1.3.1 Send: First Packet Processing 182
B.1.3.2 Send: Timeout Processing 182
B.1.3.3 Send: Insert FETC Header 183
B.2 Egress 184
B.2.1 Initialization 184
B.2.2 Receive Packet 185
B.2.3 Receive Edge Control Packet 185
B.2.3.1 Receive Edge Control: First Packet Processing 186
B.2.3.2 Receive Edge Control: Ignore Old Control Packets 186
B.2.3.3 Receive Edge Control: Retransmitted Control Packet Processing 186
B.2.3.4 Receive Edge Control: RTT Window 188
B.2.3.5 Receive Edge Control: Loss Window 188
B.2.3.6 Receive Edge Control: Accumulation Algorithm 189
B.3 Service Building Blocks 189
B.3.1 Weighted Sharing Pseudocode 189
B.3.2 Assured Bandwidth Allocation Pseudocode 190
B.3.3 Quasi Leased Line Pseudocode 193

C. EC's Single Path Requirement: Load Balancing, Tunneling, and NAT 195

D. ERICA Extensions and Our TCP Rate Controller 197

D.1 Achieving Weighted Max-Min Fairness 197

D.2 Queue Control 198

D.3 Parameter Tuning 200

LIST OF TABLES

1.1 Differences between DiffServ and EC Building Blocks 21

2.1 Traffic Control Schemes 38

3.1 Constants and Variables Used in Analyses 79

4.1 TCP versus UDP Statistics 98

A.1 Simulation Parameters 136

A.2 Class of Service Steady-State Results 138

A.3 Fairness under Variance Simulation Results 151

B.1 Ingress Parameters in Equations and Pseudocode 174

B.2 Ingress Variables 175

B.3 Egress Parameters in Equations and Pseudocode 184

B.4 Egress Variables in Equations and Pseudocode 184

D.1 Mapping of ERICA ATM Variables onto EC Equivalents 198

D.2 ERICA-based TCP Rate Control Parameters 201

LIST OF FIGURES

1.1 Edge-to-edge Control Example Application: Multi-Provider Networks 2

1.2 Network Using Time Division Multiplexing Switches 6

1.3 Store-and-forward Packet-switched Network 7

1.4 Cell-switched Network 10

1.5 Two DiffServ-enabled Networks 12

1.6 Network Using Label Switching 15

1.7 The Private Network Customer's View of Edge-to-edge Control 16

1.8 Traditional versus Edge Control 17

1.9 Edge-to-edge Control over DiffServ 18

2.1 Simple Two Bottleneck Network 25

2.2 WFQ Implementation Using Two Heaps and Linked Lists 31

2.3 Congestion Avoidance Maintains the System Near the Knee 34

2.4 Constant-size Window Subjected to Bottleneck Capacity Changes 35

2.5 AIMD Allocation Graph 39

2.6 Phases of TCP Tahoe Congestion Control 41

3.1 Traffic Control Model 65

3.2 Accumulation Computation Timeline 66

3.3 Steady-State Dynamics 70

3.4 AIMD2ER Timeline 70

3.5 Queue Growth Prior to Negative Feedback 71

3.6 Riviera with Disabled Decreases versus Changing Capacity 72

3.7 Vegas and Riviera with Sudden Increase in Propagation Delay 74

3.8 Timeline Showing Control Packet Loss 76

3.9 Timeline Showing Data Packet Loss 77

3.10 Increase/Decrease Dynamics in Detail 78

3.11 Multi-hop Loop's Throughput Versus Number of Hops 83

3.12 Riviera Single Bottleneck Topology 84

3.13 Riviera Steady-State Queue Length Maxima versus Number of Control Loops 85

3.14 Slow Start Queue Length Maxima versus Number of Control Loops 89

4.1 Example TCP Rate Control Ingress 91

4.2 Weighted TCP Connections 91

4.3 Congestion Window versus Time for Weighted TCP Connections 92

4.4 Backlog Moved from Bottleneck to Edge 94

4.5 Distributed Buffer Management: FRED versus Riviera+FRED 94

4.6 Topology Used in Comparing Near Zero Loss Building Block 95

4.7 Buffer Management versus Near Zero Loss Building Block 97

4.8 TCP versus UDP with both Regulated by Riviera 98

5.1 User Utility Functions 101

5.2 Riviera with Weighted Accumulation Differentiation 103

5.3 Riviera Weighted Sharing With Additional Backoff 104

5.4 QLLs Through a Single Bottleneck 109

5.5 Bandwidth Allocation for QLL over Multiple Bottlenecks 110

5.6 Quasi-Leased Line Analysis Topology and Variables 110

5.7 Analytic and Simulated Max Queue Length vs. b 114

5.8 Behavior of an EC Trunk and QLL for $b = 0.5$ 115

5.9 Assured Capacity Allocation Uses Tighter Backoff When below Assurance . . 118

5.10 Assured Bandwidth Allocation for TCP versus UDP 119

5.11 Assured Bandwidth Allocation over Single Bottleneck 120

5.12 Assured Bandwidth Allocation over Multiple Bottlenecks 120

5.13 Assured Bandwidth Allocations during Oversubscription 121

5.14 Smallest First Assured Bandwidth Allocation with Oversubscription ($a_x = .3\mu, a_y = .8\mu$) 121

5.15 Heaviest Weight First Assured Service with Oversubscription 122

A.1 Class of Service Topology . . . 138
A.2 Single Bottleneck Topology . . . 140
A.3 Riviera Single Bottleneck Results . . . 140
A.4 Multiple Bottleneck Topology . . . 141
A.5 Case for k=8, m=1 with Reno, Vegas, and Riviera . . . 142
A.6 Multiple Bottleneck Results for m=5 . . . 143
A.7 Simulated Heterogeneous Delay Topology . . . 146
A.8 Riviera's Performance over the Heterogeneous Delay Topology . . . 147
A.9 Topology for Fairness under Variance Simulations . . . 150
A.10 Riviera and Reno versus Random Loss . . . 153
A.11 Small Buffer Topology . . . 154
A.12 Riviera and Reno with Small Buffers . . . 155
A.13 Staggered Starts . . . 156
A.14 Reverse Path Congestion Topology . . . 157
A.15 Reverse Path Congestion Results . . . 159
A.16 RTPD Jumps at 5 Seconds . . . 160
A.17 Weighted TCP Per-Flow Service Building Block Results . . . 161
A.18 Distributed Buffer Management Results with FRED Edges . . . 163
A.19 Distributed Buffer Management Results with FIFO Edges . . . 164
A.20 Weighted Building Block Variants with Small Weights . . . 166
A.21 Diverging With Heavy Weights and Min-of-the-last-k . . . 169
A.22 Weighted Building Block Variants with Heavy Weights . . . 170
A.23 Rate Limits For Examples with Heavy Weight . . . 171
A.24 Sample Simulation for g=95Mbps . . . 171
A.25 Quasi-Leased Line Multi-bottleneck Results . . . 172
A.26 Single Bottleneck Assured Bandwidth Allocation Results . . . 173
A.27 Oversubscription for Single Bottleneck Bandwidth Allocation . . . 173

D.1 ERICA Algorithm Modified for Weighted Max-Min Fairness 199

D.2 TCP Rate Control with 100 Connections and 100Mbps Bottleneck 201

ACKNOWLEDGMENT

To the stampeding dictator without whose cudgel this thesis would not exist, thank you Shiv.

To the noble legions, your marching orders are: more simulations, more analysis, more implementation. Thank you Josh Hort, Karan Gupta, Sung-Gon Lee, Sthanu Ramakrishnan, Kishore Ramachandran, Vijay Ramani, Roopesh Varier, and Yong Xia.

Whose inspiration guided the early stages and whose competition stirred me to arms, thank you Prasad Bagal.

To my committee for their guidance and patience.

And to my wife and parents.

ABSTRACT

This dissertation presents Edge-to-edge Control (EC), a new traffic control and service differentiation architecture for communication networks. Unlike traditional QoS architectures which require complex scheduling at bottlenecks, EC requires only that intermediate nodes provide FIFO queuing and a separate class for EC traffic. Similar to DiffServ, EC concentrates functionality at internetwork edges where existing or proposed building blocks can be employed for traffic control, measurement, policing, congestion-sensitive admission control, congestion-sensitive pricing, and differentiated services. Unlike DiffServ, EC operates by establishing closed-loop regulated traffic trunks between EC edges. This traffic trunking building block is called an *EC trunk*. EC trunks use congestion avoidance to push congestion back from intermediate network nodes distributing the congestion across EC edges where the smaller congestion problems are handled with more stateful and sophisticated methods. EC places *no new implementation or upgrade requirements* at bottlenecks, requires *no new packet format requirements* at the IP-level, and requires no upgrades from end-systems. We also propose *Riviera*, a rate-based congestion avoidance algorithm that implements congestion avoidance for EC trunks.

1. Introduction

Quality of Service (QoS) can roughly be defined as the problem of distributing scarce resources among a set of users or applications (web, streaming media, voice) in a manner that best suits their respective needs and willingness to pay. The Internet is *not* a QoS network. Instead, the Internet allocates network resources roughly equally in response to congestion.

In this dissertation, we introduce a QoS architecture, Edge-to-edge Control (EC). In some contexts *QoS architecture* has been taken to include functions from the management level down to the packet level. ATM[44] takes this approach. Other architectures intentionally leave functions unspecified to facilitate inter-operation with many existing and future building blocks. DiffServ[15] for example places few requirements on routing, signaling, user-network interfaces, monitoring, or other functions relegated to what we later call the control and management planes. Instead, DiffServ focuses on building blocks that sit in the path of packets. Such building blocks are said to operate in the *data plane*. In this introduction we consider *EC* as an architecture in the same sense as DiffServ, and therefore we discuss functions outside the data plane only when they are constrained by EC, interoperate with EC, or are needed for illustration.

To limit our topic to a single dissertation, we focus on an enabling traffic trunking building block named the *EC trunk* and discuss how it is used in combination with other building blocks to achieve *scalable* data plane operation over *heterogeneous* networks. Our building blocks potentially scale in the sense of high bitrates ($>$ 100 Mbps) with many customers aggregated across a moderate number of traffic trunks (e.g., multi-site private networks overlaid on a packet-switched network like the Internet). Our building blocks operate over heterogeneous networks in the sense of operating over many data link layers across many orders of magnitude difference in bandwidth and delay. We discuss a third and more difficult problem, *coordination*, where appropriate to the data plane but with the recognition that many daunting coordination issues remain for *all* QoS architectures.

To ameliorate scale and heterogeneity problems within the data plane, we propose to provide QoS via closed-loop traffic control. In the context of the Internet, closed-loop traffic control is usually viewed as an end-to-end problem: each source monitors loss or other characteristics of its own traffic, and then adjusts demand to avoid overloading the network. TCP's end-to-end traffic control already arguably masters the *scale* and *hetero-*

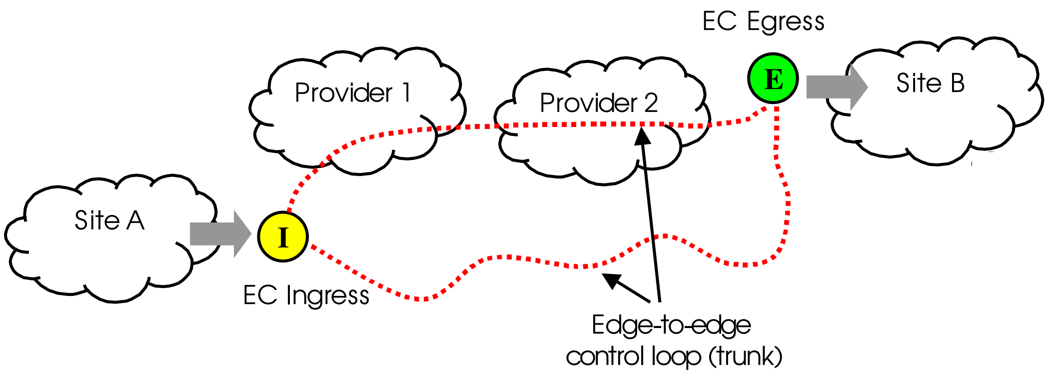


Figure 1.1: Edge-to-edge Control Example Application: Multi-Provider Networks

geneity of the Internet. At least TCP scales in the sense of approximating fairness and achieving high utilization for many connections, provided the connections have sufficiently long duration and the bottlenecks have enough buffer. TCP triumphs over heterogeneity in part by not imposing any functionality on Internet routers other than packet forwarding, low error rates, and infrequent packet reordering.

Unfortunately, the end-to-end model is unsatisfactory for providing QoS: *end-systems cannot be trusted.* For a network to provide bandwidth guarantees, providers must either overprovision such that worst-case end-system behavior never violates service-level agreements[1], or providers must isolate end-systems that misbehave. We study the latter alternative, because it usually requires far less bandwidth for similar quality of service. Isolating an end-system's misbehavior implies placing devices somewhere in the end-to-end path. Traditional QoS architectures take the extreme approach of isolating misbehavior of individual connections at every switch in the end-to-end path. We present examples of the scaling problems encountered with implementing isolation at every switch in Section 2.4.4.

Edge-to-edge control means literally to apply closed-loop traffic control between points within the network. These points, called *EC edges*, are placed so that each EC control loop encompasses the likely bottlenecks in the path, and are placed far enough from heavily-loaded bottlenecks to permit sophisticated and stateful QoS functions, while also far enough from end-systems to aggregate at logical administrative boundaries. For example, one possible deployment point is the enterprise access point (see Figure 1.1). By using closed-loop traffic control we also deal with the scale and heterogeneity of the

[1] Overprovisioning may be economical in an optical core, but it is not clear that such is true in access networks or multi-provider networks

Internet with similar skill as TCP. Misbehaving end-to-end flows subjected to EC are isolated at EC edges. EC is protected from misbehaving non-EC traffic by moving EC traffic into a separate class isolated at the bottleneck, for example, by using a separate queue for EC traffic.

As we discuss in detail later (see Section 1.2), due to limitations inherent in closed-loop traffic control, Edge-to-edge Control is only suitable for data or jitter-tolerant multimedia (not two-way-live). For reasons we discuss presently, we frequently use QoS for private data networks as an example EC application.

Edge-to-edge Control exploits three commonly occurring properties of private networks: *significant aggregation*, *limited number of sites*, and *moderate to long lifespan*. Aggregation arises in the sense that links in a private network are abstractly implemented as traffic trunks overlaid on top of a transit network (typically PSTN, ATM, or frame relay) or the Internet (in the case of VPNs). These traffic trunks *aggregate* all traffic passing between the sites joined by the private network. EC can be applied between the ends of the traffic trunk. As described later, EC obtains only loose control over queueing, delay, and jitter. In particular, EC's buffer requirements increase roughly linearly with the number of EC control loops and therefore EC requires aggregation to support many end-to-end control loops without experiencing congestive loss. Furthermore, with aggregation, overhead necessary to the operation of the trunk (e.g., control packets) amortizes across the set of active end-to-end flows. Because the sites connected by a private network are often corporate partners or branches of a single corporation, private networks may frequently contain less than thousands of sites. Less sites implies fewer trunks terminating at each site. Fewer trunks terminating at each site implies fewer traffic trunks maintained by each edge. We claim that private networks have *moderate (hours) to long (months, years) lifespan*, because corporate partners or branches do not change frequently. Thus any overhead in establishing a traffic trunk is amortized over the lifespan of the trunk.

To understand the services enabled by Edge-to-edge Control, we first cover previously implemented and proposed switched network architectures with respect to their abilities to provide QoS. Once we have discussed switched network architectures, we fully describe the Edge-to-edge Control architecture and its interoperation with the DiffServ architecture and contrast it with prior QoS architectures. In the following chapters we provide a representative survey of mechanisms used in QoS architectures for packet-switched networks. Then we define, analyze, and simulate Riviera, a congestion avoidance scheme that can be used to implement our traffic trunking building block.

In this introduction we discuss each architecture abstractly divulging details only where necessary to illustrate the different properties of each architecture. One should consider how each architecture compares in terms of *deployability*, *efficiency*, *fairness*, *protection*, *complexity*, and *responsiveness.*

- *Protection* refers to the isolation of each flow from the misbehavior of other flows.

- We use *efficiency* in two different contexts. For services that provide resource guarantees, we define efficiency as the number of flows supported for a given service commitment. For services that fairly divide resources, an efficient scheme achieves high utilization with low delay.

- *Fairness* refers to the ability of a system to divide excess network resources equitably between equally important traffic flows. We provide more precise definitions of fairness in Section 2.2.

- *Complexity* refers to implementation difficulty as well as computational overhead, control overhead, and state maintenance.

- *Responsiveness* is not one but several related metrics. *Response time* generally refers to the time between a request and completion of service. In our context, response time refers to the delay between the initiation of communication and the time the user receives contracted service. For systems that use closed loop feedback, we use the term *convergence time* to refer to the time it takes the system to reach a state of efficiency or fairly sharing capacity depending on context. We discuss the terms *convergence time* and the related concept of *transient response* in more detail in Section 2.3.

- *Deployability* contextually refers to the ease of introducing QoS mechanisms into a network. Architectures that require wide-scale upgrades are less deployable. *Incremental deployment* refers to introducing mechanisms slowly or in stages with benefit increasing throughout the deployment. Incremental deployability is paramount for the success of any QoS architecture in the context of an existing large heterogeneous internetwork such as the Internet.

1.1 Switched Network Architectures

We begin this discussion of communication architectures with the Public Switched Telephone Network (PSTN) and how its limitations led to the development of packet-switched and virtual-circuit-switched networks. Therein we discuss two prominent examples: the Internet as a packet-switched network, and Asynchronous Transfer Mode (ATM[44]) as a virtual-circuit-switched network. Both represent trade-offs that eventually led to the development of label-switching (MPLS) and packet QoS architectures namely Integrated Services (IntServ[55]), and Differentiated Services (DiffServ[15]).

1.1.1 PSTN

Telephone networks are engineered around the delivery of a single service: 3.5kHz analog calls with an average length of about four minutes. Over the last century, as telephone networks grew and technologies changed, rather than upgrade all phones, increasing *complexity* naturally appeared *within* the telephone networks. As an example, most telephones are comprised of not much more than a ringer, a microphone, a speaker and tone generators. The phones still provide analog signals while the telephone networks perform echo cancellation and analog-to-digital conversion. Within the digital core of the telephone networks, switches maintain the illusion of a contiguous physical wire by using Time Division Multiplexing (TDM). TDM requires that each switch must maintain per-call state. Heavy call volumes thus incur significant *overhead* to setup, tear down, and maintain state. Because the telephones are comparatively simple, the telephone network architecture is sometimes described as a *smart* network with *dumb* peripherals. As discussed in the next section, the Internet is a *dumb* network with *smart* peripherals. The relative lack of complexity within Internet networks leads to smaller upfront costs in providing Internet service that has created a competitive Internet Service Provider (ISP) market.

Part of the cost associated with providing data services between two parties directly over a telephone circuit can be blamed on TDM. TDM is *inefficient* in the sense that customers are constrained to a statically-provisioned bandwidth specified as a multiple of the bandwidth for a single voice circuit. This constraint applies even in off-peak hours when most of the capacity of a telephone network is idle.

Data applications also do not usually obey the four minute average period of communication. A single user may use the Internet for hours at a stretch, but communication

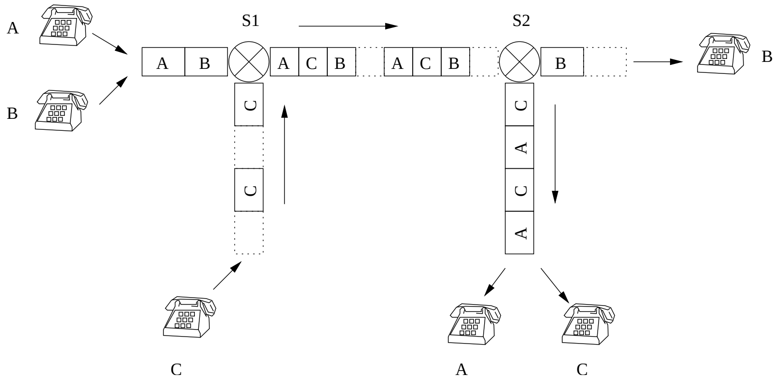


Figure 1.2: Network Using Time Division Multiplexing Switches

between any two pairs of computers is usually short and bursty. If all data communications occurred over end-to-end circuits, the cost of establishing and tearing down connections would prevent bursty applications from arising (e.g., the web). Where telephone circuits are used to carry data traffic between many parties, the circuits usually operate as single links within an overlying packet-switched network. Circuits are used in this manner because they are pervasive, reliable, provide consistent performance, and in some cases because they provide *protection* from the behavior of other parties.

We provide an example of TDM in Figure 1.2. As voice samples arrive at a Time Division Multiplexing (TDM) switch they are buffered and then retransmitted in the appropriate time slot on the appropriate outgoing link. Each switch determines a sample's "appropriate" time slot and outgoing link solely based on the time slot and link on which the sample arrives. In this example, the call volume underutilizes the telephone network and thus some time slots remain idle.

1.1.2 Internet

In the 60s Baran proposed store-and-forward packet-switched networks motivated primarily by their efficiency compared to TDM networks. Instead of establishing a circuit through a network by reserving TDM time slots at each link, packet-switched networks bundle each data into separately addressed units known as *packets*. Thus each switch (a.k.a., *router* in packet-switching terminology) forwards packets toward the destination specified in each packet. A *store-and-forward* network handles the simultaneous arrival of multiple packets destined for the same output link by storing the excess packets and forwarding them as the link becomes available (see Figure 1.3). Since packet transmissions are not constrained to periodic time slots, sources can adjust their send rates to utilize available capacity. Baran's ideas motivated design decisions for the ARPAnet built in 1969.

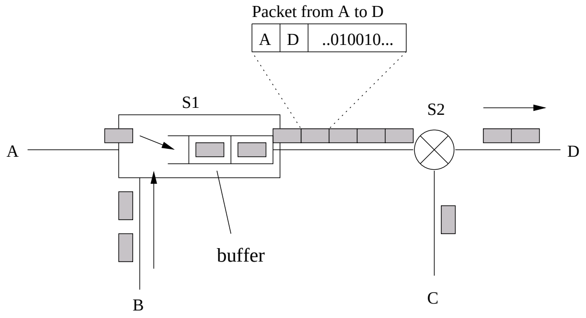


Figure 1.3: Store-and-forward Packet-switched Network

When the Department of Defense decided to interconnect the ARPAnet with other networks, the Internet was born [20]. DARPA mandated that the Internet architecture operate over existing networks with the top priority that communication must continue despite the loss of intermediate networks or routers [26]. Packet switching was used because the researchers had experience with the ARPAnet, because packet-switching well suits the intended applications such as remote logins and file transfers, and because the networks to be connected were packet-switching networks [26]. The ability to withstand the loss of networks or routers was interpreted as meaning that communicating parties should be able to continue communicating without disrupting application-level state despite an intermediate failure. As a result Internet routers do not notify the communicating parties when an intermediate network outage occurs, instead the network transparently reroutes packets to maintain connectivity. Any lost packets are eventually detected and retransmitted by the end-system. Furthermore no state regarding a particular communication is maintained in the network because this state would be lost if a router or network failed. Instead reliable, in-order delivery is maintained entirely at the end-systems using TCP.

In contrast to the telephone network architecture, the Internet is a *dumb* network with *smart* peripherals. With the existence of a telephone network, ISPs have and still use telephone circuits to create links between packet switches. The ISPs can then multiplex many customers over the same bandwidth. Because data communication is often bursty, several customers can share the same bandwidth without appreciable degradation in performance. Ergo more cost-effective use of bandwidth. The packet switches themselves are simple compared to telephone switches since most functions are handled by end-systems.

Routers do not have to maintain per-flow (i.e., call) state. Thus low-end routers are easier to engineer, less expensive to purchase, and cheaper to operate.

The relative simplicity of the network service, the initial willingness of users to put up with poor quality intermittent service, combined with the ability to overlay Internet links on an existing telephone infrastructure allowed ISPs to get off the ground with a relatively small initial investment. Consequently, the number of competitors in the market grew at an alarming rate during the mid-90's. The resulting low prices and battle for market share benefits consumers today at the expense of ISP profits. As the Internet has become economically critical, business users demand better service, Service Level Agreements (SLAs), and accountability. Each requirement increases complexity and introduces network externalities that favor larger service providers; a shake-out has been under way for some time.

Even with the user reliability requirements approaching those attributed to telephone networks, the Internet is still more cost effective relative to purely circuit-switched networks for providing data services. Assuming satisfactory reliability, low delay and low jitter can be achieved without resorting to per-call state at every switch, packet-switching may also become a viable and cost-effective alternative for toll-quality voice.

By removing *complexity* from the network, the Internet architecture scales better than the telephone network architecture, but removing complexity from the network complicates providing QoS. Intermediate networks are not required to provide any mechanisms that control the sharing of resources (buffers and bandwidth). This has the side-effect that fairness, efficiency, and convergence time are dependent on properly functioning end-systems. Thus the Internet provides *no protection* from a malfunctioning end-system that floods a path with packets. The problems are further exacerbated by the possibility that neighboring interconnected networks might be designed for different performance objectives. As a result one network might trade-off large delay variability to obtain reliability (e.g., X.25) and thus thwart attempts to provide a tight delay bound. Fortunately today, hardware in core transit networks has become so reliable that almost all loss, delay, and jitter (delay variance) is introduced by queuing [61] at Internet routers. As long as bottlenecks occur at Internet routers then both protection and delay problems can be solved by placing mechanisms at Internet routers that control each communicating party's access to buffers and bandwidth. This exact approach has been taken with the IntServ and DiffServ architectures as we discuss later.

1.1.3 ATM

Parallel to the development of the ARPAnet, Fraser [45] developed the Spider and then Datakit [46] networks that later evolved into Asynchronous Transfer Mode (ATM) [44]. These networks introduce the concept of a Virtual Circuit (VC). Virtual-circuit-switched networks are similar to circuit-switched networks in that a path is established at the beginning of communication. However, unlike with circuits in telephone networks, TDM slots are not reserved for each virtual circuit. Instead ATM networks store-and-forward short, fixed-length packets each bearing an id associating the packets with the appropriate virtual circuit (see Figure 1.4). In ATM nomenclature, these fixed-length packets are called *cells*. In order to avoid requiring every virtual circuit to use a globally unique id, a virtual circuit's id is allowed to change from link to link. To forward cells, each switch maintains a table that maps from a cell's virtual circuit id on the incoming link to the appropriate virtual circuit id on the appropriate outgoing link.

By switching cells rather than performing TDM, ATM and its predecessors gain the *efficiency* of packet-switched networks. ATM switches also implement schedulers and rate controllers at each switch that allow for *efficient*, *fair*, *responsive*, and *protected* allocation of resources. More efficient, fair, responsive, and protected than currently achieved with the control implemented only at the end-systems as in the traditional Internet. But ATM's benefits come with a cost, switches are *complex* in the sense that the system requires per-VC signaling and per-VC routing, and requires each switch to maintain per-VC state and implement mechanisms to enforce per-VC service commitments. As with telephone switches such complexity potentially limits scalability and increases cost. Also, as with telephone circuits, the cost to response time and communication overhead associated with signaling to setup a VC is not appropriate for short-lived flows (e.g., web traffic) unless there is a pre-existing VC. For some applications such as in backbone networks, VCs can be setup ahead-of-time for highly aggregated traffic thus creating an *overlay network* between the edges of the ATM network. In fact ATM VCs often aggregate IP traffic because ATM provides good traffic engineering and reliability features.

To avoid much of the overhead of ATM, these overlay networks often use ATM's Unspecified Bit-Rate (UBR) service which circumvents ATM's mechanisms for *fair*, *responsive*, and *protected* allocation of resources. Lastly ATM is not typically used end-to-end because deploying ATM hardware across multiple provider networks is intractable. The difficulty of getting all intermediate networks to agree on and then deploy a particular network architecture makes ATM far *less deployable* than the Internet (which simply connects existing networks).

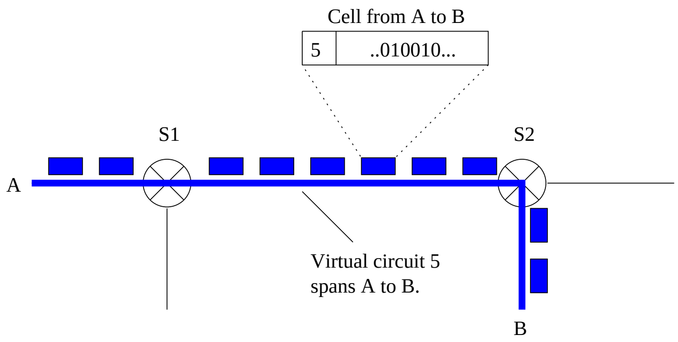


Figure 1.4: Cell-switched Network Where Cells from A to B pass through VC 5

1.1.3.1 ATM QoS Building Blocks

ATM from its inception provides six key QoS building blocks. The Internet has no widely-deployed equivalent to some of ATM's QoS building blocks:

Flow or Class Specification In ATM all QoS parameters are associated with the virtual circuit. Prior to definition of the DiffServ[15] byte and the IPv6 flow id field [31], the Internet lacked any mechanism for the explicit specification of the set of packets with the same desired QoS treatment. For QoS in the traditional Internet, each router must classify packets on a set of host id's, network prefixes, or port numbers. For the traditional Internet, aggregation of classification state sacrifices the ability to provide service differentiation for specific hosts or applications.

Stable Paths By design, virtual circuits provide paths that typically do not change within the lifespan of the virtual circuit. The traditional Internet allows for routes to change at any time without end-system notification. Further in the Internet, QoS is not integrated with the routing protocols thus when a path suddenly changes, routers along a flow's new path have no idea what resources have been allocated to the rerouted flow, and routers along the old path cannot immediately reallocate the rerouted flow's resources. The Internet's Resource reSerVation Protocol (RSVP [120]) partially mitigates this problem by having hosts periodically signal the network to refresh reservations. When a path changes, reservations along the old path eventually expire. Moreover, RSVP is not yet widely enabled even though many routers and host implementations support it.

QoS routing poses an alternative solution. *QoS routing* protocols are QoS-aware and thus can often move paths and QoS state without violating service commitments. However QoS routing requires a tight coupling between the QoS and routing subsystems, and adds significant state maintenance and communication overhead while increasing associated complexity. Coupling systems also complicates upgrades. QoS routing is not supported in most of the Internet's routing domains.

User Network Interface ATM provides a service interface specifying the parameters passed between the host and the network that characterize source behavior and specify the desired service. The Internet lacks an analogous service interface. IntServ provides a minimalist service interface [105], but IntServ is not widely deployed.

Signaling Mechanisms ATM provides mechanisms to signal desired reservations associated with each virtual circuit. For the Internet, RSVP is the only IETF-proposed signaling mechanism, but as already stated RSVP has not been widely deployed.

Admission Control In ATM all switches implement mechanisms for determining whether to allocate resources to a virtual circuit. The Internet historically allows any number of flows to pass through each router.

data plane Mechanisms Data plane mechanisms enforce resource allocations on a cell-by-cell (packet-by-packet) basis. ATM switches have provided necessary data plane mechanisms since ATM's inception. Only recently have Internet routers begun to implement schedulers and traffic conditioners capable of enforcing fair or weighted fair resource allocations. We describe some of these mechanisms in detail in chapter 2.

1.1.4 IETF IntServ and DiffServ

The Internet Engineering Task Force (IETF) has established two approaches to Internet QoS: Integrated Services (IntServ[55]) and Differentiated Services (DiffServ or DS[15]). Both approaches retain the Internet's packet-switching model and by default use existing best-effort forwarding. However, the two approaches differ in their distribution of functionality between network nodes. Integrated Services requires every router to perform per-flow classification, per-flow admission control, and per-flow scheduling. Differentiated services divides the labor between domain *edges* and *interior routers*. Interior routers perform simple classification, scheduling and buffer management algorithms that avoid per-flow or per-customer state. Per-flow or per-customer QoS functions occur only at edges (network boundaries) or end-hosts.

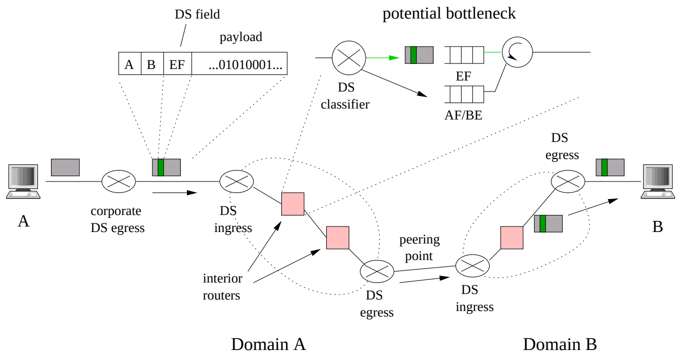


Figure 1.5: Two DiffServ-enabled Networks (domains A and B)

In IntServ end-systems use signaling (potentially with RSVP) to place per-flow reservations at each router along the path taken by a flow. Thus the state maintained at each router grows in proportion to the number of flows. This state is potentially large on backbone routers. To be fair, flows do not necessarily span between end-hosts. Flows spanning between some routers can aggregate state for flows between end-hosts. Even so, IntServ does not attach a flow id to each packet and thus as already mentioned requires every router to perform potentially complex classification based on sets of network addresses, prefixes, and/or port numbers.

With DiffServ, edges stamp each packet with a service mark (see Figure 1.5). The service mark alone defines the treatment that a packet receives in the interior and thus eliminates the need for interior routers to perform per-flow classification. The service mark is implemented as an id stamped into 6 bits in the IP packet header known as the DS field. In Figure 1.5, the DS field bears the EF id (we describe EF in more detail later in this section). Each id maps packets onto a particular Class-of-Service (CoS). The every day example of CoS is the postal system where mail stamped as priority mail is delivered more quickly than first class, which is in turn delivered more quickly than bulk mail. In DiffServ, the forwarding behavior expected at each interior router is defined as a class's Per-Hop Behavior (PHB). Each per-hop behavior is designed such that it can be implemented without requiring per-flow state in interior routers. In Figure 1.5, EF is implemented as a separate queue shared by all EF marked traffic.

We now go into more detail about the DiffServ architecture then we did with prior architectures because DiffServ is closely related to our proposed architecture, Edge-to-edge Control.

There are two types of PHBs that have received significant attention in the IETF: the Assured Forwarding (AF [52]) PHB group and the Expedited Forwarding (EF [58]) PHB. The Assured Forwarding PHB group defines several different levels of drop precedence. The AF PHBs approximately say that when overload occurs packets marked "out" should be dropped before packets marked as "in." In fact there are more than two levels of drop precedence but the idea remains the same. The original intent was that the AF PHBs would be used as a building block in implementing an "assured service" where each customer receives an expected bandwidth allocation [24] equal to the customer's "in" rate plus a fair share of the remaining capacity. This has proven difficult to achieve [102, 54] and so the community appears to have settled on a more "qualitative" set of services enabled by the AF building block in which those with more "in" marked packets receive a lower drop rate.

In terms of our primary metrics for judging QoS architectures, services based on the AF PHB can be *efficient* in the sense that sources can adjust their send rates to absorb any excess capacity. How well services based on AF perform along other metrics depends highly on the mechanisms used to implement AF-based services. We cover one set of proposed mechanisms in Section 2.7.1.

The Expedited Forwarding PHB approximately says that at every node the service rate must always exceed the arrival rate so that any EF marked packet transiting DiffServ nodes will experience tight jitter and delay. The EF PHB has two negative side-effects: 1) the EF traffic class must be provisioned for the worst-case, and 2) flows admitted to the EF traffic aggregate cannot grow to utilize any of the EF class's unused bandwidth. A network with only EF traffic is as inefficient as TDM; however, the IETF rightly assumes that EF traffic is not the only traffic on the network. Best-effort or other traffic classes can potentially absorb any capacity not currently being used by the EF class.

With DiffServ, services are built on top of PHBs by providing appropriate traffic conditioning at edges. Traffic conditioning functions include *marking*, *shaping*, and *policing*. Marking implies stamping a bit pattern in passing packets to specify the appropriate Per-Hop Behavior each packet will receive in the interior. Shaping means to delay packets until they conform to a particular traffic profile. Policing implies dropping packets to force a traffic flow to conform to a traffic profile. We describe broadly how these mechanisms are used to implement services based on the EF and AF PHBs in chapter 2.

Because IntServ and DiffServ are built on top of the Internet, they can operate across almost any data link layer including using ATM as a data link layer. Unlike ATM which requires ATM hardware everywhere within an ATM network, IntServ and DiffServ only require upgrades at potential bottlenecks. DiffServ also requires upgrades at network edges that are not yet DiffServ-capable. Networks edges are typically potential bottlenecks and would be upgraded in the context of IntServ as well. For DiffServ, network administrators then configure traffic conditioners at each edge and configure the PHBs at each bottleneck. As we discuss in Section 2.7.1 properly configuring the mechanisms implementing the AF building block has proven difficult.

In summary of DiffServ and IntServ, DiffServ requires less state than IntServ, ATM, or PSTN and hence less *complexity* at interior nodes. At least theoretically DiffServ should scale better than IntServ. Both IntServ and DiffServ are *efficient* in the sense of allowing flows to absorb available capacity. Though efficiency in the sense of the number of flows supported for a given service commitment depends highly on the mechanisms used to implement DiffServ and IntServ. Although the effort to deploy DiffServ is nontrivial, reduction in upgrades and complexity represent strides toward tackling the Three Principle Problems.

1.1.5 MPLS

Neither IntServ nor DiffServ directly provide mechanisms for associating resources with stable (pinned) paths. Multi-Protocol Label Switching (MPLS) [6] provides such functionality for the Internet. As packets enter an MPLS cloud, the ingress wraps each packet with zero or more small headers each containing a forwarding label. As a packet traverses nodes in the MPLS cloud, each node forwards all packets with the same outermost label to the same output. Optionally each node may add, swap, or remove these small headers to control how the packet is forwarded at downstream routers. By adding (pushing) the same label to the front of packets with differing inner labels, a router aggregates traffic in the sense that downstream routers maintain forwarding state only for the aggregate's label rather than for each flow in the aggregate. When a router removes the outermost label (popping), the router de-aggregates traffic (see Figure 1.6). A set of labels directing traffic along a path creates a *Label-Switched Path (LSP)*. MPLS's ability to arbitrarily aggregate forwarding state allows MPLS to pin traffic trunks to LSPs with per-label but *not* per-LSP state at every router. By avoiding per-LSP state at every router, MPLS scales better than ATM (which requires per-VC state at every switch in the path).

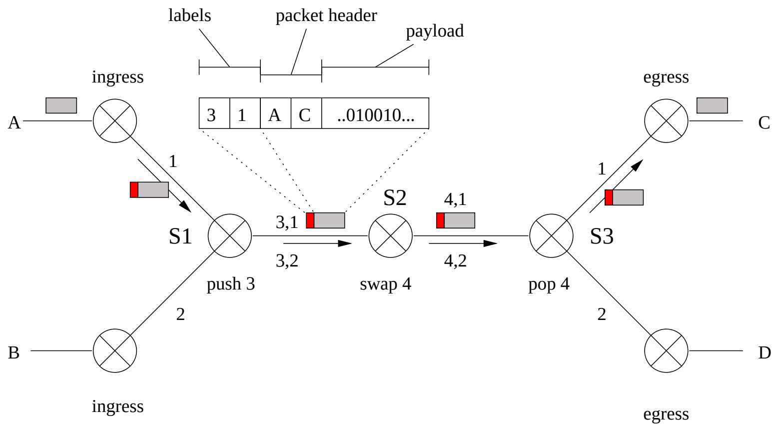


Figure 1.6: Network Using Label Switching

In Figure 1.6 packets from A to C pass through an MPLS cloud. As A's packets enter the MPLS cloud, the ingress pushes a label aggregating all traffic heading to C. Switch S1 pushes a label that aggregates traffic heading to C or D so that switch S2 only needs to maintain state for label 3 and *not* 1 or 2. At S2, label 3 is used for some other aggregate so S2 swaps all packets entering with label 3 onto label 4. S3 finally de-aggregates traffic heading toward C and D.

Together with MPLS, either IntServ or DiffServ can provide a service with a minimum bandwidth guarantee with low likelihood that a path will unexpectedly change causing a new path to become oversubscribed.

1.2 Edge-to-edge Control

Lastly we come back to our proposed data plane building block: *Edge-to-edge Control (EC)*. We then describe how it operates within the DiffServ architecture. Within the confines of our example application, a customer might view edge-to-edge control as a means for providing QoS for his or her private network. If we assume the underlying network is the Internet then the private network becomes a VPN with EC edges placed in between the customer and the ISP(s). From the view of a network engineer, Edge-to-edge Control is a hybrid architecture combining ideas from traditional QoS and closed-loop traffic control. Traffic control loops spanning between VPN edges push congestion back from the bottleneck with minimal imposition on the bottleneck. At the VPN edge we apply

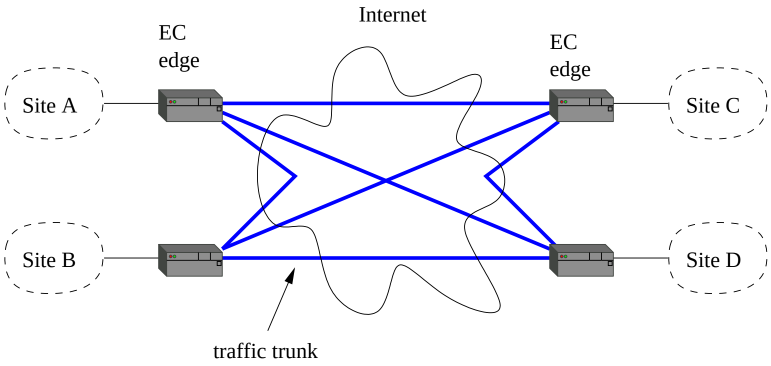


Figure 1.7: The Private Network Customer's View of Edge-to-edge Control

stateful and sophisticated QoS mechanisms (see Figure 1.8). This architecture trades off the precision and *responsiveness* of traditional QoS architectures to decrease bottleneck *complexity*.

The core of Edge-to-edge Control is the closed-loop regulated traffic trunk building block hereafter called the *EC trunk*. As with serial physical links, EC trunks are point-to-point but they do not assume fixed or symmetric paths or fixed capacity. EC trunks do however assume that within a direction, packets follow a single path on the timescale of round-trips.

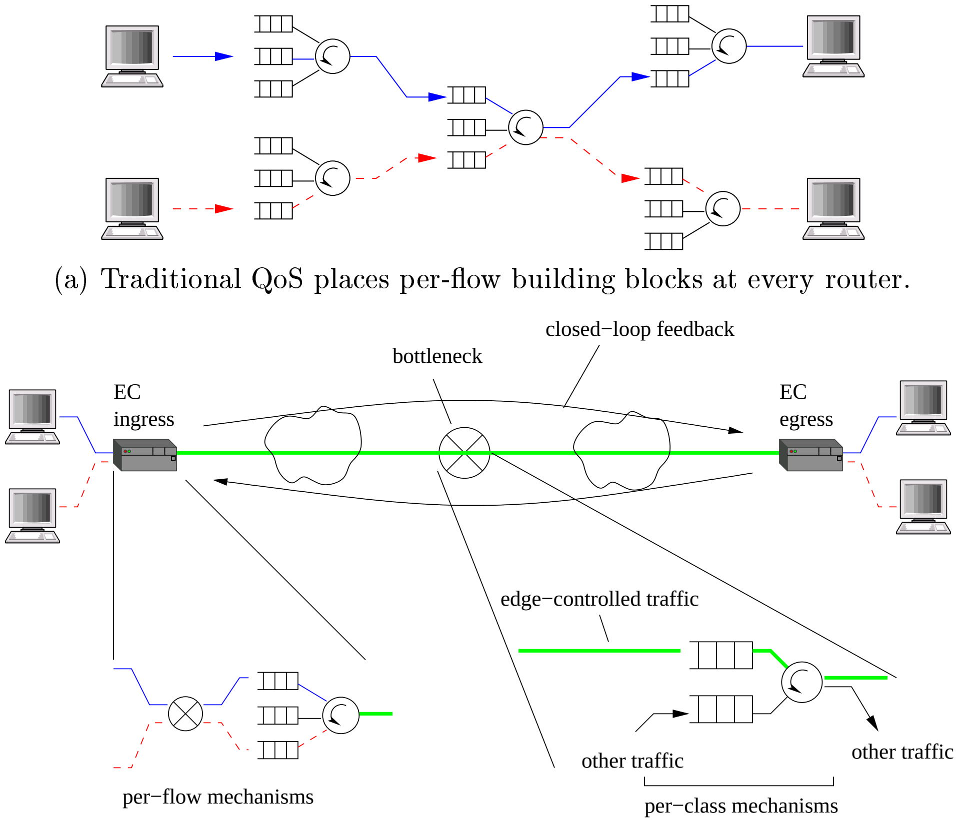
With EC trunks, QoS occurs in two stages:

1. EC trunks detect congestion without requiring loss and accordingly adjust their control loops to realize a variety of bandwidth sharing objectives (see chapter 5)

2. The EC trunk's ingress redistributes the EC trunk's bandwidth and buffers among end-to-end flows (see chapter 4).

If misbehaving end-systems can violate service level agreements for QoS traffic then separation of end-to-end and QoS traffic is a prerequisite. The approach that imposes the least bottleneck complexity of all of the mechanisms studied or surveyed in this dissertation is to place QoS traffic into a separate queue from end-to-end controlled traffic, and then serve from each queue using some scheduler.

Edge-to-edge control imposes no more than the simplest known means to achieving the prerequisite to providing QoS. By assuming a separate queue, EC allows us to implement EC trunks with any congestion control scheme we desire without requiring the scheme to compete fairly with TCP control loops. Thus we are free to study more recent congestion control schemes that may provide increased performance. This is not



(a) Traditional QoS places per-flow building blocks at every router.

(b) Edge-to-edge Control uses a CoS bottleneck with FIFO queuing.

Figure 1.8: Traditional versus Edge-to-edge Control

to say that TCP traffic cannot pass through an EC trunk, but rather that as long as loss does not occur within the network traversed by the trunk, TCP sees the EC trunk as a single lossless link with variable capacity. This enables the EC edge sitting at the head of this single lossless link to control all allocation of drops or perform other TCP-specific manipulation.

In the following sections we describe how to implement EC trunks and then contrast Edge-to-edge Control against the architectures described throughout this introduction.

1.2.1 Implementing Edge-to-edge Control

Five building blocks implement an EC trunk:

- EC traffic classification,
- EC isolation,
- Flow-to-EC-Trunk mapping,

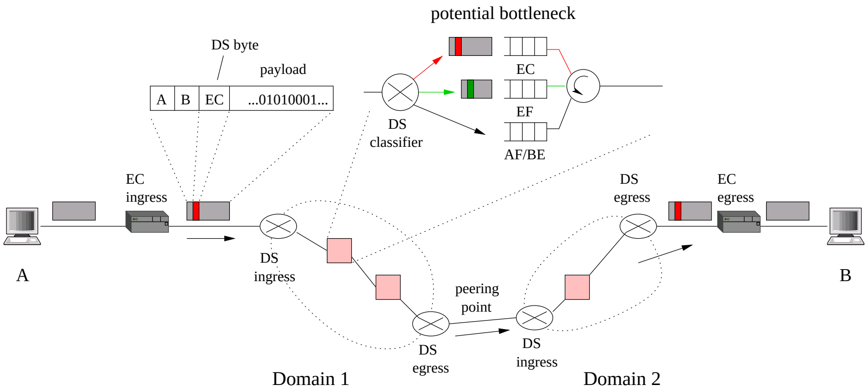


Figure 1.9: Edge-to-edge Control over DiffServ

- one-way single-path routing, and
- EC congestion avoidance.

Edge Control adopts DiffServ's classification technique; the ingress EC edges stamp each passing packet with the Edge Control service marking (EC) then intermediate routers map EC-marked packets onto the isolated traffic class. To maintain interoperability with DiffServ, the ingress can stamp EC into DiffServ's 6-bit field already defined in the IP packet header. The IETF need only define a default code point (i.e., 6-bit id) for EC traffic along with an Edge Control (EC) PHB. We show EC interoperating with DiffServ in Figure 1.9.

This EC PHB at least requires FIFO queuing and a provisioned minimum bandwidth share for the EC class. The EC PHB need not specify a maximum bitrate, because the EC class can grow to use any excess capacity.

Additionally we must define an EC Per-Domain Behavior (PDB) that requires providers to take reasonable measures to prevent non-EC traffic from entering their networks bearing the EC marking.

By implementing the EC PHB and EC PDB, the ISP achieves what we mean by *class-wise isolation*. The EC PHB might be implemented with any number of schedulers which we survey in chapter 2. The EC PDB isolation requirement can be implemented with DS edges that either remark or drop non-EC customers onto best-effort or other classes. Providers cannot easily determine whether traffic entering through a peering point is legitimate. Any stateful and/or per-customer inspection is likely to not scale. Non-stateful measures can be taken to limit damage done by illegitimate traffic such as imposing rate limits on the aggregate entering through a peering point. Unfortunately all DS services share this same problem to some degree, but the problem is exacerbated by the adaptive nature of both AF- and EC-based services, and we do not yet have a satisfactory answer.

In the context of VPNs, the problem of Flow-to-EC trunk mapping can be solved completely through static configuration at the EC/VPN edges. Each VPN edge needs to know at least the IP address prefixes of other sites in the VPN. If IPSec tunneling is used to force packets through desired VPN edges then each VPN edge also needs to know the IP address of each opposing edge. Because there is overlap between the configuration at each edge, configuration for the complete VPN might better be stored in a shared, trusted directory or communicated from a master edge using signaling. Either way shared directories or master edges reduce total human configuration cost from $\Omega(N^2)$ to $\Omega(N)$ in the number of VPN sites with an incremental cost of $\Omega(1)$ per site. Configuration cost can be higher depending on the types of services employed using EC trunks (see Chapters 4 and 5). Because we have limited ourselves to VPN QoS, the configuration and/or signaling cost of setting up an EC trunk is amortized across the expected moderate-to-long lifetime of the EC trunk.

EC trunks require one-way single-path routing (not symmetric paths) on RTT time-scales since each EC trunk has only one control loop while multiple paths may have different capacities. Multi-path routing can be prevented by running EC trunks over VCs or LSPs. However, TCP connections also require one-way single-path routing to avoid frequent packet reordering and consequent performance degradation. Thus without VCs or LSPs, a high-likelihood of single-path routing for an EC trunk can be established by using tunneling (IPSec [70]) to hide TCP headers. We discuss tunneling and an alternate approach, Network Address Translation (NAT [35]), In Appendix C.

Lastly, congestion avoidance is a central issue to this dissertation and is discussed in detail in chapter 3. Here we only state that carefully chosen mechanisms when combined with properly provisioned buffers allow for high utilization with low delay and near zero loss.

1.2.2 Virtual Circuits versus EC Trunks

EC trunks are similar to virtual circuits in that they are point-to-point and all packets in a given direction follow a single path, but there the similarity ends.

- The EC trunk building block itself does not pin a path, though it can be combined with MPLS or overlaid on VC-networks to pin an EC trunk's path when desired. As we describe in chapter 4, several interesting services can be constructed without route pinning.

- The EC trunk building block prescribes the use of traffic control. Virtual circuits do not prescribe the use of traffic control, but a virtual circuit combined with traffic control (such as ATM's ABR [66, 63, 16, 65, 64]) also moves congestion to the edges of the network (ATM cloud) where the VC's capacity can be redistributed among end-to-end flows, but the deployment space is limited to VC networks.

- VCs necessarily impose per-VC state on the bottleneck. EC trunks do not establish any state in intermediate nodes, other than per-class state necessary to isolate EC traffic.

- VCs as implemented in ATM are tightly coupled to the data link layer. EC trunks are implemented in the IP layer and thus can span multiple administrative domains and multiple data link layers.

Some of the enumerated properties have further implications. By not imposing EC trunk-specific state on any intermediate router, an EC trunk can handle route changes in intermediate networks without updating state at intermediate routers and with only transient loss of communication as some packets might be lost during the route update. Not requiring pinned routes or state at intermediate routers also decouples the EC trunk from routing, allowing EC trunks to be used with any IP routing protocol or on top of MPLS, ATM, or Frame Relay. In contrast, any VC architecture mandates integration with routing.

1.2.3 Edge-to-edge Control versus Differentiated Services

EF, AF, and EC all require CoS, all three at the minimum require reconfiguration of potential bottlenecks, and all three require policing or remarking at DS boundaries. Though for policing and/or remarking, the mechanisms differ for EF, AF, and EC (see chapter 2 for EF and AF, see chapter 4 for EC). The key difference is that EC and AF are adaptive; EF is not.

EF's strict requirement that arrival rate never exceed capacity over more than consecutive packet arrivals prevents end-to-end flows from utilizing unused capacity, and

Table 1.1: Differences between DiffServ and EC Building Blocks

	Expedited Forwarding	Assured Forwarding	Edge Control
Control Loop	open-loop	open-loop	closed-loop
Adaptive	no	yes	yes
DS Boundary Policing/Remarking	yes	yes	yes
DS Boundary Reshaping	yes	no	no
Delay/Jitter Control	tight	loose	loose
Bottleneck Upgrades	maybe/no†	maybe/yes‡	no
Bottleneck Reconfiguration	yes	yes	yes
Buffer Mgmt. Configuration	no	yes	no
Drop Precedence	no	yes	no
Point-to-Anywhere	no	yes	no
Example Application	telephony	public web	VPNs

† EF itself can be implemented without a bottleneck upgrade. However, if EF is combined with MPLS, upgrades are more probable.
‡ AF has been deployed in some routers already. However, AF may require re-deployment of newer mechanisms as the aforementioned difficulties in achieving desired bandwidth assurances[102, 54] are resolved.

requires reshaping at each DS boundary. However, the strict control over the arrival rate allows EF to obtain tight control on jitter and delay. Tight control on delay and jitter makes EF a potential building block for jitter-sensitive applications (e.g., telephony). EC- and AF-based services on the other-hand may be more suited to data and bandwidth-adaptive multimedia applications.

EC cannot obtain tight control on delay or jitter because EC cannot adapt more quickly than the propagation delay in the edge-to-edge control loop. AF cannot obtain tight control on delay or jitter because knowledge of AF incurred losses[2] must propagate through the end-to-end control loop. Again, propagation delay limits response time to fluctuations in demand. However, there is one central difference between AF and EC in the way they control end-systems. AF attempts to control end-systems by generating feedback at the bottleneck, whereas an EC traffic trunk infers the bottleneck's congestion state. The EC edge then applies stateful and application-aware mechanisms to carefully allocate loss. In the case of TCP, EC edges need not even resort to loss to control end-systems, e.g., EC edges may modify receiver advertised windows (see Section 4.3). EC's stateful control at edges based on knowledge of the bottleneck congestion state, allows EC to identify and properly constrain UDP and misbehaving TCP. AF relies on proper

[2]or marks when AF is combined with ECN (see Section 2.6.3).

TCP-like behavior of end-systems to obtain fair sharing of excess capacity. When AF bottlenecks are subjected to UDP traffic, UDP can capture all excess capacity[74, 103].

Additionally, the tight jitter constraints imposed by the EF PHB demand reshaping and admission control at each network boundary, at least to ensure an upstream service provider does not temporarily oversubscribe the downstream provider. However, the looser delay and jitter constraints imposed on EC combined with the adaptability of closed-loop control tolerate transient fluctuations in demand and thus reduce the need for intermediate shapers. Additionally for many simple services, admission control is superfluous (see chapter 4).

Lastly, AF has the advantage that no traffic trunks have to be set up prior to communication. Thus a customer can obtain AF-based services immediately when communicating with any other customer in a DiffServ-enabled internetwork. Such service has commonly been called a *point-to-anywhere* service. EC-based *point-to-set* services may be possible where bandwidth-sharing objectives are defined across a set of EC trunks terminating at a given edge, but this still requires setting up traffic trunks to each member of the set prior to communication. Such point-to-set services may be an interesting departure for future research.

The differences between DiffServ and Edge-Control building blocks are summarized in Table 1.1.

2. Survey of QoS in Packet Networks

In the introduction we presented building blocks used in providing QoS: admission control, stable (pinned) paths or QoS routing, service interfaces, signaling mechanisms, classification, schedulers, and buffer management. All of these building blocks can roughly be divided into two categories: *control plane* and *data plane.* The key difference between the two categories is that data plane components perform actions on individual packets necessary to enforce service commitments, while control plane components operate on timescales typically longer than the lifetime of a flow. By this distinction, admission control and signaling are control plane mechanisms while scheduling, traffic control, and buffer management are data plane mechanisms. On even larger timescales operate *management plane functions* such as defining policy and provisioning capacity. Edge-to-edge Control operates in the data plane and thus we survey data plane mechanisms representative of the basic approaches to QoS in packet networks.

Schedulers determine the time (and hence order) that packets are transmitted. Buffer management schemes determine whether a packet is buffered awaiting transmission or discarded. Schedulers and buffer management share the property that they control access to resources without directly adjusting demand. In other words, schedulers and buffer management schemes are open-loop building blocks. Congestion control schemes on the other hand use closed-loop feedback to adjust the source rate or window size (number of packets in transit) in an attempt to match demand to the available resources. None of these mechanisms are independent. If a flow is allocated less buffer, it receives less bandwidth. If a packet is dropped, many congestion control schemes will reduce demand. If demand from a given flow is reduced, buffer management and scheduling may reallocate buffers and bandwidth to other flows.

Prior to [28, 79, 92, 93], traffic control and service differentiation were considered separate problems. As with their work, we intend to bridge the difference between traffic control and traditional QoS in order to provide deployable Internet VPN QoS and improve performance. In keeping with our objective, we survey representative scheduling, buffer management, and traffic control mechanisms and specify how they fit into the Edge-to-edge Control architecture.

We will consider each of these mechanisms using the same basic criteria that we used to compare QoS architectures in the introduction: *complexity*, *efficiency*, *fairness*, *protection*, and *responsiveness*. We provide more explicit definitions for some of these terms in the following sections.

2.1 Complexity

When devising QoS mechanisms, a hard-learned rule has been to keep the bottleneck as simple as possible. Anything complicated that has to be done at the bottleneck limits scalability. As a corollary any complex configuration at the bottleneck reduces manageability. However, as we demonstrate throughout this survey, with less complexity inside the network performance suffers.

Similarly, within a router, placement of complexity is important. Performance can often be improved by off-loading control plane functionality to processors separate from the data plane, and by optimizing for the common case. The *critical path* refers to the path taken by most packets as they pass through the router. Any operation that takes place in the critical path adds latency to common-case packets and may reduce the router's overall throughput.

Some of the mechanisms we survey perform per-flow operations in the critical path (e.g., incrementing a counter of the number of packets transmitted in each flow). In backbones and peering points where tens or hundreds of thousands of flows may be arriving per second, the following problems arise:

- Operations in the critical path that take more than $O(1)$ time in the number of flows will reduce performance as the number of flows increases. For example, inserting or removing per-flow state from a heap takes $O(\log n)$ time.

- If per-flow state must be accessed on a per-packet basis, then a high-speed bottleneck with n flows must provide $O(n)$ expensive fast memory.

For many short flows, latency in signaling per-flow state can occupy a significant portion of a flow's lifetime, signaling can consume a significant fraction of the bandwidth, and processing signaling messages can tax the control plane's processing power.

Class-of-Service (CoS) networks eliminate per-flow state by grouping packets into *classes* based on like service requirements. In the introduction (see Section 1.1.4), we used the postal system as an everyday example of CoS. Dividing packets into a small number of classes drastically reduces state at the expense of increased granularity of the types of services offered by the bottleneck. Such solutions also often represent a trade-off of reduced *complexity* for increased *scalability*.

A hybrid alternative is to impose different levels of complexity depending on the location of a particular router in the network. As discussed in the introduction, DiffServ divides routers into two types: edge and interior routers. The interior routers implement CoS while the edge routers implement per-flow or per-customer state.

2.2 Fairness

Fairness determines how equitably an algorithm divides resources between contenders. There exist many different definitions of fairness. Among them, *max-min fairness*[101] is one of the more popular in the context of bandwidth sharing in computer networks. More recent definitions include proportional fairness[68] and min-potential delay fairness[84]. We describe each of the given types of fairness using the network in Figure 2.1.

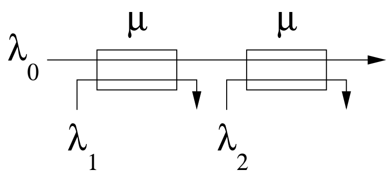


Figure 2.1: Simple Two Bottleneck Network

Max-min fairness intuitively means that the smallest demands for resources are satisfied first and whatever resources are left over are divided equally between the unsatisfied demands. Max-min fairness can be defined more precisely via the following procedure. Allocate each flow a rate of 0. Add all flows to the set of unsatisfied flows. Increase each unsatisfied flow's rate allocation equally until a flow becomes bottlenecked or all capacity in the network has been allocated. When all capacity has been allocated for a particular link, remove all flows from the unsatisfied set and add these flows to the set of bottlenecked flows. If the unsatisfied set is nonempty then resume increasing the rate allocations of the remaining unsatisfied flows. The final rate allocation is the max-min fair rate allocation.

Figure 2.1 shows a network containing two bottlenecks each with capacity μ. For this example, let $\mu = 1$. Flow 0 passes through two bottlenecks and sends with rate λ_0. Flows 1 and 2 each pass through only one bottleneck and send with rates λ_1 and λ_2 respectively. In this example, max-min fairness leads to the allocation $\langle \lambda_0, \lambda_1, \lambda_2 \rangle = \langle \frac{1}{2}, \frac{1}{2}, \frac{1}{2} \rangle$.

Max-min fairness does not consider the possibility that reducing one flow might allow multiple other flows to increase their rates. Consider Figure 2.1, if we decrease flow 0's share by $\frac{1}{4}$ then *both* flow 1 and flow 2 can increase their shares by $\frac{1}{4}$ leading to a net gain of $\frac{1}{4}$ in the sum of the rates. One might be tempted to formulate a fairness criterion that maximizes the total throughput across all flows, but consider that in our example such a criterion would allocate zero capacity to flow 0. Presumably flow 0's user would value the last bit of bandwidth taken by max-total-throughput fairness more than flow 1 and flow 2 would appreciate the small increase compared to their already large shares. Perhaps some fairness criterion in between max-min fairness and max-total-throughput fairness is in order.

Kelly [68] solves the dilemma posed between max-min and max-total-throughput fairness by finding the rate allocation that maximizes social welfare. He defines a *proportionally fair* allocation as the allocation that maximizes $\sum_i \log \lambda_i$ where $\sum_i \lambda_i = 1$. The assumption here is that every flow has a logarithmic utility function. Note that a logarithmic utility function is a differentiable, nondecreasing, concave downward function; therefore, it characterizes a user that experiences a diminishing increase in return for each increase in bandwidth. To understand proportional fairness consider again the network in Figure 2.1. With this network, max-min fairness led to allocations of 1/2, but as we will show in a moment, proportional fairness yields an allocation of $\langle \frac{1}{3}, \frac{2}{3}, \frac{2}{3} \rangle$. We show this analytically as follows. First note that at the ith hop, there are two rates λ_0 and λ_i. Thus when the bottleneck is fully utilized, $\lambda_i = 1 - \lambda_0$. Consequently our optimization problem equation becomes:

$$\log \lambda_0 + \sum_i \log(1 - \lambda_0) = \log \lambda_0 + 2\log(1 - \lambda_0)$$

Now set the derivative of our objective equal to zero to find the maximum.

$$\begin{aligned} \frac{1}{\lambda_0} - \frac{2}{1 - \lambda_0} &= 0 \\ \lambda_0 &= \frac{1}{3} \\ \lambda_1 = \lambda_2 &= \frac{2}{3} \end{aligned} \tag{2.2}$$

Massoulié[84] proposes yet a third fairness objective that minimizes potential transfer time. Given that a large portion of Internet traffic (e.g., ftp and http) transfers files, minimizing transfer time seems like a reasonable objective. Formally, min-potential-delay-fairness is defined as the rate allocation that minimizes the sum $\sum \frac{1}{\lambda_i}$. Through a similar derivation as used for proportional fairness, we find that the corresponding rate allocation for the example in Figure 2.1 is $\langle \lambda_0, \lambda_1, \lambda_2 \rangle = \langle \frac{1}{1+\sqrt{2}}, \frac{\sqrt{2}}{1+\sqrt{2}}, \frac{\sqrt{2}}{1+\sqrt{2}} \rangle$. Note that min-potential-delay fairness gives flow 0 more capacity than proportional fairness but less than max-min fairness. In general, min-potential-delay fairness punishes long flows less than proportional fairness and more than max-min fairness.

Riviera, the possible congestion avoidance scheme for implementing EC presented in chapter 3, approximates proportional fairness for a moderate number of bottlenecks (see Section A.4).

2.3 Responsiveness

As mentioned in the introduction, responsiveness is a set of metrics. Of the prior mentioned metrics, *convergence time* and *transient response* are the most relevant to data plane mechanisms. In this dissertation, convergence time refers to the time it takes for a previously idle system to converge on the steady state. Transient response refers to the time it takes for a system to recover when the system is disturbed from the steady-state. Transient response comes into play whenever a system is subjected to the arrival of one or more new flows thus requiring a reallocation of the bottleneck's resources. This is more important in systems that use closed-loop feedback as it may take many round-trips through the control loop before the system can converge on a new desired allocation of resources.

In simulations, we measure convergence time and transient response in terms of four metrics: utilization, queue, fairness, and efficiency (high utilization with small queue). A system is said to have converged on utilization when the utilization measured over some interval stays within $1 - \epsilon_u$ and 1. A system has converged on a small queue when the queue length remains below ϵ_q. Similarly fairness occurs when all rate r_i remain within ϵ_f of their fair share. Fair share can be defined by any of the specified fairness metrics. Here each ϵ is determined empirically.

2.4 Scheduling

Scheduling provides fine control over the allocation of bandwidth between flows or aggregates. With some types of schedulers, a network of schedulers can provide end-to-end bandwidth, delay, and/or delay-jitter guarantees when the network is subjected to well-specified traffic. In the following sections, we provide a short survey of scheduling disciplines and then discuss how schedulers fit into the context of Edge-to-edge Control.

2.4.1 First-In, First-Out

Perhaps the simplest scheduler is First-In, First-Out (FIFO), also known as First-Come, First-Serve (FCFS). As implied by the name, packets are served in the order they arrive as quickly as the router can forward them. When the arrival rate λ exceeds bottleneck capacity μ, the received service rate tends toward the ratio of a flow's input rate to the total input rate $\mu_i = \frac{\lambda_i}{\lambda}\mu$; here the subscript denotes the share attributed to flow i. FIFO rewards the flow with the largest send rate and thus provides incentive for flows to send as quickly as they can. Unfortunately λ consistently exceeding μ leads to unbounded queues. When a non-adaptive flow sends near or faster than μ, the non-adaptive flow forces all adaptive flows to a near zero share. This is known as a *denial-of-service attack*. FIFO provides no *protection* from misbehaving flows. Thus *efficiency*, *fairness*, and *responsiveness* are determined by the behavior of end-systems.

Despite FIFO's limitations, FIFO has a strong advantage: it has low *complexity* and is thus simple to implement at high bitrates. This may be the strongest reason behind the acceptance of FIFO as the default scheduling behavior in the Internet today.

2.4.2 Priority Scheduling

If we consider networks with potential bottlenecks where end-hosts cannot be trusted to adapt to available capacity then FIFO is not a suitable scheduler for providing any form of QoS. One possible solution to the problem is to merely prioritize traffic so that important flows are not subject to the behavior of lower priority flows. As might be expected, this behavior is known as *priority scheduling.*

Priority scheduling can be implemented with a separate queue for each priority. Assuming the scheduler is not preemptive, whenever there are packets waiting at a higher priority level, they are served as soon as the link finishes transmitting the current packet.

Priority scheduling does allow coarse-grained service differentiation, but it lacks certain desirable traits. For one, a low priority packet may never be served if there are always higher priority packets to be served. This situation is known as *starvation*. Furthermore, a priority scheduler does not itself *protect* flows at the same priority. If all packets within the same priority are served in FIFO order then there still exists the incentive for a bandwidth hog to force out adaptive flows.

Priority scheduling is important in our survey only because it has been suggested as a potential building block in implementing services based on Expedited Forwarding in Differentiated Services (see Section 2.7.2).

2.4.3 Generalized Processor Sharing

To address the problems of FIFO and priority schedulers, a reasonable alternative is to implement a scheduler that attempts to fairly distribute capacity between flows according to some fairness metric. In a network where all packets are the same size, *max-min fairness* (see Section 2.2) at a single bottleneck can be closely approximated using a round-robin scheduler. The round-robin scheduler in each round services one packet from each flow.

To handle networks with packets of multiple sizes, consider an ideal and unimplementable scheduling discipline known as *Generalized Processor Sharing (GPS)*. A GPS scheduler is a round robin scheduler that services an infinitesimal amount from each backlogged flow irrespective of packet boundaries. GPS is unimplementable because in each round GPS serves infinitesimal portions rather than whole packets. Let $B(t)$ denote that set of backlogged flows at time t. Thus given n flows in $B(t)$ and a bottleneck capacity μ, each flow receives a service rate of exactly $\frac{\mu}{n}$. When a flow does not have a packet backlogged at the bottleneck, the unused share is divided evenly between the backlogged flows. Thus a GPS scheduler realizes *max-min fairness* as defined on a single bottleneck.

GPS has the desirable property that over any time interval, GPS provides a locally fair share among the backlogged flows regardless of the excess traffic arriving from other backlogged flows. Thus GPS provides *protection* for each flow from the misbehavior of other flows.

GPS schedulers are also *efficient* in the sense that as long as there are backlogged packets, the GPS server will serve packets. When any scheduler performs work as long as there is work to be done, the scheduler is said to be *work-conserving*. All work-conserving schedulers are efficient in the stated sense. GPS schedulers are also *efficient* in the sense that they allow adaptive flows to absorb any unused capacity.

A GPS scheduler can also be visualized using a fluid model wherein all backlogged flows are served simultaneously. Among backlogged flows, each flow i with weight ϕ_i is served with a guaranteed rate g_i of

$$g_i = \frac{\phi_i}{\sum_{j \in B(t)} \phi_j} \mu \tag{2.3}$$

The behavior in the above equation motivates the study of GPS in designing schedulers for guaranteed bitrate services.

In the following two sections we will discuss schedulers that use GPS as a reference model.

2.4.4 Weighted Fair Queuing

Weighted Fair Queuing (WFQ)[32, 96] is one of the first schedulers proposed for Internet QoS and is the foundation of many GPS schedulers. In this section, we describe WFQ in some detail to impress upon the reader the complexity in providing per-flow *protection* and *max-min fairness* at each bottleneck. In Section 2.4.5, we discuss how WFQ and its descendants relate to Edge-to-edge Control.

As each packet p arrives at the bottleneck, WFQ computes p's finish time denoted F_p according to a simulated GPS scheduler and then services packets in order of increasing F_p. If a packet arrives when the buffer is full, packets are dropped in order of decreasing F_p until there is sufficient room to buffer the arriving packet.

We can better describe WFQ by introducing the concept of a *round number*. We start by simplifying GPS to a round robin scheduler and considering only the case when all flows have equal weights. With this view, we define the *round number* as the number of rounds performed by a bit-by-bit round robin scheduler. We also define the kth packet in flow i as p_i^k, its arrival time as a_i^k, its finish number as F_i^k, and its length as l_i^k. When packet p_i^k arrives at the beginning of round number n and flow i is not backlogged, WFQ computes the finish number F_i^k as simply $n + l_i^k$. When packet p_i^k arrives at round number n and flow i is backlogged, WFQ sets F_i^k equal to $F_i^{k-1} + l_i^k$.

In reality, packets will usually not arrive exactly when 1 bit would have been served from each flow by a GPS scheduler. To handle this situation we define a real-valued version of round number and call it *virtual time* $V(t)$. We also redefine F_i^k to be the virtual finish time of the kth packet for the ith flow. $V(t)$ is zero when the server is idle. In a busy period starting at time zero, $V(t)$ advances as

$$\frac{\partial V}{\partial t} = \mu \,. \tag{2.4}$$

With this definition of virtual time,

$$F_i^k = max(V(a_i^k), F_i^{k-1}) + l_i^k \,.$$

When sources are shaped by a leaky bucket shaper [111], a network of WFQ schedulers achieves bounded delay that grows linearly in the number of hops and does not depend on the behavior of other flows[97]. The existence of such a delay bound motivates the use of WFQ in QoS networks. In fact the IETF paid much attention to WFQ and its variants when designing IntServ.

2.4.4.1 Implementation of WFQ

Keshav[72] implements WFQ's behavior of servicing in order and dropping in reverse order of virtual finish times using the data structure shown in Figure 2.2. The heaps (i.e., priority queues) contain references only to the head or the tail of each flow's queue. Each queue is implemented as a doubly-linked list, so that packets can be popped from the front when choosing a new packet for service or from the back when dropping a packet. This combined data structure uses the property that all packets within a queue have monotonically rising finish times based on the order of their arrival, thus the packets need not be sorted further. In the worst-case any insertion or deletion to the heaps takes $O(\log n)$ time, where n is the number of backlogged connections. Each insertion or removal to a doubly-linked list takes $O(1)$ time, thus the processing time for inserting or removing packets from this data structure is $O(\log n)$.

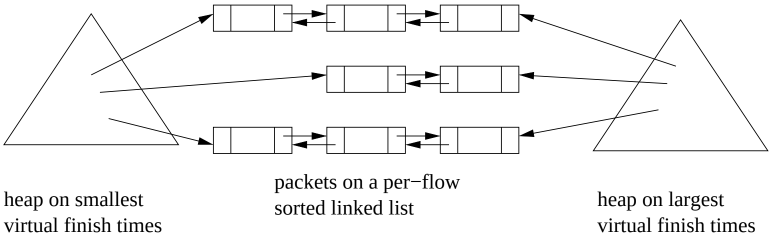


Figure 2.2: WFQ Implementation Using Two Heaps and Linked Lists

However we have omitted the largest component to WFQ's time complexity: computation of finish times. WFQ computes the virtual time when a packet arrives or departs.

Note that the departure time does not necessarily occur when the virtual time reaches a packet's virtual finish time. Consider the case when two packets of equal length L arrive at the same time to an idle scheduler with unit service rate. The packets are transmitted in arbitrary order. According to the GPS scheduler both packets finish at virtual time L. In reality we service packets one at a time and the first completes service when the virtual time is $\frac{L}{2}$. However, in the GPS scheduler, virtual time continues to progress with slope $\frac{1}{2}$ even after the first packet departs because in the GPS scheduler both packets are still receiving service. This example demonstrates how the packets at a WFQ scheduler may differ from the packets at the reference GPS scheduler. In fact multiple packets may depart from the GPS scheduler between arrivals or departures of packets to the WFQ scheduler, and hence multiple packets may depart from the GPS scheduler since the last finish time computation. Each such departure increases the virtual time's slope $\partial V/\partial t$ and hence may result in more GPS departures between virtual finish time computations. This is known as the *iterated deletion* problem [71]. Golestani[48] shows that computing virtual time requires $O(m)$ time where m is the number of flows experiencing deletions since the last computation of virtual time. In the worst case m approaches n, the number of flows passing through the WFQ scheduler, and thus computing virtual time requires $O(n)$ in the worst case.

Because the per-packet time complexity increases linearly in n, WFQ will not scale to large n at least when compared to FIFO, which performs only constant-time operations.

2.4.5 Other Schedulers and Their Relation to Edge-to-edge Control

In summary, WFQ provides reasonably tight delay bounds, approximates max-min fairness, and provides bandwidth guarantees on the timescale of a few packet transmissions. Each of WFQ's descendant's improves on WFQ in some way, e.g., decreasing the cost of the virtual time computation (SCFQ [30], SFQ [49]), decreasing the delay bound (SFQ), or improving fairness over timescales of a few packets (WF^{2}Q [14]). However, all of these descendants share the same basic properties: they impose per-packet time complexity on the bottleneck that increases linearly or logarithmically with the number of flows, they impose state that increases linearly with the number of backlogged packets (i.e., service tags), and they impose state that increases linearly with the number of flows (i.e., entries in each heap).

We could survey a wide array of related schedulers that solve some of the problems of

WFQ and its descendants. For example, Deficit Round Robin (DRR [106]) performs only $O(1)$ computations at each packet transmission and eliminates per-packet service tags, heaps, and the virtual time computation in favor of a single counter per flow. However, DRR still requires $O(n)$ state where n is the number of flows. In short, DRR reduces space and time complexity at the cost of unfairness over short time scales. The class of Rate Control Scheduling (RCS [117, 118, 47]) disciplines decouples the rate allocation from the delay bound thus allowing smaller flows to obtain tight delay bounds, which is impossible with all schedulers that emulate GPS. However, when used to provide per-flow QoS as with IntServ, all of the aforementioned schedulers impose complexity increasing with the number of flows.

When WFQ operates on classes rather than flows, the time complexity decreases significantly. Typically the number of classes is a small constant and thus all $O(n)$ operations become $O(1)$ operations. Furthermore, the per-flow component of space complexity reduces to a constant. This same reasoning applies to all of WFQ's descendants, DRR, and RCS. In the context of Edge-to-edge Control, any of these schedulers will probably suffice to provide "classwise isolation" of the Edge Controlled traffic class. Even so, these schedulers are all more complicated than FIFO, but we currently see no way around this limitation.

Luckily, most routers in the Internet today support some form of fair queuing. In Appendix A.2, we study EC over CoS with two classes: one for best-effort and the other for EC traffic. We study the effect of bursty best-effort traffic competing with EC traffic over a CoS network in Appendix A.6.

Edge-to-edge Control is orthogonal to scheduling in that EC trunks use schedulers only to provide class-wise isolation and possibly to provide per-flow QoS at the EC trunk's ingress edge.

2.5 Traffic Control

Traffic control uses closed-loop feedback to adjust demand to the available capacity. Jain [100] differentiates traffic control schemes into two categories: congestion avoidance and congestion control. The objective of *congestion avoidance* algorithms is to maintain the system near a point of high efficiency. One measure of efficiency, *network power*, is defined as the ratio of the throughput to the delay. In Figure 2.3, network power is maximized at the knee. Conversely, *congestion control* algorithms allow the load to grow until the system nears the cliff before backing off. In the Internet, the role of imposing congestion control has traditionally been the job of TCP. The most widely deployed version of TCP, TCP Reno, is a congestion control scheme; TCP Reno waits until buffers overflow before reducing load (see Section 2.5.2). TCP continues to use *congestion control* rather than *congestion avoidance* partially to maintain backwards compatibility and partly because it is difficult to create robust congestion avoidance algorithms that do not depend on support from the bottleneck. In this section we survey both congestion avoidance and congestion control schemes.

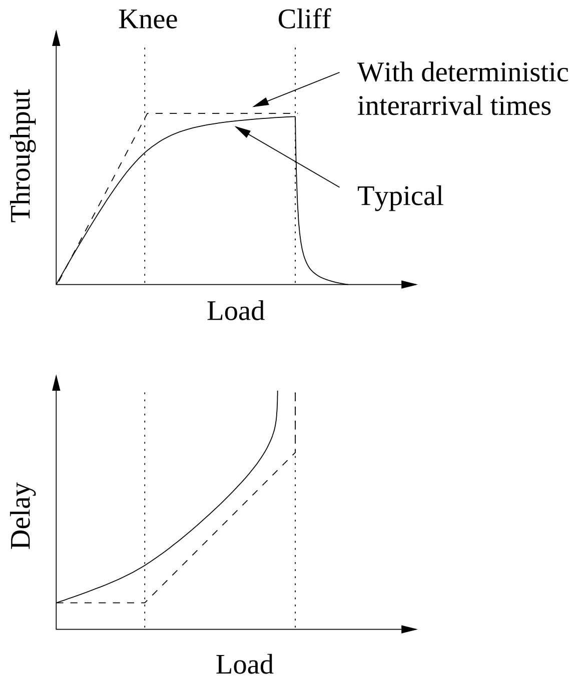


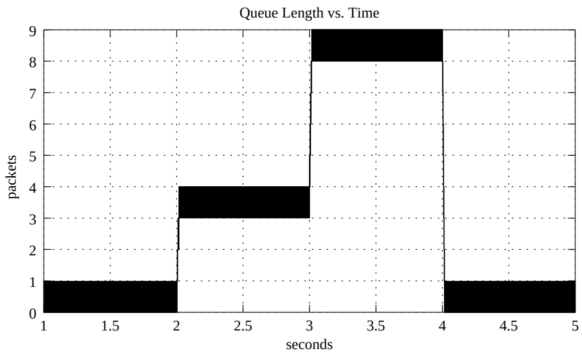
Figure 2.3: Congestion Avoidance Maintains the System Near the Knee

In addition to classifying schemes based on whether they implement congestion control or congestion avoidance, we classify schemes based on the following properties.

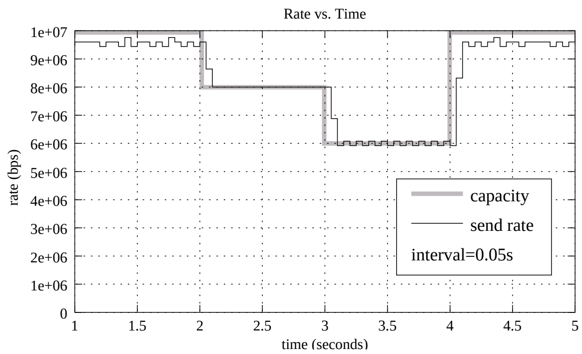
Explicit versus Implicit Traffic control algorithms that rely on feedback from the bottleneck use *explicit* control. Traffic control algorithms that rely on measurements taken only at the end-systems use *implicit control.* Current TCP schemes rely on implicit control but many can respond to explicit control (see Section 2.6.3).

End-to-end, Hop-by-hop, and Edge-to-edge End-to-end schemes operate between end-systems and thus an end-system's knowledge of the bottleneck state is limited by its end-to-end time delay, by state communicated from the bottleneck, and by what the end-system can measure. Hop-by-hop mechanisms operate right at the bottleneck, have more complete and timely knowledge of the bottleneck state, and thus can exercise greater control but only at the cost of increasing the bottleneck complexity. This dissertation presents schemes that operate edge-to-edge in the sense that the ends of the control loop may not be end-systems. Edges closer to the bottleneck will tend to have more aggregate knowledge of the bottleneck state and can exploit this knowledge to provide per-flow services and higher performance control.

Window, Credit, or Rate Window- and credit-based algorithms adjust the number of packets in transit in order to match the bandwidth-delay product of the path between the sender and receiver. Rate-based algorithms adjust the send rate to match the bottleneck capacity. We assume that the reader has basic knowledge of window-based and rate-based control. A description can be found in any introductory networking textbook. Credit-based schemes periodically exchange update packets that grant the sender permission to send a batch of new packets. We describe credit-based schemes in more detail in this traffic control survey. Here we merely contrast the relevant properties of window-, credit-, and rate-based schemes:



(a) queue length versus time



(b) average rate versus time

Figure 2.4: Constant-size Window Subjected to Bottleneck Capacity Changes

- Window-based protocols obey what Van Jacobson [56] calls the *conservation of packets principle*[3]. As long as the window is full and no loss occurs, a new packet enters when and only when an acknowledgment (ACK) is received. Van Jacobson uses the term *ack clocking* to describe sending a new packet as each ack arrives. When

[3]Van Jacobson actually talks about TCP, but the principle applies to window-based schemes in general.

packets are waylaid at a queue so are their corresponding acks and as such a window-based traffic control algorithm automatically slows down to match the bottleneck capacity. In this manner, sources slow down *even if the window maintains static size.* Rate-based algorithms do not inherently have such an ack clocking mechanism. In Figure 2.4 we show simulation results of queue length and arrival rates for a single bottleneck subjected to a single flow. The single flow uses window-based flow control with a static window size. In this simulation the bottleneck capacity periodically changes. Notice that rate changes to match changing bottleneck capacity. The queue length grows to absorb the difference between the window size and the bandwidth-delay product of the path between the sender and receiver.

- Window-based control couples error control and traffic control. This coupling introduces significant implementation complexity, and is irrelevant for protocols that do not provide reliable or in-order delivery.

- Window-based control requires a stream of acknowledgments from the receiver to the sender. For unreliable data transports (e.g., UDP), acknowledgments represent significant overhead.

- Window-based control requires that all packets have a sequence number. Again, this is fine for reliable or in-order transports, where sequence numbers are needed anyway. Because edge-to-edge and hop-by-hop schemes cannot rely on access to TCP sequence numbers when bearing non-TCP or encrypted traffic, window-based edge-to-edge or hop-by-hop schemes would require an additional sequence number field in each IP packet.

- Credit-based schemes do not require sequence numbers or an ack stream. The periodic exchange of update packets that occurs with credit-based schemes represents less overhead than an acknowledgment stream, but only at the expense of losing the ack-clocking property of window-based schemes. As with rate-based schemes, credit-based schemes decouple error control from traffic control.

- Window-based and credit-based control have the advantage that they do not require a fine-grained timer to clock packets into the network. End-systems are notorious for having inaccurate and coarse-grained clocking mechanisms, but this may not be a problem with routers (and hence ingress edges) where fine-grained timers can be provided in hardware.

- Rate-based schemes space out packets thus reducing bottleneck buffer requirements and reducing delay variance. Both credit-based and window-based schemes can require significant bottleneck buffers to absorb transient bursts.

In this survey we overview mechanisms from each class: scheduling, traffic control, and buffer management. In chapter 3, we propose implicit and explicit, edge-to-edge, rate-based congestion avoidance. We motivate our decisions in chapter 3 and reiterate the differences between end-to-end and edge-to-edge control that motivate these design decisions.

In Table 2.1 we summarize the properties of the traffic control schemes mentioned in this survey.

2.5.1 DECbit

DECbit[99, 100] is an early *explicit*, *window-based*, *end-to-end* congestion *avoidance* scheme. As with all window-based schemes, DECbit maintains a *sliding window*. DECbit increases the sliding window linearly until congestion is detected and then backs off multiplicatively. This is known as Additive Increase/Multiplicative Decrease (AIMD) and is more explicitly characterized as follows:

- if no congestion, $cwnd = cwnd + \alpha$
- if congestion, $cwnd = \beta * cwnd$

Here *cwnd* (congestion window) bounds the maximum size of the window, and α and β are input parameters where $\alpha > 0$ and $0 < \beta < 1$. Chiu and Jain[23] show for a single bottleneck in a rate-based context with synchronized increases and decreases and persistent flows that AIMD converges on an equal distribution of the bottleneck capacity. This can intuitively be shown for two sources using an allocation graph (see Figure 2.5). In this graph, we show two sources X and Y. Each axis refers to the send rate of the respective source. Thus the coordinate (x, y) means that X is sending with rate x, and Y is sending with rate y. Let μ denote the bottleneck capacity. The line $x + y = \mu$ is known as the *efficiency line*. Any point above this line corresponds to an arrival rate that exceeds the bottleneck capacity. Because X and Y send with equal rates when the system falls along the line $x = y$, this line is called the *fairness line*. When both flows

Table 2.1: Traffic Control Schemes

scheme	end-to-end	edge-to-edge	hop-by-hop	type	control/ avoidance	explicit/ implicit	detect method	features
DECbit	•			window	avoidance	explicit	avg. queue	a
TCP Tahoe	•			window	control†	implicit	loss	a,e,t
TCP Reno	•			window	control†	implicit	loss	a,da,e,t
TCP SACK	•			window	control†	implicit	loss	a,sa,e,t
TCP RBP	•			rate-window	control†	implicit	loss	a,da/sa⋆,e,t
TCP Vegas	•			window	avoidance	implicit	delay	a,da,e,p,t
Tri-S	•			window	avoidance	implicit	dT/dW	a,un
CARD	•			window	avoidance	implicit	power	a,un
Mo-Walrand	•			window	avoidance	implicit	delay	a,p,un
NETBLT	•			rate	avoidance	explicit	rate diff	c,sa
ERICA	•	•		rate	avoidance	explicit	z,queue	c,f,ur
Packet-Pair	•			rate	avoidance	implicit	ack spacing	fq,sm
Credit			•	credit	avoidance	explicit	avail. buffer	c,f,ur
MK			•	rate	avoidance	explicit	queue	c,f,s,ur
CM	•			rate-window	control†	implicit	loss	c,da/sa/g⋆⋆,e
TCP Trunks		•		rate-window	control†	implicit	loss	a,cn,dp,ur,e
Riviera‡		•		rate	avoidance	implicit	accumulation	c,g,p,r,t,ur

† These schemes become congestion avoidance schemes when combined with proper buffer management (see Section 2.6) at potential bottlenecks.
‡ Riviera is introduced in chapter 3
⋆ (da) and (sa) are orthogonal to Rate-Based Pacing (RBP).
⋆⋆ A window-based transport notifies CM of congestion detected via (da) or (sa). CM uses (g) when the overlying transport does not generate frequent feedback.

a	ack-clocking
c	uses separate control packets
cn	uses a separate control "connection"
da	uses duplicate acknowledgments to detect loss
dp	uses drop preference to achieve loss-less behavior. Assumes some buffer management algorithm at bottleneck such as RIO.
e	becomes explicit when using Explicit Congestion Notification (ECN). ECN is usually considered orthogonal to other congestion control mechanisms and can be used with any of the TCP variants.
f	requires per-flow or per-VC state at each switch/router.
fq	requires per-flow fair queuing (e.g., WFQ).
g	uses packet counts to detect loss.
p	requires estimation of propagation delay.
r	rate-clocking (see Section 3.2.3)
sa	uses selective acknowledgments.
sm	uses SMART[73] retransmission algorithm.
t	uses timeouts to detect loss.
un	retransmission mechanism is unspecified.
ur	unreliable (provides no mechanisms for retransmitting packets)

increase linearly with the same rate the system moves up at a 45° angle. When congestion is detected, each flow backs off from (x, y) to $(\beta x, \beta y)$. The line drawn between (x, y) to $(\beta x, \beta y)$ passes through the origin, and notice that with each iteration the slope of the back off line moves toward the fairness line. After enough iterations the system converges near the intersections of the fairness and efficiency lines. This intersection is called the *optimality point*.

AIMD has been widely used in both window-based (see Section 2.5.2) and rate-based schemes (see Section 2.5.10). Later in Section **??** we will introduce a new rate-based form of AIMD based on the arrival rate at the receiver-side of the control loop.

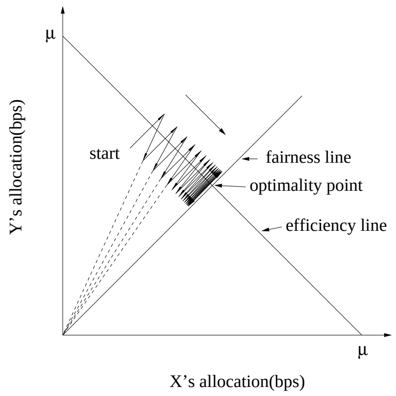


Figure 2.5: AIMD Allocation Graph

DECbit detects congestion when the *bottleneck* marks (flips) a bit in the network layer header of passing packets. The receiver copies the bits into the transport-layer ACK. The receiver records the bits over the current and previous window size and applies a filter. The filter determines that congestion has occurred when 50% of the bits received in an interval are marked. When the filter decides that there is congestion DECbit backs off otherwise it increases.

The bottleneck marks packets whenever the *average* queue length exceeds a threshold (set to 1). The queue length is averaged over an interval that is dynamically adjusted to match the bottleneck's previous busy + idle period plus the current busy period. The bottleneck is busy as long as it sends packets and is otherwise idle. Averaging over two busy periods takes into account that early in the current busy period there is little data from the current busy period on which to base an average.

2.5.1.1 Advantage of DECbit

- Because the bottleneck marks packets when the average queue length exceeds a threshold, DECbit does not incur large queue lengths or packet loss before adjusting demand.
- Because the system relies on only aggregate state (i.e., the average queue length), it does not impose as much *complexity* on the bottleneck as WFQ, or its variants.

2.5.1.2 Disadvantages of DECbit

- To deploy DECbit into the existing Internet would require upgrading all potential bottlenecks and end-systems.
- Additive increase requires $\frac{\mu}{\alpha}$ time for a single flow to go from a bottleneck with zero utilization to a fully utilized bottleneck. Thus DECbit only slowly utilizes bottleneck capacity. One could solve the problem by increasing α but the resulting α would not be appropriate across a wide range of μ. Large α would also incur large queue oscillations in the steady-state.
- Because DECbit marks all packets when above the average queue length, DECbit can exhibit a bias against bursty flows [43].

2.5.2 TCP Tahoe and Reno

Van Jacobson's TCP Tahoe [56] is an *implicit*, *window-based*, *end-to-end* congestion *control* scheme.

TCP Tahoe operates in three different phases: *slow-start*, *congestion avoidance*, and *timeout*. During slow-start, the window increases by one for each ack received. This has the effect of doubling the window each round-trip time (RTT). Such multiplicative increase allows a source to fully utilize the available capacity in $\log_2 \mu$ round-trips, thus solving DECbit's sluggish start-up times. When TCP Tahoe first opens a connection, it enters slow-start and stays in slow-start until loss occurs under the assumption that loss is almost always caused by a buffer overflow at the bottleneck. The loss is detected only after a *timeout* waiting for an acknowledgment. Once loss is detected TCP Tahoe sets *ssthresh* (slow-start threshold) to $cwnd/2$ then sets $cwnd$ to 1. After the timeout, the system recovers by going into slow-start until $cwnd$ reaches *ssthresh* at which point the system enters the *congestion avoidance phase* (see Figure 2.6). In congestion avoidance TCP slowly increases the window size and *ssthresh* with each acknowledgment.

$$cwnd \leftarrow cwnd + \frac{1}{cwnd}$$

The above equation has the effect that after a window worth of acknowledgments has been received, the window size has increased by roughly $\frac{1}{cwnd} * cwnd = 1$. Thus TCP Tahoe performs additive increase. Because *ssthresh* is decreased multiplicatively from *cwnd*, TCP Tahoe uses multiplicative decrease. This provides TCP Tahoe with similar *fairness* as DECbit.

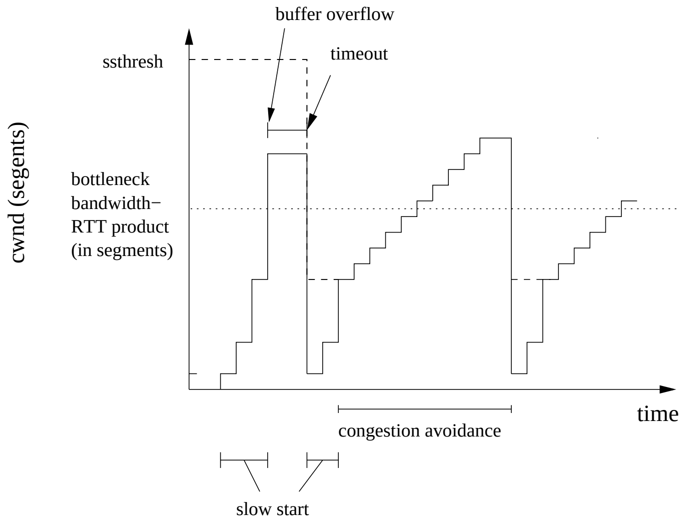


Figure 2.6: Phases of TCP Tahoe Congestion Control

The $\frac{1}{2}$ backoff factor is motivated at least partly by the doubling factor of slow start[56]. The idea is that no losses occurred at $cwnd/2$ so $cwnd/2$ is probably below the network's capacity. TCP thus returns to the point where no losses occurred and increases slowly (additively) from there.

If a retransmitted packet is lost, TCP Tahoe invokes the *exponential retransmit timer backoff algorithm*. The exponential backoff algorithm dictates that with each retransmission of the same packet, TCP must double its timeout. This mechanism allows TCP connections to send at much less than 1 packet per RTT and thus allows for many connections to share a bottleneck without causing instability. When a packet is finally successfully retransmitted, TCP goes back into slow start starting with $cwnd = 1$. This sharp change from exponential backoff to slow-start has certain negative effects which we will come back to when discussing the disadvantages of TCP Tahoe and Reno.

TCP Tahoe has one significant problem that Van Jacobson quickly fixed. TCP

Tahoe backs off too much in response to a sporadic (single packet) loss. Such loss could be due to burstiness rather than persistent overload and thus some lesser backoff is in order. TCP Reno [108] solves this problem by introducing two mechanisms:

- fast retransmit
- fast recovery

TCP Tahoe and Reno use cumulative acknowledgments, thus gaps in arriving sequence numbers trigger the receiver to repeat the cumulative acknowledgment for the last in-order received packet. The *fast retransmit* algorithm infers packet loss from three duplicate acknowledgments (dupacks) and consequently retransmits the missing or out-of-order packet without waiting for a timeout. *Fast recovery* infers from each duplicate acknowledgment that a packet has left the network and thus another packet can be clocked into the network. However, a duplicate acknowledgment does not advance the window, thus when the window is full, each arriving duplicate acknowledgment temporarily *inflates* the window by one segment. Similarly the arrival of duplicate acknowledgments indicates that transient rather than persistent loss has occurred and thus *fast recovery* reduces *cwnd* by half rather than $cwnd = 1$. In detail,

- $ssthresh = cwnd/2$
- $wnd = cwnd + ndup$ where $ndup$ is the number of consecutive duplicate acknowledgments.
- when the first non-duplicate acknowledgment arrives, *cwnd* is reset to *ssthresh*, and Reno leaves fast recovery.

2.5.2.1 Advantages of TCP Reno

TCP Reno operates across a wide range of capacities and delays and is widely deployed. Because it assumes little of the bottleneck, Reno operates over many data link layers. Reno is the dominant congestion control algorithm in the Internet today.

2.5.2.2 Disadvantages of TCP Reno

TCP Reno's success has motivated intense study. Consequently, much is known about Reno's (and to a certain extent Tahoe's) properties:

- A burst arriving at a full buffer experiences multiple losses. Multiple packet losses in a single window often defeats fast retransmit causing substantial unfairness to bursty connections [41].

- TCP Reno is biased against connections with larger round-trip delays [114, 41, 42, 78, 119]. Connections with shorter round-trip delay increase their window sizes more frequently.

- TCP Reno is biased against connections that pass through multiple FIFO bottlenecks [33, 41]. A connection passing through multiple FIFO bottlenecks has a higher likelihood of experiencing a drop. Whether this is a disadvantage depends on your definition of fairness. Certainly Reno does not achieve max-min fairness, but neither does Reno achieve proportional or min-potential delay fairness since these definitions ignore propagation delay. According to [114], Reno implements what they call F_A^h fairness.

- TCP Reno assumes FIFO and thus provides *no protection* from misbehaving flows.

- When the buffer overflows, multiple Reno connections may experience loss causing synchronous backoff, underutilization, and wide oscillation[51].

- Synchronization can arise over the last packet in the buffer causing a subset of the connections to experience disproportionate packet drop rates and substantial unfairness[42]. Slight changes to the propagation delay can change the set of flows receiving service and the set starved or nearly starved. Floyd[42] calls this a *phase effect*.

- Loss to a connection with less than a four packet window triggers a timeout rather than fast retransmit. Thus TCP Reno connections with tiny windows experience substantial unfairness[80].

- As the number of connections exceeds well beyond the number of packets in the network's bandwidth-delay product, the drop rate increases dramatically and as a result a significant portion of the flows remain in timeout. Morris[91] simulates loss rates ¿ 20% and provides analysis that suggests that drop rates can grow as high as 50%.

There are various solutions to each of these problems. Some of them involve adding support at the bottleneck. We will cover some of these when surveying buffer management (see Section 2.6). Most of the problems enumerated above are due to waiting until loss occurs before decreasing window sizes. Edge-to-edge Control (see chapter 3) avoids these problems most of the time by using congestion avoidance and by pacing packets to smooth out burstiness.

2.5.3 Extensions to TCP

TCP Reno experiences timeouts when multiple packets are lost within a single window despite the existence of the fast retransmit and fast recovery algorithms. Various solutions have been proposed to this problem including Select Acknowledgments (SACK [57, 86] and FACK [85]) and New-Reno [39].

To reduce TCP's bias against flows with tiny windows, limited-transmit [3] inflates the window whenever a duplicate acknowledgment arrives, i.e., limited-transmit does not wait for three duplicate acknowledgments.

To avoid burstiness and hence TCP's bias against burstiness, several researchers have suggested pacing TCP transmissions to correct for ack compression [121], spacing when acknowledgments are not available such as after loss [53] or when an idle connection resumes [113]. In Table 2.1 we collectively refer to these solutions as TCP RBP (Rate-Based Pacing). Counter-intuitively, [1] finds that pacing can decrease throughput and increase delay, because by smoothing out transient queue growth, pacing delays packet loss until the average queue length is much closer to the buffer size. Also when a bursty source and a paced source compete, a paced source is more likely to have a packet within each burst and thus more likely to experience loss during burst loss [1]. Each of these pacing problems arise due to the use of pacing in combination with *congestion control* (as opposed to *congestion avoidance*).

2.5.4 TCP Vegas

Brakmo, *et al*,[19] introduce TCP Vegas, an ingenious window-based congestion avoidance scheme. Vegas estimates the Round-Trip Propagation Delay (RTPD) as the minimum RTT measured so far. Vegas denotes the RTPD estimate as $BaseRTT$. Each Vegas connection estimates the bandwidth-RTPD product as the connection's current throughput times $BaseRTT$ and then adjusts its window size to fill the bandwidth-RTPD product plus between α and β packets.

Once per RTT, the sender remembers the transmission time of a distinguished segment. When the corresponding ack arrives, the sender computes the RTT and does the following:

1. The sender computes the *Expected* throughput as $\frac{wnd}{BaseRTT}$.

2. The sender then computes the *Actual* throughput as $\frac{wnd}{RTT}$.

3. Vegas then computes the backlog estimate, $diff = (Expected - Actual) \times BaseRTT$

4. The sender then adjusts the window

$$cwnd = \begin{cases} cwnd + 1 & \text{if } diff < \alpha \\ cwnd - 1 & \text{if } diff > \beta \\ cwnd & \text{otherwise} \end{cases}$$

When there is sufficient buffer for Vegas to maintain between α and β packets per connection, Vegas stabilizes meaning that congestion windows remain near constant. When there is insufficient buffer for Vegas to stabilize, Vegas reverts to Reno (multiplicative decrease, fast retransmit, fast recovery). However, when Vegas stabilizes, it achieves proportional fairness [83] without bias against connections with longer propagation delays [50, 89] and is more *efficient* in terms of both throughput and delay than the same set using Reno.

However, TCP Vegas has several known problems:

- Reno connections do not back off until loss occurs, meanwhile Vegas connections yield to reduce RTT[2]. At worst, Reno maintains large backlog and nearly starves Vegas[89]. Let i refer to the ith connection. Let b_i denote $BaseRTT$. Let q denote the average queue delay incurred by Reno connections. Let d_i denote the round-trip propagation delay. Assume $b_i = d_i$ and that the estimated backlog stays within α and β. We then expect that TCP Vegas connections should obtain average window sizes around,

$$\begin{aligned}
diff &= cwnd_i(1-\frac{b_i}{d_i+q}) = cwnd_i(1-\frac{d_i}{d_i+q}) \leq \beta, \\
cwnd_i &\leq \beta\frac{d_i+q}{q}, \\
\lim_{q\to\infty} cwnd_i &\leq \beta.
\end{aligned} \tag{2.8}$$

[89] provides an alternate analysis validated with simulation for a Reno connection competing with a Vegas connection. The analysis shows the ratio of Vegas to Reno throughput as approximately $\frac{2\beta}{B-\beta}$ where B is the buffer size. For large buffer size, Vegas throughput approaches zero. Thus Vegas might only be deployed into the Internet if it is isolated to its own traffic class.

- Vegas interprets an increase in propagation delay as congestion. Connections backoff until the backlog estimate is $\leq \beta$ and RTT $\to p_i$,

$$\begin{aligned}
cwnd_i(1-\frac{b_i}{d_i}) &\leq \beta, \\
cwnd_i &\leq \beta\frac{d_i}{d_i-b_i}, \\
\lim_{d_i\to\infty} cwnd_i &\leq \beta.
\end{aligned} \tag{2.9}$$

A similar analysis appears in [17]. We simulate increases in propagation delay in Section A.10.

Mo and Walrand [89] suggest limiting the history on the RTPD estimate by computing $BaseRTT$ as the minimum of the last k RTT samples, but as we show in Section A.10, this policy can lead to unbounded queue.

- Mo and Walrand[89] point out that when one or more connections arrive at a bottleneck with a standing queue, the new arrivals obtain an exaggerated RTPD estimate. The new arrivals then adjust their window sizes to incur a *backlog* between α and β in addition to the standing queue. As long as nothing disturbs the system, the standing queue can remain forever. Mo suggests using RED (see Section 2.6.1) to solve these problems. We would prefer a method that does not require support from the bottleneck to clear standing queues.

- As a consequent of the prior problem, when multiple connections consecutively start, latter connections tend to grab a larger share of the bottleneck.

  [17] provides a close facsimile of the following analysis and validates it with simulation. Assume all connections settle with $diff$ approximately equal to α.

$$cwnd_i = \frac{\alpha}{1 - \frac{b_i}{d_i + q}}. \tag{2.10}$$

  The throughput for connection i is thus

$$rate_i = \frac{cwnd_i}{RTT_i} = \frac{\alpha}{q + (d_i - b_i)}. \tag{2.11}$$

  Thus any connection that overestimates $BaseRTT$ obtains larger throughput.

Most of TCP Vegas's problems arise from inaccurate RTPD estimates. Riviera, introduced in chapter 3 also relies on RTPD estimation, but Riviera introduces mechanisms that allow for frequent queue drain and hence frequent occasion for the min RTT to approximate RTPD.

2.5.5 Other Implicit, Window-based, End-to-end, Congestion Avoidance Schemes

Besides Vegas, a number of implicit congestion avoidance schemes have been proposed in the literature. Among them Tri-S [116], CARD [62], and Mo-Walrand [90]. Tri-S determines congestion based on the derivative of the throughput with respect to window size. CARD maximizes network power, where network power is defined as the ratio of throughput to delay. Network power is maximum at the knee in Figure 2.3.

Mo-Walrand uses Vegas's backlog estimator, but Mo-Walrand connections adjust their window size in proportion to their distance from a target backlog. Mo and Walrand have shown that this scheme achieves proportional fairness. The Mo-Walrand scheme shares the same problems with Vegas resulting from poor RTPD estimation.

2.5.6 ATM ABR

ATM Available Bit Rate (ABR) is an adaptive bandwidth service that aggregates traffic between the edges of an ATM cloud. By using rate control, ATM ABR separates traffic control from error control and allows for simple application of congestion avoidance to an aggregate. For similar reasons, we use rate-control in the design of Riviera in chapter 3.

As mentioned in the introduction, ATM relies on significant bottleneck support to provide *fair*, *efficient*, and *responsive* traffic control. ATM ABR relies on explicit rate congestion avoidance applied between the edges of an ATM cloud. If the sender and receiver implement ATM then the ATM ABR can be used end-to-end.

ATM ABR actually specifies only the communication protocol. Many traffic control schemes have been proposed, and the standard does not mandate a particular scheme.

Mike Hluchyj[36] proposed the first rate-based scheme for the ABR service. His scheme is a rate-based variant of DECbit (see Section 2.5.1).

ATM ABR's rate-based bit based schemes share some of DECbit's and TCP's problems:

- All AIMD schemes must oscillate many times to reach fairness.
- All covered schemes experience "beat-down" in the sense that flows passing through multiple switches have a higher probability of encountering congestion and thus obtain less throughput [13]. As before 'beat-down" may be a desirable property though the fairness of ATM ABR's bit-based schemes have not been thoroughly studied.

ATM ABR's rate-based bit-based schemes also have problems peculiar to themselves:

- Rate-based schemes lack ack-clocking. Instead the queue length grows with the size of the rate mismatch, and thus rate-based control may incur significant transient queues. Consider a case where capacity changes frequently due to competition from other traffic classes, a single drop in capacity may require multiple downward adjustments, and all the while the queue grows linearly with the size of each rate mismatch.
- As with DECbit, the switches use queue length to determine whether to mark a control packet, but with rate-based control inducing significant queue oscillation, queue length is a difficult metric to use. Consider that with window-based control, the queue length is never larger than the sum of the windows of all connections passing through the bottleneck. With rate-based control the queue length can grow as a

function of the size of the rate mismatch and each source's transient response to congestion. The queue lengths can grow quite large as multiple downward adjustments to rate may be necessary to alleviate congestion.

One of our proposed schemes for Edge-to-edge Control is based on bit-based feedback, but only in the forward direction. In the reverse direction, our scheme feeds back the receiver (egress) measured rate. This additional information allows the sender (ingress) to backoff quickly to a rate close to but below the rate received at the egress. As a result queue growth is stopped quickly and little oscillation occurs.

To solve many of the problems associated with bit-based rate control, Charny *et al* [22] suggest that switches compute the exact rate that each source should send and then feed this rate back to each sender. Thus the system would quickly converge on *efficiency* and *fairness* though explicit rate computations impose significant *complexity* on every switch. Charny's approach has since been improved [66, 64, 107, 110]. All of the algorithms are similar and here we present the cogent aspects of one of them, ERICA [66].

With ATM ABR each source sends a control packet known as a Forward Resource Management (FRM) cell every nth cell. The FRM cell contains an explicit rate field typically set to the source's peak rate. Backward Resource Management (BRM) cells flow from the destination to the source. Each switch monitors the sends rates of each VC and then reduces each BRM's explicit rate to the VC's max-min fair share.

Once per averaging interval, ERICA computes two things: the overload factor (z) and an even allocation of the target ABR capacity (i.e., target ABR capacity / number of active VCs). The overload factor is the ratio of the ABR input rate to the target ABR capacity.

As BRM cells arrive for each VC, the ERICA switch computes the appropriate *explicit rate* for the BRM's VC. If the bottleneck is underutilized (i.e., $z < 1$), ERICA computes the VC's explicit rate as the max of two values: the VC's current send rate divided by z and the maximum rate allocated in the previous averaging interval. By dividing by z, the ABR input rate rapidly approaches efficiency. By allocating to each VC at least the max of the previous interval, ERICA approximates fairness. If the bottleneck is fully utilized, then ERICA computes the VC's explicit rate again as the max of two values: 1) the VC's current send rate divided by z, and 2) the even allocation computed in the last averaging interval. Thus each VC gets at least an even share. Then only if the VC's computed explicit rate is smaller than the BRM's explicit rate does ERICA set the

BRM's explicit rate to the computed explicit rate.

The ERICA algorithm converges on a global max-min fair rate allocation, and typically achieves efficiency within two or three RTT's. With a proper choice of algorithms to set the target ABR capacity, it can even be shown to achieve bounded delay [66]. In all, ERICA is more *efficient*, *fair*, and *responsive* than TCP Reno, Tahoe, Vegas, or DECbit.

ERICA has lower time complexity than WFQ or its variants because it performs only $O(1)$ computation on each cell arrival, but it still requires $\Theta(n)$ storage to maintain a table of VC current send rates. DECbit maintains only $O(1)$ state, while TCP Tahoe, Reno and Vegas require no congestion state variables at the bottleneck. Lastly, ERICA is less deployable than TCP Tahoe, Reno, or Vegas, or WFQ and its variants because ERICA requires that all switches within an ATM cloud implement explicit rate control and that all end-systems or edges of an ATM cloud respond to explicit rate feedback. Further, extending an ATM cloud from end-to-end requires deploying ATM end-to-end.

2.5.7 TCP Trunks

Kung, Wang, and Chapman's *TCP Trunks* [79, 21] are quite similar to EC Trunks. TCP Trunks aggregate flows passing between two network points, may overlie LSPs or VCs, and use TCP-like congestion control between the ends of the TCP Trunk. At the TCP Trunk sender-side, each aggregate is buffered and accompanied by a "management TCP connection." The "management TCP connection" spans between the ends of the TCP Trunk and every VMSS (Virtual Maximum Segment Size[115]) bytes the TCP Trunk sender-side injects a packet carrying no data. Here VMSS is a parameter set on the same order of magnitude as the MSS used by end-systems. The TCP Trunk receiver-side acknowledges these management packets as soon as they arrive but without returning any data. As each management TCP ACK arrives at the TCP Trunk sender-side, the TCP Trunk sender-side *ack clocks* VMSS worth of bytes from the aggregate into the network. The management TCP connection reacts to loss just as a normal TCP connection and thus regulates the flow of user packets entering the trunk.

To provide lossless behavior within the TCP Trunk, TCP Trunks rely on intermediate routers to drop management packets sufficiently long before dropping user packets so that the TCP Trunk has adequate time to backoff. For example, TCP Trunks can overlie a DiffServ-enabled network that implements the AF building blocks. All management packets are then marked as OUT and user packets marked as IN.

In combination with a token bucket and admission control, the TCP Trunk sender can admit an additional guaranteed bit rate worth of packets into the network. We construct a similar service for EC trunks called a Quasi-Leased Line in chapter 5.

Employment of a token-bucket is the reason we say that TCP Trunks use rate-window hybrid congestion control in Table 2.1. Otherwise, TCP Trunks are window-based.

TCP Trunks and EC trunks differ as follows: TCP Trunks do not mandate isolation. EC mandates isolation and thus can utilize congestion avoidance that is incompatible with TCP. TCP Trunks are compatible with TCP traffic and thus can share a queue with best-effort traffic, but TCP Trunks are then subject to best-effort denial of service attacks. To be fair, TCP Trunks can be isolated from best-effort traffic. Besides, TCP Trunks use TCP because TCP has been widely studied and because buffer management algorithms have been designed for TCP[21]. TCP Trunks require buffer management at the bottleneck to avoid loss. EC trunks do not require buffer management at the bottleneck. Each TCP Trunk introduces a control packet every VMSS bytes and thus roughly doubles the number of packets on the network. Riviera (see chapter 3), our congestion avoidance algorithm proposed for EC trunks, generates only one control packet per RTT.

2.5.8 Congestion Manager

Balakrishnan and Seshan introduce a Congestion Manager (CM)[7, 34] component at end systems that shares congestion state between flows with the same end-to-end path and provides applications with the ability to control what bytes are sent at transmission time (ALF[25]). Our interest in the CM derives from its handling of aggregate flows.

The CM provides only congestion control and leaves error control to the transport. Applications or their transports can notify the CM via an API when error control detects a loss or when an ECN CI marked packet arrives (see Section 2.6.3 regarding ECN). The CM then schedules the next packet transmission using TCP-like AIMD and later notifies the application via callback when the application is allowed to send.

To provide congestion control for transports that may not implement error control (e.g., UDP) or that generate infrequent feedback, the CM engages in its own congestion control protocol. The sender CM injects a control packet twice per RTT. The control packet (called a *probe*) carries a probe sequence number and a count of the number of bytes sent since the last probe. If the receiver sees a difference between the number of bytes received and the number reported in a probe, the receiver signals loss back to the sender. The probe sequence number is used to detect lost probes. This byte counting method will properly detect loss so long as a probe packet is not reordered with a data packet. When a probe is reordered, subsequent probes will correct the byte count, and thus the CM can recover from a false positive loss detection by undoing the backoff.

Riviera (see chapter 3) uses the same CM protocol mechanisms for detecting loss of data packets, though Riviera ignores the effects of packet reordering under the assumption that reordering occurs rarely, and Riviera uses a necessarily modified version of the CM protocol mechanisms for detecting and recovering from lost control packets.

2.5.9 End-to-end Closed-loop Differentiated Service

All of the traffic control schemes discussed achieve some form of unweighted fairness. In the context of differentiated services we want to weight bandwidth shares according to customer needs and willingness to pay. Crowcroft and Oechslin [28] achieve weighted TCP-style fairness.

Crowcroft and Oechslin implement MulTCP[28]. MulTCP tricks a TCP connection to behave as n TCP connections by increasing the single MulTCP connection by n packets in each RTT, and then when a loss occurs the TCP connection reduces its congestion window by only $\frac{n-0.5}{n}$. However, even with TCP SACK, MulTCP cannot achieve accurate weighted shares above 10-to-1. They argue that this is because one MulTCP is more likely to experience a timeout than 10 individual TCP connections.

Nandogopal *etal*[92] also attempts to modify end-system congestion control to achieve weighted sharing. Both [92] and [28] rely on loss as the primary indicator of congestion and attempt to retain compatibility with existing TCP and thus allows weighted connections to share bottlenecks with best-effort connections. However, from our prior discussion, we know that mixing best-effort and QoS subjects QoS traffic to potential denial-of-service attacks. If QoS traffic is separated from best-effort traffic then one can abandon TCP compatibility provided there is a better alternative.

[93] similarly provides end-to-end service differentiation but relies on an AIMD, rate-based, congestion-avoidance scheme that detects congestion based on inter-packet separation. The use of inter-packet separation, however limits the domain of applicability to ¡100Kbps bandwidths[93].

2.5.10 Other Traffic Control Schemes

NETBLT [27] is one of the earliest rate-based schemes. It is also one of the earliest congestion avoidance schemes. As with TCP Tahoe, Reno, and DECbit, NETBLT uses AIMD, but in a rate-based context. As with Tahoe and Reno, NETBLT is implicit. NETBLT breaks up communication into blocks and then negotiates a rate for each block where a block is then divided into some number of packets. The rate r is unchanged while transmitting the block. NETBLT detects congestion when the rate received at the receiver is less than the transmission rate and then backs off multiplicatively to βr for the next block where β is a parameter less than 1.

NETBLT has two basic problems: 1) when rate differences are small, congestion may go undetected leading to unstable queue growth, and 2) rates are only renegotiated once per block possibly leading to unnecessarily long periods of loss. Our proposed implementations of Edge-to-edge Control do not use rate difference to detect congestion and change the rate once per RTT rather than once per block.

Mishra and Kanakia (MK) [88] present a rate-based, hop-by-hop scheme. Because the traffic controller sits at each switch, the control mechanisms have better knowledge of the bottleneck state and can rapidly adjust the switch input rates by notifying the upstream switches. Thus MK is efficient. However, MK requires per-flow state and thus MK is *complex*. MK is also only *deployable* within a network domain that supports MK traffic control at every switch.

Following Mishra and Kanakia, Kung *etal*[76, 75] introduce a credit-based, hop-by-hop scheme that competed with explicit-rate based schemes for ATM ABR. One key benefit of Kung's credit-based scheme is the ability to eliminate all congestive loss: each downstream switch periodically notifies upstream switches of the number of available buffers (i.e., available credit) for each VC. Upstream switches then stop sending a VC's cells when the VC has exhausted its credit. Unfortunately as MK, Kung's credit-based scheme imposes per-VC state on each switch and is deployable only within a domain that implements the credit-based scheme at every switch.

2.6 Buffer Management

Buffer management algorithms determine the packets that are buffered and those that are dropped. By carefully allocating drops, buffer management algorithms improve TCP performance: lower delay, improve fairness, reduce burst drops, and reduce timeouts.

The seminal buffer management algorithm for the Internet is RED[43]. Recommendations for early random dropping [56, 51] preceded RED, but RED became the gold standard for later work: ARED[37], BRED[4], SRED[95], and FRED[80]. In this section we discuss only RED and FRED in detail.

2.6.1 RED

Random Early Detection (RED)[43] buffer management provides congestion avoidance by randomly dropping or marking packets. As stated, the goal of congestion avoidance is to keep the system in the region of high throughput with low delay. RED attempts this by either dropping or marking packets early enough to avoid large queues and late enough to avoid dropping or marking packets due to transient burstiness. RED has the additional stated goals: avoid bias against bursty flows and avoid the global synchronization that occurs when multiple connections experience simultaneous loss. This global synchronization leads to wide oscillation between large queue delay and low utilization. RED does not take into consideration that TCP connections with tiny windows experience timeouts more readily than TCP connections with larger windows. We return to this problem in Section 2.6.2.

The RED algorithm does not distinguish internally between dropping and marking. When congestion is detected, the gateway uses some external knowledge to decide whether a drop or mark is appropriate (see Section 2.6.3). RED assumes that end-systems will adjust their send windows in a similar manner to a mark as to a drop. For the rest we will refer to drop events meaning either drop or mark.

The drop probability increases linearly from 0 to max_p as average queue length, avg, increases from ϵ_{min} to ϵ_{max}. Once the average increases above ϵ_{max} all packets are dropped. By properly setting decay constants in its averaging algorithm, network administrators can tune the point between transient and persistent queue in order to accommodate bursty connections. By varying the drop probability, the algorithm accommodates a range of traffic loads. When few connections traverse a bottleneck, the bottleneck need only drop a handful of packets to notify sufficient connections of congestion to reverse queue growth. Because each connection increases linearly during TCP's congestion avoidance phase, fewer connections implies slower queue growth. As a result, one would expect the system to oscillate around a point somewhat above above but near ϵ_{min}. Conversely a load containing more connections should oscillate about a point with larger average queue length.

RED interoperates with existing TCP. Does not require upgrades from end-systems. RED operates at the bottleneck without cooperation from other components and is thus incremental deployable. RED does however suffer from placing some complexity at the bottleneck, but this is often considered tolerable because RED performs only constant time, constant space operations on each packet and does not require per-flow state. RED's largest drawback is the difficult in setting parameters. Such difficulty spawned variants such as ARED[37] and SRED[95]. In simulation, ARED and SRED improve performance under a wide range of loads but with the expense of slightly increased complexity at the bottleneck. RED also does not handle non-adaptive flows well[4].

May*et al*[87] further demonstrate that in realistic settings, RED may not improve performance *regardless of parameter settings.* May *et al* performed experiments with real routers given reasonably realistic traffic [87] containing ftp and web bursty transfers and audio/video streams across a single router. As with all such studies, the results may not extrapolate to larger topologies with more flows, but the results are suggestive enough to urge caution before spending the effort to deploy mechanisms that may provide little or no benefit.

2.6.2 FRED

Lin and Morris[80] demonstrate that TCP exhibits substantial unfairness to flows with windows smaller than four packets. TCP Tahoe requires three duplicate acknowledgments to trigger fast retransmit. When a connection has a window size of less than four packets then a single drop triggers a timeout. Because connections with larger windows are less likely to experience timeouts, large connections tend to remain large, thus TCP is said to exhibit a bias against connections with tiny windows.

FRED aims to improve TCP's fairness over the region where there is sufficient buffer for most connections to achieve a window size of at least four. When the average queue length is less than ϵ_{max}, FRED performs RED dropping for only those connections with at least four packets buffered. If the average queue length exceeds ϵ_{max} then as with RED, all packets are dropped.

FRED attempts to improve fairness in the presence of non-adaptive flows by dropping arriving packets from connections that have equal or more than max_q packets backlogged. In addition, each time a packet arrives in connection i when that connection has more than max_q packets or when i has more than twice the average number of packets in the queue and $avg > \epsilon_{max}$, FRED increments a *strike* count by 1. Once a connection has obtained more than 1 strike, all packets from the connection are dropped whenever the connection has more than the average number of packets backlogged. If a UDP flood occurs, the flood is constrained to an even share of the occupied buffer.

One could argue that strikes should be counted on larger than per-packet time-scales since connections cannot adapt to drops more quickly than 1 RTT. However, we leave such considerations to others if FRED.

FRED requires per-flow state for flows with any packets backlogged. Thus FRED has bounded but certainly not trivial state requirements. FRED is an example of how increased complexity provides for better performance, at least for infinitely long flows in simulation.

2.6.3 ECN

As stated, RED and its descendants can either mark or drop packets. When packets are marked, it is up to the end-system to interpret a mark as a loss and properly adjust congestion window size. In the IETF, bottleneck marking is called Explicit Congestion Notification (ECN[98]). ECN has certain advantages. It has been shown to reduce loss rates [38], and a bottleneck can mark a packet with less than a four packet window size without incurring a timeout. However, a non-conformant source can mark packets as ECN capable to reduce loss rates without actually backing off when the CE bit is set. Also unlike buffer management algorithms that simply drop packets, ECN requires upgrades both at bottlenecks *and end-systems*. Fortunately ECN-capable and ECN-incapable traffic can share the same bottleneck allowing ECN to be incrementally deployed.

2.6.4 Buffer Management Summary

All buffer management algorithms share a common failing, they impose *complexity* on the bottleneck. As stated in the introduction, EC requires nothing more than tail dropping bottlenecks. Buffer management algorithms may often be deployed at EC edges to perform final loss/mark assignment (see chapter 4).

Although buffer management algorithms require upgrades of bottlenecks, RED and its descendants can be deployed one router at a time. ECN requires a longer deployment horizon but it too can be deployed one router and one end-system at a time supposedly increasing in benefit throughout the deployment. EC may not require upgrades at end-systems, but as stated EC's classwise isolation imposes both bottleneck reconfiguration and policing at network edges. The policing requirement makes EC harder to deploy than RED or its descendants, but we see no way around requiring isolation for QoS (see Section 1.2).

2.7 Differentiated Services

In the introduction (see Section 1.1.4), we introduced DiffServ's AF and EF PHB building blocks. In this section we describe how AF or EF PHBs can be combined with other building blocks to implement data plane component of two services.

2.7.1 AF and RIO

Clark and Wang[24] created the Explicit Allocation Framework by assuming two bit markings: in and out, which are essentially similar to the AF PHBs. In Clark's Explicit Allocation Framework, the network is broken into domains and each domain's edges mark or remark packets as "in" or "out" depending on policy implementation. The bottleneck then applies drop precedence. The objective is to allow customers to obtain a bit rate that is above the contracted bitrate. This model inspired DiffServ.

Clark and Wang propose an ingress marker that averages rate over a time window and then marks each connection's packets as "in" so long as the connection's average rate remains below 1.33 times the connection's target rate. Ideally TCP should then oscillate between 0.66 and 1.33 times the target rate maintaining a long-term average equal to the target rate.

Bottlenecks then implement RED with In and Out (RIO). RIO is two parallel RED algorithms. One RED drops "in" packets based on the number of "in" packets in the queue and the other RED drops "out" packets based on the total queue length (i.e., in+out). The drop thresholds for each RED algorithm are then set so that RIO will begin dropping "out" packets sufficiently long before dropping "in" packets to avoid dropping "in" packets during normal operation. The network must also be properly provisioned to avoid persistent oversubscription.

RIO has certain problems: it relies on RED which is difficult to configure. Providing customers with the desired assurance has proven difficult to achieve for large assurances[102, 54]. RIO as with RED does not protect well against non-adaptive flows[74, 103]. Appropriate buffer management, such as FRED with In and Out (FRIO), may better handle UDP but FRIO increases the complexity of the bottleneck.

2.7.2 Virtual Wire and Expedited Forwarding

The Expedited Forwarding Per-Hob Behavior (PHB) is defined in [59]:

> The EF PHB is defined as a forwarding treatment for a particular DiffServ aggregate where the departure rate of the aggregate's packets from any DiffServ node must equal or exceed a configurable rate. The EF traffic SHOULD receive this rate independent of the intensity of any other traffic attempting to transit the node. It SHOULD average at least the configured rate when measured over any time interval equal to or longer than the time it takes to send an output link MTU sized packet at the configured rate.

In other words, the service rate must exceed the arrival rate, even on small timescales.

Van Jacobson *et al* [60] propose a *virtual wire* building block that emulates a physical wire. This virtual wire is constructed by shaping at the ingress edges such that the traffic entering the virtual wire never exceeds the EF's configurable rate at any DS nodes. Packets may queue at the ingress but little delay jitter is experienced within the network. If the intermediate network perfectly implements the EF PHB then the flows transiting such networks experience bounded delay and bounded delay jitter regardless of the topology and other traffic sharing the same links.

This architecture allows both non-bursty voice or similar delay and jitter sensitive traffic to exist in the same network with bursty data traffic. Capacity is available immediately for non-bursty traffic delayed only by the time to complete transmission of the current packet plus any queued EF packets. However in the absence of EF traffic, the best-effort traffic is allowed to use the entire capacity of the link. In this sense EF is more *efficient* then TDM.

Nichols *et al*[94] suggest an approximate implementation that elaborates on an earlier talk presented by Van Jacobson at the 1997 IETF. They propose that the EF PHB be implemented by passing all EF marked packets through a priority queue and all other traffic passes through a second queue. Such an implementation is attractive because of its simplicity and scalability.

Bennett *et al*[12] demonstrate a class of networks in which the delay jitter bound is violated. They demonstrate that the argument in [60] ignores jitter accumulation due to non-EF traffic, and that fixing the problem requires redefining the EF PHB and loosening the delay and jitter guarantees. The modified virtual wire building block can then be implemented using a priority queue as well as several other schedulers.

As an aside, the virtual wire building block assumes stable routing. As long as route changes occur infrequently then no route-pinning or QoS routing is necessary. Internet2[4] is experimenting with virtual wires without any route pinning mechanisms under the assumption that any route change is likely due to an outage which would result in inevitable rerouting. Where graceful handling of route changes is deemed necessary, one can use MPLS (see Section 1.1.5).

2.7.3 Core Stateless Fair Queuing

Core Stateless Fair Queuing (CSFQ)[109] proposes an architecture that follows the same spirit as DiffServ in keeping the core simple by moving some of the complexity to the edges. In particular the architecture is *stateless* in the sense that core routers maintain no per-flow state. Instead edge routers stamp the average rate into packets and then core routers use the per-packet labels to probabilistically drop packets from faster flows before the packets enter the transmission queue. The transmission queue is simply served in FIFO order. Since faster flows are penalized with more drops the system approximately fairly shares the bottleneck.

More specifically stated, at each packet arrival in flow i the ingress edge exponentially averages the flow arrival rate r_i. This rate is then stamped in the passing packet. The core router estimates a fair share rate α then drops arriving packets with probability

$$p = \max\left(0, 1 - \frac{\alpha}{r_i}\right) \tag{2.12}$$

Equation 2.12 has the property that if a flow is below its fair share then no packets are dropped in the flow. However, if a flow is above its fair share then an appropriate portion of its packets are dropped so that only a fair share of the flow's packets actually enter the queue.

The tricky part is in estimating α without relying on per-flow state. First let K_c denote a time interval length and μ denote the bottleneck capacity. The fair share estimation algorithm records the maximum rate label seen during each interval. If the bottleneck is uncongested then α is set to the maximum rate seen in the previous interval. When congestion arises, the rate F that the dropper admits packets into the queue is exponentially averaged. Since prior to congestion α followed the maximum flow rate, the drop drop probability is probably too low thus the dropper admits more packets into the queue

[4] from correspondence with Stanislav Shalunov.

then it should. CSFQ fixes the situation by adjusting α downward until F and μ become equal. More specifically, with each packet arrival

$$\alpha = \alpha * \mu / F \tag{2.13}$$

To understand how this works, consider the situation where all flows are increasing their rates slowly to absorb available capacity and then each flow slows down when it experiences loss (e.g., this is the behavior of TCP). At the beginning of congestion, α equals the fastest flow's rate, thus all flows are sending below the fair share estimate and the dropper initially drops no packets. Consequently, the transmission queue grows. The fair share estimator detects that the queue is growing (i.e., F exceeds μ) and adjusts α downward. Since α is now lower than the fastest flow's send rate, the dropper begins to drop packets from the fastest flow. The fastest flow backs off and another flow becomes the fastest flow. The process continues always pushing down faster flows and allowing slower flows to send without experiencing loss.

CSFQ is *efficient* in the sense that it does not drop packets from flows when the arrival rate is less than the bottleneck capacity even if the capacity is unevenly shared. CSFQ has low *complexity* compared to WFQ, SCFQ, SFQ, and Virtual Clock because core routers require only $O(1)$ time and space complexity.

CSFQ likely has a smaller *convergence time* to fairness then Edge-to-edge Control because it relies on mechanisms placed at the bottleneck to explicitly drop packets from larger flows. However, CSFQ relies on loss to control end-to-end flows, Edge-to-edge Control does not. CSFQ is also less *deployable* than Edge-to-edge Control because it requires that all core routers and edge routers be upgraded, and it requires that routers and edge routers be upgraded simultaneously within a single domain. Edge-to-edge Control requires only that potential bottlenecks have a separate queue set aside (a configuration not an upgrade) and then edges can be added one customer at a time. Further CSFQ may have similar complexity to Edge-to-edge Control because CSFQ imposes a dropper but does not require isolation to a separate class. However, we would be remiss if we did not point out that if Edge-to-edge Control used the same deployment model as CSFQ then we could require that all entry points to a domain implement Edge-to-edge Control. At which point *all* traffic within the domain would be Edge Controlled and there would be *no need* to set aside a separate class for Edge Controlled traffic.

2.8 Performance Enhancing Proxies

TCP is a widely deployed, general-purpose, reliable transport protocol. As a result, TCP slowly evolves as it takes much time to upgrade the installed base, and TCP trades off generality for performance. *Performance Enhancing Proxies* (PEPs) are devices placed in the path of packets to modify the behavior of end-systems in order to introduce newer control mechanisms without upgrading end-systems or to introduce mechanisms that accommodate specific link characteristics or both. As with edges in Edge-to-edge Control, PEPs exploit their location in the network. For example, a proxy is usually near either the sender or receiver where there are typically fewer flows and thus more sophisticated mechanisms can be deployed. PEPs often operate across multiple protocol layers (network, transport, and possibly application layers). For example, SNOOP[8] drops duplicate acknowledgments to prevent end-systems from backing off when loss is not due to congestion. Detecting duplicate acknowledgments requires transport layer access.

Edges differ from PEPs only in application. PEPs are typically for managing congestion at or around a specific link rather than to differentiate service through wide-area networks. Instead of modifying TCP behavior to accommodate specific link characteristics, edge-to-edge control exploits aggregation of TCP and UDP flows to improve congestion control across wireline networks. As with PEPs, ingress edges may implement sophisticated, layer-violating control mechanisms. We survey relevant mechanisms in this section.

2.8.1 TCP Rate Control

Packeteer implements a PEP that uses TCP Rate control (TCPR [67]). TCP rate control takes as input the rate of the PEP's access link. The PEP monitors TCP traffic transiting the access link in order to determine each connection's rate and round-trip propagation delay. The TCP rate controller then employs a rate control algorithm such as Charny's algorithm [22] or ERICA [66] to allocate a rate, $rate\_allocation_i$, to each connection i. The PEP then computes the appropriate window size for each connection as

$$wnd_i = rate\_allocation_i * rtt_i. \tag{2.14}$$

The PEP then copies wnd_i into the receiver advertised window field in the TCP header of each acknowledgment heading back to each sender. Since each TCP sender will not increase its window size above the receiver advertiser window, TCP rate control can upper bound the number of packets transiting the bottleneck without requiring loss.

The exact rate control algorithm used by Packeteer is proprietary but Karandikar, Kalyanaraman, Bagal, and Packer [67][5] present a similar algorithm that periodically computes send rates according to Charny's algorithm[22] ATM switch algorithm. The connections converge on a max-min fair allocation of the access link.

To reduce burstiness, the Packeteer PEP also paces acknowledgments according to the desired rate. This does not require buffering acknowledgments. Instead the Packeteer PEP modifies the sequence number in each passing ack so as to avoid triggering fast retransmits and to maintain ack spacing. The PEP necessarily remembers the latest acknowledged sequence number for each connection to avoid acknowledging beyond sequence numbers it has seen in arriving acks. When rate allocations are forced to less than 1 MSS per RTT, the PEP sets each receiver advertised window to 1, however the PEP still spaces acks according to the rate allocation. Because of TCP's ack clocking, increasing ack spacing allows the TCP rate controller to further decrease each TCP's send rate to an arbitrarily slow speed.

TCP rate control requires per-connection state and access to TCP headers and acknowledgments. The complexity of the prior may prevent the switch from scaling to backbone rates, which is acceptable since TCP rate control is used on access links. The latter prevents TCP rate control from being used anywhere where data and acks do not pass through the rate controller (i.e., asymmetric paths) or where TCP headers are encrypted. Both events occur frequently inside ISP networks, but less frequently at access points.

Note that the Packeteer PEP attempts some control even when it does not have access to acknowledgments, but the exact mechanisms are not publicly available.

We are interested in TCP rate control as a compliment to EC. Once EC has pushed congestion back to the ingress edges, we can apply TCP rate control to divide the EC trunk's capacity between TCP connections without requiring loss. By weighting the max-min fair allocation, we can also implement edge-based service differentiation between TCP connections sharing an EC trunk (see Appendix D).

[5] Bob Packer is a co-founder of Packeteer, Inc.

3. Congestion Avoidance

We present a possible scheme for implementing the congestion avoidance component of Edge-to-edge Control: *Riviera*. Riviera employs a novel backlog estimator coupled with a novel control policy. Together they reduce the requirement for buffer management at bottlenecks and address problems with the backlog estimator used by TCP Vegas.

In the introduction and the survey chapters, we compared architectures and QoS mechanisms in terms of deployability, efficiency, fairness, protection, complexity, and responsiveness. We discussed complexity, deployability, and protection with respect to the EC architecture in Section 1.2. For the Riviera's design, we consider the constraints imposed by the EC architecture and appropriately restate the remaining criteria as common goals for data services (in order of decreasing importance):

1. **Avoidance of Congestion Collapse and Persistent Loss:** If the queue should grow to the point of loss, the system must backoff sufficiently to avoid persistent loss and the potential consequence: congestion collapse[56]. Congestion collapse refers to the occasion when significant loss leads to retransmissions dominating the network load thereby reducing the effective throughput to near zero. Edge-to-edge control does not perform retransmissions and thus cannot lead to such congestion collapse though end-to-end transports that implement error control without congestion control certainly might force congestion collapse if edge-to-edge control does not back-off sufficiently in response to loss. In consequence of this goal, Riviera degrades to Reno-like performance when there is insufficient buffer or bandwidth. This does not imply that EC control loops compete fairly with TCP connections.

2. **Avoidance of Starvation or Gross Unfairness:** Gross unfairness is a known problem of Reno under heavy loads and of Vegas when Round-Trip Propagation Delay (RTPD) is poorly estimated. Countermeasures must avoid starvation and gross unfairness under the widest possible operational range even at the expense of utilization.

3. **Stability Without Loss:** The queue length should remain bounded provided sufficient buffers and bandwidth exist at all queuing points within the network. When traffic trunks start, the initial transients may incur significant backlog. Loss dur-

ing transients is more tolerable than loss under steady-state conditions. Provided loss under transients is minimal and does not adversely affect performance when compared to existing end-to-end control schemes. TCP relies on loss to detect congestion and thus in the absence of bottleneck buffer management requires wide oscillations (i.e., TCP's saw-tooth). With bursty loss these oscillations tend to synchronize across multiple control loops leading to swings between long delay and underutilization[18]. Buffer management can smooth out these effects with some additional complexity at the bottleneck; complexity that we want to avoid if possible.

4. **High Utilization, Proportional Fairness, Delay Minimization, Transient Response Minimization:** We defined proportional fairness in Section 2.2 and transient response in Section 2.3.

   All of these are desirable performance optimization goals, but the ordering differs depending on the intended application. Furthermore there usually exists a trade-off. For example, delay is often minimized when utilization is lowered. Performance optimization only makes sense once somebody obtains non-negligible service (i.e., achieves goal 1), and few or no people are starved (i.e., achieves goal 2).

We organize this chapter as follows. We describe Riviera in Sections 3.2. We then provide further discussion of the properties of Riviera with supporting simulation. A wider array of simulations appear in Appendix A.

The full pseudocode for Riviera appears in Appendix B.

3.1 Differences between End-to-end and Edge-to-edge Control

In the survey (see chapter 2), we classified congestion control and avoidance mechanisms and considered their applicability to Edge-to-edge Control. We summarize the most cogent points here to provide context for the design.

Edge-to-edge Control operates on traffic trunks aggregating end-to-end flows. End-to-end flows within each trunk may use TCP, UDP, or any other transport. Edge-to-edge Control cannot

- depend on the existence of a sequence number field in every packet,
- require access to transport layer headers,

- impose additional fields in IP headers,
- assume the existence of acknowledgments passing in the reverse direction,
- nor incur the overhead of imposing edge-to-edge acknowledgments.

These constraints rule out window-based control leaving rate-based, credit-based, or rate-window control. We have chosen rate-based control because it reduces burstiness. We leave rate-window hybrids to future research.

3.2 Riviera

Riviera implements edge-to-edge congestion avoidance that like Vegas adjusts each control loop's load such that each control loop maintains a range of bytes backlogged in the path. Unlike Vegas, Riviera is rate-based, employs a (σ, ρ)-shaper at each edge to enforce a rate limit r, and this rate limit is modified according to a novel control policy designed to frequently drain queues in the path across a given operating region. Riviera attempts to drain queues in the path to enable accurate estimation of the Round-Trip Propagation Delay (RTPD) and thus solve one of Vegas' plaguing problems. In particular for Vegas (see Section 2.5.4), substantial increases in RTPD lead to gross unfairness and drastic underutilization (violations of goal 2 and 4).

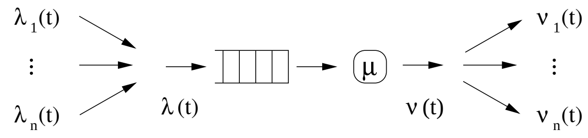


Figure 3.1: Traffic Control Model

In the description of Riviera we use the model shown in Figure 3.1. In particular λ_i refers to the send rate provided by the ith control loop. ν_i refers to the ith control loop's received (egress) rate. μ refers to the service rate of the bottleneck.

3.2.1 Accumulation: Riviera Backlog Estimation

To differentiate backlog incurred by a single control loop along its path from other notions of backlog, we introduce the term *accumulation*. Riviera estimates accumulation as the number of packets that have entered the network up to some time t and the number of packets that have left by time $t + b_i$. Riviera uses the departure of a control packet from the egress t^{out} as time t, and then counts the number of arrivals from $t^{out} + b_i$ until

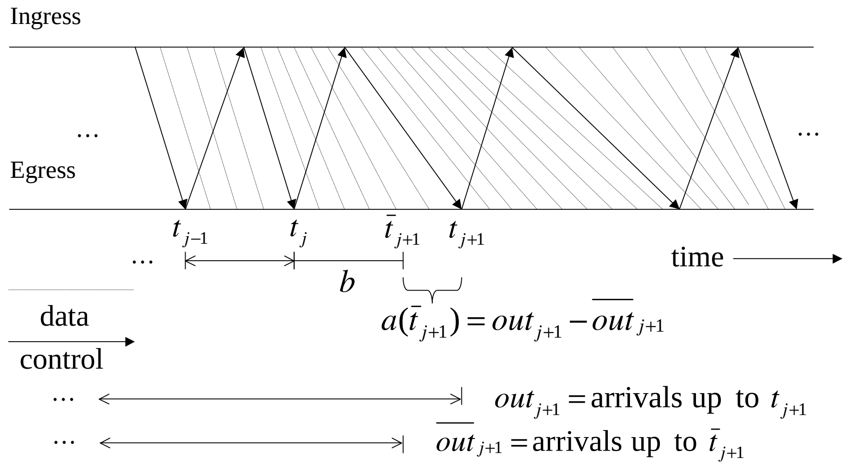


Figure 3.2: Accumulation Computation Timeline

the next arrival of the control packet (see Figure 3.2). With Riviera, the accumulation estimate is potentially based on several packet arrivals.

As shown in Section 2.5.4, Vegas estimates the ith control loop's accumulation q_i as the difference between the window size w_i and the expected number of packets propagating (i.e., not backlogged), $\hat{q}_i^v = (Expected - Actual) \times BaseRTT = \frac{w_i}{T_i}(T_i - b_i)$, where $\hat{q}_i^v$ estimates q_i, b_i denotes i's BaseRTT, and T_i denotes i's RTT.

Unfortunately Riviera cannot apply the Vegas accumulation estimator directly because Riviera is not window-based. Riviera could use a rate-based approximation of the Vegas accumulation estimator: $\hat{q}_i^v = \nu_i(T_i - b_i)$. As Vegas is implemented in `ns`, the RTT used in the backlog estimation is averaged across the round-trip times of each segment in the last window. Riviera cannot depend on the existence of acknowledgments in which to perform such a fine-grained round-trip estimate. Because Riviera bounces a single control packet back and forth, at best Riviera can measure round-trip time once per round-trip. Accumulation potentially reduces this problem.

With Little's Law, we now provide a trivial proof that for general arrival and service time distributions and a system in equilibrium that Vegas' backlog estimator $\hat{q}_i^v$ is unbiased, that is $E[q_i] = E[\hat{q}_i^v]$, under the condition that

$$E[T_i - b_i] = E[\text{i's queue delay in the path}] \ . \tag{3.1}$$

Condition (3.1) holds if b equals RTPD and there is no bias in the RTT samples. Because Vegas can average the RTT over all segments in the previous window, the RTT used in the Vegas accumulation estimator is likely unbiased.

By removing condition (3.1) and adding the condition that packets are "roughly" evenly spaced, we prove that accumulation is also an unbiased estimator. We define "roughly" more rigorously in a moment.

With these constraints the proof that the Vegas backlog estimator is unbiased boils down the following:

Theorem 3.2.1 *Vegas' accumulation estimator $\hat{q}_i^v$ is an unbiased estimator of q with condition (3.1) and no loss occurs.*

Proof.

Let q_i be a random variable denoting the current amount of traffic queued or in service in the path for control loop i. Consider a system that comprises the queues in a network. Let T_i denote the average RTT. Note that because of the steady-state condition, $\lambda_i = \nu_i$. We see that from Little's Law

$$E[q_i] = E[\text{number in net}] - E[\text{number on wire}] = \lambda(T_i - b_i) = \nu(T_i - b_i) = E[\hat{q}_i^v] \ . \quad (3.2)$$

□

Now we consider Riviera's accumulation estimator $\hat{q}(t)$. Riviera provides accumulation estimates at only discrete times corresponding to the arrival times of the control packet at the egress. Let t_j denote the jth arrival time of the control packet at the egress. Let O_j denote the amount of traffic arriving at the egress and departing the network over the time interval $[t_{j-1}, t_j]$. Let $\overline{O}_j$ denote the amount of traffic arriving at the egress over the interval $[t_{j-1}, t_{j-1} + b]$. Let $\hat{q}_j$ denote the accumulation computed at time t_j. Let q_j denote the ith control loop's backlog in the path at time t_j. On T, $\hat{q}_j$, and q_j, we omit the subscript i because all variables are taken with respect to control loop i.

Theorem 3.2.2 *Riviera's accumulation estimator $\hat{q}_j$ is an unbiased estimator of q_j with the conditions that (1) BaseRTT b equals the round-trip propagation delay, (2) no loss occurs, and (3) packets are "roughly evenly spaced" meaning $E[\overline{O}_j] = \frac{b}{T}E[O_j]$.*

Proof.

We must prove that $E[\hat{q}_j] = E[q_j]$.

By definition,

$$\begin{aligned}\hat{q}_j &\equiv a(t_j) \equiv out_j - \overline{out}_j \ , &(3.3)\\ O_j &\equiv out_j - out_{j-1} \ , &(3.4)\\ \overline{O}_j &\equiv \overline{out}_j - out_{j-1} \ , &(3.5)\end{aligned}$$

Subtracting (3.4) from (3.5) yields (3.3), thus

$$\hat{q}_j = O_j - \overline{O}_j \ . \tag{3.6}$$

The equality $E[\overline{O}_j] = \frac{b}{t}E[O_j]$ and (3.2) implies

$$\begin{aligned}E[\hat{q}_j] &= E[O_j](1 - \frac{b}{T}) = \frac{E[O_j]}{T}(T - b)\\ &= \nu(T - b) = E[q_j] \ .\end{aligned}$$

□

The "roughly evenly spaced" condition is necessary to account for recurring groups of back-to-back packets. With window-based schemes, if the control packet resides at the beginning of a window, all packets in the window tend to arrive tightly packed after the control packet leading to a gross backlog underestimate. Unfortunately, due to ack-clocking such problems tend to persist across many round-trips. Consequently, the Riviera accumulation estimator does not work well with window-based schemes. We saw this exact effect in simulation with a modified version of TCP Vegas that uses accumulation. With rate-based schemes, cross traffic and different packet sizes may lead to back-to-back packets as packets transit the edge-to-edge path, but because the ingress spaces back-to-back packets we would not expect such groups to persist from round-trip to round-trip.

Proving that Riviera implements an unbiased accumulation estimator under specified conditions is hardly sufficient to demonstrate the estimator's effectiveness. We must study the relationship between Riviera and Vegas accumulation estimators thoroughly in later work.

As with the Vegas accumulation estimator, the Riviera accumulation estimator requires an RTPD estimate. As suggested by [89] for Vegas, Riviera uses the minimum of the last k RTT samples. Riviera's backlog and RTPD estimators combine to provide stable queuing and fairness across a wider operating range than possible with Vegas as we demonstrate in Section 3.2.4.

3.2.2 Riviera Control Policy

Riviera detects congestion when the accumulation estimate exceeds a maximum threshold, ϵ_{max}. Riviera then generates negative feedback until the accumulation estimate reduces to a minimum threshold, ϵ_{min}. The lower threshold is set low enough to achieve frequent queue drain within the desired operating range. As we discuss in Section 3.2.4, frequent queue drain aids Round-Trip Propagation Delay (RTPD) estimation.

Riviera increases additively and decreases multiplicatively from the egress rate. We call this control policy "AIMD2ER," the two appearing to denote that both increases and decreases occur with respect to the egress rate,

$$r_i = \begin{cases} \nu_i + \frac{m}{T} & \text{if no congestion} \\ \beta\nu_i & \text{if congestion ,} \end{cases}$$

$$\begin{array}{ll} \multicolumn{2}{l}{\text{where } T \text{ is RTT, } m \text{ is MTU, and}} \\ \text{if } q_i > \epsilon_{max} & \text{congestion begins,} \\ \text{if } q_i \leq \epsilon_{min} & \text{congestion ends .} \end{array}$$

AIMD2ER is illustrated in Figures 3.3(b) and 3.4. AIMD2ER continues sending at a steady-rate in the absence of feedback. Riviera eventually backs off when a timeout occurs.

In the next section we motivate this choice of increase/decrease policies by showing that AIMD2ER has similar behavior to window-based AIMD in terms of queue growth and response to changes in capacity. Though they are not similar in all regards; during congestion AIMD2ER holds the send rate steady with drain capacity $(1-\beta)\mu$ whereas AIMD backs off from the current send rate in each subsequent RTT so long as congestion is detected.

3.2.3 Rate-Clocking versus Ack-Clocking

With TCP, ack clocked (see Sections 2.5) additive increase ensures that the number of packets entering the network equals the number of packets leaving the network plus

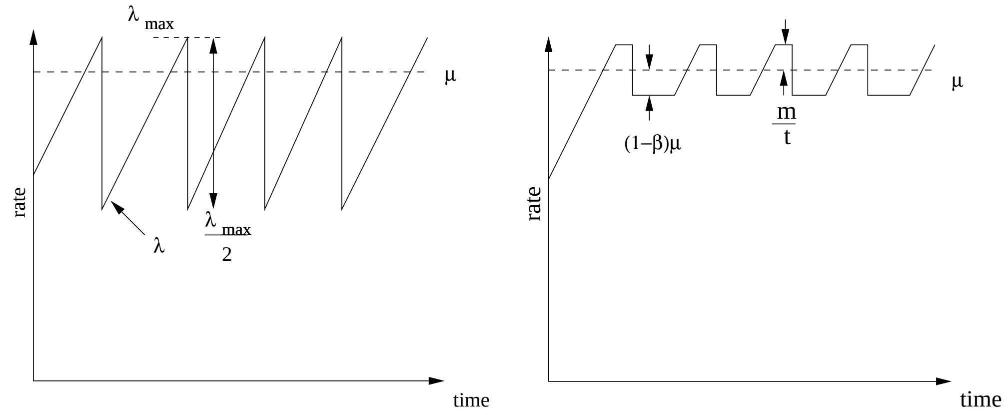


Figure 3.3: Steady-State Dynamics

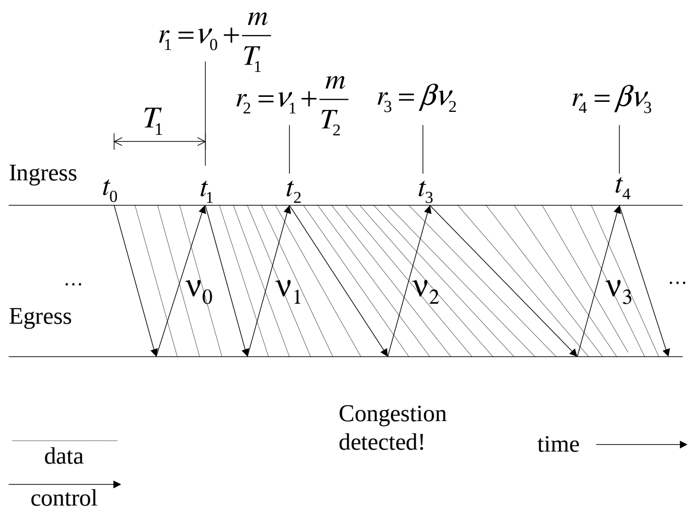
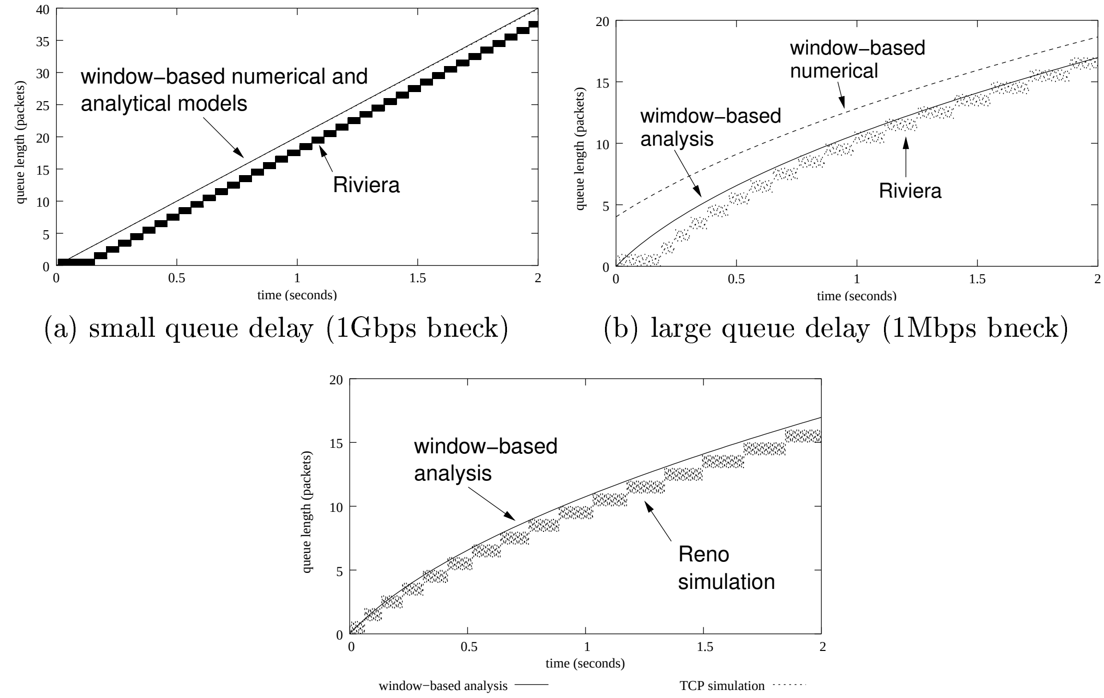


Figure 3.4: AIMD2ER Timeline

roughly one MSS per RTT. By always adjusting the rate limit with respect to the egress rate, Riviera ensures that the *rate* of packets entering the network approximately equals the *rate* of packets leaving the network plus one MTU per RTT. Due to the similarity, we use the term *rate clocking* to describe the behavior of adjusting rate limit relative to egress rate. As with TCP's ack clocked additive increase, when the bottleneck becomes saturated, Riviera's rate clocked additive increase stops increasing the input rate and RTT begins increasing by no more than one MTU transmission time per RTT. When simplified to a zero-burstiness fluid model, Riviera behaves identically to window-based control as we now show.

Approximating window-based control with a zero-burstiness fluid-flow model with



(a) small queue delay (1Gbps bneck) (b) large queue delay (1Mbps bneck)

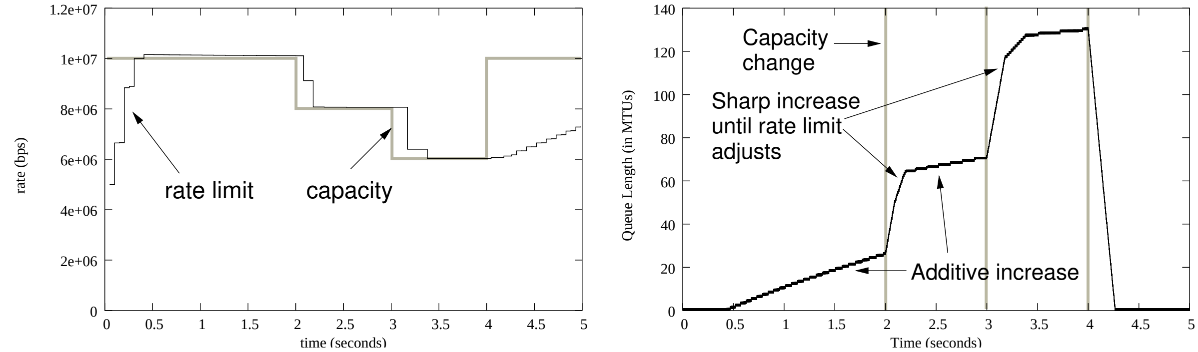
(c) Reno small queue delay (1Mbps)

Figure 3.5: Queue Growth Prior to Negative Feedback

persistent demand and where fluid is continuously acknowledged, the input rate equals the output rate plus one MSS per RTT. For convenience, we overload m to now denote MSS, and assume that the propagation delay exceeds the queuing delay sufficiently such that we can treat T as a constant. Then,

$$\begin{aligned} \dot{q} &= \lambda - \nu \ , \quad q(0) = 0, \\ \lambda &= \nu + \frac{nm}{t} \ , \\ q &= \frac{nm}{T} t \ . \end{aligned} \tag{3.7}$$

With window-based control, when the queue delay is a significant fraction of the end-to-end delay, the queue tends toward sublinear queue growth with the amount of sublinearity depending on the size of the queue delay. To see this, we use the same model except we allow propagation delay, d, to dominate round-trip time. Under such conditions, the system is formulated as



(a) Rate Limit (b) Bottleneck Queue Length

Figure 3.6: Riviera with Disabled Decreases versus Changing Capacity

$$
\begin{aligned}
\dot{q} &= \lambda - \nu \ , \quad q(0) = 0, \\
\lambda &= \nu + \frac{nm}{T} \ , \\
\left(\frac{q}{\mu} + d\right) dq &= nm \ dt \ , \\
q &= -\mu d + \sqrt{\mu^2 d^2 + 2nm\mu t} \ , & (3.8) \\
q &\approx \sqrt{2nm\mu t} \ . & (3.9)
\end{aligned}
$$

In Section 2.5, we also demonstrate that window-based schemes adjust within one RTT to changes in capacity even prior to the detection of loss.

Figure 3.5 shows window-based increase and Riviera increase in the absence of negative feedback (i.e., very large ϵ_{max}) for a single control loop for cases with (a) small and (b) large queuing delay as compared to RTPD. The analytical models in 3.5(a) and (b) are equations (3.7) and (3.8) respectively. Thus, when queue delay is a small component of end-to-end delay, both window-based and Riviera control tends toward linear queue growth; when queue delay is large, window-based and Riviera control tend toward queue length that grows with the square root of time.

As with ack-clocking, when capacity reduces rate-clocking reduces the bottleneck's arrival rate even prior to detecting congestion. In Figure 3.6, we show a single traffic trunk spanning a 10Mbps bottleneck subjected to changes in capacity at 2, 3, and 4 seconds into the simulation. `max_thresh` has been set sufficiently high to avoid congestion detection. The queue grows sharply following a change in capacity as knowledge of the capacity changes propagates through the control loop. Once the rate limit settles at the new capacity, the queue resumes queue growth according to additive increase.

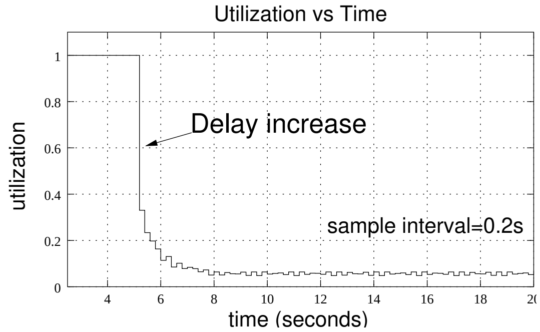
3.2.4 Round-trip Propagation Delay Estimation

Riviera's queue oscillation is intended to frequently drain the queues in the path. By draining the queues, control packets are allowed to transit the network experiencing little queuing delay. Draining the queues frequently is critical to obtaining a good estimate of the Round-Trip Propagation Delay (RTPD). As discussed in Section 2.5.4, obtaining a good estimate of RTPD is a necessary condition to avoiding gross unfairness and drastic underutilization (goals 2 and 4). Furthermore, accurate RTPD estimation is more important for traffic trunks than end-to-end flows given the potential number of connections affected by a single poor estimate. We now demonstrate the unfairness through simulation and the fixes enabled by the AIMD2ER control policy.

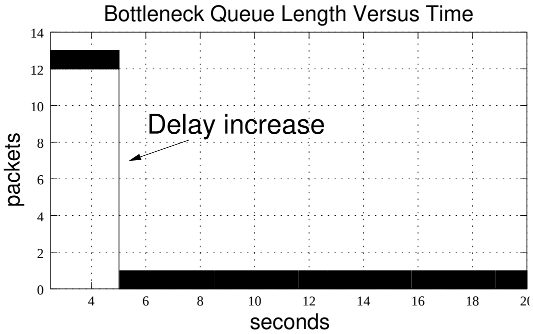
For example, consider ten long TCP Vegas file downloads competing over a 100Mbps bottleneck. At five seconds into the simulation, the bottleneck delay suddenly increases causing the RTPD to increase from 16ms to 54ms, such an increase in RTPD could occur due to a route change. Since all connections are affected by the increase in this example, they all see the increase in propagation delay as congestion and backoff until $\nu_i(d_i - b_i)$ drops to β (as derived in Section 2.5.4 and demonstrated in Figure 3.7(a) and (b)). If Vegas measures the RTPD over the last k samples then the queue grows unbounded as demonstrated in Figure 3.7(d). We shall refer to Vegas that sets BaseRTT to the minimum of the last k samples as Vegas-k.

Riviera as with Vegas experiences a decrease in utilization[6] when the propagation delay suddenly increases, but unlike Vegas it recovers. Vegas-k does not drop to zero, but it exhibits an unstable queue. Vegas-k recovers from the drop in queue more rapidly than Riviera because Vegas performs RTT measurements on every segment rather than once per RTT or once per window. We implemented min-of-the-last k literally as implied by [89]. Thus more than k RTT samples may occur within a single window. Because it is impossible for a control loop to react to queue faster than within one RTT, Vegas-k should set BaseRTT to the minimum over the last k *windows* or something else that operates on the correct timescale. Min over the last k windows can still be implemented with $O(k)$ state. For example, with a circular buffer, the current entry simply refers to the minimum seen so far in the current window. The circular buffer then advances at the end

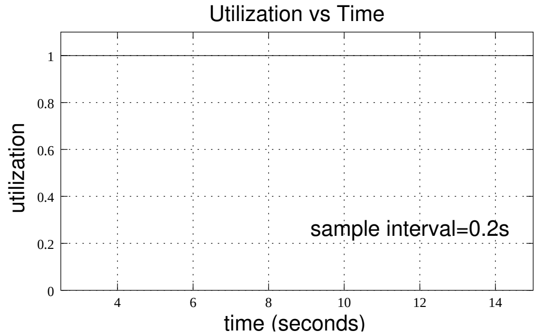
[6]We are currently working on mechanisms that diminish this temporary drop in utilization.



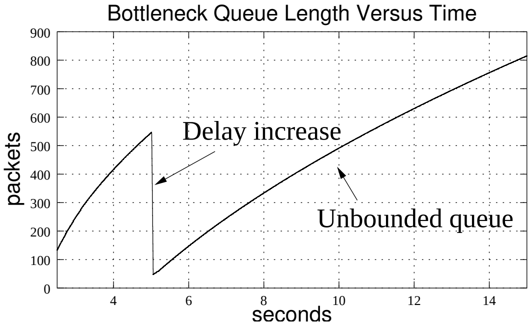
(a) Vegas Utilization



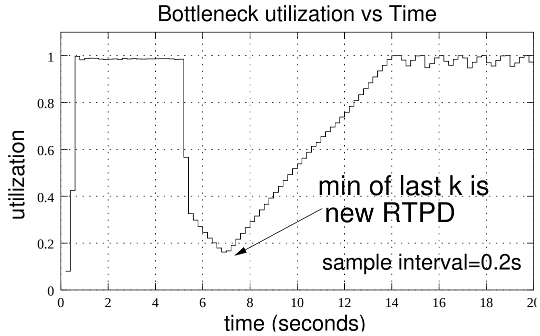
(b) Vegas Queue Length



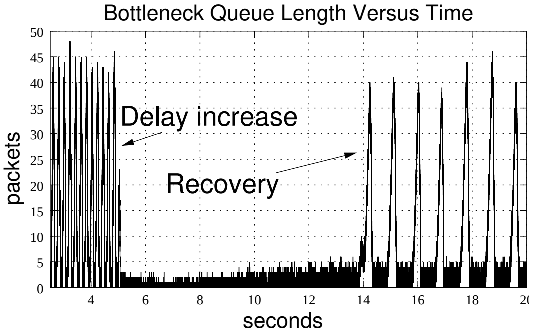
(c) Utilization for Vegas-k, k=30



(d) Queue Length for Vegas-k, k=30



(e) Utilization for Riviera k=30



(f) Queue Length for Riviera k=30

Figure 3.7: Vegas and Riviera with Sudden Increase in Propagation Delay

of each window. Riviera is unaffected by this pitfall because it only measures the RTT once per round-trip of a control packet. Even if min-of-the-last-k were implemented over k windows, Vegas-k would still grow unstable under persistent demand.

We present expanded simulation results for this example in Appendix A.10.

Riviera suffers from the problem that it cannot guarantee that all queues drain in the path at any given time, because each control loop detects congestion independently and those control loops with smaller shares tend to detect congestion later or not at all. As a consequence the effectiveness of our accumulation estimator in conjunction with control policy is more easily studied through experimentation and simulation. We show that for large numbers of traffic trunks or small bandwidths, the queue will still fail to drain leading to queue growth. Although, such queue growth sacrifices goal 3, such queue growth does not necessarily lead to heavy loss (see goal 1) or gross unfairness (see goal 2), because Riviera degrades similar to TCP Reno. When loss occurs, Riviera backs off by one-half from the egress rate (ν_i) if feedback is available or one-half from the input rate (λ_i) otherwise.

3.2.5 Riviera Control Packets

Control packets headed from the ingress to the egress are called *Forward Edge Traffic Control (FETC)* packets. Control packets heading from the egress to the ingress are called *Reverse Edge Traffic Control (RETC)* packets. All control loops in Riviera are currently unidirectional. Bidirectional control can of course be employed using two control loops. All control packets bear the following information:

```
<tripcount, action=loss/increase/decrease, nsent, egressrate>
```

The `tripcount` field contains the number of round-trips made by the control packet. It is similar to the `probeseqno` field in [7]. The ingress alone increments the `tripcount` counter. The ingress measures round-trip time as the time between the arrivals of a particular sequence number. The egress measures round-trip times as the time between the arrivals of consecutive trip counts. If a control packet is lost or sufficiently delayed, the ingress times out, increments the current trip count by two and generates a new control packet (see Figure 3.8). Increasing `tripcount` by two tells the egress that this is a new control packet and thus the control packet should not be used in the round-trip estimate.

The `tripcount` field is similar to the `probeseqnum` field used in the control packets in Balakrishnan and Seshan's congestion manager protocol [7]. The only difference is the increase by two when a new control packet is generated. Balakrishnan's congestion manager does not need to increase the `probeseqnum` by two because the receiver does not compute round-trip times baesd on a shared counter and thus the receiver does not need to distinguish retransmitted control packets. A Riviera egress must compute round-trip times to update the RTPD estimate.

The `nsent` field denotes the number of bytes sent on this EC trunk since the last control packet and serves the same purpose as the field by the same name in Balakrishnan's congestion manager protocol. It is separate from the trip count because the `nsent` field

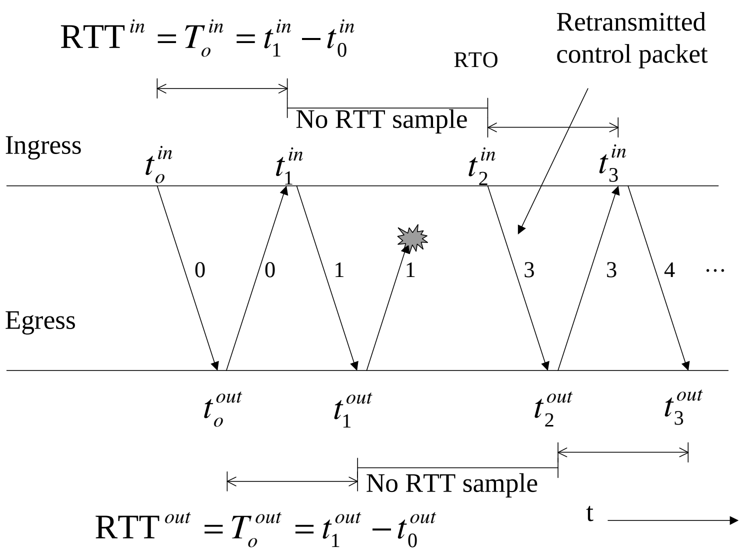


Figure 3.8: Timeline Showing Control Packet Loss

increases by an unpredictable amount on each round-trip. When a control packet arrives at the egress, if the `nsent` does not match the number of packets received then the egress signals loss to the ingress. The `nsent` field serves the same purpose as the `nsent` field in the congestion manager's control packets. The number of bytes sent is used rather than the number of packets sent to account for possible fragmentation. Unlike Balakrishnan's congestion manager, Riviera does not correct for false positive detections of loss due to reordering of a control packet with a data packet. We feel that packet reordering occurs too infrequently to justify the complexity required to correct the resulting transient error.

The `egressrate` field provides the ingress with an accurate estimate of the egress arrival rate. The ingress could measure the egress rate as the number of packets sent over the previous RTT, but this is ill-defined when the previous RTT is ambiguous, e.g., following a control packet loss. The rate measured at the egress also is not affected by congestion in the reverse path, whereas the RTT measured at the ingress increases with reverse path congestion resulting in a low egress rate measurement.

3.2.6 Additional Riviera Details

In addition to AIMD-ER, the ingress edge in Riviera implements the following:

- Riviera uses a rate-based slow start analogous to TCP slow start in order to quickly utilize capacity.
- Each ingress shapes traffic entering the network using a leaky bucket (σ, ρ) shaper,

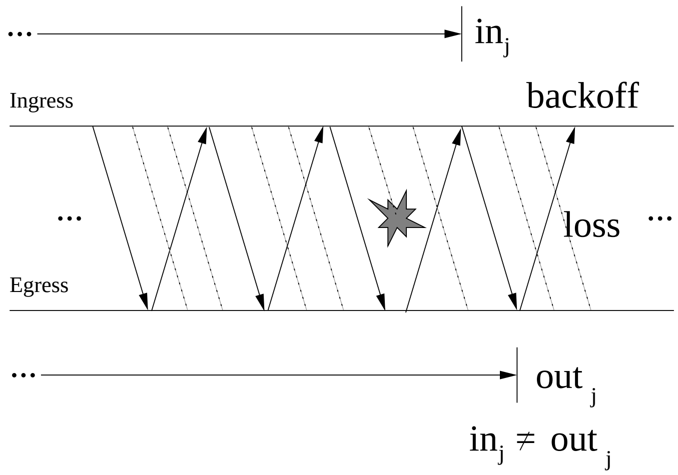


Figure 3.9: Timeline Showing Data Packet Loss

to conform to a Linearly Bounded Arrival Process (LBAP) [29], i.e.:

$$\int_{t}^{t+\tau} \lambda_i(T)dT \leq r_i\tau + \sigma \tag{3.10}$$

where r_i is the ith EC trunk's rate limit and σ is the maximum burst size.

Note that shaping traffic reduces queue length variability and smooths bursts so that packets interleave more evenly at the bottleneck. This reduces the bias against bursty flows seen with TCP Reno [43].

- During additive increase, each trunk i's rate increments by $\frac{m_i}{T_i}$ from the received (egress) rate ν_i at the beginning of each RTT. Here m_i denotes the maximum transmission unit along i's path. Currently the maximum burst size σ is set equal to m_i.

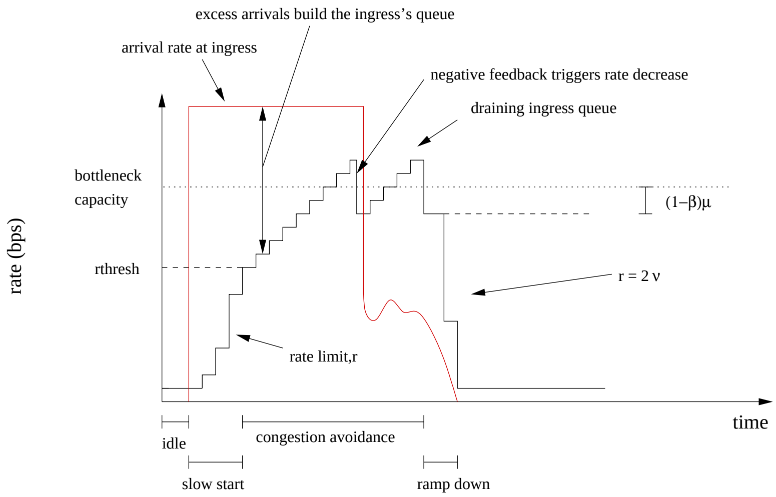


Figure 3.10: Increase/Decrease Dynamics in Detail

3.3 Properties of Riviera

We first study the stability and fairness of Riviera and then consider single bottleneck analyses of steady-state and slow-start queueing behavior.

3.3.1 Terminology

Let L denote the set of links in an arbitrary network where each link $l \in L$ has capacity C_l. Let S denote the set of control loops. Let r_i denote the set of links in the route taken by control loop $i \in S$. In other words, $r_i = \{ l \in L \mid \text{ loop } i \text{ passes through } l \}$. And let $S(l) = \{i \in S | l \in r(i)\}$, in other words $S(l)$ is the set of loops traversing link l. In addition we use the terms found in Table 3.1.

Table 3.1: Constants and Variables Used in Analyses

$\lambda_i(t)$	send rate of ith control loop
$\nu_i(t)$	egress rate of ith control loop
$\rho(t)$	aggregate rate limit
$q(t)$	queue length
$q_i(t)$	ith loop's accumulation[2]
$I[t, t+\tau]$	traffic in $[t, t+\tau]$
T	round-trip time
d	round-trip propagation delay
n	number of trunks
m	MTU
σ	max burst size, typically $= m$
a	acceleration $= \frac{m}{T^2}$
t_μ	time when $r(t) = \mu$
s	slow-start increase
ϵ	max threshold

Throughout the single bottleneck analyses of Riviera, we use the model shown in Figure 3.12 with the variables and constants given in Figure **??**. In each case we compare the models with simulation. Only illustrative simulation results appear in this chapter. The simulations are described in detail in Appendix A.

3.3.2 Stability

In this section we prove[7] global stability of Riviera using Lyapunov Theory. Let k be discrete time step. Let $p_i(k)$ denote the probability of detecting congestion. Therefore,

[7]This proof was slightly modified from a proof provided by Yong Xia.

$$p_i(k) = Prob[q_i(k) \geq \epsilon] \ . \tag{3.11}$$

then we have from the definition in Section 3.2.2,

$$\lambda_i(k+1) = (1 - p_i(k)) \cdot (\lambda_i(k) + \alpha_i) + p_i(k) \cdot \beta_i \nu_i(k). \tag{3.12}$$

Dividing by the length of the control period which is, under ideal feedback condition, the round-trip propagation delay d_i, we get

$$\begin{aligned}
\dot{\lambda}_i(k) &\doteq \frac{\lambda_i(k+1) - \lambda_i(k)}{d_i} \\
&= \frac{\alpha_i(1 - p_i(k))}{d_i} - \frac{p_i(k)}{d_i} \cdot [\lambda_i(k) - \beta_i \nu_i(k)]
\end{aligned}$$

or simply

$$\dot{\lambda}_i = m_i \cdot (w_i - s_i(\,\lambda_i - \beta_i \nu_i\,)) \tag{3.13}$$

where

$$\begin{aligned}
m_i &= 1 - p_i(k) \\
w_i &= \alpha_i \,/\, d_i \\
s_i &= p_i(k) \,/\, \big(d_i(1 - p_i(k))\big)
\end{aligned}$$

are all non-negative. This differential equation shows the dynamics of the AIMD2ER algorithm. We prove its stability by constructing a Lyaponov function of system states.

Define a function of all flows' ingress rates [69, 77]

$$U(\overrightarrow{\lambda}) \triangleq \sum_{i \in S} [w_i \log \lambda_i - s_i(1 - \beta_i)\lambda_i] - s_i \beta_i \sum_{l \in L} \int_0^{\lambda^l} p_l(c_l, x)\, dx \tag{3.14}$$

where the penalty $p_l(c_l, x)$ at link l is a non-negative, continuous, increasing function of x :

$$\begin{cases} p_l(c_l, x) = 0 & \text{if } x \leq c_l, \\ p_l(c_l, x) > 0 & \text{if } x > c_l. \end{cases} \tag{3.15}$$

It also satisfies $\sum_{l:l\in r_i} p_l(c_l, \lambda^l) = (\lambda_i - \nu_i)/\lambda_i$, where $\lambda^l \doteq \sum_{i:l\in r_i} \lambda_i$ is the aggregate input rate at link l. We have

Lemma 3.3.1 *$U(\overrightarrow{\lambda})$ is strictly concave, and has a unique interior maximum denoted as U_{max} occuring at $\overrightarrow{\lambda} = \overrightarrow{\lambda}_{max}$.*

Proof. Logarithmic functions are strictly concave. Every other component in $U(\overrightarrow{\lambda})$ is also concave. The definition domain $\overrightarrow{\lambda} > 0$ is a convex set. Plus $\forall i, \lim_{\lambda_i \to 0} \partial U(\overrightarrow{\lambda})/\partial \lambda_i > 0$ and $\lim_{\lambda_i \to \infty} \partial U(\overrightarrow{\lambda})/\partial \lambda_i < 0$ we get this result.

□

Theorem 3.3.1 *$V(\overrightarrow{x}) \triangleq U_{max} - U(\overrightarrow{x} - \overrightarrow{\lambda}_{max})$ is a Lyaponov function for the system (3.13) which is stable.*

Proof. By inspection $V(\overrightarrow{x}) \geq 0$. We have

$$\frac{\partial V}{\partial x_i} = \frac{\partial V}{\partial \lambda_i} = -\frac{\partial U}{\partial \lambda_i} = -\frac{\dot{\lambda}_i}{m_i \lambda_i}$$

and then

$$\frac{dV}{dt} = \sum_{i \in S} \frac{\partial V}{\partial \lambda_i} \cdot \frac{d\lambda_i}{dt} = -\sum_{i \in I} \frac{\dot{\lambda}_i^2}{m_i \lambda_i} \leq 0.$$

□

Corollary 3.3.1 *$U(\overrightarrow{\lambda})$ is maximized at the stable point.*

3.3.3 Fairness

In a multiple bottleneck scenario, each path may distribute its accumulation along several hops in the path. Let's return to the example used to describe proportional fairness in Section 2.2. Using the two-hop linear network in Figure 2.1, if each control loop achieves x packets of accumulation, and control loop 0 evenly distributes the accumulation across the two hops then control loop 0 will have on average $x/2$ packets backlogged at each bottleneck compared to x packets average backlog by loops 1 and 2 at their respective bottlenecks. Thus control loops 1 and 2 should get twice as much capacity as zero. Thus this example with accumulation converges on the same rate allocation as proportional fairness. We now prove that a fluid-flow approximation of Riviera achieves proportional fairness.

Define control loop i's net benefit

$$B_i(\lambda_i) \triangleq w_i \log \lambda_i - s_i \int_0^{\lambda_i} \frac{x - \beta_i \mu_i}{x}\, dx \tag{3.16}$$

which includes a utility and a penalty function in the first and second terms respectively. From this we theorize that

Theorem 3.3.2 *The maximization of $U(\overrightarrow{\lambda})$ is the same as a set of games maximizing $B_i(\lambda_i)$, namely*

$$\max_{\overrightarrow{\lambda}>0} U(\overrightarrow{\lambda}) \;\Leftrightarrow\; \max_{\lambda_i>0} B_i(\lambda_i), \forall i \in S.$$

Proof. Note $\forall i$, we have $\partial U(\overrightarrow{\lambda})/\partial \lambda_i = \partial B_i(\lambda_i)/\partial \lambda_i$, plus Lemma 3.3.1 and $\partial^2 B_i(\lambda_i)/\partial \lambda_i^2 \leq 0$ we obtain this.

□

This result shows all flows' independently maximizing their own net benefits effectively maximizes a global interest. Under ideal conditions, β_i could be set somewhat below 1. If all flows compete network resources so aggressively that congestion probabilities are very large, we have

Lemma 3.3.2 *If $\beta_i = 1$, $p_i(k) \to 1$, and flow i's utility function*

$$U_i(\lambda_i) \triangleq w_i \log \lambda_i, \tag{3.17}$$

then we obtain a constrained convex optimization,

$$\max_{\overrightarrow{\lambda}>0} U(\overrightarrow{\lambda}) \;\Rightarrow\; \max_{\lambda_i>0} \sum_{i\in S} U_i(\lambda_i) \quad \text{suchthat.} \sum_{i:l\in r_i} \lambda_i \leq c_l,\ \forall l \in L.$$

Proof. We get $s_i \to \infty$ thus $p_l(c_l, \lambda^l)$ has to be 0. According to (3.15) we have $\lambda^l \leq c_l$, $\forall l$.

□

Theorem 3.3.3 *Rate allocation at the stable point of the system (3.13) is weighted proportionally fair [69].*

Proof. Due to (3.3.1) and Lemma 3.3.2, since flow i's utility function $U_i(\lambda_i)$ is strictly concave, according to nonlinear programming optimality condition [11], if $\hat{\lambda}_i$ is the allocation to flow i associated with the system stable point, then for any other allocation λ_i,

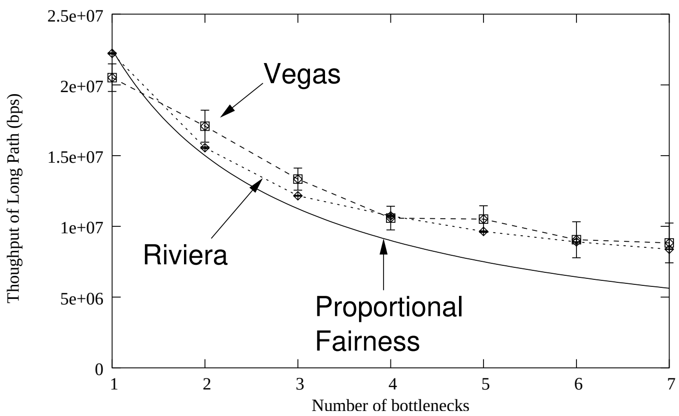


Figure 3.11: Multi-hop Loop's Throughput Versus Number of Hops

we have

$$\sum_{i \in S} U_i'(\hat{\lambda}_i) \cdot (\lambda_i - \hat{\lambda}_i) = \sum_{i \in S} w_i \cdot \frac{\lambda_i - \hat{\lambda}_i}{\hat{\lambda}_i} < 0.$$

This is the definition of weighted proportional fairness.

□

Figure 3.11 demonstrates Riviera's fairness using a linear network, discussed in detail in Appendix A.4. Riviera remains reasonably close to proportional fairness.

3.3.4 Steady-State Approximate Queue Behavior

We perform a simple fluid model analysis of the queueing behavior of Riviera for a single bottleneck using the topology in 3.12. All control loops have equal RTPD. Riviera has certain features that complicate analysis:

1. a control packet transits the bottleneck only once per round-trip time,
2. each control packet must pass through the queue thus experiencing the same queuing delay as other traffic,
3. and each control loop estimates its own accumulation and reacts independently when that estimate exceeds a threshold.

We start by ignoring these complexities. In other words, we assume that queuing delay is negligible compared to the RTT, that the only delay in detecting congestion once the per-flow backlog exceeds M is propagation delay, and assume that steady-state implies that all control loops have already converged on fairness and thus have equal rates. Equal rate control loops have equal accumulation, and thus detect congestion simultaneously.

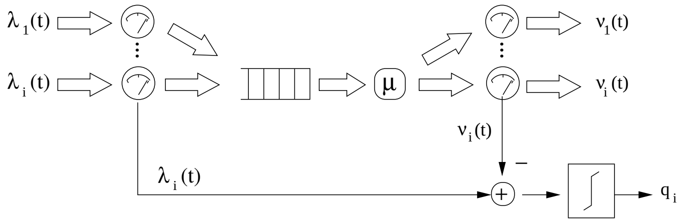


Figure 3.12: Riviera Single Bottleneck Topology

Each control loop has a leaky bucket obeying inequality (3.18):

$$I_i[t, t+\tau] \leq \int_t^{t+\tau} \rho_i(x)dx + \sigma \ . \tag{3.18}$$

With n control loops this inequality becomes:

$$I[t, t+\tau] \leq \int_t^{t+\tau} \rho(x)dx + n\sigma \ . \tag{3.19}$$

The bottleneck queue after an interval $[t, t+\tau]$ in which the queue does not reach zero is:

$$q(t+\tau) = q(t) + I[t, t+\tau] - \mu\tau \ . \tag{3.20}$$

Because the rate limit increases relative to the egress arrival rate, the aggregate rate limit's is upper bounded by

$$\rho(t) \leq \mu + \frac{nm}{T} \tag{3.21}$$

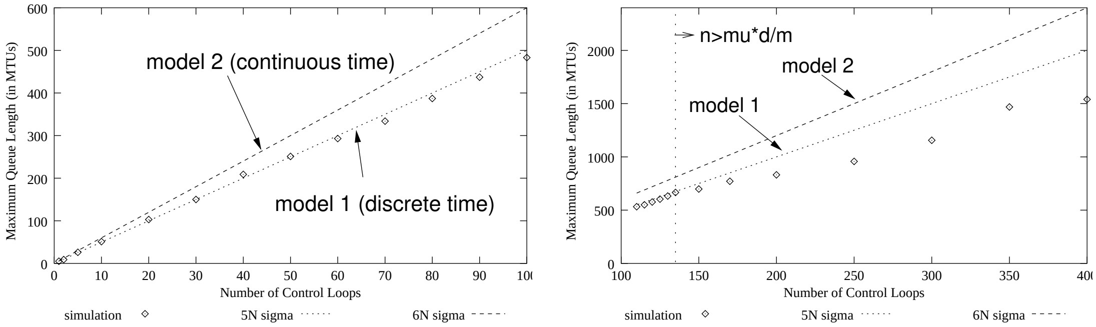
With this drastically simplified model, substituting (3.21) into (3.19) and (3.20) yields

$$q(t+\tau) \leq q(t) + \frac{nm}{T}\tau \ . \tag{3.22}$$

Again consider time in units of $T = 1$, define $t = 0$ as being the time when the contributions of all of the control loops reach ϵ. Then 3.22 becomes

$$q_{max} \leq q(1) \leq n\epsilon + nm + n\sigma \ . \tag{3.23}$$

In our simulations we set $\epsilon_{max} = \epsilon = 3m$ and $\sigma = m$, thus (3.23) becomes



(a) small n (b) big n

Figure 3.13: Riviera Steady-State Queue Length Maxima versus Number of Control Loops

$$q_{max} \leq 5n\sigma \ . \tag{3.24}$$

Figure 3.13(a) shows that the $5n\sigma$ model (labelled model 1) remains close to the simulation maxima across the range of 1 to 100 control loops for a T3 bottleneck (see Appendix A.3) though the simulation maxima occasionally violate the approximate bound by a few packets.

To eliminate the violations, we consider a continuous-time model that considers the first two enumerated complications: 1) one control packet per RTT per control loop, and 2) queuing delay affects control packets. Our continuous-time model still ignores the third complication: control loops measuring their own backlog in the path. Instead we assume all control loops have equal rate and thus detect congestion in the same RTT.

When queue delay is significant, $T = RTT$ increases with time and $\frac{m}{T}$ decreases. Following the derivation that led to equation 3.8 but taking into account the definition of the leaky bucket in (3.18), and defining $t = 0$ to be the time when all control loops experience $q_i > \epsilon$, the queue length is given by

$$\begin{aligned} \left(\frac{q}{\mu} + d\right) dq &= nm \ dt \ , \ q(0) = n\epsilon \ , \\ q(\tau) &\leq -\mu d + n\,\sigma + \sqrt{C + 2\,m\,\mu\,n\,\tau} \ , \end{aligned} \tag{3.25}$$

where $C = \mu^2\, d^2 + 2\, d\, \epsilon\, \mu\, n + \epsilon^2\, n^2$.

At time 0, $q = n\epsilon$ and in the worst case all control packets enter the queue the instant before $q = n\epsilon$ and thus congestion is not detected for 1 RTT, T_0, and the congestion response does not arrive for 2 RTTs, $T_0 + T_1$. Finding max q requires bounding the time before a feedback response arrives at the bottleneck, τ_{max}.

The duration of each RTT is given by

$$T_0 = \text{q delay} + \text{RTPD} = \frac{q(0)}{\mu} + d = \frac{n\epsilon}{\mu} + d \ , \tag{3.26}$$

$$T_1 = \frac{q(T_0)}{\mu} + d = \frac{1}{\mu}\, q\left(\frac{n\epsilon}{\mu} + d\right) + d \ . \tag{3.27}$$

τ_{max} is thus given by

$$\tau_{max} \leq T_0 + T_1 = \frac{n\epsilon}{\mu} + 2d + \frac{1}{\mu}\, q\left(\frac{n\epsilon}{\mu} + d\right) \ , \tag{3.28}$$

and substituting τ_{max} into (3.25) yields

$$q_{max} \leq -\mu d + n\sigma + \sqrt{C + 2\, m\, \mu\, n\left(\frac{n\epsilon}{\mu} + 2d + \frac{1}{\mu}\, q\left(\frac{n\epsilon}{\mu} + d\right)\right)} \tag{3.29}$$

Further expanding (3.29) results in an unwieldy equation that provides little insight. However, in the special case where $m = \sigma$ and $\epsilon = 3\sigma$, (3.29) simplifies to

$$q_{max} \leq -\mu d + n\sigma + \sqrt{d^2\, \mu^2 + 8\, d\, \mu\, n\, \sigma + n\, \sigma\, \left(17n\sigma + 2\sqrt{d^2\, \mu^2 + 8\, d\, \mu n\sigma + 15\, n^2\, \sigma^2}\right)} \ . \tag{3.30}$$

Let P denote the polynomial $d^2\mu^2 + 8d\mu n\sigma + 15n^2\sigma^2$, let $x = d\mu$, and let $y = n\sigma$. Observe that $(x^2 + 8x + 15y^2) \approx (x^2 + 8x + 16y^2)$ except around the negative roots $(-5, -3)$, which is not a problem since neither x nor y can be negative. Thus P can be closely approximated with $(d\mu + 4n\sigma)^2$. Further note that $(d\mu + 4n\sigma)^2 > P$ thus q_{max} obeys the inequality given by

$$q_{max} \leq -\mu d + n\sigma + \sqrt{d^2\mu^2 + 10d\mu n\sigma + 25n^2\sigma^2} \ ,$$

$$q_{max} \leq 6n\sigma \ . \tag{3.31}$$

Figure 3.13(a) shows that our second model (labelled model 2) less tightly bounds the maxima of the steady state queue lengths incurred in the simulations, but the simulation maxima do not violate the second model's approximate queue bound. Figure 3.13(b) demonstrates that Riviera operates across ranges where each control loop is forced to operate well below 1 MTU per RTPD.

3.3.5 Riviera Slow Start Queue Length Bound

As was briefly mentioned in Section 3.2.6, Riviera implements a rate-based slow-start that is analogous to TCP's slow start. In Appendix B we describe the entire algorithm in detail, here we present a snippet taken directly from the source code that is specific to slow start:

```
min_rate = min( departure_rate, egress_rate );        \\ 1
if ( min_rate >= rthresh_ - r_inc )                   \\ 2
  rthresh_ = min_rate + r_inc;                        \\ 3
r_ = ss_increase_ * min_rate;                         \\ 4
r_ = min( r_, rthresh_ );                             \\ 5
r_ = max( r_, r_inc );                                \\ 6
```

`rthresh` is analogous to TCP's `ss_thresh`. Line 1 handles the condition when demand (departure rate) reduces thereby making departure rate a better estimate of the load on the bottleneck in the next RTT. Lines 2 and 3 allow `rthresh` to increase linearly when the departure rate (i.e., λ) is near to `rthresh`. Line 4 and line 5 allow a multiplicative increase by `ss_increase` from the departure rate but upper bound the increase to `rthresh`. Due to the upper bound, we expect slow-start behavior to have little if any impact on steady-state behavior. When the load nears capacity, lines 4 and 5 also allows the load to vary by a factor of as much as `ss_increase` in an RTT without reducing the rate limit, and thereby reduces bias against particularly bursty sources.

We use the same topology and assumptions appearing in the Riviera continuous-time model (i.e., model 2). Riviera's slow start multiplicatively increases from the egress rate, thus for a leaky-bucket fluid model given constant capacity μ, the aggregate rate of the control loops $\rho(t)$ never exceeds $s\mu$ except to admit up to σ burstiness per control loop. Again starting from equation (3.20),

$$
\begin{aligned}
q(t+\tau) &\leq q(t) + \int_t^{t+\tau} \rho(x)dx + n\sigma - \mu\tau \ , \\
q(t+\tau) &\leq q(t) + (s-1)\mu\tau + n\sigma \ ,
\end{aligned}
$$

In the worst case, a packet from each flow arrives at the bottleneck queue the instant before the queue is sufficient to trigger congestion detection. Let $t = 0$ denote this time, thus $q(0) = n\epsilon$. The queue is then allowed to build for one full RTT, T_0, before the next round of control packets transit the bottleneck arrive. Assuming all control loops are sending at equal rate, at most one burst from each control loop transits the bottleneck before all control loops detect congestion thus adding $\frac{n\sigma}{\mu}$ to the next RTT, T_1. The feedback response then arrives no later than two RTTs from time zero.

T_0 and T_1 are given by

$$
\begin{aligned}
T_0 &= \frac{q(0)}{\mu} + d = \frac{n\epsilon}{\mu} + d \ , && (3.32) \\
T_1 &= \frac{q(T_0)}{\mu} + \frac{n\sigma}{\mu} + d \ . && (3.33)
\end{aligned}
$$

The maximum queue occurs at time $\tau_{max} = T_0 + T_1$ and thus τ_{max} is given by

$$
\tau_{max} \leq T_0 + T_1 \leq \frac{1}{\mu}\big((n\epsilon + \mu\, d)\,(s+1) + 2\,n\,\sigma\big) \ , \tag{3.34}
$$

and q_{max} is given by

$$
q_{max} \leq \mu\, d\, \left(s^2 - 1\right) + n\ \epsilon\, s^2 + (2\,s - 1)\,n\,\sigma \ . \tag{3.35}
$$

Figure 3.14 demonstrates that our analysis (labelled "model") applies across a range well into the range where there are more control loops than MTUs in the bandwidth-RTPD product (i.e., $n > \frac{\mu d}{m}$).

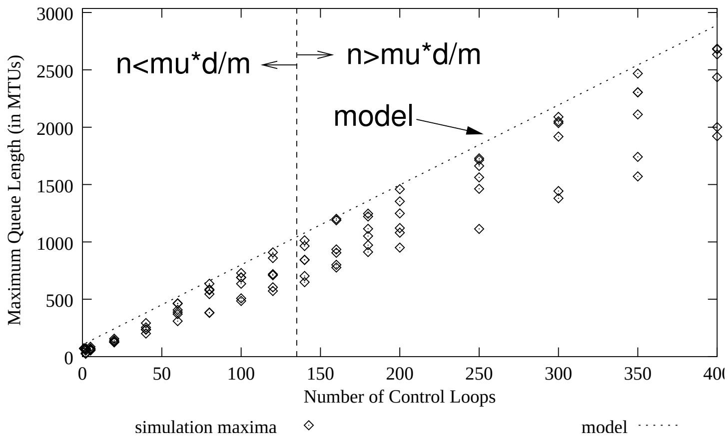


Figure 3.14: Slow Start Queue Length Maxima versus Number of Control Loops

4. End-system Control Building Blocks

In the EC architecture, *end-system control building blocks* sit within the data plane allocating resources between flows. These building blocks may be placed at the bottleneck or at the entrance to an EC trunk. Examples include TCP rate control (see Section 2.8.1 and FRED (see Section 2.6.2). In the next chapter we discuss *per-trunk service building blocks*, which allocate resources between EC trunks. Because all of the end-system control building blocks in this chapter adapt to the trunk's available capacity, any of the end-system control building blocks can be combined with any of the per-trunk building blocks described in the next chapter. However, for this chapter we assume that the EC traffic trunks implement Riviera as described in Chapter 3 without per-trunk service building block extensions. Thus, each EC trunk competes for a proportionally fair share of the capacity in the path.

In comparison to DiffServ (see Sections 1.1.4 and 2.7), building blocks that allocate AF "in" marks between end-to-end flows fit our definition of end-system control building blocks. DiffServ lacks a direct analog to EC traffic trunks or per-trunk service building blocks. However, when AF or EF are used in the construction of traffic trunks, both the AF marker that sets marks to achieve per-trunk target resource allocations and the EF rate limiter that admits packets into a traffic trunk fit our definition of per-trunk service building blocks. As discussed in Chapter 1, the key difference between DiffServ and EC is EC's use of closed-loop feedback allowing the edges to infer the available capacity in the path. Providing knowledge of capacity to the ingress edge enables a wide range of building blocks implementing improved performance and policy-based per-flow QoS.

All simulations presented in this chapter are provided in greater detail in Appendix A.

4.1 Weighted TCP Service Building Block

TCP rate control (see Section 2.8.1) enables the administrator to adjust each TCP connection's receiver advertised window according to policy. Our implementation of TCP rate control achieves weighted max-min fairness (see Appendix D). In this section, the TCP rate controller sits at the head of an EC trunk and adjusts window allocations to utilize the EC trunk's rate limit (see Figure 4.1). As stated in the introduction to this chapter, each EC trunk competes for a proportionally fair share of the capacity in the path.

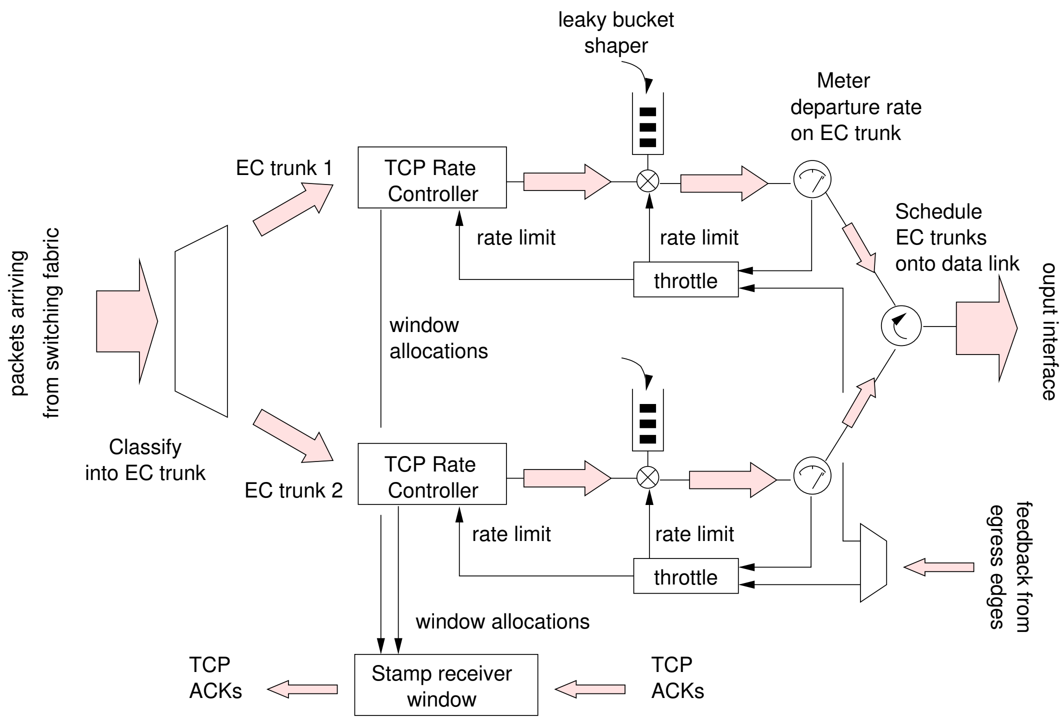


Figure 4.1: Example TCP Rate Control Ingress

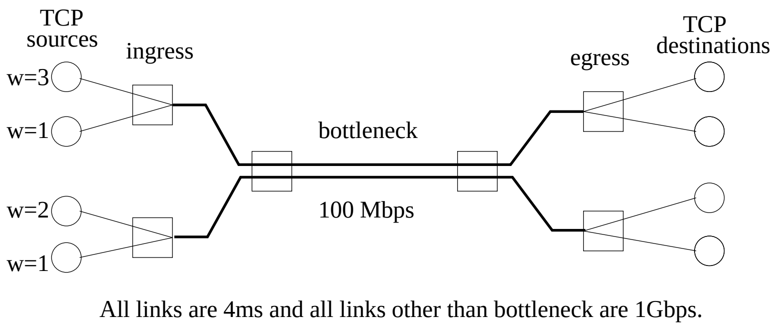
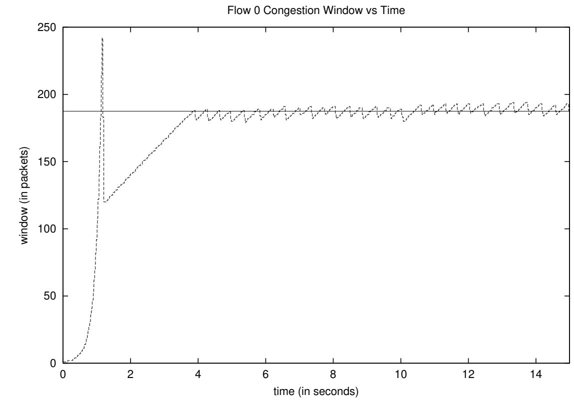


Figure 4.2: Weighted TCP Connections

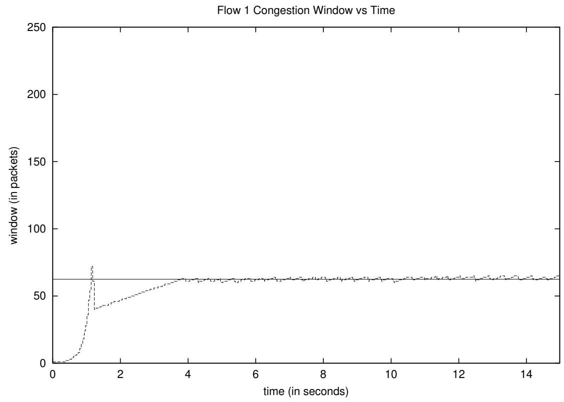
In the case of a single bottleneck, each of n EC trunks fights for $1/n$ of the bottleneck's capacity.

Consider the topology in Figure 4.2. This topology is simple for descriptive purposes. More realistic scenarios appear in Appendices A.2 and A.6.

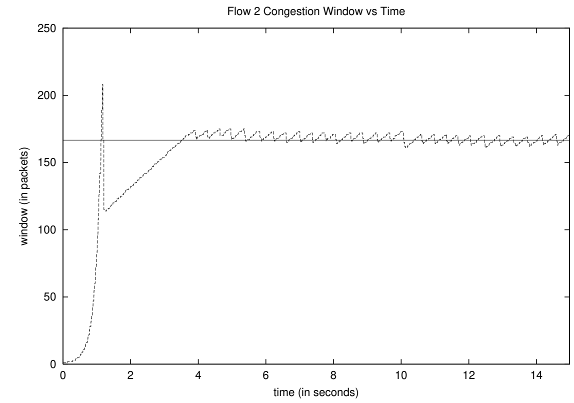
Four TCP bulk transfers enter a network. Two connections enter each ingress edge. Each ingress edge maintains a single Riviera EC trunk, and the two EC trunks share a common bottleneck. To EC traffic, the bottleneck appears as a FIFO with constant capacity of 100Mbps. This topology has a bandwidth-RTPD product of 500 packets of



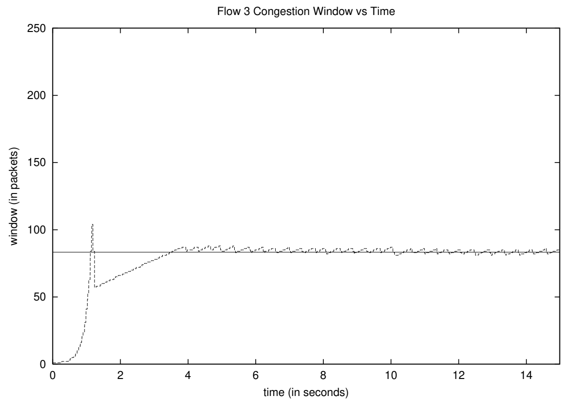
(a) Flow 0, weight=3



(b) Flow 1, weight=1



(c) Flow 2, weight=2



(d) Flow 3, weight=1

Figure 4.3: Congestion Window versus Time for Weighted TCP Connections

size 1000 bytes. Because all paths have homogeneous delay, we expect that each EC trunk should obtain 250 packets of the bandwidth-RTPD product. Each edge then uses a weighted version of the ERICA [66] algorithm to compute weighed fair shares of their respective EC trunk capacities. The edges' TCP rate controllers then convert the rate allocations into the appropriate advertised windows. The advertised windows are then stamped in passing acknowledgements constraining each flow to its appropriate window size. In this case we expect flow 0 with weight 3 to oscillate around $\frac{3}{4}$ of 250 packets, i.e., 187.5 packets. Flow 1 should oscillate around $\frac{1}{4}$ of 250 packets, i.e., 62.5 packets. Similarly flow 2 should oscillate around 166.7 packets and flow 3 around 83.3 packets. Figure 4.3 shows that our edges correctly constrained the TCP connections to their weighted fair shares.

Additionally the weighted TCP service does not adversely affect the aggregate performance of the EC trunks. For example, in the last five seconds of the simulation, the bottleneck utilization is 98.5%, maximum queue length is 8 packets, maximum *ingress* queue length is 7 packets, and the mean end-to-end delay is a mere 20.87ms. This is only slightly above the 20ms propagation delay.

The full results for this simulation appear in Appendix A.11.

4.2 Distributed Buffer Management

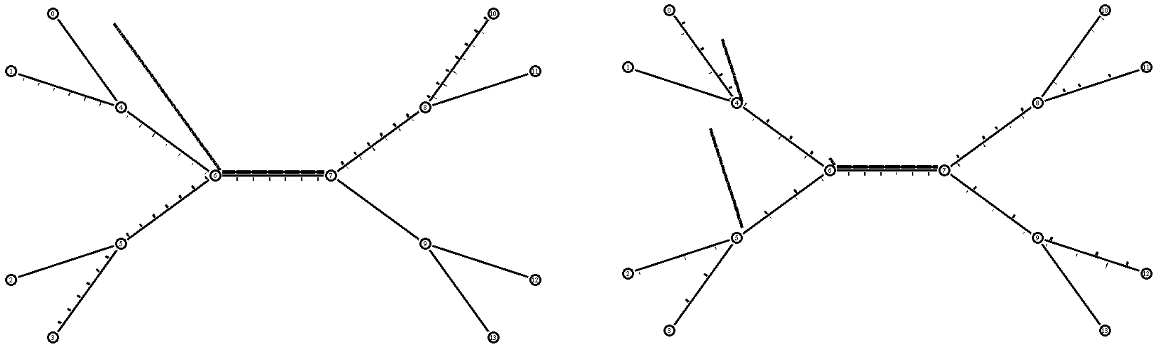
Edge-to-edge Control not only notifies edges of available capacity but EC also distributes backlog (see Figure 4.4). By moving buffer management algorithms from the bottleneck(s) to the ingress edges, the EC architecture enables *distributed buffer management*. Dividing backlog between the edges increases the effective number of buffers. With more buffer, more TCP connections obtain the critical four packet window size that survives loss without timing out. As we demonstrate momentarily, more buffer implies less unfairness induced by TCP's bias against tiny flows [80]. As demonstrated in Figure 4.5(a), using Riviera to distribute the backlog reduces the coefficient of variation in goodput[8] when many TCP connections compete for the bottleneck. As expected this improvement increases as we distribute the congestion across more edges.

Buffer management suffers a fundamental trade-off imposed by TCP fast retransmit. The network cannot drop packets and avoid timeouts without allotting sufficient buffers to accommodate 4 segments from each connection that experiences loss. Distributed buffer management accommodates more packets by distributing backlog across multiple queues. The result is an increase in delay as we distribute the backlog across more edges. Figure 4.5(b) demonstrates one-way delay in excess of propagation and transmission delays.

It is worth noting that buffer management can avoid the stated trade-off between fairness and delay by fairly distributing timeouts rather than avoiding timeouts. Distributing timeouts implies short-term unfairness in order to reduce delay and obtain long-term fairness. Here "short-term" implies seconds since TCP's coarse-grained timers operate within this regime. Alternatively we could require end-systems to implement ECN (see Section 2.6.3) thereby avoiding drops and consequent timeouts so long as transient burstiness does not overflow the buffers in the path.

TCP rate control in conjunction with Riviera avoids all three of the above trade-offs by using the receiver advertised window rather than loss to control end-systems (see next section). Of course TCP rate control is highly stateful, works only with TCP, requires

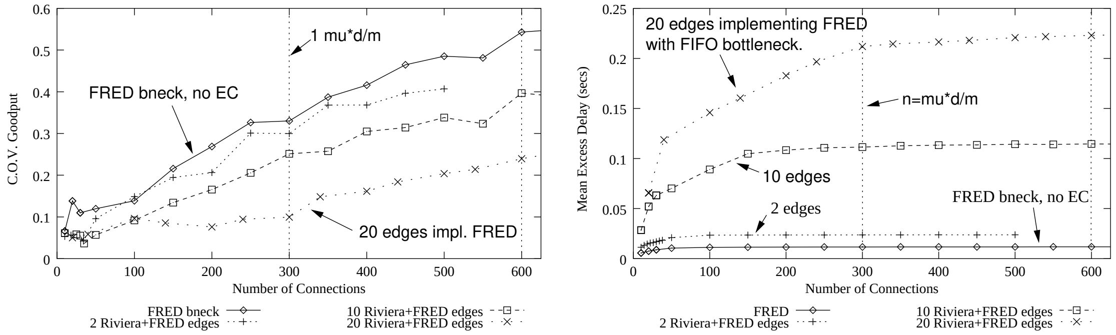
[8] Goodput is defined here as the number of transmitted payload bits excluding retransmissions per unit time.



(a) queue at bottleneck (b) queue distributed across edges

Figure 4.4: Backlog Moved from Bottleneck to Edge

access to TCP headers, and requires access to both data and acknowledgements. Thus distributed buffer management makes sense in environments where TCP headers or acknowledgements are unavailable or where per-flow state introduces unacceptable limits on scalability.



(a) Improved fairness (lower C.O.V.) [a] [b] (b) But larger mean excess delay [a] [b]

[a] 30 second simulation. Statistics gathered from 10s to end.

[b] 100Mbps bottleneck. Vary number of edges in Figure 4.4. One bandwidth-delay product worth of buffer.

Figure 4.5: Distributed Buffer Management: FRED versus Riviera+FRED

Full results appear in Appendix A.12.

4.3 Near Zero Congestive Loss TCP End-System Control Building Block

In environments where TCP rate control can be deployed (see Section 2.8.1), TCPR eliminates virtually all congestive loss under even extreme loads ($<$ 1 packet in the bandwidth-RTPD product per connection) provided there is sufficient buffer at the bottleneck to absorb Riviera's queue oscillation (see Sections 3.3.4 and 3.3.5) and sufficient

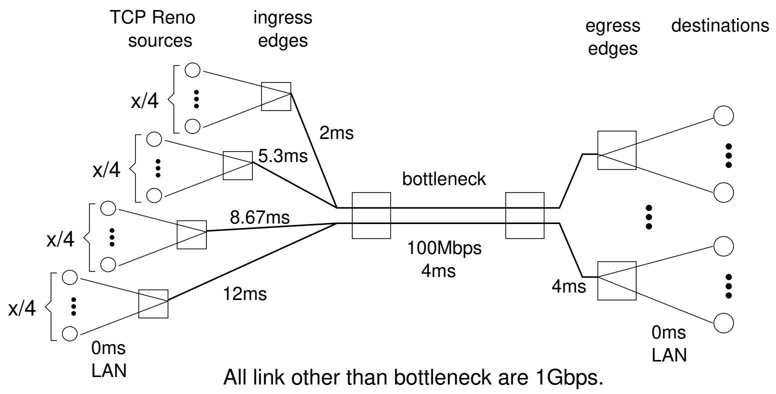


Figure 4.6: Topology Used in Comparing Near Zero Loss Building Block

buffer at the edges to absorb burstiness ($\geq$ 1 packet per connection). Note that this is significantly less than the 4 packets per connection required to avoid timeouts using buffer management. We compare Riviera+TCPR to distributed buffer management building blocks in Figure 4.7 using the topology in Figure 4.6. The buffers in the topology in 4.6 store up to one mean bandwidth-RTPD product of packets. Here we study the effects of infinitely long file transfers. We study the effects of web traffic over Riviera+TCPR in the Appendices respectively.

Figures 4.7(c) and (d) demonstrate Riviera's ability to distribute backlog from the bottleneck to the ingress edges. However unlike distributed buffer management, by setting the receiver advertised window Riviera+TCPR does not resort to loss to reduce queue, and thus under moderate loads achieves low excess delay (a) without incurring low utilization (b), retransmissions (e), timeouts (f), or great unfairness (g). FRED reduces delay below FIFO by aggressively dropping packets resulting in the increased retransmissions and timeouts shown in 4.7(e) and (f) respectively. When there is insufficient buffer to obtain 4 packets per connection in transit, FRED simply reduces the available buffer resulting in worse performance than FIFO.

As the number of connections increases, TCP rate control is unable to avoid queuing. When the wire is full, each connection is constrained to a 1 packet window size. When 1 packet per connection exceeds 1 bandwidth-RTPD product (i.e., 375 packets), the excess packets are queued at the ingress edges resulting in a sharp linear increase right of the 1 bandwidth-RTPD line in delay and ingress queue length as shown in Figures 4.7(a) and (d) respectively. When 1 packet per connection equals 2 bandwidth-RTPD products, 1 bandwidth-RTPD resides on the wire and the rest are distributed across the ingress edges achieving marginally higher than the predicted mean backlog of 375/4 packets as shown in Figure 4.7(d). Thus under moderate loads in steady state conditions, TCPR improves delay and fairness. Under heavy loads, TCPR as with distributed buffer management trades off delay for increased fairness; however, TCPR does not resort to retransmissions or timeouts until so many connections compete that the ingress queues overflow.

4.4 UDP Congestion Control and Isolation of Denial-of-Service Attacks

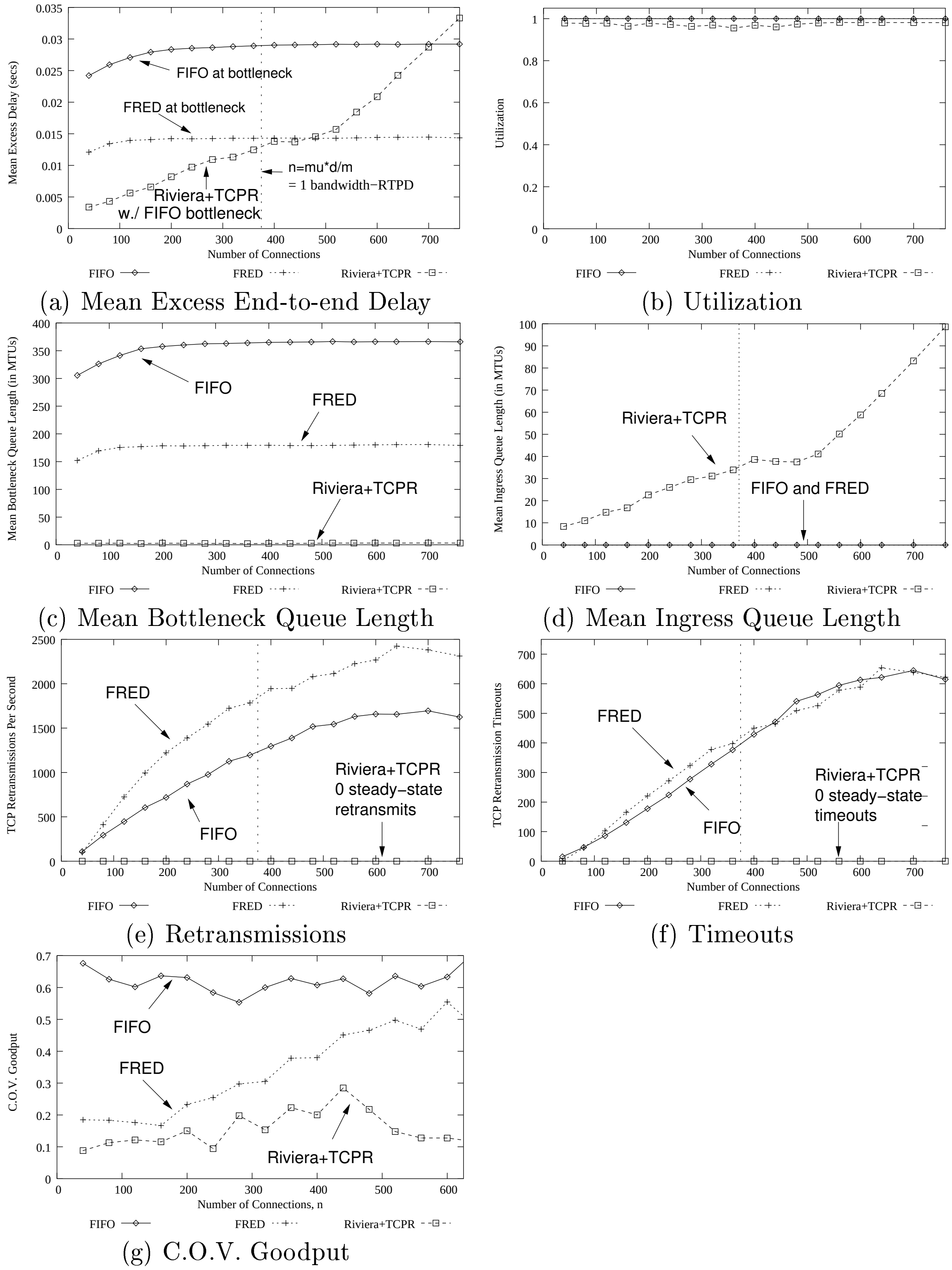
Edge-to-edge Control can control UDP as well as TCP. Using WFQ or any number of other schedulers (see Section 2.4) the edge can allocate a rate to each UDP flow. As long as each UDP flow stays within its share it experiences no drops.

UDP congestion control also provides moderate protection from bandwidth-based Denial of Service (DoS) attacks (e.g., ping floods, smurf attacks, fraggle attacks) against the Edge-Controlled traffic class. As long as the attacker passes through a non-compromised ingress edge, the arrival rate of the attackers packets is constrained to the bandwidth allocated to the EC trunk. Using the same topology as in Figure A.2 with 375 packet buffers, we demonstrate the effects of a UDP flood constrained by edge-to-edge control in Figures 4.8 and 4.1 competing against an EC trunk carrying a TCP flow. Without Edge-to-edge Control, the TCP connection is forced down to 1200bps of a 100Mbps bottleneck. Riviera constrains the UDP attack to the EC Trunk's fair share of the bottleneck (49.9Mbps, row 1, last column in Figure 4.1).

Second, the mere fact that Edge-to-edge Control is isolated into a separate traffic class provides protection from bandwidth-based Denial-of-Service (DoS) attacks to other traffic classes.

As an edge-based service, an EC edge can employ stateful mechanisms to constrain misbehaving TCP end-systems. Proposed methods for isolating misbehaving TCP connections have been constrained by the minimal state afforded by the bottleneck [40, 4]. Much more might be accomplished with the relaxed state constraints imposed by an edge box. We leave this up to later research.

Unfortunately edge-to-edge control does not protect against attacks that compromise or circumvent EC edges.



(a) Mean Excess End-to-end Delay

(b) Utilization

(c) Mean Bottleneck Queue Length

(d) Mean Ingress Queue Length

(e) Retransmissions

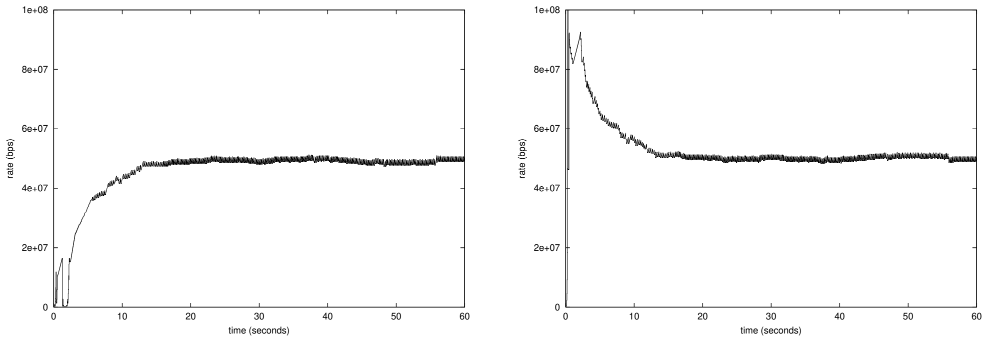
(f) Timeouts

(g) C.O.V. Goodput

Figure 4.7: Buffer Management versus Near Zero Loss Building Block

4.5 Other Potential End-system Control Building Blocks

All of the building blocks so far maintain FIFO service discipline. End-system control service blocks might reorder packets to provide flow-level delay differentiation. An



TCP Reno connection — UDP flow sending at 200Mbps

Figure 4.8: TCP versus UDP with both Regulated by Riviera

Table 4.1: TCP versus UDP Statistics

Bottle-neck[a]	Edge Scheme	Utilization	Queue Length[b] (avg/max pkts)	Drops (pkts)	Throughput (TCP/UDP in Mbps)
FIFO	Riviera	99.0%	0.886/7	2/750322/**0**[c]	**49.1/49.9**
FRED[d]	none	100%	42.5/175	702448[e]	40.5/59.5
FIFO	none	100%	373.5/375	500008[f]	**1200bps/100.0**

[a] Statistics collected starting at 20 seconds in a 60 second simulation.
[b] Bottleneck queue length. We ignore ingress queue length for now.
[c] Drops in ingress serving TCP, ingress serving UDP, and interior respectively.
[d] In all simulations we used the default FRED parameter set given in [80].
[e] Drops at the bottleneck. Mostly UDP drops else TCP would receive less share.
[f] Drops at the bottleneck. Mostly UDP drops because TCP is nearly starved.

end-system control service building block might avoid dropping retransmitted packets in order to reduce the probability of timeouts [81]. Another building block might provide per-flow bandwidth assurances to certain connections at the expense of others with over-subscription occurring whenever the EC trunk's capacity reduces below the sum of the assurances. For UDP, application-aware building blocks might preferentially drop less important packets. For example, an application-aware UDP service might guard against dropping key frames in a video stream. We present here an example of congestion control of a single UDP flow and a single TCP flow each passing through a separate EC trunk. We leave these and other building blocks up to later study.

5. Per-trunk Service Building Blocks

In this chapter we modify the Riviera control policy in response to accumulation to realize a variety of bandwidth sharing objectives. By controlling the aggregate rate of a traffic trunk, Riviera allows the traffic trunk to converge on a desired bandwidth allocation. Bandwidth, delay, and loss allocation between end-to-end flows within the Riviera control loop is controlled at the edge using mechanisms already discussed in Chapter 4.

First we define a set of user utility functions not previously studied and describe their motivation. We then provide supporting analyses and simulation of our new building blocks that implement these utility functions. In the appendix A we provide an expanded set of simulations for these building blocks.

5.1 Bandwidth Allocation Basics

Consider an arbitrary network using the notation given in Section 3.3. In addition let p_i denote the amount paid by loop i.

Low [82] treats congestion at each link as a price per unit of bandwidth. When our congestion measure is backlog as in [83], a control loop's share of the accumulation is proportional to the loop's total price per unit bandwidth. An alternative interpretation of the price paid is the weight assigned to a source in a weighted proportional fair bandwidth allocation. For simplicity let's ignore the fact that Riviera's control policy oscillates each loop's accumulation, and instead assume that each control loop converges on a target accumulation and that each control loop is free to set this target accumulation now denoted ϵ_i. Combining these points together,

$$\forall i \in S, \quad w_i \propto \epsilon_i \propto p_i \quad .$$

If we choose some canonical values for parameter ϵ_{min} and ϵ_{max} then we can achieve weighted proportional fairness by modifying the Riviera control policy to

$$\begin{aligned} &\text{if } q_i \;>\; w_i \cdot \epsilon_{max} && \text{congestion begins ,} \\ &\text{if } q_i \;\leq\; w_i \cdot \epsilon_{min} && \text{congestion ends .} \end{aligned}$$

We now apply one of Kelly's more important results to show that Riviera in combination with a proper payment mechanism maximizes total utility for an arbitrary set of loops over an arbitrary topology. According to Kelly [69], for a network that realizes

$$\begin{aligned} \text{maximize} \quad & \sum_{i \in S} w_i \ \log \lambda_i \\ \text{subject to} \quad & \sum_{i \in S(l)} \lambda_i \le c_l, \ \lambda_i \ge 0, \ w_i > 0 \end{aligned} \tag{5.2}$$

and subject to certain constraints on U_i, when each source maximizes its benefit by setting the amount the source pays,

$$\text{maximize} \quad U_i(\lambda_i) - p_i \lambda_i \tag{5.3}$$

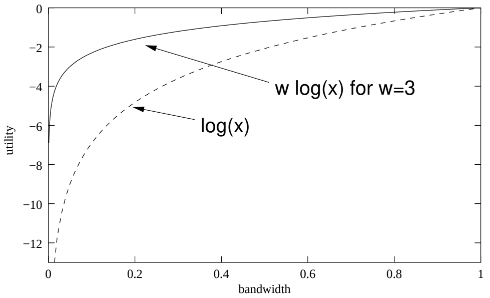
$$\text{subject to} \quad \lambda_i \ge 0 \ , \tag{5.4}$$

the system optimizes total utility

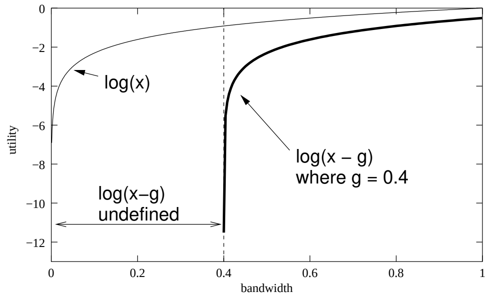
$$\begin{aligned} \text{maximize} \quad & \sum_{i \in S} U_i(\lambda_i) \\ \text{subject to} \quad & \sum_{i \in S(l)} \lambda_i \le c_l, \ \lambda_i \ge 0 \ \ . \end{aligned} \tag{5.5}$$

Kelly constrains user utility U_i such that U_i must be strictly concave, increasing, and continuously differentiable. Shenker [104] refers to traffic having a strictly concave, increasing, continuously differentiable utility function as *elastic*.

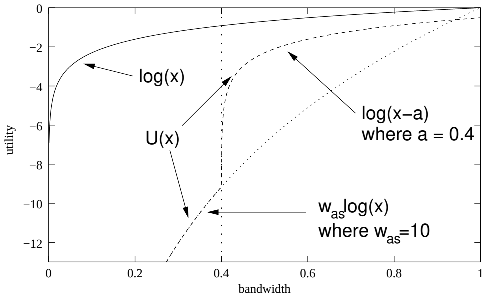
In this chapter we relax Kelly's constraints on the utility function to include certain *inelastic* applications. For these utility functions, the user inputs a weight, a required bandwidth, or a desired bandwidth. We have not yet incorporated the price concept. In particular, we consider three example user utility functions that have simple implementations. To the best of our knowledge, the last two of the three utility functions below have not been previously studied in the context of bandwidth sharing. The last two also correspond to *inelastic* applications:



(a) Weighted User Utility, U^w



(b) Guaranteed User Utility, U^g



(c) Assured User Utility, U^a

Figure 5.1: User Utility Functions

$$U_i^w(\lambda_i) = w_i \log \lambda_i \tag{5.6}$$

where $w_i > 0$, $\lambda_i \geq 0$.

$$U_i^g(\lambda_i) = \begin{cases} w_i \log(\lambda_i - g_i) & \text{if } \lambda_i \geq g_i \\ \text{undefined} & \text{otherwise} \end{cases} \tag{5.7}$$

where $w_i > 0$ and $\lambda_i \geq g_i \geq 0$.

$$U_i^a(\lambda_i) = \begin{cases} w_{Bi} \log(\lambda_i - a_i) & \text{if } \lambda_i > y_i \\ w_{Ai} \log \lambda_i & \text{otherwise} \end{cases} \tag{5.8}$$

where $w_{Ai} > w_{Bi} \geq 0$, $\lambda_i \geq 0$, and $y_i = a_i/(1 - e^{w_{Bi}/w_{Ai}})$.

Definition (5.6) represents the utility function of a user that desires weighted proportional fair service. As already stated, U^w is trivially implemented, since Riviera already approximates weighted proportional fairness.

Definition (5.7) represents a user or application that requires a minimum bandwidth (see Figure 5.1(a)) but is elastic to bandwidth changes above this minimum. Any bandwidth below g_i results in an undefined utility. The undefined region $\lambda_i \leq g_i$ violates the user utility function constraints imposed by Kelly resulting in unbounded price as λ_i approaches g_i from the right, but U_g is still implementable provided precautions are taken to ensure that the system does not operate in the region where the utility is undefined. We present an implementation of this utility function in Section 5.3. Furthermore, U^g results in well-defined bandwidth allocations even when other users sharing the same network use U^w, U^g or any strictly concave, increasing, and continuously differentiable utility function. We analyze U^g further in Section 5.3.

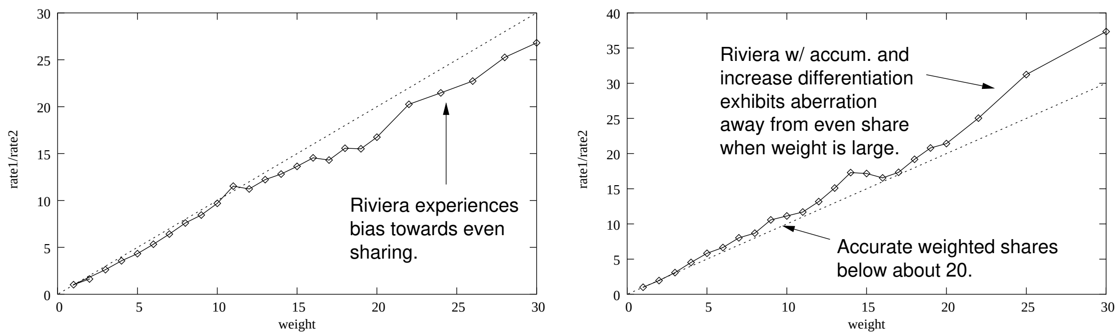
Definition (5.8) represents a user that desires a minimum bandwidth a_i (denoting "assurance") but can tolerate less bandwidth. As demonstrated in Figure 5.1(b), U^a contains a cusp. U^a represents a *piece-wise concave* user utility function. The point where the two curve segments intersect is neither differentiable nor concave. When all assurances are satisfiable, meaning the sum of the assurances is less than the capacity in the network, all users find themselves in the right concave segment and thus a unique optimal solution exists. When all of the assurances exceed the capacity in the network then all of the user's find themselves in the left concave segment and thus again a single unique optimum exists. When only some of the assurances arise, multiple optima may exist.

When only a subset of the assurances can be satisfied, our objective is to use policy to direct the system to a unique equilibrium that is often suboptimal, yet with policy-based knobs we are able to move the equilibrium point to control which assurances are likely to be satisfied first.

We describe more what this utility function means and present an approximate implementation in Section 5.4. There is substantial left work in determining the equilibria and devising policy-based mechanisms for controlling convergence.

5.2 Weighted Proportional Fair Service Building Block

In the previous section we already specified how to implement a weighted proportional fair share by scaling canonical values for ϵ_{min} and ϵ_{max} by weight w_i. We call this strategy *accumulation differentiation.* We demonstrate this control policy in Figure 5.2(a). As in Figure 5.2(a), our simulations reveal that accumulation differentiation alone results in a bias toward even sharing as weights increase. In this section we demonstrate that the



(a) Accumulation Differentiation (b) Accum. Plus Increase Differentiation

Two control loops over single 100Mbps bottleneck topology with 24ms RTPD on each path. Control loop 1 has weight x. Loop 2 has weight 1.

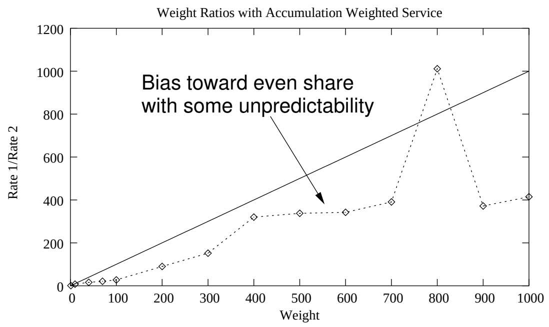
Figure 5.2: Riviera with Weighted Accumulation Differentiation

problem gets much worse at higher weights and demonstrate a possible solution.

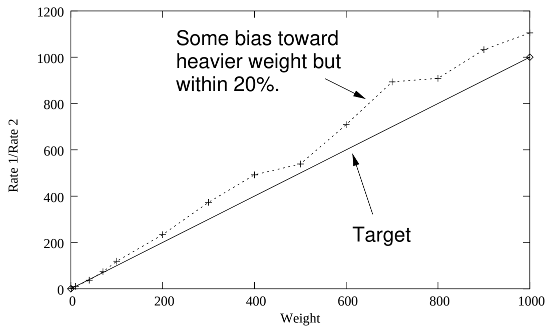
The bandwidth allocation obtained by optimizing $\sum_i \log \lambda_i$ does not consider the control policy used to obtain the allocation. With AIMD2ER, the additive increase is the same for all loops rather than proportional to the loop's weighted share. Thus during increase, the loop with the larger share falls behind its weighted share. Figure 5.2(b) demonstrates that weight proportional additive increase overcompensates resulting in a bias in favor of the loop with the larger weight. Hereafter we refer to setting the additive increase proportional to the weight as *increase differentiation*. We now demonstrate that as weights increase the problem grows far worse without increase differentiation.

At higher weights, Figure 5.3 demonstrates that increase differentiation behaves in a more predictable manner albeit it still demonstrates subtle bias in favor of the loop with the heavier weight. However, the ability to obtain reasonable accuracy at weights as high as 1000-to-1 is an order of magnitude better than achieved in prior work [92, 28] based on differentiation via congestion control. We expand on these results in Section A.13.

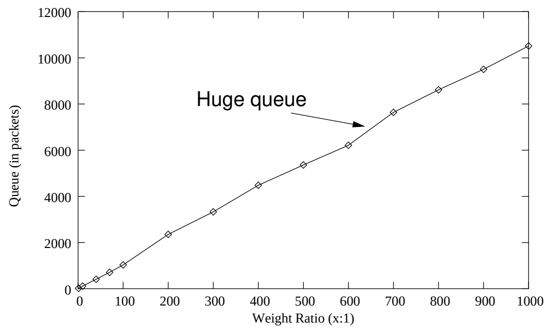
Obtaining such large weighted shares has the trade-off of incurring large maximum queues (see Figure 5.3(c)). Large queue lengths are an inherent limitation of requiring each loop to incur an accumulation proportional to its weight. However, if we permit ourselves to introduce active queue management at the bottleneck we might be able to control queue length without losing the ability to vary weight over a large range. Low [5] uses REM AQM to control Vegas. Since Vegas's control policy is also based on accumulation estimation, REM may be applicable to Riviera. We leave this to further research.



(a) Weighted Accumulation w/ $\beta = 0.8$



(b) Weighted Accum. and Inc. w/ $\beta = 0.8$



(c) Queue Length w/ Weighted Accum. and Inc.

Figure 5.3: Riviera Weighted Sharing With Additional Backoff

To motivate the use of an additive increase step proportional to the weight, we now employ a simple single bottleneck fluid model where each loop has equal RTPD. We now demonstrate that increase differentiation causes the optimal point (i.e., equal shares) to be an equilibrium point. Starting from the optimal point at $t = 0$, all control loops have a rate allocation $\lambda_i(0) = \frac{w_i}{W}\mu$ where $W = \sum_i w_i$ and μ is the bottleneck service rate. Each control loop increases with acceleration αw_i. Therefore,

$$\lambda_i(t) = \frac{w_i}{W}\mu + \alpha w_i t \ .$$

Each control loop's share of the arrival rate is

$$\frac{\lambda_i(t)}{\lambda} = \frac{\frac{w_i}{W}\mu + \alpha w_i t}{\mu + W\alpha t} = \frac{w_i}{W} \ . \tag{5.10}$$

Thus each control loop's share of the bottleneck remains constant as long as all control loops are increasing. Because each control loop's contribution to the queue length grows proportional its share of the arrival rate,

$$q_i(t) \approx \frac{\lambda_i(t)}{\lambda(t)} q = \frac{w_i}{W} q \ . \tag{5.11}$$

When a control loop detects congestion its $q_i = w_i \epsilon_{max}$. Because all control loops increase proportional to their weights, in our model all control loops reach the max threshold simultaneously when $q = \frac{W}{\epsilon}_{max}$. Because a multiplicative backoff does not change the relative shares of each control loop, (5.10) still holds immediately after the backoff. Furthermore each control loop has drain capacity,

$$(1-\beta)\nu_i(t) = (1-\beta)\frac{\lambda_i(t-rtt)}{\lambda(t-rtt)}\mu = (1-\beta)\frac{w_i}{W}\mu \ . \tag{5.12}$$

Thus the drain time t_{di} for the ith control loop is governed by

$$\begin{aligned} (1-\beta)\frac{w_i}{W}\mu t_{di} &= w_i q - w_i \epsilon_{min} \\ t_{di} &= \frac{q - \epsilon_{min}}{1-\beta} W \end{aligned} \tag{5.13}$$

Notice that t_{di} is not a function of w_i and thus in our fluid model all control loops leave congestion at the same time and once again begin weight proportional additive increase maintaining Invariant (5.10).

5.3 Guaranteed Bandwidth Building Block

To determine the appropriate bandwidth allocation for U^g,

$$\begin{array}{ll} \text{maximize} & \sum_{i \in S} w_i \log(\lambda_i - g_i) \\ \text{subject to} & \sum_{i \in S(l)} \lambda_i \le c_l, \lambda_i \ge g_i \ge 0, w_i > 0 \\ \text{for all} & l \in L, i \in S \ . \end{array} \tag{5.14}$$

Optimization (5.14) is a special case of the following problem. Let $U_i^h(\lambda_i)$ denote a function that is increasing, differentiable and strictly concave for $\lambda_i \ge g_i$ where g_i is a nonnegative constant. (5.14) is thus a special case of

$$\begin{aligned}
\text{maximize} \quad & \sum_i U_i^h(\lambda_i) && (5.15)\\
\text{subject to} \quad & \sum_{i \in S(l)} \lambda_i \leq c_l, \lambda_i \geq g_i \geq 0 && (5.16)\\
\text{for all} \quad & l \in L, i \in S\ .
\end{aligned}$$

Because the feasible region (5.16) is compact and the objective function of (5.15) is differentiable and is a strictly concave function of λ_i within the feasible region, the optimal vector λ exists and is unique.

Let $U_i^u(\lambda_i - g_i) = U_i^h(\lambda_i)$, let $u_i = \lambda_i - g_i$, and let $v_l = c_l - \sum_{i \in S(l)} g_i$. Substituting u_i and v_l into 5.15 yields

$$\begin{aligned}
\text{maximize} \quad & \sum_i U_i^u(u_i) && (5.17)\\
\text{subject to} \quad & \sum_{i \in S(l)} u_i \leq v_l, u_i \geq 0 \\
\text{for all} \quad & l \in L, i \in S\ .
\end{aligned}$$

Given any set of utility functions $\mathbf{U}$ that are differentiable and strictly concave over $\forall i \in S : \lambda_i \geq 0$, Optimization 5.17 reveals that by translating each $U_i \in \mathbf{U}$ by g_i results in the bandwidth allocation vector $\lambda = \mathbf{g} + \lambda'$ where λ' is the bandwidth allocation obtained on a network with links L' but without guarantees that is identical to the original network except that each link $l' \in L'$ has capacity $c_l' = c_l - \sum_{i \in S(l)} g_i$ where l is the corresponding link in L.

Notice that Optimization 5.17 has the exact same form as Optimization 5.2. Thus, when every user has utility function $U^u = U^g$ then each user obtains a guarantee plus a weighted proportional fair share of any capacity not allocated to guarantees.

5.3.1 Guaranteed Building Block Implementation

In this section we implement a *Quasi-Leased Line (QLL)*. A QLL implements the guaranteed utility function U^g. We use the term “quasi-” because the QLL provides a bandwidth guarantee but provides only loose control over delay and delay jitter due to the limitations imposed by the propagation delay in the control loop.

To implement U^g we consider a reference network with links L' where each link $l' \in L'$ has a corresponding link in the real network except the capacity of l' is $c_l' = c_l - \sum_{i \in S(l)} g_i$. We then partition the traffic entering each traffic trunk so that only that traffic above the trunk's guarantee is seen by the reference network. All traffic below each trunk's guarantee is admitted to the real network but never seen by the reference network.

First we consider how to estimate the accumulation in the reference network from characteristics measurable from the real network. By employing Little's Law, we can estimate accumulation for an arbitrary network with arbitrary service time distributions and subjected to an arbitrary set of trunks with general arrival time distributions. Let N denote the number of packets in the real network and N' in the reference network. Let λ equal the aggregate rate of packets entering the real network and λ' for the reference network. Little's Law tells us that the average number of packets in a system operating in the steady-state equals the average arrival rate times the average time each packet spends in the system. Thus,

$$E[N'] = E[\lambda'] \cdot E[rtt]$$

Since Little's Law applies to an arbitrary system in the steady-state, we can consider only the traffic generated by a single control loop. All other traffic is "part of the system." Therefore,

$$E[N_i'] = E[\lambda_i'] \cdot E[rtt_i] \ .$$

Furthermore we exclude packets that are propagating (i.e., not queued) by substracting $basertt$ from the time packets spend in the system. Let q_i' denote the ith control loop's accumulation in the reference network, and let $d_i = rtt_i - basertt_i$. Therefore,

$$E[q_i'] = E[\lambda_i'] \cdot E[d_i] \ . \tag{5.20}$$

Since λ_i' is the traffic seen by the reference network,

$$\lambda_i' = max(0, \lambda_i - g_i) \ . \tag{5.21}$$

The "max" function appears in Equation (5.21) because λ_i may be less than g_i. Note that in deriving Optimization 5.17 we assume that the network is the bottleneck.

Here λ_i is the demand provided by the application which may be limited either by the application itself or by the network, therefore we define the guarantee as applying only when the network is the bottleneck.

In the steady-state $E[\lambda_i] = E[\nu_i]$, therefore, we make a steady-state approximation,

$$\lambda'_i \approx max(0, \nu_i - g_i) \ . \tag{5.22}$$

We combine equations (5.22) and (5.20) to create an estimator of q'_i.

$$\hat{q}'_i = max(0, \nu_i - g_i) \cdot d_i \ . \tag{5.23}$$

Assume the egress knows g_i as an input parameter. The egress can measure ν_i, rtt_i and $basertt_i$. Thus, if we were using the Vegas accumulation estimator, we could apply Equation (5.23) directly. To estimate $\hat{q}'_i$ based on the Riviera accumulation estimator $\hat{q}^R_i$ we use the fact that $\hat{q}^R_i$ is an unbiased estimator of the average number of packets backlogged in the path,

$$E[\hat{q}^R_i] = E[\lambda_i] \cdot E[d_i] \ . \tag{5.24}$$

Again making a steady-state approximation and substituting the results into (5.23) yields

$$\hat{q}'_i = max(0, \nu_i - g_i) \frac{\hat{q}^R_i}{\nu_i} \tag{5.25}$$

Once $\hat{q}'_i$ is known, each loop i in the reference network adjusts its rate limit as,

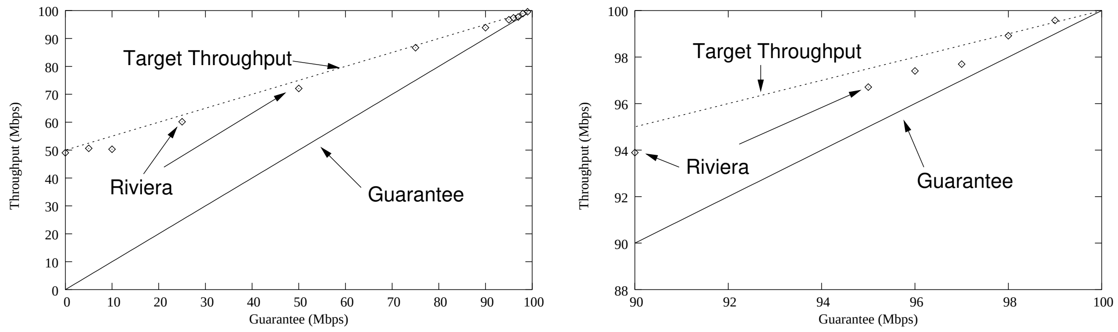
$$r_i \leftarrow \begin{cases} \nu'_i + w_i \cdot \frac{m}{rtt_i} & \text{if no congestion} \\ \beta \nu'_i & \text{if congestion} \end{cases}$$

where

$$\begin{array}{ll} \text{if } \hat{q}'_i > w_i \cdot \epsilon_{max} & \text{then congestion begins ,} \\ \text{if } \hat{q}'_i \leq w_i \cdot \epsilon_{max} & \text{then congestion ends .} \end{array}$$

From this we derive the control policy for the real network,

$$r_i \leftarrow \begin{cases} \nu_i + w_i \cdot \frac{m}{rtt_i} & \text{if no congestion ,} \\ \beta(\nu_i - g_i) + g_i & \text{if congestion ,} \end{cases}$$



(a) Throughput for Small Guarantees (b) Throughput for Large Guarantees

Figure 5.4: QLLs Through a Single Bottleneck

$$r_i \leftarrow max(r_i, g_i) \ .$$

where congestion ends and begins at the same times as in the reference network.

The maxima prevents the rate limit r_i from dipping below g_i. The maxima also eliminates the need to take the max of 0 and $\nu_i - g_i$ in the decrease step of the real network's control policy.

Note that the above control policy leaves out the complexity incurred by slow start. We simply state here that the maxima applies to slow-start as well, meaning that the control loop starts the slow-start from g_i.

5.3.2 QLL Sample Simulation Results

In Figure 5.4 we show results for a series of simulations run on the single bottleneck topology given in Figure A.2 for two control loops. One control loop has no guarantee and we vary the guarantee for the other control. In this simple simulation, the achieved throughput in all cases exceeds the guarantee and approximately follows the target throughput though there does appear to be a slight bias in favor of the control loop without a guarantee. We ran other simulations not shown here, and the results are consistent that the QLL loses part of its fair share to loops without guarantees, but the loss in bandwidth is usually within 10-15%.

In Figure 5.5 we show a QLL operate over the topology in Figure A.4 with one cross-flow at each hop and one flow passing through multiple hops. The bandwidth diminishes with the number of hops as dictated by proportional fairness but adjusted to exclude the portion of the bandwidth allocated to guarantees.

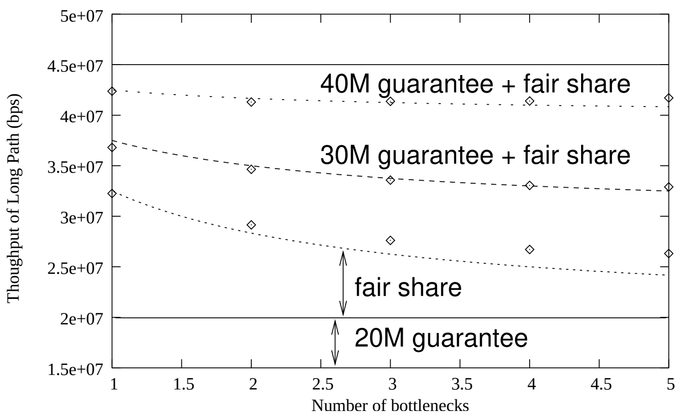


Figure 5.5: Bandwidth Allocation for QLL over Multiple Bottlenecks

We discuss the results in Figures 5.4 and 5.5 in more detail in Appendix A.14, including a demonstration of the trade-off in steady-state queue length versus size of the guarantee. As with the weighted service, the quasi-leased line incurs substantial queue as the guarantee increases, and we see only one potential solution to support large guarantees: add AQM at the bottleneck's where we cannot cost effectively allocate sufficient buffer to avoid q congestive loss. In a network that is dedicated mostly to best-effort traffic with only a small fraction of the bandwidth allocated to guarantees and with a moderate number of EC control loops, AQM may not be necessary.

5.3.3 Quasi-Leased Line Startup Queue Length Bound

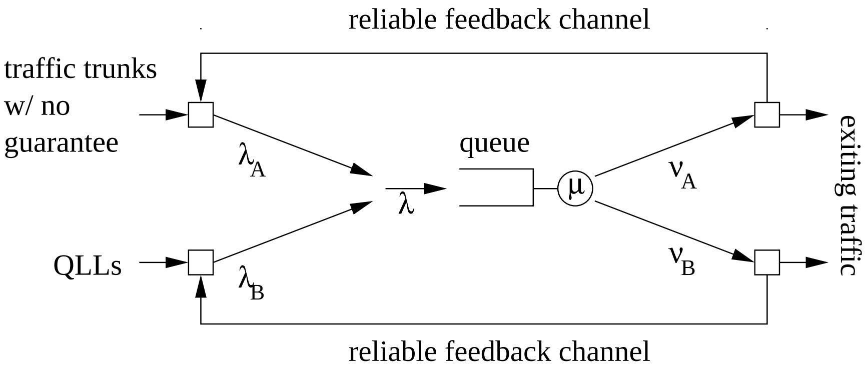


Figure 5.6: Quasi-Leased Line Analysis Topology and Variables

In this section we provide a simple analysis that derives a queue length bound

$$\mathsf{qmax} = \frac{b}{1-b}$$

on the queue incurred by a bounded jump in demand as caused by a starting quasi-leased line. Here b is the fraction of the bottleneck capacity allocated to guaranteed traffic and thus represents a bound on the jump in demand seen at the bottleneck, and qmax is in units of bandwidth-RTT products.

For this analysis we use a simplified discrete time model based on the topology in Figure 5.6. A is a set of EC trunks that implement U^w all with weight 1. B is a set of quasi-leased lines. At the beginning of the analysis, the EC trunks in A have fully utilized the link. At time 0, the quasi-leased lines perform a maximum jump in demand to incur a large backlog $\forall n : q[n] \leq \mathsf{qmax}$. $\lambda_A[n]$ represents the aggregate arrival rate of the EC trunks in A at the nth time step. $\lambda_B[n]$ represents the aggregate arrival rate of the quasi-leased lines in B at the nth time step. Each time step represents one interval T. In keeping with discrete time, we assume all virtual links respond to congestion (i.e., $\lambda > \mu$) at the next time step. Our analysis has the following givens:

- $b\mu$ is the portion of the bottleneck allocated to quasi-leased lines, $0 \leq b < 1.0$
- $\forall n \geq 0 : \lambda_B[n] = b\mu$
- $\lambda_A[0] = \mu$
- q[0] $= 0$
- when congestion arises each EC trunk in A backs off to $\beta\nu_i$ where $\sum \nu_i = \nu_A$.

Theorem 5.3.1 $\mathsf{qmax} < \frac{b}{1-b}$ *in bandwidth-RTPD products.*

Proof.

First, we observe that the queue length is a function of the arrival rate in excess of the capacity as in equation 5.26 below. The purpose of this analysis is to show that the queue grows slowly enough to converge below the hypothesized bound.

$$\forall i : q[i] = \mathsf{max}(q[i-1] + \lambda[i] - \mu, 0) \tag{5.26}$$

For the purposes of this analysis, it does not matter how many EC trunks are in A. The aggregate back off is always $\sum \beta\nu_i = \beta \sum \nu_i = \beta\nu_A$.

In the worst-case $\beta = 1.0$ meaning that the EC trunks only backoff to the arrival rate at the egress. Thus the aggregate rate λ does not backoff below the capacity. We will show in the rest of this analysis that the queue length converges on a finite bound even when $\beta = 1.0$. Intuitively such a bound for $\beta = 1.0$ should also apply for $0 < \beta < 1.0$, but I do not rigorously show this here. Without loss of generality we also assume all rates are in units of the bottleneck capacity, thus $\mu = 1$.

$$\begin{aligned} \lambda[0] &= \lambda_A[0] + \lambda_B[0] = 1 + b && (5.27) \\ \lambda[i] &= \lambda_A[i] + b && (5.28) \end{aligned}$$

As long as there is no loss, each control loop obtains a share of the bottleneck proportional to its arrival rate:

$$\nu_A[i] = \frac{\lambda_A[i]}{\lambda[i]}\mu = \frac{\lambda_A[i]}{\lambda[i]}, \quad \nu_B[i] = \frac{\lambda_B[i]}{\lambda[i]}\mu = \frac{\lambda_B[i[}{\lambda[i]} \tag{5.29}$$

Feedback arrives in the next time step:

$$\begin{aligned} \lambda_A[i] &= \beta\nu_A[i-1] = \frac{\lambda_A[i-1]}{\lambda[i-1]} \\ \lambda[i] &= \frac{\lambda_A[i-1]}{\lambda[i-1]} + b && (5.30) \end{aligned}$$

Because of equation 5.28, we know that $\lambda_A[i-1] = \lambda[i-1] - b$, thus 5.30 becomes,

$$\begin{aligned} \lambda[i] &= \frac{\lambda[i-1] - b}{\lambda[i-1]} + b \\ \lambda[i] - 1 &= \frac{b}{\lambda[i-1]}(\lambda[i-1] - 1) && (5.31) \end{aligned}$$

In order to show that equation 5.26 stays within the hypothesized queue length bound, we need to 1) show that 5.31 converges as $i \to \infty$, 2) show that equation 5.31 converges to μ (i.e., 1), and 3) determine how quickly equation 5.31 converges. To accomplish these, I intend to show that $\lambda[i] - 1$ stays between zero and an exponentially decaying function. To show this we first need only prove that $\frac{b}{\lambda[i-1]} \leq b$. This is true so long as the following lemma is true:

Lemma 5.3.1

$$\forall i \geq 1 : \lambda[i] > 1$$

Proof.

We use inequality 5.32 below as a base step for induction:

$$\lambda[0] > 1 \tag{5.32}$$

Assume $\lambda[i-1] > 1$. Thus $0 < (1 - \frac{1}{\lambda[i-1]}) < 1$.

Since $0 < b < 1$,

$$0 < b(1 - \frac{1}{\lambda[i-1]}) < 1$$

From equation 5.31, we can rewrite the above equation as:

$$\begin{aligned} 0 &< \lambda[i] - 1 < 1 \\ 1 &< \lambda[i] < 2 \end{aligned} \tag{5.33}$$

By applying induction to the basis step, the assumption that $\lambda[i-1] > 1$, and equation 5.33, we know that $\lambda[i] > 1$ for all $i \geq 0$.

□

From lemma 5.3.1 we know $\lambda[i] > 1$ thus $\frac{b}{\lambda[i-1]} < b$ for all $i \geq 0$. Thus,

$$(\lambda[i] - 1) = \frac{b}{\lambda[i-1]}(\lambda[i-1] - 1) < b(\lambda[i-1] - 1)$$

Thus we only need to investigate the convergence of $(\lambda[i] - 1) = b(\lambda[i-1] - 1)$. To do this, let $x_i = \lambda[i] - 1$. Thus $x_i = bx_{i-1}$. This is a geometric series, so $x_i = b^i x_0$. Thus,

$$(\lambda[i] - 1) < b^i(\lambda[0] - 1) \tag{5.34}$$

Combining equations 5.26, 5.27, and 5.34 yields

$$\begin{aligned} q[i] &< q[i-1] + b^i(\lambda[0] - 1) + 1 - 1 \\ q[i] &< q[i-1] + b^i(\lambda[0] - 1), \quad \lambda[0] - 1 = b \\ q[i] &< q[i-1] + b^{i+1} \end{aligned} \tag{5.35}$$

Thus,

$$\lim_{n \to \infty} q[n] < \sum_{i=0}^{\infty} b^{i+1} = \sum_{i=1}^{\infty} b^i = \frac{b}{1-b} \tag{5.36}$$

The elements in this summation are increasing so the maximum occurs at $n = \infty$. Thus,

$$\mathsf{qmax} < \frac{b}{1-b} \tag{5.37}$$

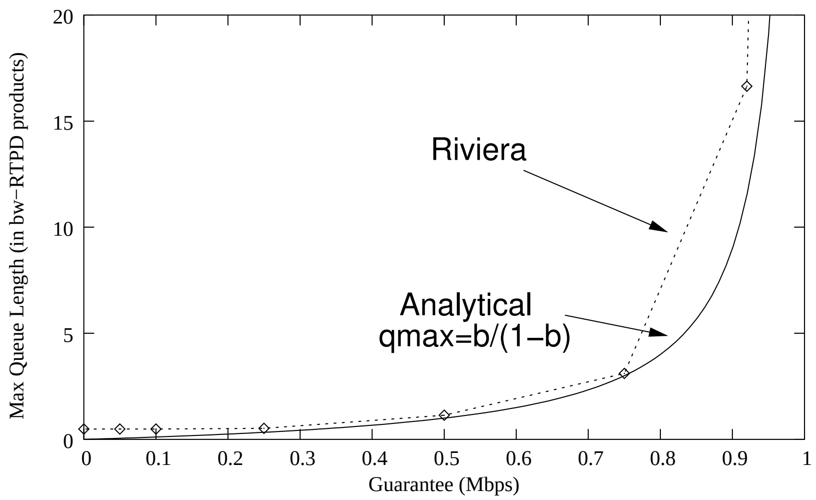


Figure 5.7: Analytic and Simulated Max Queue Length vs. b

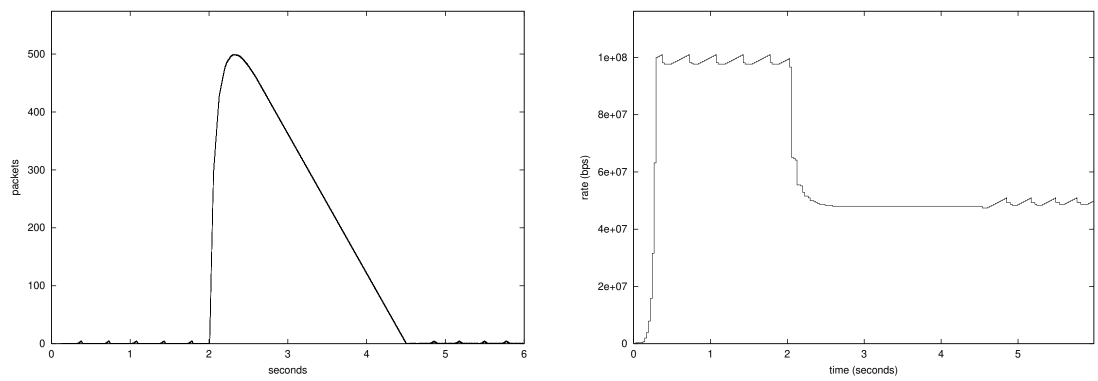
□

For the case of a quasi-leased line jumping to its full guaranteed bitrate after one EC trunk has already fully utilized the bottleneck, Figure 5.7 demonstrates both analytical and simulation results for EEC and Riviera. EEC closely matches the analysis over most of the tested range. Riviera is not quite as quick at catching the sharp increase in demand, but still it appears to follow the analysis reasonably close. We attribute this to the fact that Riviera probes for congestion only twice in each interval whereas with EEC, the bottleneck marks immediately when the queue length exceeds the mark threshold. By including a safety factor, we should be able to retain near zero loss behavior even with Riviera.

Here an EC trunk and a quasi-leased line pass through the single bottleneck topology shown in Figure A.2. The QLL is fed by a constant-bit rate source sending with rate equal to the bandwidth guarantee. Figure 5.8(a) and (b) show the bottleneck queue length and shaper rate limit for the Riviera and $b = 0.5$ case. In all of the simulations the quasi-leased line starts at 2.0s. Thus the EC trunk has two seconds to converge on a fully-utilized bottleneck.

5.4 Assured Bandwidth Building Block

The assured bandwidth building block operates under similar principles as the guaranteed bandwidth building block. However, instead of imposing a hard lower bound on the rate a user can send, the assured service employs a cusp (see Figure 5.1(c)). Below



Bottleneck queue length — Non-QLL shaper rate limit

Figure 5.8: Behavior of an EC Trunk and QLL for $b = 0.5$

the cusp the control loop pushes harder (i.e., has a heavier weight), and when above the cusp the control loop competes with a lighter weight. *Oversubscription* occurs whenever the sums of the bandwidths at the cusps exceeds the available bandwidth, i.e.,

$$\sum_{i \in S(l)} y_i > C_l \tag{5.38}$$

where y_i is the allocation at i's cusp and is defined in (5.8).

When no oversubscription exists, we can apply the same argument used to derive the bandwidth allocation for QLLs. The result is that all control loops obtain a weighted proportional fair share in excess of their respective bandwidth assurances.

When oversubscription exists, there is no guarantee that a unique optimal solution exists. However, we can set the weight that a user employs when below its assurance in order to control which assurances are likely to be satisfied first as capacity becomes available. We have not rigorously mathematically derived the equilibria or proven that the implementation we present in the next section always converges on a unique equilibrium. We are studying these properties now.

The assured building block has certain advantages over the QLL:

1. admission control is optional,
2. during oversubscription no control loops are starved,
3. and which assurances are satisfied during oversubscription is defined by policy (see Sections 5.5, 5.6, and 5.6.1).

The assured building block also has the disadvantage that because admission control is optional, the control loop cannot jump straight to a guaranteed bandwidth and refuse to backoff below the guarantee. Instead the control loop must oscillate toward the target bandwidth allocation. Thus the QLL takes far less time to obtain its guarantee than does the assured building block to obtain its assurance.

5.4.1 Assured Building Block Implementation

To implement the assured bandwidth building block, we use a similar strategy as used to implement the QLL. Without loss of generality, we break the network into two subsystems, subsystem A services the assured component λ_i^A of the traffic entering loop $i \in S$, and subsystem B services the weighted proportional fair share λ_i^B above assurance a_i. Let q_i^B denote the fair portion of control loop i's contribution to the queue and let q_i^A denote the assured portion of control loop i's contribution to the queue. Let $d_i = rtt_i - basertt_i$. From Little's Law,

$$\begin{aligned} E[q_i] &= E[q_i^A] + E[q_i^B]\ , && (5.39) \\ E[q_i] &= E[\lambda_i] \cdot E[d_i], && (5.40) \\ E[q_i^A] &= E[\lambda_i^A] \cdot E[d_i] && (5.41) \end{aligned}$$

Combining (5.39), (5.40), and (5.41) yields

$$E[q_i^B] = E[q_i](1 - E[\lambda_i^A]/E[\lambda_i]) \tag{5.42}$$

Making a steady-state approximation and introducing Riviera's accumulation estimator $\hat{q}_i^R$, we estimate q_i^B as

$$\hat{q}_i^B = \hat{q}_i^R \cdot (1 - \nu_i^A/\nu_i) \tag{5.43}$$

Here $\nu_i^A = max(0, \nu_i - a_i)$, but due to the way we detect congestion we can simplify (5.43) to

$$\hat{q}_i^B = \hat{q}_i^R \cdot (1 - a_i/\nu_i) \tag{5.44}$$

Once $\hat{q}_i^B$ is known, we modify the congestion detection portion of the control policy to

$$\begin{array}{ll} \text{if } \hat{q}_i^B > w_{Bi} \cdot \epsilon_{max} \text{ or } \hat{q}_i^R > w_{Ai} \cdot \epsilon_{max} & \text{then congestion begins ,} \\ \text{if } \hat{q}_i^B \leq w_{Bi} \cdot \epsilon_{min} \text{ and } \hat{q}_i^R \leq w_{Ai} \cdot \epsilon_{min} & \text{then congestion ends .} \end{array}$$

Therefore, congestion begins whenever the loop's accumulation estimate exceeds either the maximum threshold for the assurance plus weighted proportional fair share or the maximum threshold for the proportional fair share assigned during oversubscription.

Now we must design the increase/decrease portion of the control policy. One possibility is to back off only the portion of the ingress departure rate that exceeds the assurance, a_i:

$$\lambda_i = \begin{cases} a_i + \beta(\nu_i - a_i) & \text{if } \nu_i > a_i \\ \nu_i & \text{otherwise} \end{cases}$$

If an EC trunk is below its assurance, it backs off just enough to keep the queue from growing further, but does not backoff enough to drain its share of the bottleneck. This forces others to backoff longer to alleviate congestion and allows each EC trunk to come up its assurance more quickly. Unfortunately if oversubscription occurs then the EC trunks the queue potentially grows unstable. One possible solution is for each EC trunk to still backoff when below its respective assurance but to backoff less. For this we define two backoff parameters, β^B is used when above the assurance and thus competing for a fair share of any best-effort capacity. The other backoff β^A is used when below the assurance. The parameters obey $1 > \beta^A >> \beta^B >> 0$.

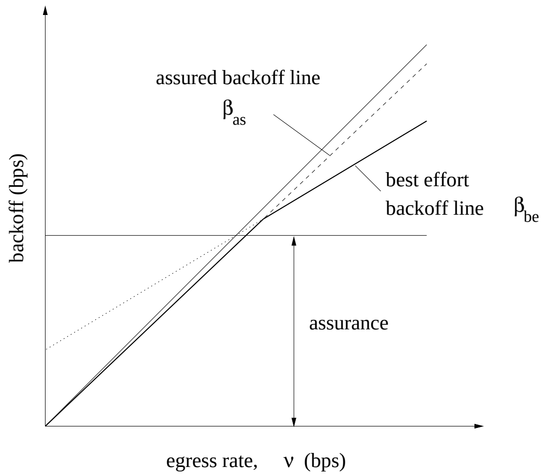
$$r_i = \mathsf{min}(a_i + \beta^B(\nu_i - a_i), \beta^A \nu_i) \tag{5.45}$$

In the equation above, β^B stands for the "best-effort" backoff. This is the fraction used when the EC trunk is receiving more than its assured allocation of the capacity. β^A stands for the "assured" backoff. It is used when the EC trunk receives capacity below its assured allocation (see Figure 5.9). Here,

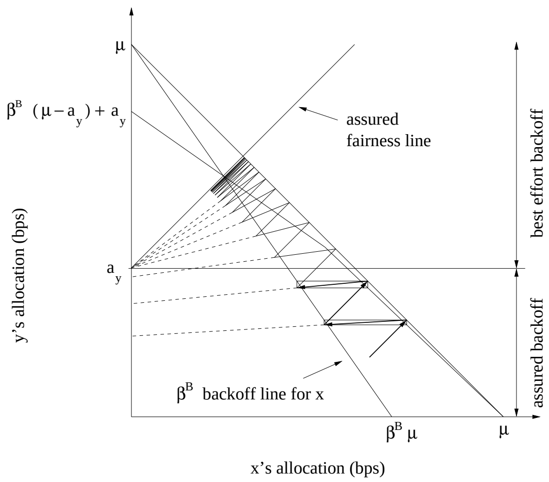
$$w^A = \frac{1}{1 - \beta_i^A} \ . \tag{5.46}$$

Equation 5.46 derives from the realization that a stability point exists where the size of each loop's decrease step equals its increase step assuming synchronous increases and decreases and equal RTPD. When all loops increase at the same rate this implies that all loops have equal increase steps as well as equal decrease steps. i.e.,

$$\nu_1(1-\beta_1) = \nu_2(1-\beta_2) = \cdots = \nu_n(1-\beta_n) \ . \tag{5.47}$$



(a) Single EC trunk



(b) Allocation Graph

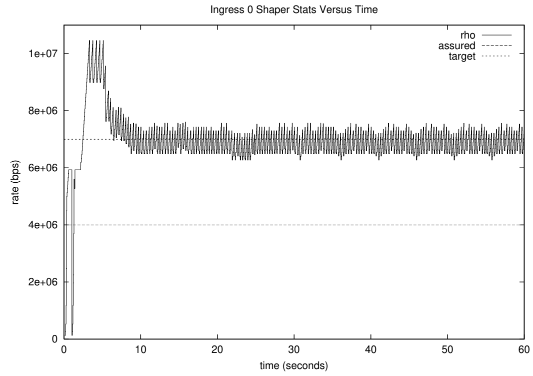
Figure 5.9: Assured Capacity Allocation Uses Tighter Backoff When below Assurance

In Figure 5.9 we use an allocation graph akin to those devised by Chiu and Jain [23]. However, they allow us to visualize rate allocations for two control loops competing over a single bottleneck.

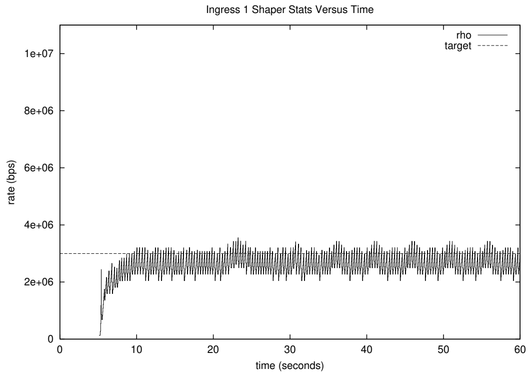
In Figure 5.10 we show results from a simulation using the single bottleneck configuration shown in Figure A.2 except the bottleneck is 10Mbps and the TCP connection has 4Mbps of assured bandwidth. Notice that the TCP connection obtains its 4Mbps assurance plus 1/2 of the remaining 6Mbps for a total of 7Mbps.

In Figure 5.11, we demonstrate the dynamic range of the assured building block by varying the assurance from 0 to 99% of the capacity. In Figure 5.12 we demonstrate that the assured building block approximately realizes U^a meaning each loop obtains its assurance plus a weighted proportional fair share of the remaining capacity.

We demonstrate the assured service operating in oversubscription in Figure **??** with two control loops over the single bottleneck topology in Figure A.2. One control loop has an assurance a which we vary in subsequent simulations. When the accumulation exceeds $w^A \cdot \epsilon_{max}$, congestion is declared regardless of the size of the assurance. Therefore once assurance a is high enough to cause oversubscription, the same bandwidth allocation arises



(a) TCP w/ 4Mbps Assured + 3Mbps Best-Effort



(b) UDP w/ 3Mbps Best-Effort

Figure 5.10: Assured Bandwidth Allocation for TCP versus UDP

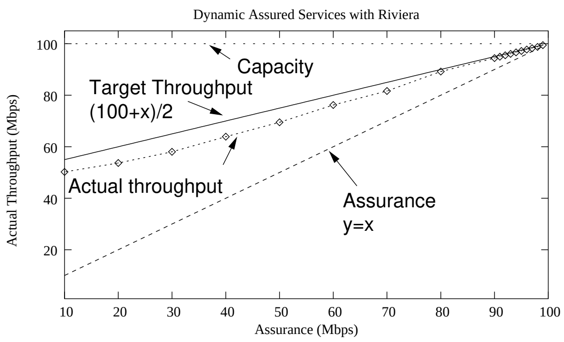
regardless of the value of a. In the following sections we describe a variety of strategies for managing oversubscription.

5.5 Smallest First Assured Bandwidth Allocation

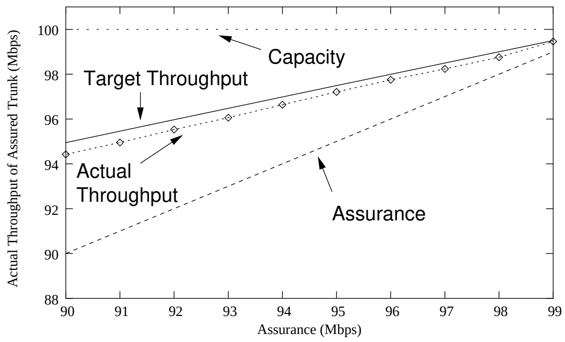
In the previous section we set all w_{Ai} equal. Therefore when oversubscription occurs all control loops push equally hard for their respective assurances and therefore as capacity increases, assurances are satisfied in increasing order of a_i. Therefore we call this the "Smallest First Assured Bandwidth Allocation." This bandwidth allocation also has the property that it satisfies the most assurances possible. This is demonstrated in the allocation graphs in Figure 5.14. In (a) the system starts with y satisfied, but because x is below its cusp it pushes with weight w^A which is heavier than y pushing with w^B. The system moves through the region where neither x or y is satisfied. In this region, both x and y push with weight w^B thus the system moves toward unweighted fairness (i.e., even shares for x and y). Before the system reaches unweighted fairness, it passes into the satisfied region for x. As a result, the system settles on the allocation where x obtains w^B weighted share above its assurance while y obtains w^A *absolute weighted share*, where *absolute weighted share* refers to a weighted share above 0.

5.6 Heaviest Weight First Assured Bandwidth Allocation

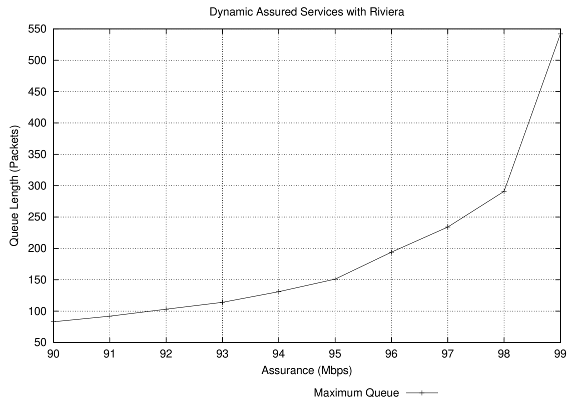
The assured capacity allocation building block proposed in previous sections satisfies the smallest assurance first because all EC trunks use the same w^A. There is no reason why this must be the case. We can use a different w_{Ai} for each EC trunk as determined by policy. Assurances are satisfied in order of decreasing w_{Ai}/a_i. This is demonstrated in



(a) Riviera approximates assurance



(b) Even large assurances between 90-99%



(c) Large Assurances Imply Large Queues

Figure 5.11: Assured Bandwidth Allocation over Single Bottleneck

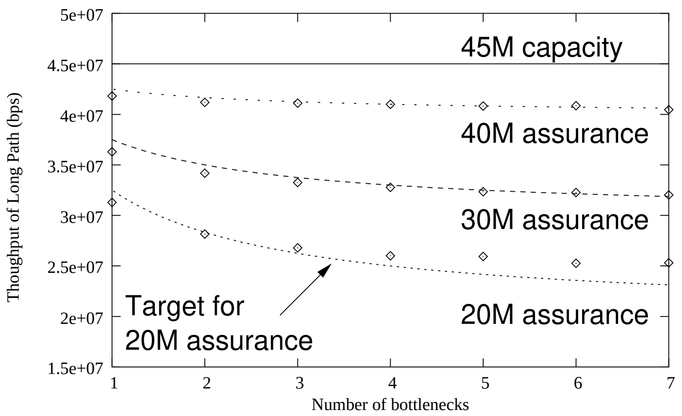


Figure 5.12: Assured Bandwidth Allocation over Multiple Bottlenecks

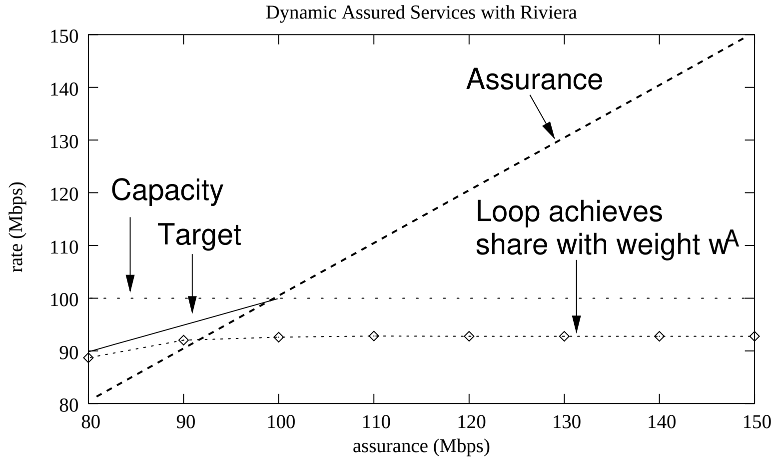
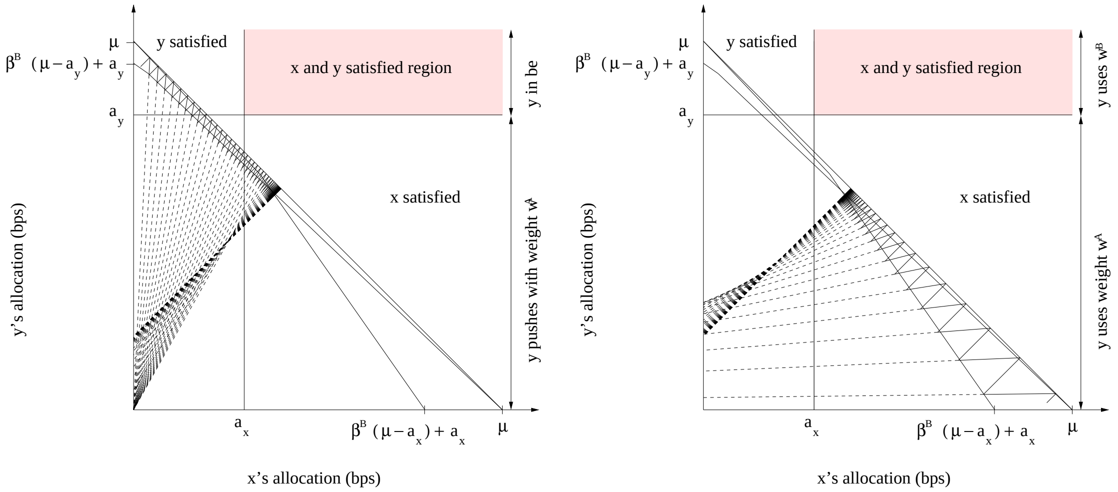


Figure 5.13: Assured Bandwidth Allocations during Oversubscription



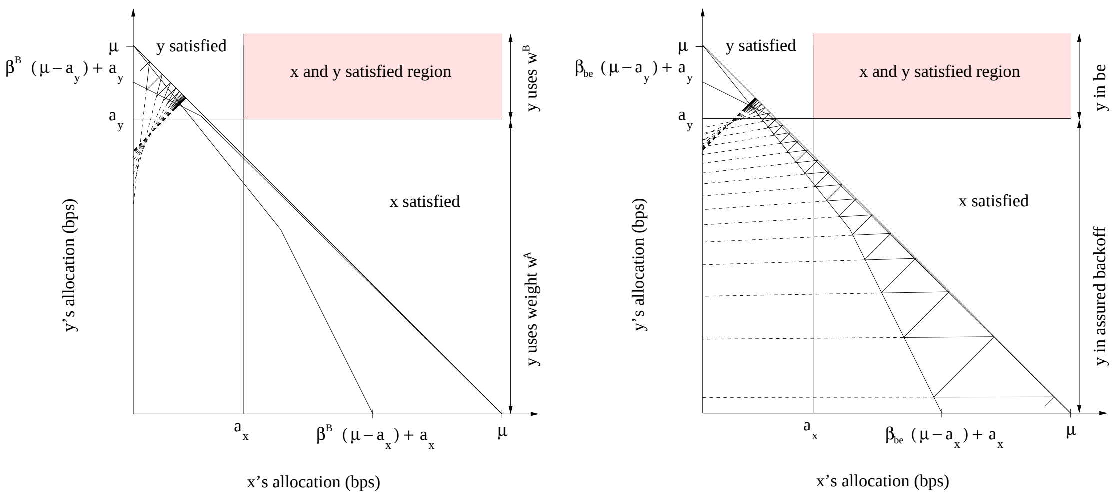
(a) Starting from $(.02\mu, .93\mu)$ (b) Starting from $(.93\mu, .02\mu)$

Figure 5.14: Smallest First Assured Bandwidth Allocation with Oversubscription $(a_x = .3\mu, a_y = .8\mu)$

Figure 5.15. Here y has a sufficiently heavy weight that it is satisfied over x even though y has a larger assurance.

5.6.1 All-or-none Assured Bandwidth Allocation

Set $w_{Ai} \propto a_i$. Thus either all assurances are satisfied or none. During oversubscription each control loop obtains a share proportional to the size of its assurance.



(a) Starting[a] from $(.02\mu, .93\mu)$ (b) Starting[a] from $(.93\mu, .02\mu)$

[a] $(a_x = .3, a_y = .8, \beta^B = 0.5, \beta_x^A = 0.8, \beta_y^A = 0.99)$

Figure 5.15: Heaviest Weight First Assured Service with Oversubscription

6. Conclusion

In this dissertation we described an architecture in which traffic control is used between the edges of a network to control traffic trunks passing between these edges in order to differentiate service, improve performance, and protect traffic from bandwidth-based Denial-of-Service attacks. We call this architecture *Edge-to-edge Control*. This architecture differs from prior architectures in that potential bottlenecks are expected to provide only a separate FIFO queue for all edge-controlled traffic. Instead all differentiation and performance enhancements are implemented at the edges. This placement of functionality represents a trade-off between reducing complexity in core nodes against looser control of delay, delay jitter, and fairness convergence times than possible with mechanisms placed directly at the bottleneck.

We then introduced a new congestion avoidance scheme for controlling a traffic trunk passing between an ingress and egress edge. The scheme differs from previous schemes in that it employs a novel backlog estimator coupled with a control policy that is less prone to *basertt* estimation errors.

By combining edge-based mechanisms with edge-to-edge control, we then demonstrated that it is possible to differentiate service between end-to-end flows within a traffic aggregate. Therein we presented the first scheme that achieves zero congestive loss for TCP 1) requiring only an isolated queue at the bottleneck, 2) provided there exists sufficient buffer at all queueing points, 3) without upgrading end-systems, and 4) without AQM or ECN at the bottlenecks.

Last we demonstrated that by modifying the control policy employed by the edge-to-edge control loop, we can cause these control loops to realize a variety of bandwidth sharing objectives. We then introduced new mechanisms that obtain optimal bandwidth allocations for previously unstudied utility functions useful for certain *inelastic* applications.

We recognize that our work only touches the surface. Future work will have to address outstanding issues such as performing a complete sensitivity analysis of all of the parameters in the system, better studying the operational region of the scheme, and extending the operational region to larger numbers of EC control loops without incurring queues that on average grow linearly with the number of EC control loops.

Together our advances lay the groundwork for developing new, scalable, implementable data plane service architectures for the Internet.

LITERATURE CITED

[1] Aggarwal, A., Savage, S., and Anderson, T. Understanding the performance of TCP pacing. In *Proceedings of INFOCOM 2000* (March 2000), vol. 3, pp. 1157–1165.

[2] Ahn, J., Danzig, P., Z.Liu, and L.Yan. Evaluation of tcp vegas: emulation and experiment. In *SIGCOMM Symposium on Communications Architectures and Protocols* (1995).

[3] Allman, M., Balakrishnan, H., and Floyd, S. Enhancing tcp's loss recovery using limited transmit. IETF RFC HERE, August 2000.

[4] Anjum, F., and Tassiulas, L. Fair bandwidth sharing among adaptive and non-adaptive flows in the Internet. In *Proceedings of IEEE INFOCOM '99* (March 1999).

[5] Athuraliya, S., Li, V. H., Low, S., and Yin, Q. REM: active queue management. *IEEE Network* (2001).

[6] Awduche, D. O., Malcolm, J., Agogbua, J., O'Dell, M., and McManus, J. Requirements for Traffic Engineering Over MPLS. *IETF Request for Comments 2702* (September 1999).

[7] Balakrishnan, H., Rahul, H., and Seshan, S. An integrated congestion management architecture for Internet hosts. In *Proceedings of ACM SIGCOMM* (August 1999), pp. 175–187.

[8] Balakrishnan, H., Seshan, S., Amir, E., and Katz, R. Improving TCP/IP performance over wireless networks. In *Proceedings of 1st ACM Conference on Mobile Communications and Networking (Mobicom)* (November 1995).

[9] Barford, P., and Crovella, M. Generating representative web workloads for network and server performance evaluation. In *Proceedings of ACM SIGMETRICS '98* (June 1998).

[10] Barford, P., and Crovella, M. A performance evaluation of hyper text transfer protocols. In *Proceedings of ACM SIGMETRICS '99* (March 1999).

[11] Bazaraa, M., Sherali, H., and Shetty, C. *Nonlinear Programming: Theory and Algorithms*, 2nd ed. ed. John Wiley & Sons, 1993.

[12] Bennett, J., Benson, K., Charny, A., Courtney, W., and Le Boudec, J.-Y. Delay jitter bounds and packet scale rate guarantee for expedited forwarding. In *INFOCOM 2001* (April 2001), vol. 3, pp. 1502–1509.

[13] Bennett, J., and Jardins, G. T. D. Comments on the July PRCA Rate Control Baseline. ATM Forum 94-0682, July 1994.

[14] Bennett, J. C. R., and Zhang, H. WF^2Q: Worst-case fair weighted fair queueing. In *Proceedings of IEEE INFOCOM '96* (San Francisco, 1996), vol. 1, pp. 120–128.

[15] Blake, S., Black, D., Carlson, M., Davies, E., Wang, Z., and Weiss, W. An architecture for differentiated services. IETF RFC 2474, December 1998.

[16] Bonomi, F., and Fendick, K. The rate-based flow control framework for the available bit rate ATM service. *IEEE Network Magazine* (March/April 1995), 25–39.

[17] Boutremans, C., and Le Boudec, J.-Y. A note on the fairness of TCP vegas. In *Broadband Communications* (2000).

[18] Braden, B., Clark, D., Crowcroft, J., Davie, B., Deering, S., Estrin, D., Floyd, S., Jacobson, V., Minshall, G., Partridge, C., Peterson, L., Ramakrishnan, K., Shenker, S., Wroclawski, J., and Zhang, L. Recommendations on queue management and congestion avoidance in the Internet. Internet RFC 2309, 1998.

[19] Brakmo, L., and Peterson, L. TCP Vegas: End to end congestion avoidance on a global Internet. *IEEE Journal on Selected Areas in Communication 13*, 8 (October 1995), 1465–1480.

[20] Cerf, V., and Kahn, R. A protocol for packet network intercommunication. *Transactions on Communications 22*, 5 (May 1974), 637–648.

[21] Chapman, A., and Kung, A. Traffic management for aggregate ip streams. In *In Proceedings of CCBR* (Ottawa, March 1999).

[22] Charny, A., Clark, D., and Jain, R. Congestion control with explicit rate indication. In *Proceedings of the IEEE International Communications Conference (ICC)* (June 1995), pp. 1954–1963.

[23] Chiu, D., and Jain, R. Analysis of the increase/decrease algorithms for congestion avoidance in computer networks. *Journal of Computer Networks 17*, 1 (June 1989), 1–14.

[24] Clark, D., and Fang, W. Explicit allocation of best-effort packet delivery service. *IEEE/ACM Transactions on Networking 6*, 4 (August 1998), 362–373.

[25] Clark, D., and Tennenhouse, D. Architectural consideration for a new generation of protocols. In *ACM SIGCOMM* (September 1990).

[26] Clark, D. D. The design philosophy of the DARPA Internet. In *ACM SIGCOMM '88* (1988).

[27] Clark, D. D., Lambert, M. L., and Zhang, L. NETBLT: A high throughput transport protocol. In *SIGCOMM Symposium on Communications Architectures and Protocols* (August 1987), pp. 353–359.

[28] Crowcroft, J., and Oechslin, P. Differentiated end-to-end Internet services using a weighted proportional fair sharing tcp. *Computer Communication Review 28*, 3 (July 1998).

[29] Cruz, R. L. *A Calculus for Network Delay and a Note on Topologies of Interconnection Networks.* PhD thesis, University of Illinois, July 1987.

[30] Davin, J., and Heybey, A. A simulation study of fair queueing and policy enforcement. *Computer Communication Review 20*, 5 (October 1990), 23–29.

[31] Deering, S., and Hinden, R. Internet protocol, version 6 (ipv6). Internet RFC 2460, December 1998.

[32] Demers, A., Keshav, S., and Shenker, S. Analysis and simulation of a fair queueing algorithm. *Internetworking: Research and Experience 1* (1990), 3–26.

[33] Demers, A., Keshav, S., and Shenker, S. Analysis and simulation of a fair queueing algorithm. *Internetworking: Research and Experience 1* (1990), 3–26.

[34] Endpoint congestion management working group. http://www.ietf.org/html.charters/ecm-charter.html.

[35] Egevang, K., and Francis, P. The ip network address translator (nat). IETF RFC 1631, May 1994. Covers NAT, but does not cover Port Address Translation (PAT).

[36] et al, M. H. Closed-Loop Rate-Based Traffic Management. ATM Forum 94-0211R3, April 1994.

[37] Feng, W., Kandlur, D., Saha, D., and Shin, K. Self-configuring RED gateway. In *Proceedings of IEEE INFOCOM '99* (March 1999).

[38] Floyd, S. TCP and Explicit Congestion Notification. *ACM Computer Communication Review 24*, 5 (October 1994), 10–23.

[39] Floyd, S. The newreno modification to tcp's fast recovery algorithm. http://www.aciri.org/floyd/papers/rfc2582.txt, April 1999.

[40] Floyd, S., and Fall, K. Promoting the use of end-to-end congestion control in the Internet. *IEEE/ACM Transactions on Networking* (February 1998).

[41] Floyd, S., and Jacobson, V. Connections with multiple congested gateways in packet-switched networks part 1: One-way traffic. *Computer Communications Review 21*, 5 (October 1991), 30–47.

[42] Floyd, S., and Jacobson, V. Traffic phase effects in packet-switched gateways. *ACM Computer Communication Review 21*, 2 (April 1991).

[43] Floyd, S., and Jacobson, V. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking 1*, 4 (August 1993), 397–413.

[44] Forum, A. http://www.atmforum.com.

[45] Fraser, A. G. Spider– an experimental data communications system. In *IEEE Conference on Communications* (June 1974).

[46] Fraser, A. G. Towards a universal data transport system. *IEEE Journal on Selected Areas in Communication 1*, 5 (November 1983).

[47] Georgiadis, L., Guérin, R., and Parekh, A. Optimal multiplexing on a single link: Delay and buffer requirements. *IEEE Transactions on Information Theory* (1997).

[48] Golestani, S. J. A self-clocked fair queueing shceme for broadband applications. In *In Proceedings of INFOCOM* (Toronto, Canada, June 1994), pp. 636–646.

[49] Goyal, P., Vin, H., and Chen, H. Start-time fair queueing: A scheduling algorithm for integrated services packet switching networks. In *Proceedings of ACM SIGCOMM '96* (Palo Alto, CA, 1996), pp. 167–168.

[50] Hasegawa, G., Murata, M., and Miyahara, H. Fairness and stability of congestion control mechanisms of tcp. In *INFOCOM '99* (March 1999), vol. 3, pp. 1329–1336.

[51] Hashem, E. Analysis of random drop for gateway congestion control. Tech. rep., Laboratory for Computer Science, MIT, 1989.

[52] Heinanen, J., Finland, T., Baker, F., Wiess, W., and Wroclawski, J. Assured forwarding phb group. Internet RFC 2597, June 1999.

[53] Hoe, J. Start-up dynamics of tcp's congestion control and avoidance schemes. Master's thesis, MIT, 1995.

[54] Ibanez, J., and Nichols, K. Preliminary simulation evaluation of an assured service. Internet Draft, August 1998. draft-ibanez-diffserv-assured-eval-00.txt.

[55] IETF. Integrated services in the Internet architecture: an overview. IETF RFC 1633, June 1994.

[56] Jacobson, V. Congestion avoidance and control. In *Proceedings of the SIGCOMM'88 Symposium* (August 1988), pp. 314–332.

[57] Jacobson, V., and Braden, R. TCP extensions for long-delay paths. IETF RFC 1072, October 1988. Discusses TCP SACK.

[58] Jacobson, V., and Nichols, K. An expedited forwarding phb. IETF RFC 2598, June 1999.

[59] Jacobson, V., Nichols, K., and Poduri, K. An expedited forwarding phb. IETF RFC 2598, June 1999.

[60] Jacobson, V., Nichols, K., and Poduri, K. The "virtual wire" behavior aggregate. Internet Draft, ftp://ftp.ee.lbl.gov/papers/vw_ba.pdf, March 2000.

[61] Jacobson, V., Nichols, K., and Poduri, K. The virtual wire per-domain behavior. ¡draft-ietf-diffserv-pdb-vw-00.txt¿, July 2000.

[62] Jain, R. A delay-based approach for congestion avoidance in interconnected heterogeneous computer networks. *Computer Communications Review 19*, 5 (1989), 56–71.

[63] Jain, R. Congestion control and traffic management in ATM networks: Recent advances and a survey. In *Invited submission to Computer Networks and ISDN Systems* (February 1995), vol. 28, pp. 1723–1738.

[64] Jain, R., Kalyanaraman, S., and Vishwanathan, R. The osu scheme for congestion avoidance in ATM networks using explicit rate inidication. In *Proceedings WATM '95 First Workshop on ATM Traffic Management* (December 1995).

[65] Kalampoukas, L., Varma, A., and Ramakrishnan, K. An efficient rate allocation alogirthm for ATM networks providing max-min fairness. In *Proceedings of the 6th IFIP International Conference on High Performance Networking* (September 1995).

[66] Kalyanaraman, S., Jain, R., Fahmy, S., Goyal, R., and Vandalore, B. The ERICA switch algorithm for ABR traffic management in ATM networks. *IEEE/ACM Transactions on Networking 8*, 1 (February 2000), 87–98.

[67] Karandikar, S., Kalyanaraman, S., Bagal, P., and Packer, B. TCP rate control. *Computer Communication Review 30*, 1 (2000).

[68] Kelly, F. Charging and rate control of elastic traffic. *European Transactions on Telecommunications* (1997).

[69] Kelly, F., Maulloo, A., and Tan, D. Rate control in communication networks: Shadow prices, proportional fairness and stability. *Journal of the Operational Research Society* (1998).

[70] Kent, S., and Atkinson, R. Security architecture for the Internet protocol. RFC 2401, November 1998.

[71] Keshav, S. A control-theoretic approach to flow control. In *Proceedings of ACM SIGCOMM '91* (September 1991).

[72] Keshav, S. On the efficient implementation of fair queueing. *Journal of Internetworking Research and Experience* (1991).

[73] Keshav, S., and Morgan, S. P. Smart: Performance with overload and random losses. In *IEEE INFOCOM '97* (April 1997).

[74] Kolarov, A. Study of the tcp/udp fairness issue for assured forwarding per-hop behavior in differentiated services networks. In *2001 IEEE Workshop on High Performance Switching and Routing* (May 2001), pp. 190–196.

[75] Kung, H., Blackwell, T., and Chapman, A. Credit-based flow control for ATM networks: Credit update protocol, adaptive credit allocation, and statistical multiplexing. In *Proceedings of ACM SIGCOMM '94* (September 1994), pp. 101–114.

[76] Kung, H., and Chapman, A. The flow-controlled virtual channels proposal for ATM networks. In *Proceedings of 1993 International Conference on Network Protocols* (1993), pp. 116–127.

[77] Kunniyur, S., and Srikant, R. End-to-end congestion control: Utility functions, random losses and ecn marks. In *Proc. INFOCOM 2000* (March 2000).

[78] Lakshman, T. V., and Madhow, U. The performance of TCP/IP for networks with high bandwidth-delay products and random loss. *IEEE/ACM Transactions on Networking 1*, 4 (October 1997).

[79] Lin, D., and Kung, H. T. TCP trunking: Design, implementation and performance. In *IEEE ICNP* (October 1999), pp. 222–231.

[80] Lin, D., and Morris, R. Dynamics of Random Early Detection. *SIGCOMM'97* (August 1997).

[81] Lo Monaco, G., Feroz, A., and Kalyanaraman, S. TCP friendly marking for scalable best-effort services on the Internet. To apear in Computer Communication Review, 2001.

[82] Low, S., and Lapsley, D. E. Optimization flow control, i: Basic algorithm and convergence. *IEEE/ACM Transactions on Networking 7*, 6 (December 1999), 861–875.

[83] Low, S. H., Peterson, L., and Wang, L. Understanding vegas: A duality model. In *Proceedings ACM Sigmetrics 2001* (June 2001).

[84] Massoulié, L., and Roberts, J. Bandwidth sharing: objectives and algorithms. In *IEEE INFOCOM '99* (March 1999).

[85] Mathis, M., and Mahdavi, J. Forward acknowledgement: Refining TCP congestion control. In *Proceedings of ACM SIGCOMM '96* (Palo Alto, CA, August 1996), pp. 281–291.

[86] Mathis, M., Mahdavi, J., Floyd, S., and Romanow, A. TCP selective acknowledgement options. Internet RFC 2018, October 1996.

[87] May, M., Bolot, J., Diot, C., and Lyles, B. Reasons not to deploy red. In *Proceedings of IEEE/IFIP IWQoS '99* (June 1999).

[88] Mishra, P. P., and Kanakia, H. R. A hop-by-hop rate-based congestion control scheme. In *Proceedings of ACM SIGCOMM '92* (August 1992).

[89] Mo, J., La, R. J., Anantharam, V., and Walrand, J. Analysis and comparison of TCP Reno and Vegas. In *Proceedings of IEEE INFOCOM '99* (March 1999).

[90] Mo, J., and Walrand, J. Fair end-to-end window-based congestion control. *IEEE/ACM Transactions on Networking* (2000).

[91] Morris, R. TCP behavior with many flows. In *Proceedings of ICNP '97* (October 1997).

[92] Nandagopal, T., lee, K.-W., Li, J.-R., and Bharghavan, V. Scalable service differentiation using purely end-to-end mechanisms: Features and limitations. In *IwQoS 2000* (June 2000).

[93] Nandagopal, T., Tae Eun, K., Sinha, P., and Bharghavan, V. Service differentiation through end-to-end rate control in low bandwidth wireless packet networks. In *IEEE International Workshop on Mobile Multimedia Communications '99* (November 1999).

[94] Nichols, K., Jacobson, V., and Zhang, L. A two-bit differentiated services architecture for the Internet. Internet Draft, ftp://ftp.ee.lbl.gov/papers/dsarch.pdf, November 1997.

[95] Ott, T. J., Lakshman, T. V., and Wong, L. H. SRED: Stabilized RED. In *Proceedings of IEEE INFOCOM '99* (March 1999).

[96] Parekh, A. K. *A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks.* PhD thesis, Massachusetts Institute of Technology, Cambridge, MA 02139, February 1992.

[97] Parekh, A. K., and Gallager, R. G. A generalized processor sharing approach to flow control in integrated services networks–the multiple node case. *IEEE/ACM Transactions on Networking* (April 1994), 137–150.

[98] Ramakrishnan, K. K., and Floyd, S. A proposal to add Explicit Congestion Notification (ECN) to IP. Internet RFC 2481, January 1999.

[99] Ramakrishnan, K. K., and Jain, R. A binary feedback scheme for congestion avoidance in computer networks with a connectionless network layer. In *Proceedings of ACM SIGCOMM '88* (August 1988), pp. 303–313.

[100] Ramakrishnan, K. K., and Jain, R. A binary feedback scheme for congestion avoidance in computer networks. *ACM Transactions on Computer Systems 8*, 2 (1990), 158–181.

[101] Ramakrishnan, K. K., Jain, R., and Chiu, D.-M. Congestion avoidance in computer networks with a connectionless network layer. part iv: A selective binary feedback scheme for general topologies methodology. Tech. Rep. DEC-TR-510, Digital Equipment Corporation, 1987.

[102] Sahu, S., Nain, P., Towsley, D., Diot, C., and Firoiu, V. On achievable service differentiation with token bucket marking for tcp. In *ACM SIGMETRICS '00* (June 2000).

[103] Seddigh, N., Nandy, B., and Pieda, P. Bandwidth assurance issues for tcp flows in a differentiated services network. In *IEEE GLOBECOM '99* (December 1999), pp. 1792–1798.

[104] Shenker, S. Fundmental design issues for the future internet. *IEEE Journal on Selected Areas in Communications* (1995), 1176–1188.

[105] Shenker, S., and Wroclawski, J. General characerization parameteres for integrated service network elements. IETF RFC 2212, September 1997.

[106] Shreedhar, M., and Varghese, G. Efficient fair queueing using deficit round robin. In *Proceedings of ACM SIGCOMM* (September 1995).

[107] Siu, K., and Tzeng, T. Intelligent congestion control for ABR service in ATM networks. *ACM Computer Communication Review 24*, 5 (October 1995), 81–106.

[108] Stevens, W. R. TCP slow start, congestion avoidance, fast retransmit and fast recovery algorithms. *Internet RFC 2001* (January 1997).

[109] Stoica, I., Shenker, S., and Zhang, H. Core-stateless fair queueing: Achieving approximately fair bandwidth allocations in high speed networks. In *Proceedings of the ACM SIGCOMM '98* (1998), pp. 118–130.

[110] Tsang, D., and Wong, W. A new rate-based switch algorithm for ABR traffic to achieve max-min fairness with analytical approximation and delay adjustment. In *Proceedings of IEEE INFOCOM '96* (March 1996), pp. 1174–1181.

[111] Turner, J. New directions in communications, or which way to the information age? *IEEE Communications Magazine* (1986).

[112] Villamizar, C., and Song, C. High performance TCP in ANSNET. *ACM Computer Communication Review* (1995).

[113] Visweswaraiah, V., and Heidemann, J. Improving restart of idle TCP connections. Tech. Rep. Technical Report TR97-661, University of Southern California, November 1997.

[114] Vojnović, M., Le Boudec, J.-Y., and Boutermans, C. Global fairness of additive-increase and multiplicative-decrease with heterogeneous round-trip times. In *IEEE INFOCOM 2000* (March 2000), pp. 1303–1312.

[115] Wang, S. Y. *Decoupling Control from Data for TCP Congestion Control.* PhD thesis, Harvard University, 1999.

[116] Wang, Z., and Crowcroft, J. A new congestion control scheme: Slow start and search (tri-S). *ACM Computer Communication Review, SIGCOMM 21*, 1 (1991), 32–43.

[117] Zhang, H., and Ferrari, D. Rate-controlled static priority queueing. In *Proceedings of IEEE INFOCOM '93* (San Francisco, 1993).

[118] Zhang, H., and Ferrari, D. Rate-controlled service disciplines. *Journal of High Speed Networking* (1994).

[119] Zhang, L. A new architecture for packet switching network protocols. Tech. Rep. MIT LCS TR-45, Laboratory for Computer Science, Massachusetts Institute of Technology, August 1989.

[120] Zhang, L., Deering, S., Estrin, D., Shenker, S., and Zappala, D. Rsvp: A new resource reservation protocol. *IEEE Network Magazine* (1993).

[121] Zhang, L., Shenker, S., and Clark, D. Observations on the dynamics of a congestion control algorithm. the effects of two-way traffic. *ACM Computer Communication Review* (September 1991).

APPENDIX A
Simulations

When studying Riviera's properties we rely on simulation to study the robustness to various pathologies and to support analyses using a controlled environment. When using simulation we use `ns-2.1b5`. With a few exceptions noted where appropriate, we use the parameters shown in Table A.1. We motivate this choice of parameter settings in the parameter sensitivity analysis.

Table A.1: Simulation Parameters

Parameter	Riviera
backoff, β	0.95
MTU, m	1000 bytes
max burst, σ	8000 bits
min threshold, ϵ_{min}	1000 bits
max threshold, ϵ_{max}	3000 bits
init BaseRTT	500ms
init rtt	500ms
rtt window size, k	30
time stamp window size	4

A.1 Simulation Models

In several simulations we use web traffic. For these simulations we use the models developed by Barford and Crovella in [9, 10]. In topologies subjected to web traffic, we always show traffic flowing from servers-to-clients, the direction taken by most web traffic.

When an EC trunk experiences "persistent demand" this implies that the trunk always sends at its rate limit as occurs when the ingress always has backlog. When TCP traffic provides "persistent demand" this implies that the connection transfers an infinitely long file. When UDP traffic provides "persistent demand" this implies that the UDP source sends faster than the aggregate bottleneck capacity, i.e., UDP floods the bottleneck(s).

Unless stated otherwise, when an EC trunk is fed with TCP, the ingress employs TCP rate control (see Section 2.8.1) to upper bound the receiver advertised window of each connection so that the TCP aggregate rate follows the EC trunk's rate limit. Our TCP rate controller employs ERICA [66] to allocate rates among the connections. We modify ERICA in four ways: 1) ERICA's "link capacity" input is the EC trunk rate limit, 2) the interval between rate computations is the time between EC trunk rate limit changes, 3) our version of ERICA provides *weighted* max-min fair sharing of the EC trunk's capacity, whereas the original ERICA supports only max-min fairness, and 4) our version modifies ERICA's queue control function to use RTT estimates. Per-connection RTT estimates are needed anyway for the rate-to-window conversion and such information was assumed unavailable to the original ERICA algorithm. Our modifications to the ERICA switch algorithm and associated parameter settings are detailed in Appendix D.

A.2 Steady-State over CoS

To understand the steady-state behavior of our schemes over a CoS network, we simulate EC and BE running over the heterogeneous delay topology given in Figure A.1. The EC class uses Riviera and TCP rate control (Riviera+TCPR). Each class is subjected to the same number of connections, 500, so that we can reasonably compare the performance obtained in each class. All TCP connections provide persistent demand since fairness is more difficult to define under transient conditions. Further if we were to use more realistic traffic (bursty, short transfers), BE and EC would be able to use the unused capacity from the other class, which is a highly desirable trait, but such sharing makes it difficult to compare performance with and without CoS. We study transient conditions and larger topologies in later sections. Each connection sends with a maximum packet size uniformly distributed in $[40, 1000]$ bytes[9]. For Reno, in our TCP model only packets in the SYN and FIN exchange plus ACKs are less than the loop's maximum packet size. Since we are interested in steady-state performance, the SYN and FIN exchange are irrelevant. The classes are serviced by a Deficit Round Robin (DRR[106]) scheduler with 500 MTUs (slightly less than 500K) of buffer shared between the two classes. Because DRR drops packets from the class with the most buffered packets and because TCP grows queues until loss, we expect most if not all drops to occur on the best-effort class. The fair scheduler attempts to provide 50% of the bandwidth to each traffic class. Source-to-bottleneck link delays are evenly spaced in the range $[2, 50]$ ms. The links from the edges to the end-systems are zero ms emulating a LAN as would occur in a typical access-to-access deployment of EC.

To verify that our statistics represent the steady state, we looked at graphs of queue

[9]Those familiar with ns know that ns provides more than one implementation of TCP. The earliest implementation operates in units of packets. The "full" TCP implementation operates in units of bytes. We use the "full" TCP implementation.

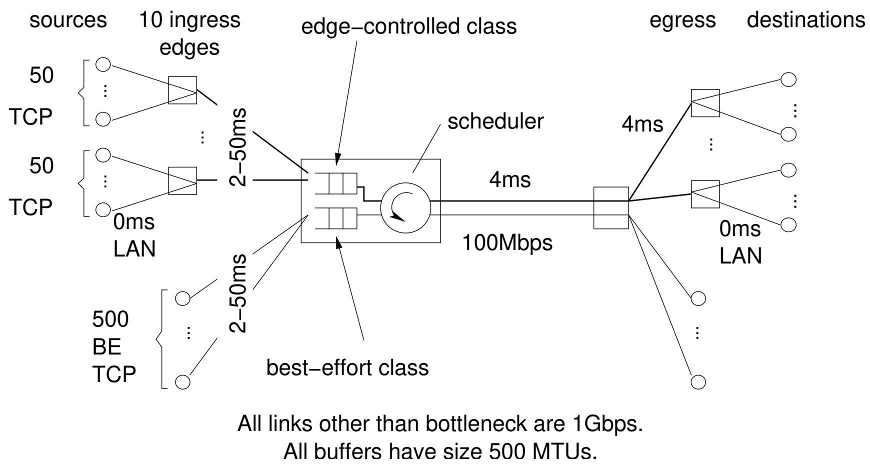


Figure A.1: Class of Service Topology

Table A.2: Class of Service Steady-State Results

Class	EC	BE	EC-only[a]	BE-only[a]
Throughput (Mbps)	49.2	50.8	49.3	50.0
Ingress Queue (avg/max MTUs)	16.5/75.5	n/a	16.6/74	n/a
Bneck Queue (avg/max MTUs)	4.53/23.1	454./500.[b]	3.99/22.0	477./500.
Excess Delay[c] (ms)	18.0	63.5	17.6	70.6
Retransmissions[d]	0	23034	0	21808
RTO[e]	0	5039	0	5234
C.O.V. Goodput[f]	0.635	1.082	0.632	0.915

Statistics are given for the last 20 seconds of a 30 second simulation in order to allow the system to converge on steady-state.
[a] 50Mbps bottleneck with only one class.
[b] Maximum queue is equal to the buffer size.
[c] End-to-end delay excluding propagation and transmission delays.
[d] Total TCP retransmissions within class.
[e] Total TCP retransmission timeouts within class.
[f] Coefficient of Variation (C.O.V.) in goodput. Goodput excludes retransmission, and TCP and IP headers.

length versus time plus EC rate limit versus time for several EC edges. The queue length began steady-steady oscillation and those EC rate limits looked at all stabilize well within the first ten seconds of the simulation. We therefore collected results over the last 20 seconds of 30 second simulations. All control loops begin sending at time zero.

As shown in Table A.2, the EC and Best-Effort (BE) classes obtain an almost even share of the bottleneck. In this scenario, EC outperforms Best-Effort (BE) in every metric including most markedly EC's elimination of steady-state loss and reduction of the excess end-to-end delay . The last two columns of Table A.2 demonstrate that EC and BE do not behave appreciably differently when provided a bottleneck unto themselves with half the capacity. This lends some support to the claim that BE and EC have been successfully isolated from each other in this scenario.

As a last few comments, we choose ten EC trunks arbitrarily. With fewer ingress edges and proportionally more connections per edge, EC obtains even better results, at the expense of increasing the load on the more stateful TCP rate controllers. We can also reduce the C.O.V. goodput for both classes by removing the delay heterogeneity, but that reduces the already tenuous realism provided by any simulation. We discuss the effects of delay heterogeneity in Appendix A.5. We could also bring down the delay experienced by the BE class by introducing bottleneck buffer management (see Section 2.6) for the BE class, but buffer management increases the complexity of the bottleneck and provides an unfair advantage to BE unless we also introduce comparable buffer management for EC.

A.3 Single Bottleneck

In support of the steady state queue length bound analysis (see Section 3.3.5), and slow-start queue length bound analyses (see Section 3.3.4), we provide a simple simulation using the topology in shown in Figure A.2 with no outside sources of variance so that the steady-state is well defined. All sources use persistent UDP with MTU-sized (1000 byte) packets. We recognize that such topologies and workloads lack realism but realistic scenarios are difficult to analyze.

We need to determine an appropriate run time and a time when we should start gathering steady-state statistics. We hereafter refer to the latter time as the "reset time," because statistics gathered before the "reset time" represent initial transients and statistics gathered after the "reset time" represent the steady-state. We determined appropriate run and reset times by running several simulations including the extrema (1 and 100 control loops). We then looked at queue length versus time plus the rate limit versus time for several EC edges. The simulations with more control loops tend to take somewhat longer to converge, but all converge in under 10 seconds; therefore, we choose a reset time of 15 seconds. We choose a run time of 60 seconds somewhat arbitrarily but certainly long enough to obtain reasonable averages.

Figure A.3 repeats several graphs that occur earlier in the dissertation (see Section 3.3) but it includes a new graph as well, utilization. Figure A.3(a) demonstrate that high utilization is achieved across the entire studied range of the number of control loops.

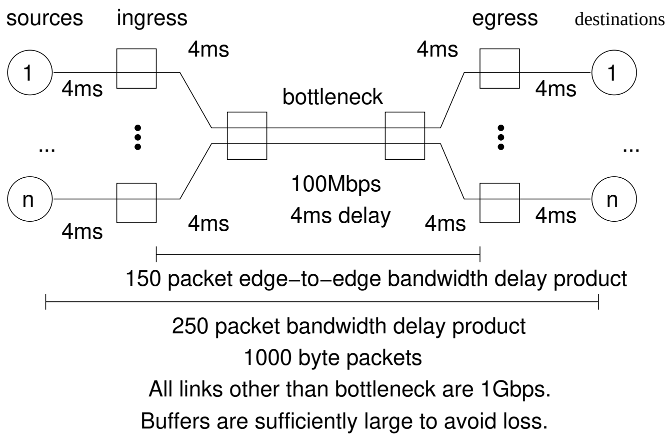
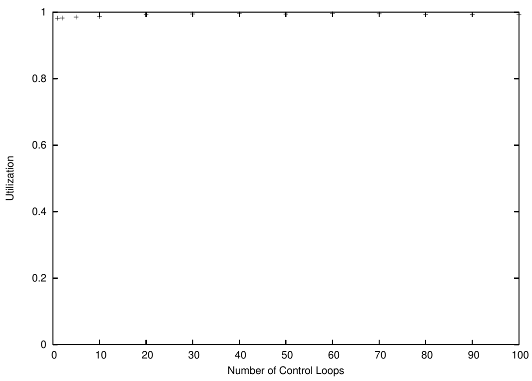
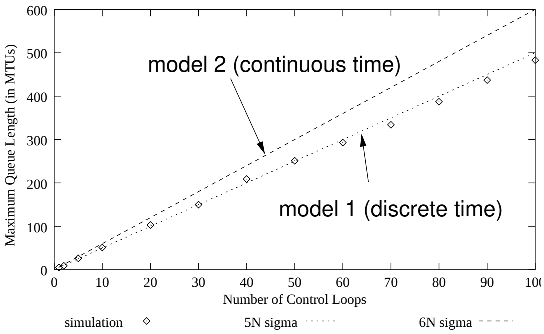


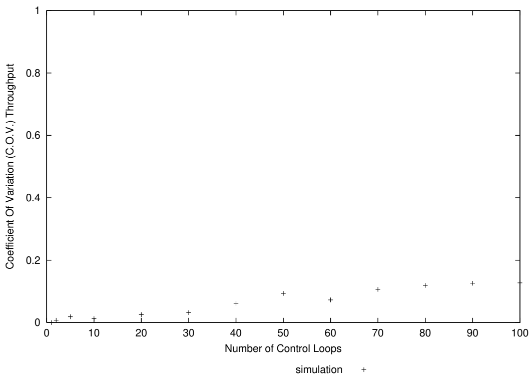
Figure A.2: Single Bottleneck Topology



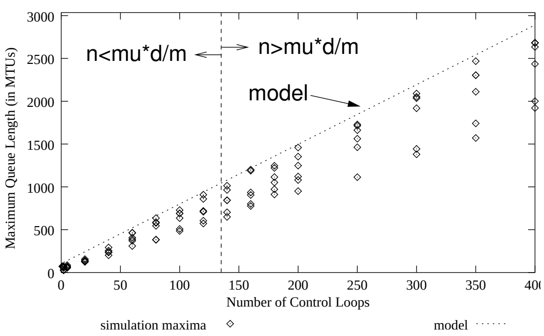
(a) Utilization[a]



(b) Steady State Max Queue Length[a]



(c) C.O.V. Throughput[a]



(d) Slow Start Max Queue Length

Figure A.3: Riviera Single Bottleneck Results

[a] Statistics gathered for last 45 seconds of 60 second simulations to allow the system to settle on the steady state.

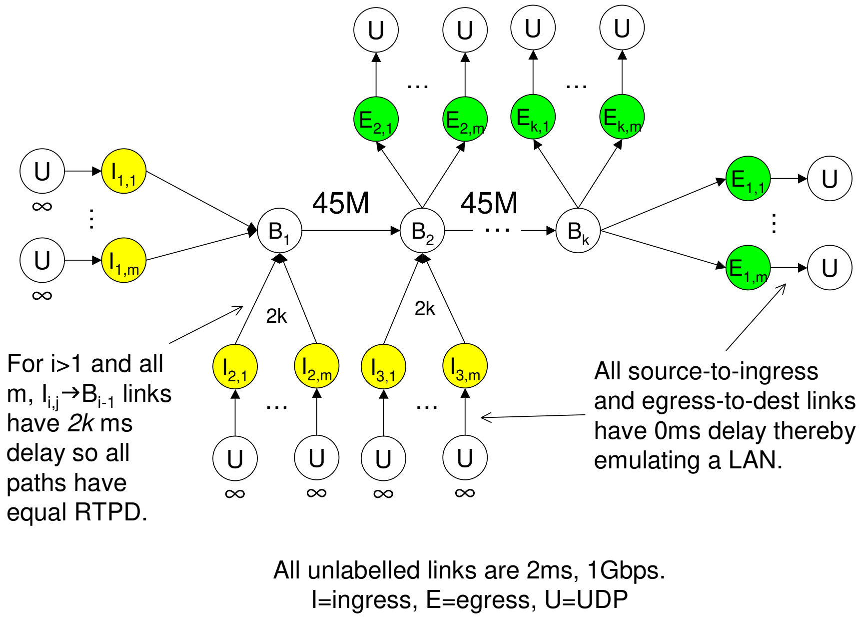


Figure A.4: Multiple Bottleneck Topology

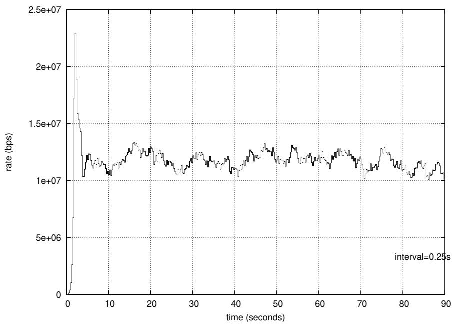
No loss occurs at the bottleneck, i.e, the maximum queue in Figures A.3(c), and A.3(d) reflect the maximum incurred before congestion is detected[10].

A.4 Steady-State over Multiple Bottlenecks

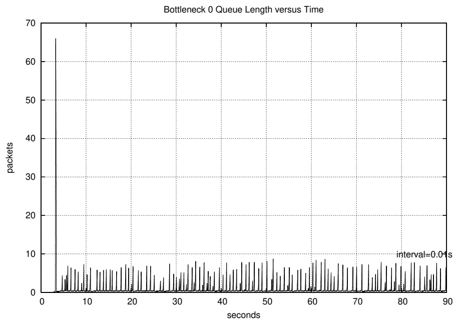
To study the steady-state fairness of Riviera when subjected to multiple bottlenecks, we use the topology shown in Figure A.4. A more realistic multi-bottleneck topology appears in Appendix A.6. To isolate effects we consider a topology wherein all paths have equal RTPD. We come back to the case with unequal RTPD in Appendix A.5. In designing this simulation we consider proper

- initial conditions,
- bottleneck capacity,
- buffer sizes,
- number of control loops per path,

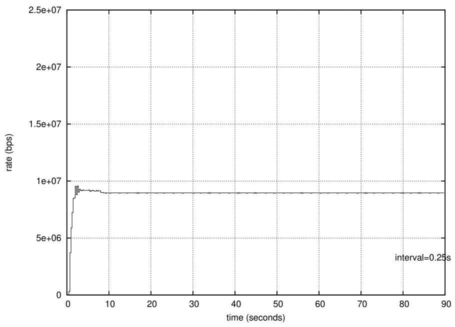
[10]Substantial loss would occur at the ingress edges since a UDP flood does not slow down regardless of congestion; however, our simulations use a variant of our shaper that generates packets as necessary in order to emulate an ingress shaper that always has a non-empty queue. This emulation has two effects: 1) the emulation ensures that the edge-to-edge control loop always experiences persistent demand regardless of rate limit, 2) simulations run much faster because extra UDP packets are not generated and then immediately discarded at the ingress edge when the ingress buffer overflows.



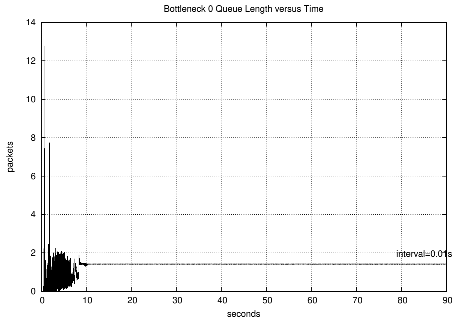
(a) Riviera Throughput[1]



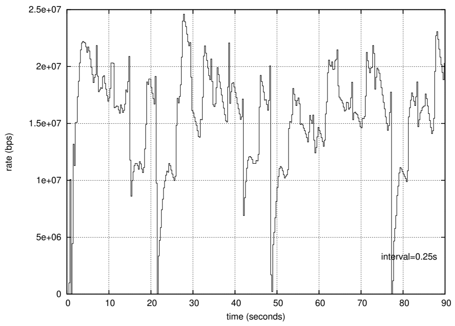
(b) Riviera Bneck (B_0, B_1) Queue



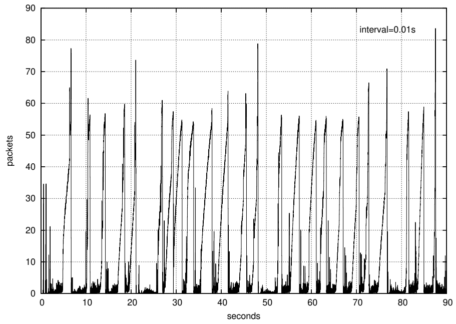
(c) Vegas Throughput[1]



(d) Vegas Bneck (B_0, B_1) Queue



(e) Reno Throughput[1]



(f) Reno Bneck (B_0, B_1) Queue

1 Throughputs are averaged over 0.25 second intervals and measured from the last bottleneck to the egress edge of the multi-bottleneck path.

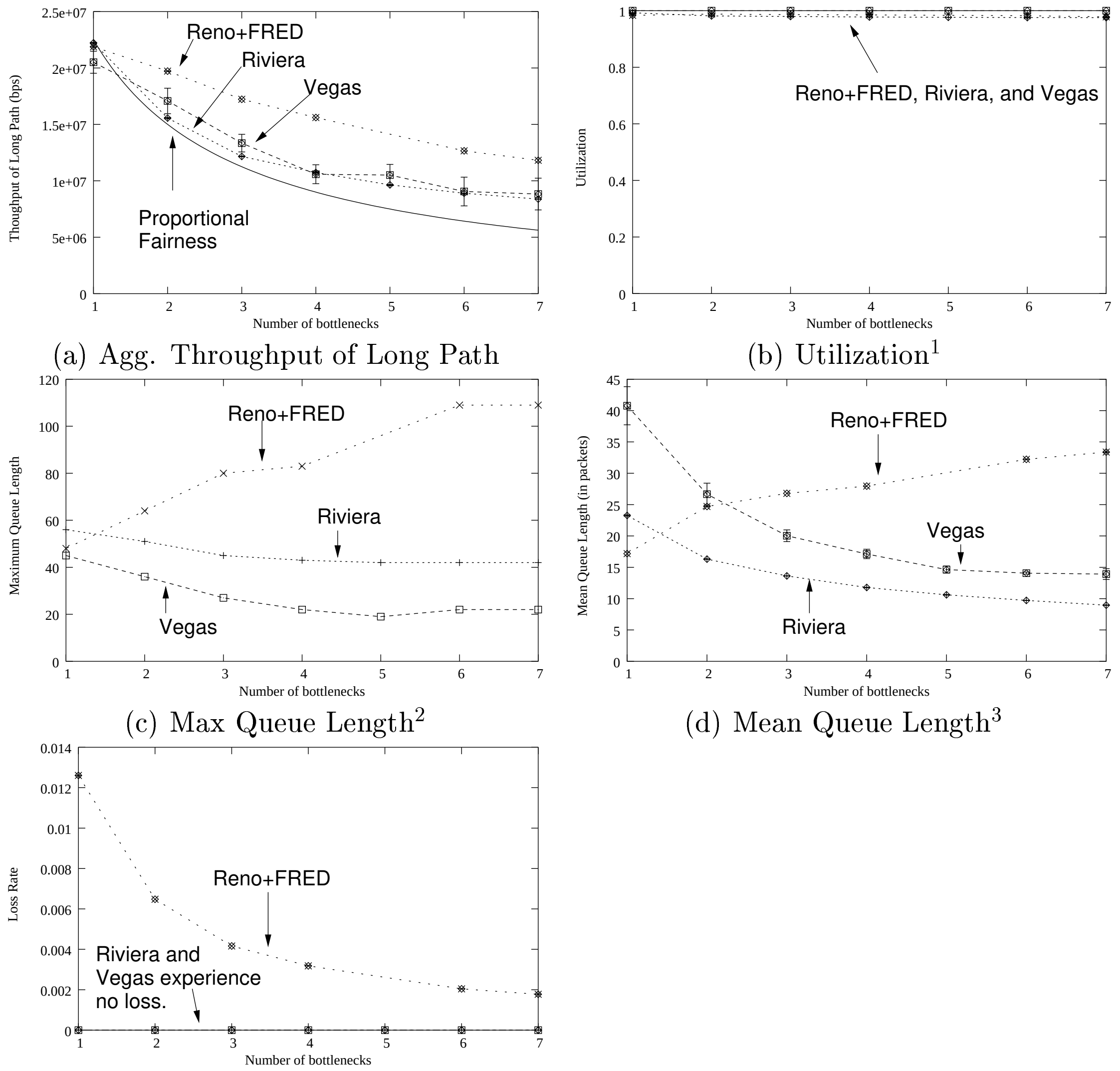
2 Queue lengths are averaged over 0.01 second intervals. Instantaneous queue lengths are well-defined, but the generated files are unwieldy in size.

Figure A.5: Case for k=8, m=1 with Reno, Vegas, and Riviera

- simulation run time,
- and time to begin gathering statistics to reflect steady-state.

These issues are not independent.

To randomize initial conditions, each source begins at a random time uniformly distributed in $[0, 2]$ seconds. Two seconds is many times the RTPD and more than sufficient for any given control loop to slow-start up to the capacity.



(a) Agg. Throughput of Long Path

(b) Utilization[1]

(c) Max Queue Length[2]

(d) Mean Queue Length[3]

(e) Loss Rate

[1] Mean of utilizations across all bottlenecks. [2] Max of maximum queue lengths across all bottlenecks. [3] Mean of mean queue lengths across all bottlenecks.

Figure A.6: Multiple Bottleneck Results for m=5

All bottlenecks have 45 Mbps (T3) capacity. We choose the same capacity for all bottlenecks to simplify computation of proportional fairness (see Section 2.2). The choice of 45Mbps is arbitrary.

All buffers have one bandwidth-RTPD product worth of buffer. Since RTPD increases as we increase the number of bottlenecks so does the buffer placed at each hop. The buffer size is irrelevant in the steady-state for Vegas and Riviera since they experience no steady-state loss in the multi-bottleneck steady-state simulations. For Reno, one bandwidth-RTPD product worth of buffer allows for all control loops to backoff by one-half and the buffers will just empty. Of course timeouts and burst loss still cause periods of underutilization. Therefore, the choice of buffer size is still somewhat arbitrary, but others[91, 112] have recommended one bandwidth-RTPD product buffer minimum as a rule-of-thumb.

We consider the case of $m = 5$. We motivate this choice momentarily. In each case we also study TCP Reno with a FIFO bottleneck, TCP Reno with a FRED bottleneck, and TCP Vegas. In each TCP scenario, we replace the UDP sources with the appropriate TCP sources and remove EC. Note that the end-to-end and edge-to-edge RTPD are equal in all cases.

Our objective is to determine the steady-state allocation of rates. Since all control loops along a given path should obtain equal rate according to any of the definitions of fairness that we have discussed (for definitions of max-min-fairness, proportional fairness, and min-potential-delay fairness see Section 2.2), we consider only the aggregate rate allocated to a given path and send the same number of control loops along each path.

To determine a proper run time and reset time for these simulations, we use the case with $k = 8$ (i.e., $k - 1$ bottlenecks). We ran the simulations for 90 seconds for Riviera, Reno, and Vegas to generate the graphs shown in Figure A.5. Riviera converges on around a rate allocation about 30 seconds into a 90 second simulation, whereas Reno appears to not converge within the time studied.

Now is appropriate to describe through simulation the different steady-state behaviors of Reno, Riviera, and Vegas. It is common in our simulations for Riviera and Vegas to converge on steady-state rate allocations far more rapidly than Reno. In the case where there are few connections involved and large bandwidth it's questionable as to whether we can say that Reno ever converges to a desired rate allocation. It is clear from Figure A.5 that it makes little sense to measure Reno fairness with $m = 1$ except on timescales far beyond demonstrated in the given plots, whereas Riviera and Vegas with even a single control loop in each path converges to a reasonably tight rate allocation. These figures mean only to illustrate the basic behaviors of Riviera, Vegas, and Reno and allow us to determine an appropriate simulation run time so that our statistics represent steady-state conditions. In Figure A.5, we learn only that we should use more Reno connections so that the queue oscillates more frequently thereby allowing Reno to move toward an equilibrium point. One should not immediately condemn Reno based on these results for they are based on highly idealized conditions.

Appendix A.6 presents more plausible conditions for comparison.

With $m = 5$ control loops on each path, Reno oscillates more quickly and with each oscillation Reno moves toward a rate allocation. Larger m exhibits even smoother Reno behavior, but with each increase m we must reduce the number of bottlenecks due to memory constraints. The throughput along the long path measured over the last half of each simulation does not change significantly between 60 and 90 second simulations. To introduce a safety factor, we ran simulations for 120 seconds and collected results only for the last 60 seconds of each simulation. Unfortunately with 5 control loops on each hop we were only able to run up to 7 bottlenecks before running into memory constraints.

We show the results with the appropriately set "reset time" in Figure A.6. We reran each simulation 4 times with different seeds and show the resulting mean and 99% confidence intervals. The randomization affects only the start times of the various control loops. In (a) we show the throughput of the control loop spanning multiple bottlenecks versus the number of bottlenecks that control loop traverses. Vegas and Riviera exhibit similar fairness, and both are reasonably close to proportional fairness at least for $m \leq 4$. We have not determined what causes the deviation at larger k. Although not shown here, simulations with smaller m ($m = 1$ and $m = 2$) reveal Reno's fairness dipping further with larger k. All this reveals is that Reno's aggregate behavior is heavily influenced by the number of control loops at least within the tested range of m, TCP Reno is not close to proportional fairness.

Figures A.6(c) and (d) provide somewhat counterintuitive results, the maximum and mean queue length actually dip as the number of bottlenecks increases. We expect that at least partially this is due to the distribution of the long control loop's accumulation across a multiple bottlenecks. Also Riviera's larger queue length than Vegas can at least partially be attributed to Riviera's queue oscillation.

Figure A.6(e) simply reflects the difference between loss-based and accumulation-based traffic control. FRED drops packets to force some reasonable degree of fairness. Riviera and Vegas do not incur loss when there exists sufficient buffer.

A.5 Heterogeneous Delay

The topology in Figure A.7 shows $n = 5$ control loops following a short path with RTPD equal to 8 ms and another group of $n = 5$ control loops following a long path with

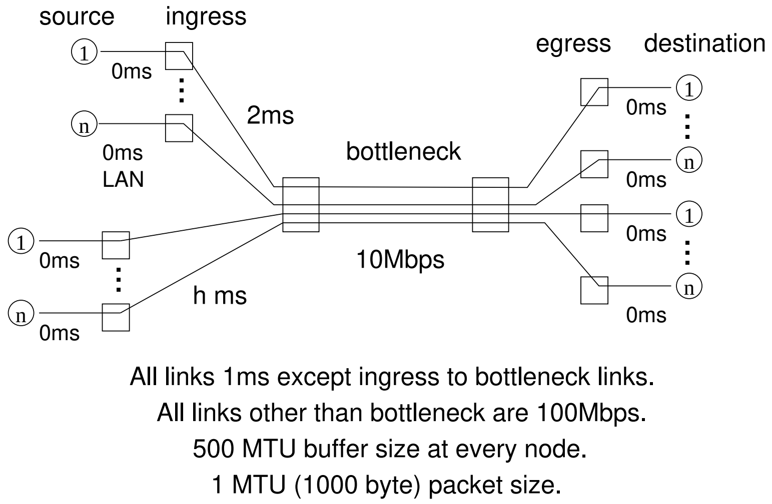
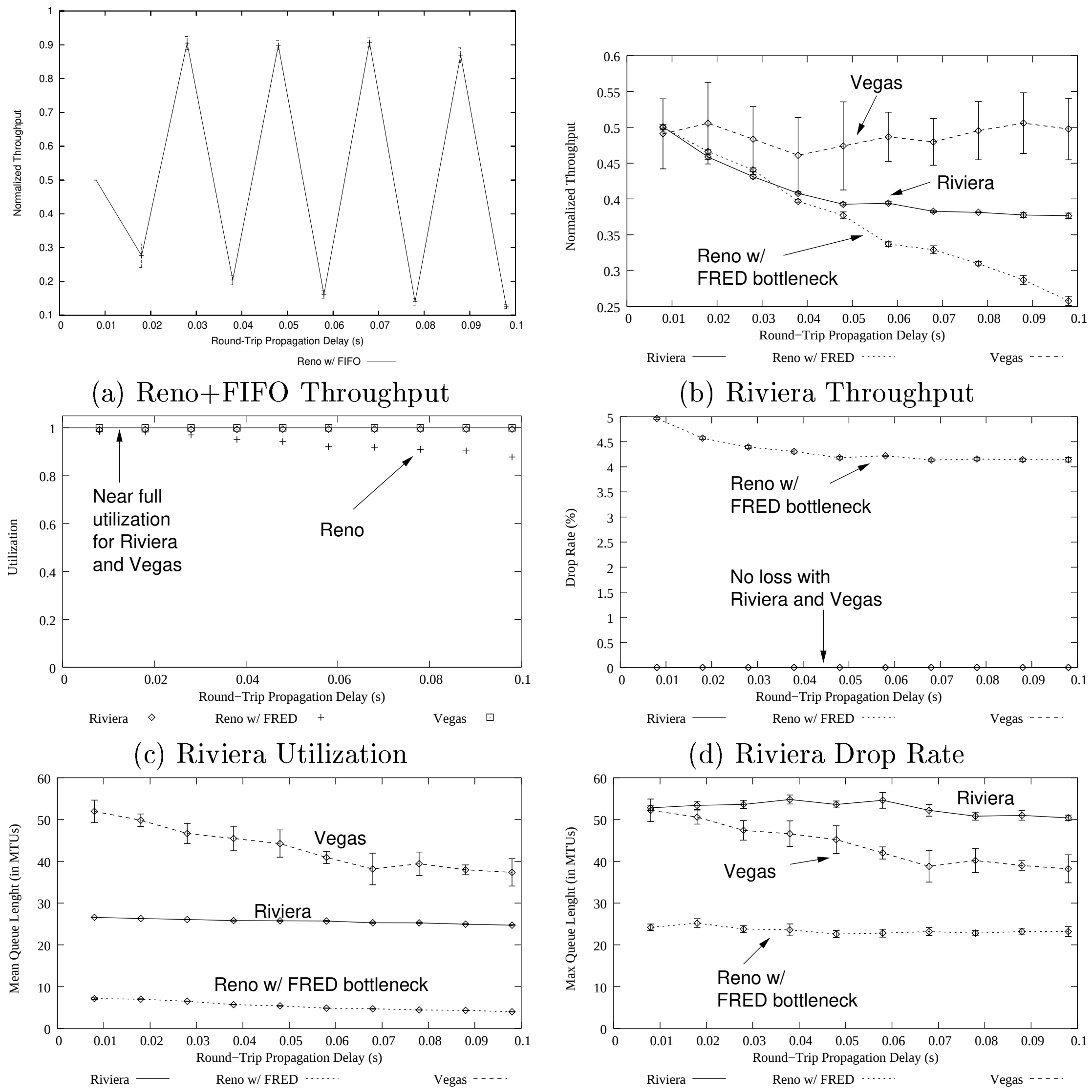


Figure A.7: Simulated Heterogeneous Delay Topology

RTPD equal to $2h + 4$ ms where h is the length of the ingress to bottleneck links. We choose to use multiple control loops on each path, because we experience low utilization when running a single connection across each path since TCP reacts drastically to loss. Aggregates of TCP connections are less responsive to loss and perhaps slightly more representative of typical TCP behavior. Each control loop starts at a random time uniformly distributed in $[0, 2]$ seconds. We discuss reset and run times momentarily.

In these simulations we compare congestion avoidance schemes in isolation from each other; therefore, the Reno simulation contains only Reno connections without EC. Likewise, the Vegas simulation contains only Vegas connections without EC, and Riviera contains only EC trunks. In each case the control loops are fed persistent demand. For Riviera this implies that the edges are flooded with UDP packets.

Normalized throughput denotes the ratio of the aggregate throughput of control loops following the long RTPD path to the total throughput of the bottleneck. In Figure A.8(a), we show the normalized throughput for Reno with a FIFO bottleneck versus the round-trip propagation delay of the long path. For the uninitiated, Reno's wide fluctuations in capacity look like an error or insufficient sampling of a high variance random variable. We ran each simulation 5 times for 500 seconds for each RTPD. Each simulation uses a different seed and thus changes the start times of the various connections. The points in Figure A.8(a) represent the means of these runs. The error bars represent the 99% confidence interval. The confidence intervals are so small as to be almost unnoticeable.



(a) Reno+FIFO Throughput (b) Riviera Throughput

(c) Riviera Utilization (d) Riviera Drop Rate

(d) Riviera Mean Queue Length (e) Riviera Max Queue Length

Figure A.8: Riviera's Performance over the Heterogeneous Delay Topology

Because the system repeatedly converges on the same throughput distributions regardless of initial conditions, we are reasonably assured that the results are either a real effect or a bug in the simulation and not due to insufficient sampling. Prior work by Sally Floyd and Van Jacobson [42] lends support to this being a real effect. Floyd and Jacobson attribute similar results to synchronization over the last packet worth of buffer (see Section 2.5.2). Because synchronization can dominate other effects, we simply omit further consideration of Reno with a FIFO bottleneck in our study of heterogeneous propagation delay. Because FRED randomly drops packets, FRED breaks synchronization effects, so we proceed with

Reno over FRED bottlenecks as a point of comparison to Riviera and Vegas. Neither Vegas or Riviera are affected by loss-based synchronization in the given regime because neither experiences loss; therefore, we keep the FIFO bottleneck for Vegas and Riviera simulations.

We initially ran several simulations for 60 then 100 seconds. Unfortunately the results for the longer path lengths differ significantly between 60 and 100 seconds except for Riviera. Aside, it would be interesting to contrive these experiments to measure convergence times, though we did not think of it at the time. Given our interest in steady-state throughput, we ran simulations for the largest RTPD for 100, 200, 300, 400, and then 500 seconds. We saw little change in normalized throughputs for a given seed for simulations running longer than 200 seconds, but to include a safety factor we ran all simulations for 400 seconds and collected statistics for only the last 50 seconds. As with the simulations for Reno with a FIFO bottleneck, for each scheme we ran each simulation five times for each RTPD and computed 99% confidence intervals.

Figure A.8(b) shows the mean normalized throughput when all control loops use Riviera, then Vegas, then Reno+FRED. Vegas distributes capacity relatively evenly regardless of the studied Ride's, though even after 400 seconds, the 99% confidence intervals are more than an order of magnitude wider for Vegas than for the other schemes. Riviera and Reno+FRED exhibit bias against the long path; however, Riviera does substantially better than Reno+FRED. Bias along longer paths is a side effect of our choice of control policies and is common to AIMD algorithms [114, 41, 78, 119].

Figure A.8(c) reveals that only Reno+FRED has difficulty fully utilizing the bottleneck under conditions of such delay. Initially we thought this is due to the increasing size of multiplicative decreases as bandwidth-RTPD product increases. For example, at 0.08 second long-path RTPD, the long-path aggregate obtains approximately 0.3×0.9 of the bottleneck capacity. Assuming this capacity is divided roughly evenly between the connections in the long-path aggregate, this comes to approximately 540kbps per connection. A back-off of one half due to loss results in a decrease in window size from 5.4 MTUs to 2.7 MTUs. A decrease of 2.7 MTUs is hardly sufficient to drain a queue with average queue length around 5. Our supposition could still be the cause for the decrease in utilization but only if multiple losses commonly occur in rapid succession. We do not currently consider this sufficiently interesting to devise further simulations to determine the exact cause of the low utilization.

Figure A.8(d) also shows a decrease in the drop rate, which could be due to the longer period of time it takes for the system to oscillate with longer control loops (i.e., drops occur only at the peak demand in the oscillation, but longer RTPD implies longer periods. Longer periods implies fewer peaks).

Figures A.8(e) and (f) reveal that Riviera and Reno+FRED queueing behavior is not sensitive to the length of the control loop. For Reno+FRED this might be expected since drops occur at a specific bottleneck queue length irrespective of end-to-end delay with a slight drop coinciding with the decreased utilization. Vegas's max and mean queue lengths are almost equal because Vegas stabilizes at a given queue length. Riviera exhibits a significant difference between max and mean queue lengths because Riviera oscillates to drain the queues.

One should not draw any strong conclusions from the queue length graphs regarding overall performance of these schemes in terms of delay or transfer times. Riviera experiences shaping and queuing delays at the ingress that also impact end-to-end delay. These combined effects may have greater impact on transfer times than bottleneck queue length.

A.6 Fairness under Variance

A general notion of fairness is difficult to define under conditions where steady state does not exist. However, in the special case when two control loops experience identical travails (i.e., demand, cross traffic processes, capacity, propagation delays), we can define fairness as each control loop achieving equal long-run throughput regardless of initial conditions. This definition lacks precision by leaving open the question of what constitutes sufficient duration to properly measure "long-run" throughput. We address this issue momentarily.

To test Riviera according to our notion of fairness, we use the topology shown in Figure A.9. This topology contains two "main" traffic trunks ($I_0 \rightarrow E_0, I_1 \rightarrow E_1$) with persistent UDP demand traveling the same path through the bottlenecks. The best-effort and edge-controlled traffic classes are separated at each bottleneck (B) into their own queues together serviced by an unweighted Deficit Round Robin (DRR[106]) scheduler. Thus the two classes obtain roughly even shares when each class is saturated. Each buffer in this topology holds up to 500 MTUs shared between the two classes. There are $k = 5$ web servers each servicing 25 users. The 25 users are distributed evenly across $k = 5$ client nodes. The web servers are connected to the first bottleneck by links with

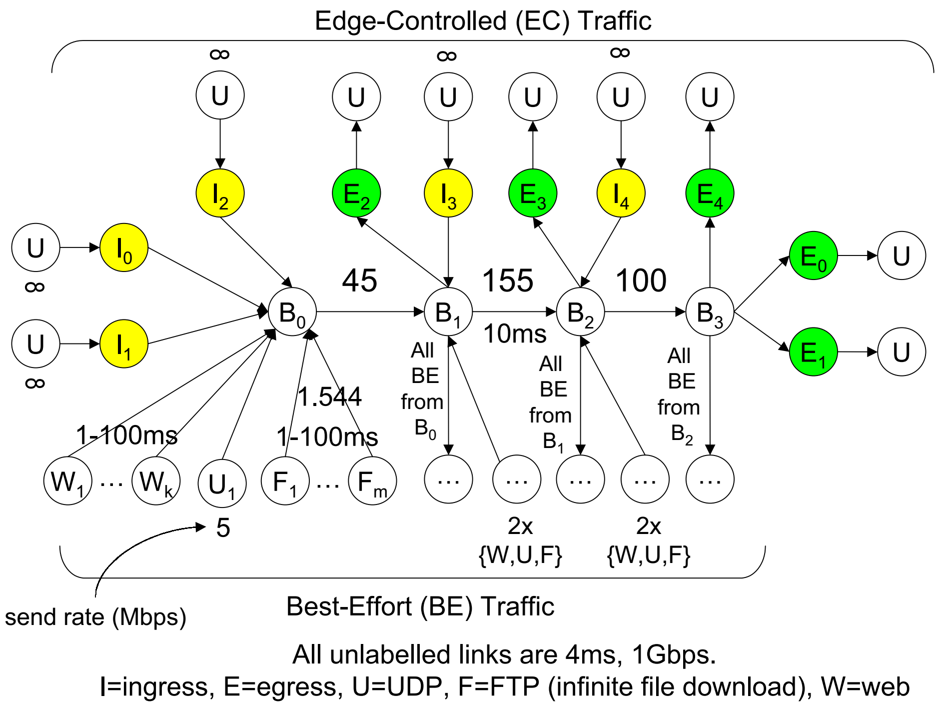


Figure A.9: Topology for Fairness under Variance Simulations

propagation delay evenly spaced over the range $[1, 100]$ ms. The FTP servers have the same propagation delays and distribution of users as the web servers and clients except $m = 8$. All sources whether EC, web, FTP, or UDP start at a random time distributed uniformly in $[0, 1]$ seconds. Prior to gathering results, we ran the simulation several times to tune the distribution of traffic such that best-effort web traffic occupies about 25% of the 45Mbps link's capacity (i.e., roughly 1/2 of the best-effort capacity). With an average of 25% web traffic and FTP traffic occupying only 1.544×8 Mbps, the remaining BE capacity lies idle unless used by the EC class. The burstiness of the web traffic ensures that the EC class sees capacity variability. To randomize initial conditions, all sources begin at random times uniformly distributed over the first second of the simulation.

For comparison we present results for the same topology but replace the EC trunks with TCP Reno, then TCP SACK, then TCP Vegas connections. To obtain comparability, the connections replacing EC trunks still pass through the EC traffic class though the EC traffic class no longer employs an edge-to-edge control loop.

Returning to the problem of determining what constitutes a sufficient interval to measure throughput, we ran the simulations several times with differing durations. After

Table A.3: Fairness under Variance Simulation Results

Scheme	Riviera	Reno	SACK	Vegas
Throughput 0 (Mbps)	6.14/.107[a]	5.19/2.02	4.56/2.42	5.76/0.25
Throughput 1 (Mbps)	6.11/.124	6.3/2.26	4.56/2.48	5.8/0.256
Utilization[b]	.844/.0173	.932/.00776	.992/.00264	.895/.00762
EC Avg. Q (MTUs)[c]	1.73/.1	197/38.5	255/9.45	4.5/0.342
BE Avg. Q (MTUs)[c]	43/16.9	42.4/10.3	47.4/18.8	45.7/11.7
EC Max Q (MTUs)	58.8/14.3	500[d]/0	500[d]/0	120/19.2
EC Drops	0/0	52.2/16.7	83/60.5	0/0
BE Drops	5526/1651	5084/1352	5512/1162	4957/1232

Simulation run time is 60s. We did not use a reset time since steady-state probably does not exist. "Drops" is the sum across all bottlenecks.
[a] All results given as "mean/standard deviation".
[b] Mean of utilizations across all three bottlenecks.
[c] Average of queue lengths and max of max queue lengths across all three bottlenecks. Each bottleneck is weighted equally in the averages.
[d] Reno and SACK drive the queue length to the point of loss, thus the maximum queue length obtained by RENO and SACK is 500 packets.

running simulations of 60 seconds, some of the schemes' throughput measurements still exhibited substantial variability. Unfortunately if we increase duration with a 155Mbps bottleneck, we risk overrunning the signed byte counters. Rather than re-implementing the simulator to handle larger numbers, we instead report "mean/standard deviation" of each metric generated from 10 runs with each scheme (see Table A.3).

Most striking is perhaps the difference in throughputs between the various schemes, but we put this issue aside because the topology and workload are too complex to determine the potential causes. Simulation results in Section A.5 hint at the potential reasons: Riviera exhibits some bias against control loops passing along longer RTPD paths and through multiple bottlenecks, though this bias is far less than exhibited by Reno+FRED. More importantly, as expected all of the schemes provide roughly even throughputs between the two EC trunks. However, both Reno as EC and SACK as EC exhibit standard deviation in throughputs exceeding Riviera's by approximately *twenty-fold.* Vegas's standard deviation approximately doubles Riviera. Unfortunately Riviera achieves lower utilization than any of the other schemes (84% versus Reno's 93%, SACK's 99%, and Vegas's 90%).

Furthermore these results demonstrate that Riviera and Vegas retain zero congestive loss under substantial noise provided there exists sufficient buffer, and Riviera, and Vegas all retain tight control on queue lengths. However, Vegas is not viable as an edge-to-edge control scheme because it is window-based and reacts poorly to increases in propagation delay (see Sections 3.2.4 and A.10).

A.7 Loss

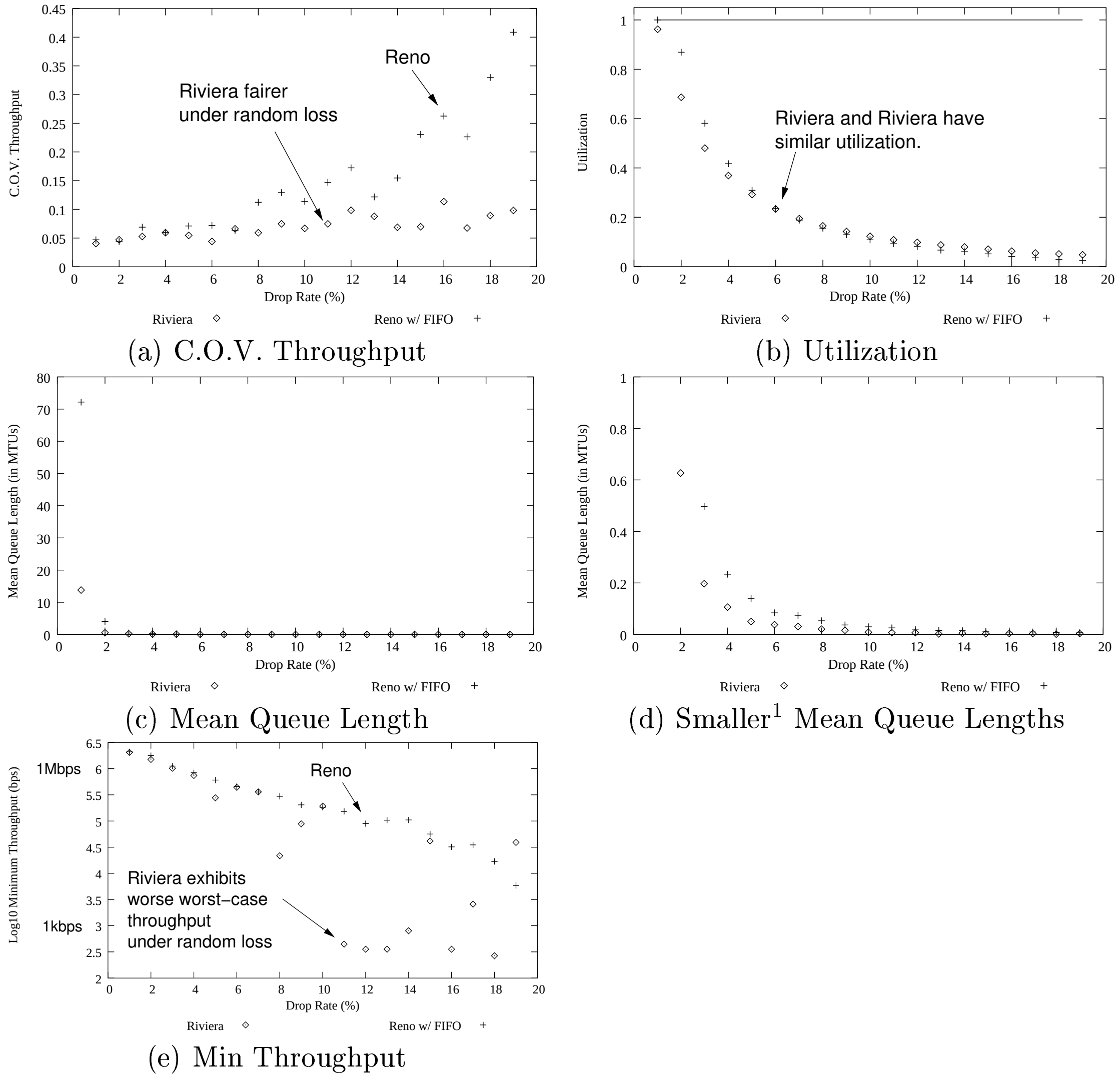
In this section, we consider the effects on Riviera of two cases that lead to loss: 1) random loss introduced by a noisy link and 2) insufficiently provisioned buffers.

Riviera reacts to loss by decreasing the rate limit and `rthresh` by one-half, even if this means reducing the rate limit below 1 MTU/RTPD. However, we do not allow the rate limit to reduce below 1 MTU/RTO where RTO refers to the time between timeouts. Since the RTO doubles following each subsequent control packet loss, lower bounding the rate limit by 1 MTU/RTO does not lead to queue instability or even cause the algorithm to behave unlike Reno. If the rate limit is allowed to dip below 1 MTU/RTO then the control loop may not send a control packet before the next timeout. Since the rate limit halves with each timeout, it can take excessively long before a control loop caught in this condition sends a control packet. In other words, if the rate limit is allowed to dip below 1 MTU/RTO, Riviera exhibits a bias against control loops with small rate limits. By removing this condition, we avoid one source of gross unfairness (Goal 2).

To isolate effects we consider only the simplest single bottleneck topology with homogeneous delay (see Figure A.2) with 45Mbps capacity, $n = 20$ control loops, and each link has 1 bandwidth-RTPD product of buffer. Our capacity choice is arbitrary. All control loops are fed by persistent demand and start at $t = 0$. We used analogous methods as used in Appendix A.4 to determine appropriate run time and reset time resulting in a choice of 120s and 60s respectively.

At the very least Figure A.10 demonstrates that Riviera handles loss. Goal 1 is to avoid congestion collapse, which we interpret as meaning the system should degrade toward Reno-like behavior under loss. For all of the given metrics except minimum (worst-case) throughput, Riviera and Reno follow similar behavior. Under heavy loss, Riviera appears marginally more aggressive than Riviera in terms of utilization, but this increase in utilization is not a problem because Riviera also obtains smaller mean queue length (see Figures A.10(c) and (d)). Lower mean queue length across the entire range implies that the queue is not being allowed to run away which would result in persistent loss and potentially congestion collapse.

Riviera's worst-case throughput is disappointing but tolerable in terms of Goal 2, because no starvation occurs across the studied range of loss rates.



Statistics gathered for last 90 seconds of 120 second simulation

[1] We mean "smaller" in the sense that the y-axis is appropriately scaled to differentiate the smaller mean queue lengths for larger drop rates.

Figure A.10: Riviera and Reno versus Random Loss

Over wireline networks we expect buffer overflow to account for most loss rather than random error (e.g., due to noise). So we now consider a topology in which we modify the buffer size for each subsequent simulation in order to explore the effects of insufficient buffer to avoid loss. We use the topology shown in Figure A.11 with $n = 60$ control loops. As with the CoS scenario (see Appendix A.2), the source-bneck links have length evenly spaced in the range $[2, 50]$ ms, each control loop begins at a random time uniformly distributed in $[0, 2]$ seconds, and each loop sends with a random maximum packet size

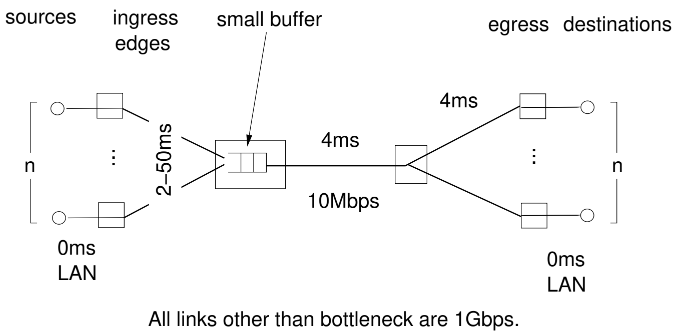


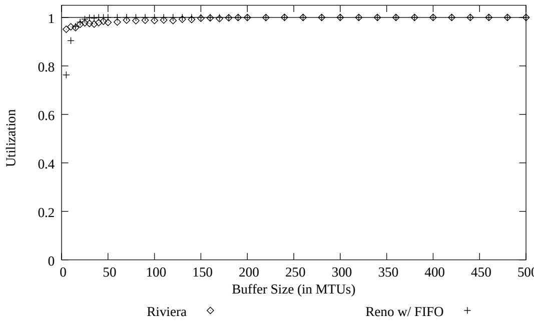
Figure A.11: Small Buffer Topology

uniformly distributed in $[40, 1000]$. All control loops experience persistent demand. We use a shorter run time for these simulations since they run operate at lower bandwidth, and heavy loss tends to push the system rapidly to a point of low utilization. Statistics gathered from a reset time of 10 seconds do not change significantly whether statistics are gathered for 20, 30, or 60 seconds so we use 10 and 30 second reset- and run times respectively.

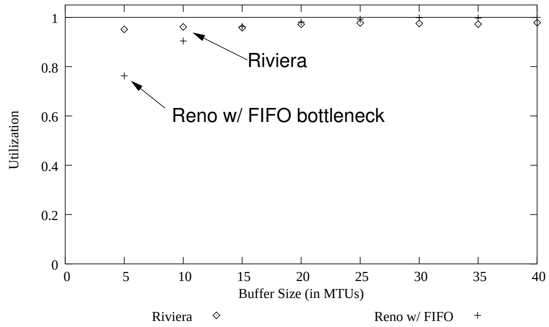
Figures A.12 show buffers sized from 5 to 500. The results are similar to the results for random loss, Riviera is somewhat more aggressive than Reno in terms of utilization and less with regard to queue length. Because the queue is kept under control and loss rates are below 5% even for buffers as small as 5 MTUs (< 5K), Riviera avoids persistent loss across all reasonable buffer sizes for the studied load. We see no violation of Goal 1.

Figure A.12(g) demonstrates that across the entire range studied, Riviera does not experience starvation (Goal 2). Although Reno has lower C.O.V. throughput when there are less than about 30 MTUs worth of buffer (see Figure A.12(d)), Riviera has significantly better fairness for all reasonably sized buffers. More importantly, Figure A.12(g) demonstrates that Reno exhibits gross unfairness for even moderately large buffers, and in one case Reno exhibits starvation (buffer size = 500 MTUs).

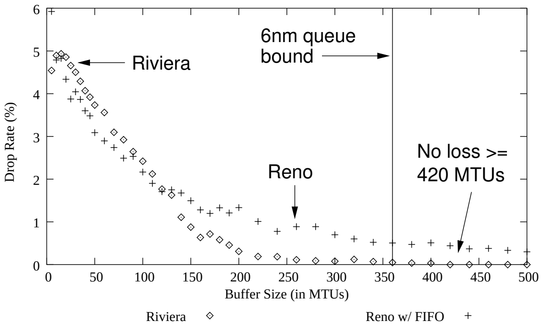
Interesting though of less importance than Goals 1 and 2, the queue length bound (Inequality 3.31 in Section 3.3.4) appears to be violated since Riviera's maximum queue length levels off at about 410 rather than 360 MTUs (see Figure A.12(f). However, there is no violation: 1) the queue bound applies only to a fluid-flow model, and 2) the model assumes that all control loops have equal RTPD. The control loops in Figure A.11 have unequal RTPD.



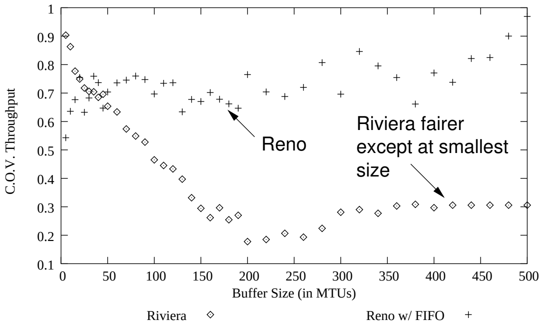
(a) Utilization



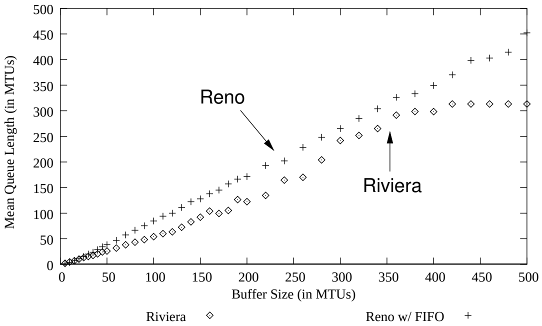
(b) Utilization for Small Buffer



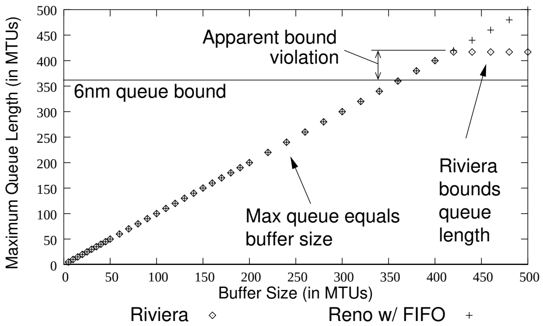
(c) Drop Rate



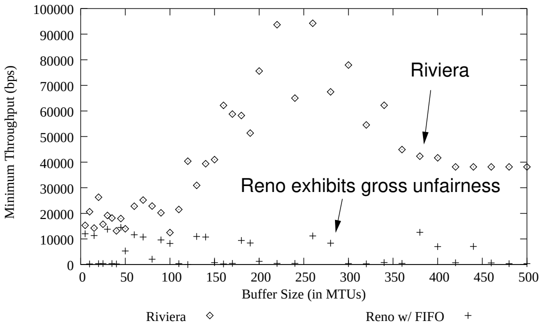
(d) C.O.V. Throughput



(e) Mean Queue Length

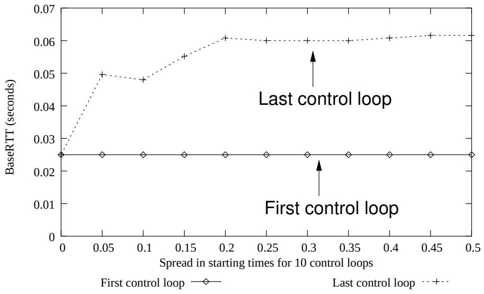


(f) Max Queue Length

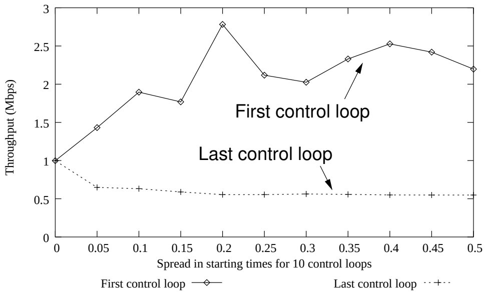


(g) Min Throughput

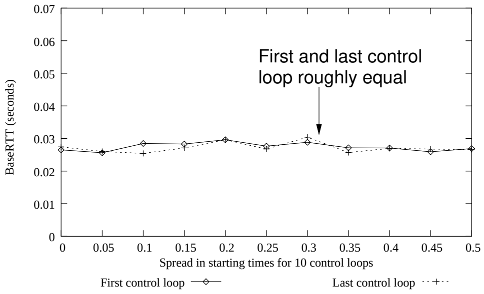
Figure A.12: Riviera and Reno with Small Buffers



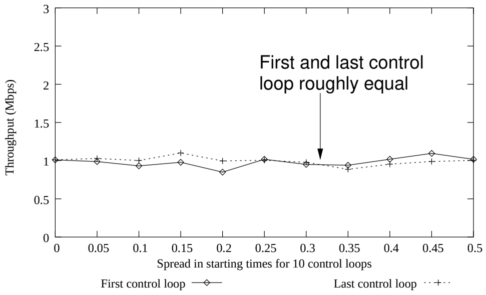
(a) Vegas's BaseRTT vs. Start Time



(b) Vegas's First and Last Throughput vs. Start Time



(c) Riviera's BaseRTT vs. Start Time



(d) Riviera's First and Last Throughput vs. Start Time

Figure A.13: Staggered Starts

A.8 Staggered Starts

Vegas is known to perform unfairly when many sources begin in rapid succession since later connections encounter non-zero queue and thus obtain artificially large RTPD estimates (see Section 2.5.4). We already know from Section 3.2.4 that changes in RTPD cause Vegas to exhibit unfairness. We now reveal a similar effect due to control loops with staggered starts and then show that Riviera does not exhibit this unfairness.

We run 10 Riviera traffic trunks and then 10 TCP Vegas connections across the single bottleneck topology shown in Figure A.2 with a 10Mbps bottleneck and sufficient buffer to avoid loss. All control loops are subject to persistent demand. When running TCP over the topology we remove the EC edges and set the source-to-bottleneck and bottleneck-to-dest links to 4ms in order to maintain the same RTPD for both Riviera and Vegas. Each control loop starts `start_time` time from each other. We vary `start_time` to demonstrate the increase in unfairness as the start times are staggered. To avoid unfairness caused by setting Vegas's α not equal to β as described in [50], we set $\alpha = \beta = 3$ MTUs.

As in prior sections, we determined a proper running time of 30 seconds with a reset time of 15 seconds by running simulations at the extrema and determining when the system appears to enter steady state and then adding a substantial safety factor. In this case, the test simulations all converged within 10 seconds. We run simulations with each

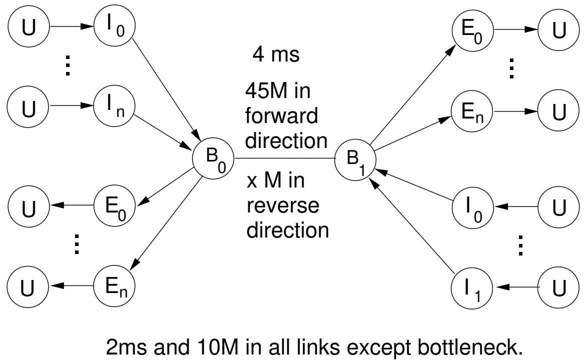


Figure A.14: Reverse Path Congestion Topology

source starting between 0 and 0.5 seconds apart.

Figure A.13(a) demonstrates how the last Vegas connection to start obtains a gross overestimate of the BaseRTT probably due to the queue incurred by the prior nine connections. Figure A.13(b) demonstrates how erroneous in the BaseRTT estimations of RTPD result in unfairness. The last of the ten connection consistently obtains a larger share than the first connection.

Because Riviera intentionally oscillates demand to drain queues, the queues left by prior control loops eventually drain out allowing the latter control loops to obtain a reasonable RTPD estimate (see Figure A.13(c)). As a result, Riviera does not exhibit the same unfairness as Vegas in response to control loops with staggered starts (see Figure A.13(d)).

A.9 Reverse Path Congestion

In this appendix we show the performance of various schemes when subjected to reverse path congestion. For Reno as congestion occurs in the reverse path we expect loss will cause the window size in the forward path to decrease. The result is underutilization of bandwidth in the forward direction. For Riviera we do not expect reverse path congestion to cause loss provided there is sufficient buffer. However, because control packets perform round-trips sharing queues in both directions, congestion in either direction will delay control packets causing an increase in the estimated accumulation. Remember that we define accumulation as a one-way estimate of backlog in the path; thus, queues in the reverse path introduce error in the accumulation estimate, and as with Reno, we expect reverse path congestion to result in underutilization in the forward direction.

To test these hypotheses we use the single bottleneck topology shown in Figure A.14

for $n = 7$. All control loops experience persistent demand and start sending at $t = 0$. All buffers are 90 packets in size. We ran a few simulations to determine proper run time and reset time of 50s and 15s respectively. We then ran a series of simulations each with a different reverse path capacity between 10Mbps and 45Mbps.

When the reverse path capacity is small, Reno, Riviera, and Vegas all suffer underutilization in the forward direction though Vegas is hit hardest (see Figure A.15(a)). Unfortunately with Vegas, underutilization occurs even when the forward and reverse path have equal capacity. Both Vegas and Riviera experience similar delay as shown by the mean queue lengths in the forward and reverse directions in Figures A.15(g) and (h) respectively; therefore, some other feature of Riviera must account for the difference in utilization especially when forward and reverse path bandwidths are near equal. One possibility is that rate-based control continues sending at a steady rate even when a control packet gets delayed by reverse path queue. With window-based control, when acknowledgements are delayed the sender stops sending.

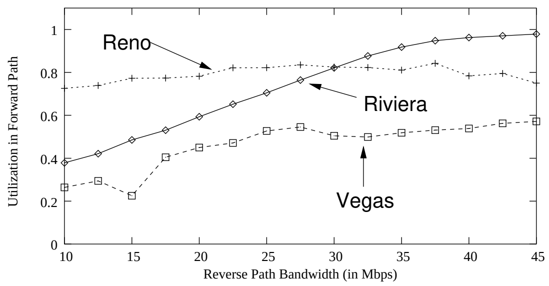
Reno obtains higher forward path utilization (a) than either Vegas or Riviera but this is achieved by incurring substantially larger queues as shown in (c),(d),(e), and (f), and by incurring substantial loss as shown in (g) and (h).

A.10 Changing Round-Trip Propagation Delay

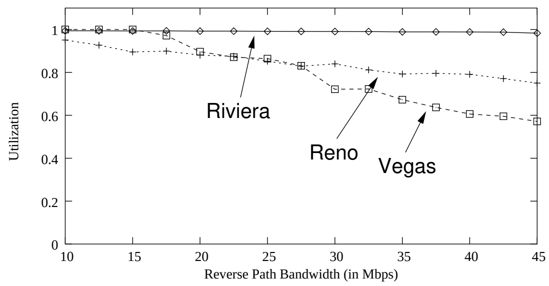
In this appendix we describe the simulations used to generate the results appearing in Section 3.2.4 in Figure 3.7. We use the single bottleneck topology (see Figure A.2) with a 100Mbps bottleneck, sufficient buffer to eliminate loss, 10 control loops, and all links having delay of 2ms except the bottleneck which has initial delay of 4ms. All control loops experience persistent demand and begin sending at $t = 0$. At 5 seconds into the simulation, the bottleneck link's delay in the forward direction suddenly jumps to 44ms causing the round-trip propagation delay between the edges to jump from $2 \times (2ms + 2ms + 4ms) = 16ms$ to $(2ms + 2ms + 44ms) + (2ms + 2ms + 4ms) = 56ms$.

For Vegas and Vegas-k we use the same topology but we remove EC and set the src-bneck and bneck-dest links to 2ms so that the TCP end-to-end control loops now have the same RTPD as Riviera's edge-to-edge control loops. We subject Vegas and Vegas-k to the same jump in delay in the forward direction.

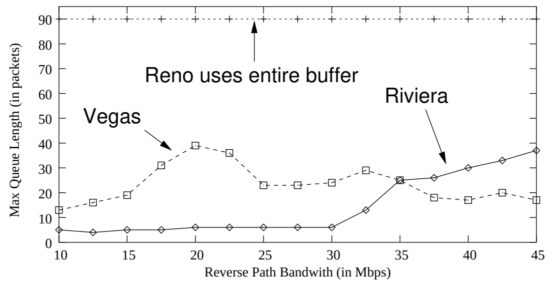
For reference we include the window size of a sample Vegas and Vegas-k control loop and the rate limit graph of a sample Riviera control loop in Figure A.16.



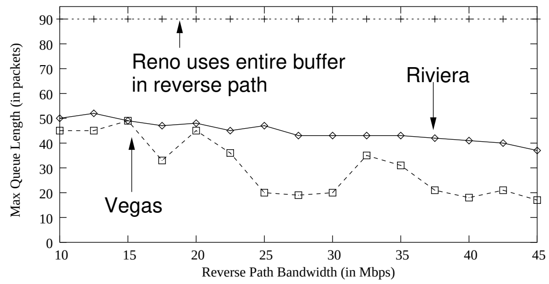
(a) Forward Bottleneck Utilization



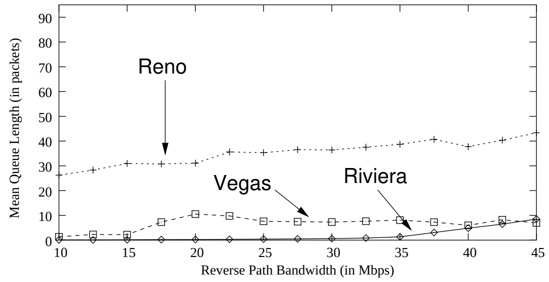
(b) Reverse Bottleneck Utilization



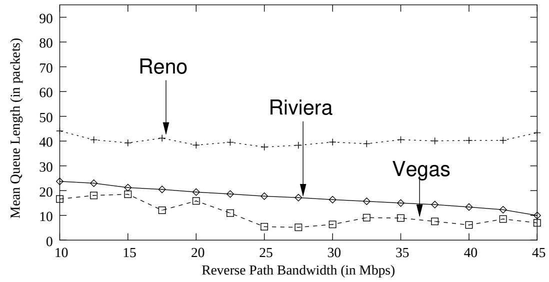
(c) Forward Max Bneck Queue Length



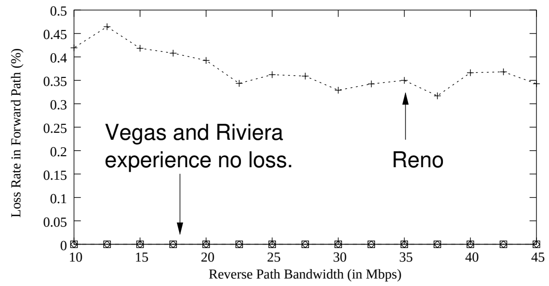
(d) Reverse Max Bneck Queue Length



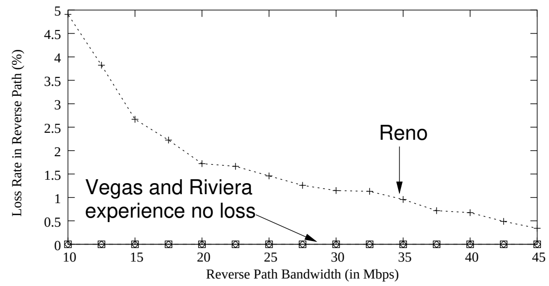
(e) Forward Mean Bneck Queue Length



(f) Reverse Mean Bneck Queue Length

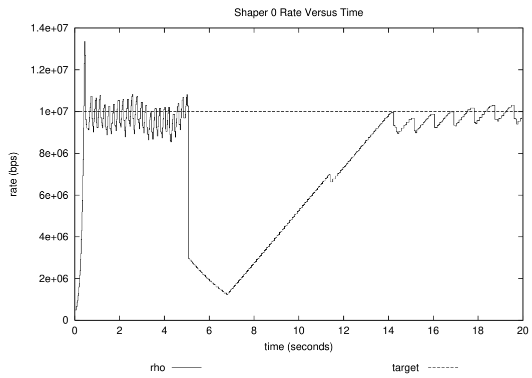


(g) Forward Bneck Loss Rate

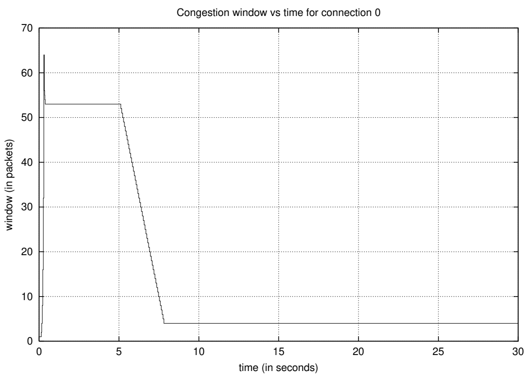


(h) Reverse Bneck Loss Rate

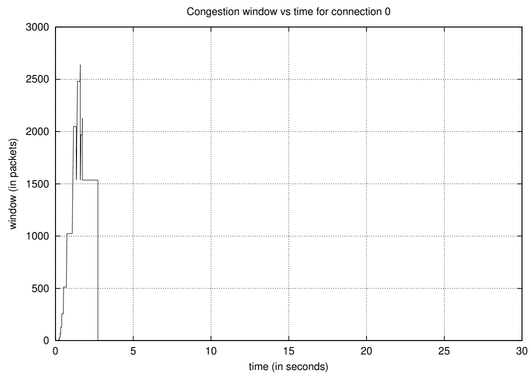
Figure A.15: Reverse Path Congestion Results



(a) Riviera Rate Limit



(b) Vegas CWND



(c) Vegas-k CWND

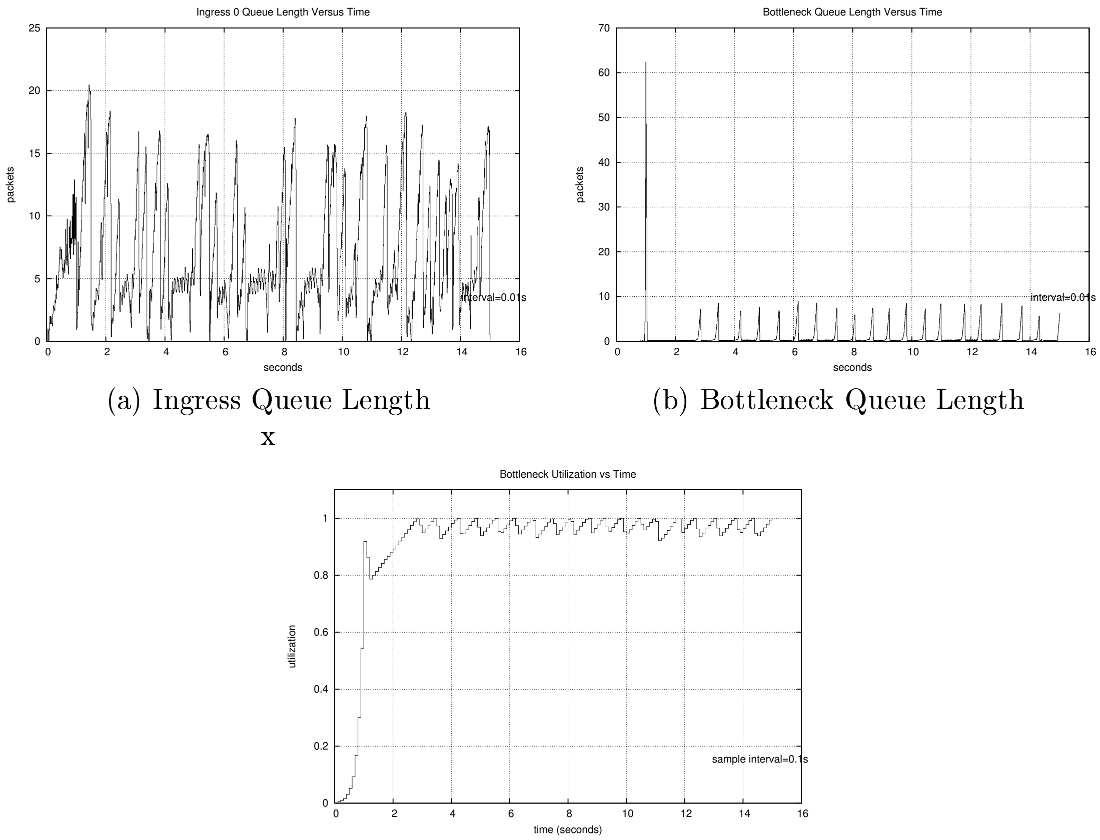
Figure A.16: RTPD Jumps at 5 Seconds

A.11 Weighted TCP Per-Flow Service Building Block

In Section 4.1 we present partial results for a weighted TCP building block. With this building block each connection obtains a weighted share of the capacity allocated to its EC trunk by employing TCP Rate Control (see Appendix D) at each edge. We run a simulation over the topology in Figure 4.2 to generate the results found in Figure 4.3. Each control loop is subjected to persistent demand and start sending at $t = 0$. Figure 4.3 only reveals the congestion windows of each connection, we provide further statistics in Figure A.17.

The bottleneck queue length shown in Figure A.17(b) follows the expected oscillatory behavior. The queue maintains zero most of the time because there are only two edge-to-edge control loops contending for the bottleneck. This also accounts for the dips in utilization seen in Figure A.17(c).

The ingress queue length shown in Figure A.17(a) appears larger than the queue



(a) Ingress Queue Length
x

(b) Bottleneck Queue Length

(c) Utilization

Figure A.17: Weighted TCP Per-Flow Service Building Block Results

length at the bottleneck. The larger queue length is expected given the ingress target queue length $q_0 = 10$ as described in Appendix D. The queue does not settle exactly around 10 packets because the capacity provided by the edge-to-edge control loop (i.e., the control loop's rate limit) is a moving target. Since the edge-to-edge control loop spends most of its time in additive increase, the TCP rate controller adjusts behind the EC control loop resulting in an ingress queue length that tends to be smaller than the target.

A.12 Distributed Buffer Management

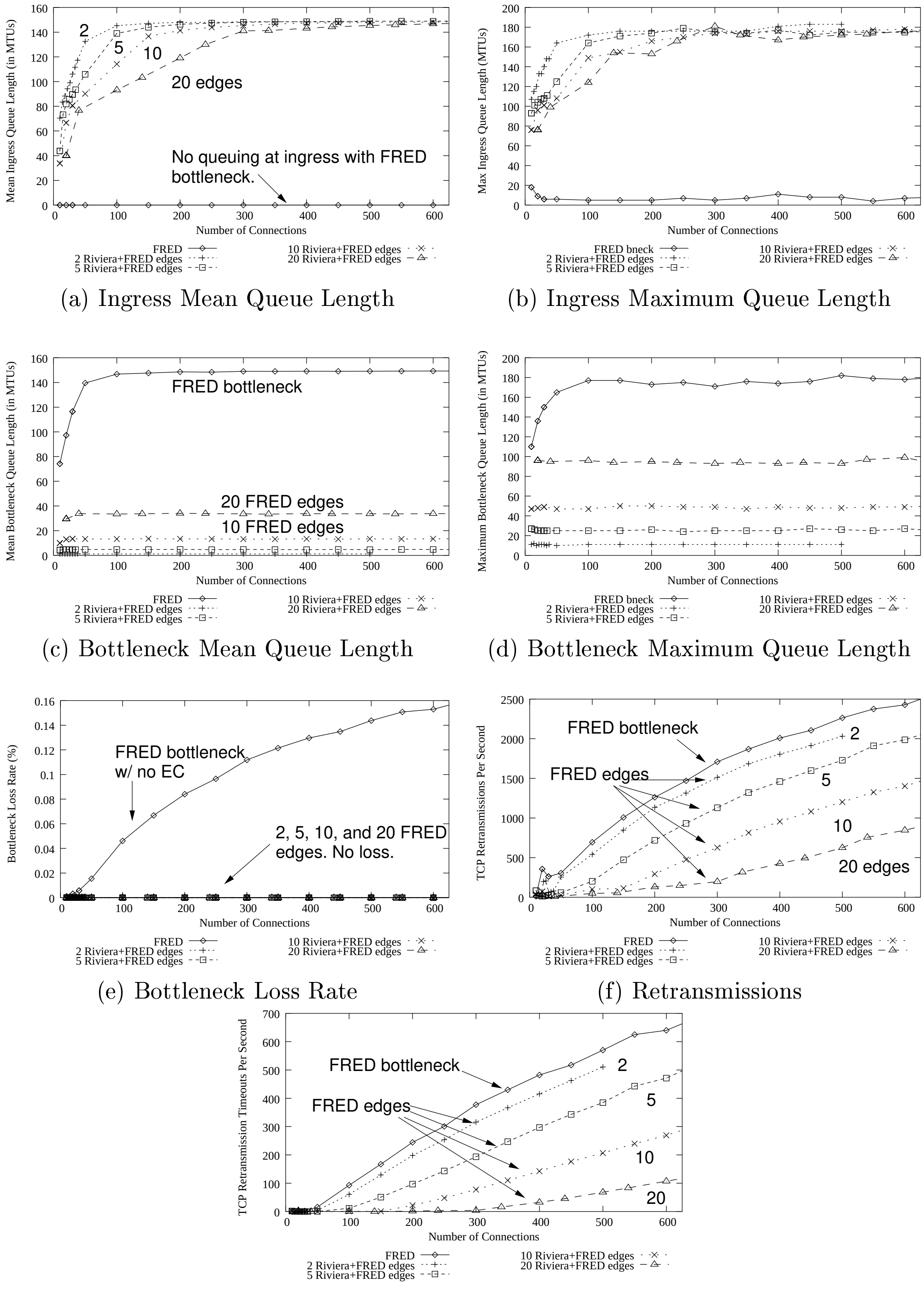
In this appendix we present further results for the simulations described in Section 4.2. We use the single bottleneck topology in Figure A.2, but we place a FRED queue at the ingress to control TCP Reno connections running from end-to-end. We refer to placement of FRED at the edge as "distributed FRED." All links have 300 packet buffers. Each connection experiences persistent demand starting at a random time uniformly distributed in $[0, 5.0]$ seconds. We vary the number of connections from 10 to over 600 distributed evenly across the edges. We also vary the number of edges from 2 to 20. We determined experimentally using the same technique as in previous sections that a run time of 30 seconds and a reset time of 10 seconds are sufficient to obtain steady-state statistics.

In Figure A.18 we show the related results for distributed FRED. We replace the FRED edges with FIFO edges in Figure A.19.

Most striking, distributed FIFO fails to improve C.O.V. (see Figure A.19(a)), even though distributed FIFO reduces the number of retransmissions (h) and although not shown here, distributed FIFO reduces the number of retransmission timeouts. Some factors other than the number of retransmissions appears to dominate fairness. Perhaps synchronization? Regardless of the source, the evidence suggests that FIFO is not a good choice for the buffer management algorithm applied at the edge.

As with FRED buffer management, Figure A.19(c) reveals the bottleneck mean queue lengths are much smaller than the mean queue lengths without EC, and the bottleneck mean queue length does not appear to be related to the number of end-to-end control loops. Thus, we reveal that at least for FRED and FIFO, we can redistribute the bottleneck queues across the ingress edges. We see no reason why EC should not work with any buffer management algorithm in this way.

As demonstrated in Section 3.3 the length of the bottleneck queues is a function of the number of edge-to-edge control loops. So long as the number EC control loops passing through the bottlenecks in the network is significantly smaller than the number of end-to-end flows, as would be expected for VPN applications, we would expect decreased bottleneck queues and successful redistribution of queues to the edges where more stateful mechanisms can be employed.



(a) Ingress Mean Queue Length

(b) Ingress Maximum Queue Length

(c) Bottleneck Mean Queue Length

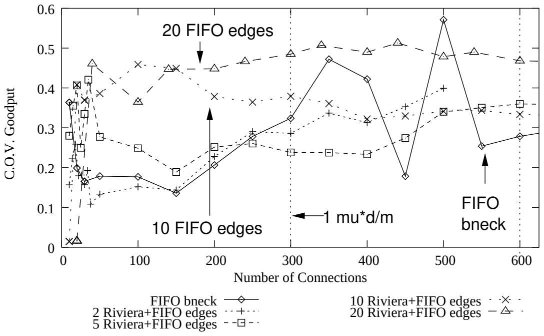
(d) Bottleneck Maximum Queue Length

(e) Bottleneck Loss Rate

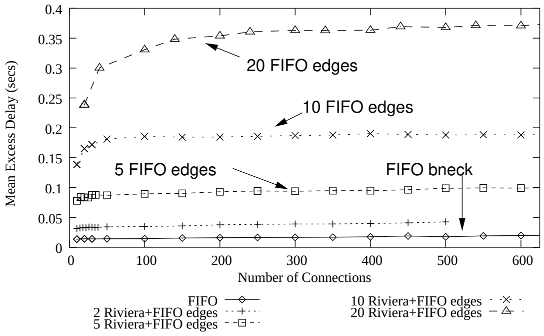
(f) Retransmissions

(g) Retransmission Timeouts

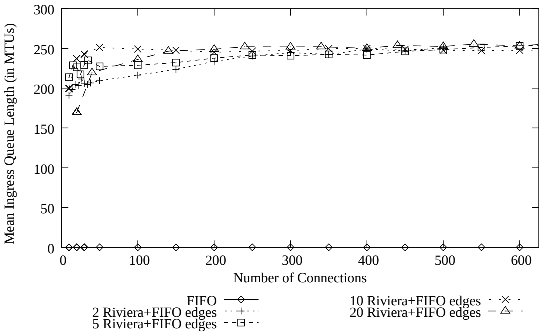
Figure A.18: Distributed Buffer Management Results with FRED Edges



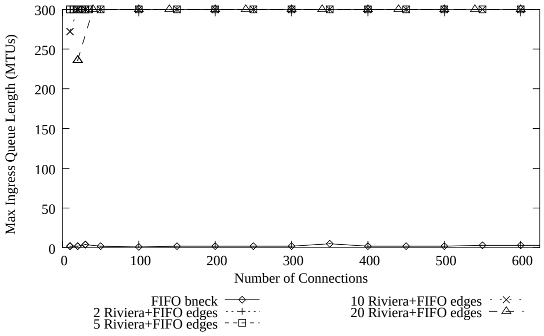
(a) Coefficient of Variation



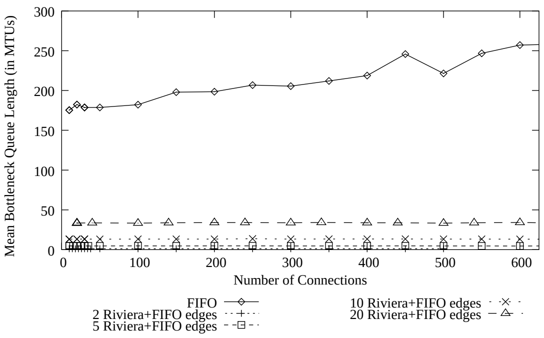
(b) Mean Excess Delay



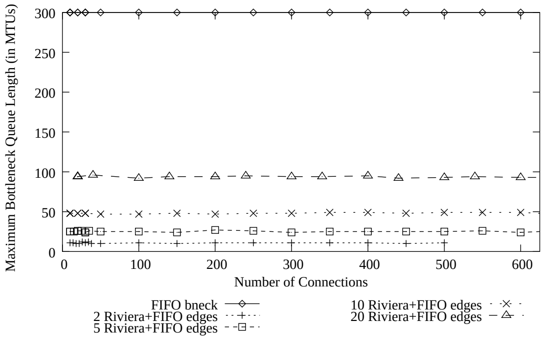
(c) Ingress Mean Queue Length



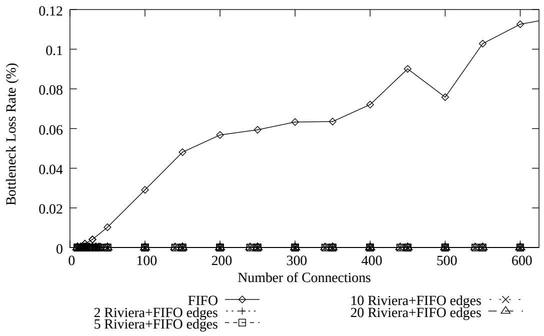
(d) Ingress Maximum Queue Length



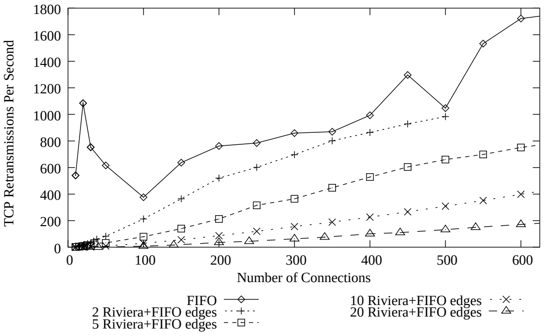
(e) Bottleneck Mean Queue Length



(f) Bottleneck Maximum Queue Length



(g) Bottleneck Loss Rate



(h) Retransmissions

Figure A.19: Distributed Buffer Management Results with FIFO Edges

A.13 Weighted Proportionally Fair Service Building Block

This section provides expanded simulation results for the simulations presented in the Weighted Proportional Fair Service Building Block section (see Section 5.2).

All of the simulations in this section use the simple single bottleneck topology shown in Figure A.2 with two control loops, but we vary the weight for one of the control loops while keeping the other weight at one. All control loops experience persistent demand starting at time $t = 0$. Given time we will have to come back and better study the multiple bottleneck scenarios for the weighted service.

Results for Riviera control loops with equal weights passing through multiple bottlenecks appear in Appendix A.4.

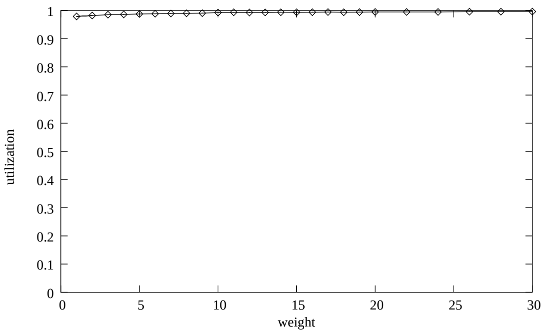
We use enough buffer in these simulations to avoid any loss. For weights less than or equal to 30, all simulations are run for 30 seconds and statistics are gathered over the last 20 seconds to avoid initial transients. We deal with the case of weights larger than 30 later in this appendix. As in prior simulations, we ran several simulations including the extrema to determine proper run and reset times. We show additional statistics in Figure A.20 for the simulations shown in Figure 5.2. As expected utilization increases with weight but is always near one. Mean queue length and max queue length increase roughly linearly with the sum of the weights, which is to be expected since the sum of the maximum thresholds before congestion is detected also increases linearly.

The diagonal lines on graphs (e) and (f) are the steady state queue length bounds as derived in Inequality (3.24) except we take into consideration the sum of the max thresholds and for (f) increase differentiation. Without increase differentiation,

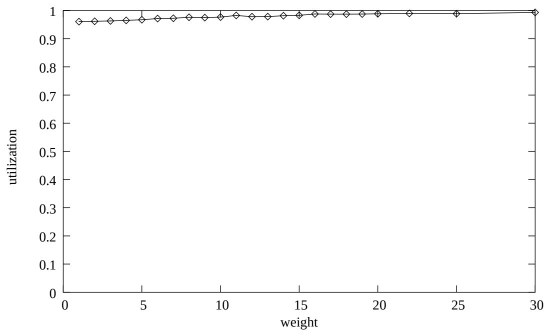
$$
\begin{aligned}
\rho(t) &\leq \mu + \frac{nm}{T} \ , \\
q_{max} &\leq q(0) + I[0,1] - \mu \ , \\
q_{max} &\leq \epsilon \sum_i w_i + \int_0^1 (\mu + n\, m) dy + n\, \sigma - \mu \ , \\
q_{max} &\leq \epsilon W + n\, m + n\, \sigma
\end{aligned}
\tag{A.1}
$$

where $W = \sum_i w_i$.

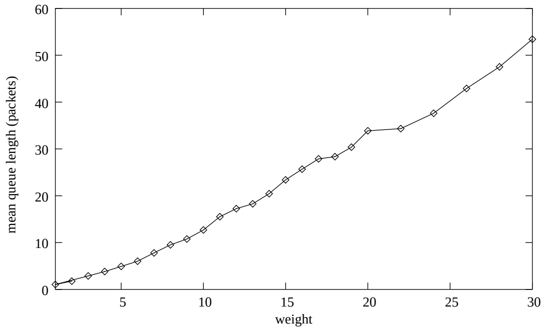
In our simulations shown in Figure A.20, $\sigma = m$, $\epsilon = 3\sigma$, $n = 2$, $w_0 = x$, and $w_1 = 1$, thus $q_{max} \leq (3x + 7)\sigma$. Figure A.20(e) reveals that at least within the range of $1 \leq w_0 \leq 30$, the simulations do not violate Inequality A.1 in the steady state.



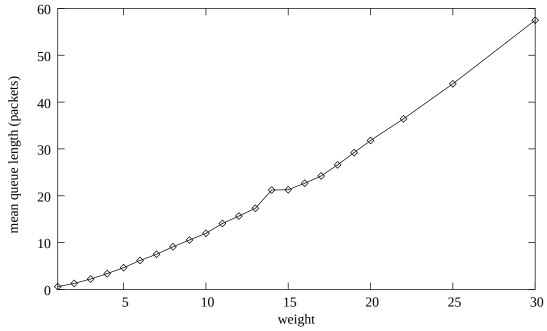
(a) Utilization (ADiff[1])



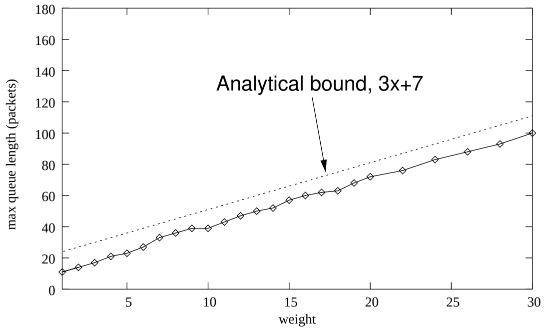
(b) Utilization (AIDiff[2])



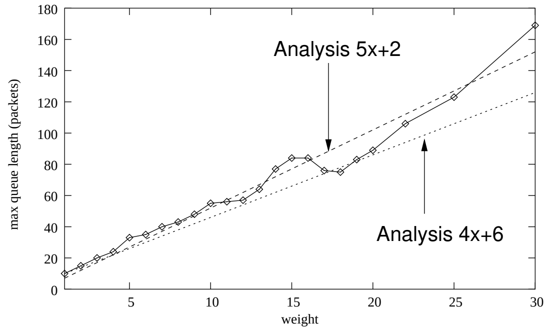
(c) Mean Steady State Queue Length (ADiff)



(d) Mean Steady State Queue Length (AIDiff)



(e) Max Steady State Queue Length (ADiff)



(e) Max Steady State Queue Length (AIDiff)

[1] ADiff refers to Accumulation Differentiation.
[2] AIDiff refers to Accumulation Plus Increase Differentiation

Figure A.20: Weighted Building Block Variants with Small Weights

With increase differentiation, the rate limit at each increase step is bounded by

$$\rho(t) \leq \mu + W\frac{m}{T} \ .$$

Therefore,

$$\begin{aligned} q_{max} &\leq \epsilon W + \int_0^1 (\mu + W\, m) dy + n\,\sigma - \mu \ , \\ q_{max} &\leq n\,\sigma + (\epsilon + m)\, W \ . \end{aligned} \tag{A.3}$$

For our simulations, Inequality A.3 becomes $q_{max} \leq (4x+6)\sigma$. This analysis appears as the bottom line in Figure A.20(e). Unfortunately this analysis does not reasonably follow the simulation results. So we now derive a result along the same lines as for Inequality 3.31.

$$\begin{aligned} (\frac{q}{\mu} + d) dq &= m\, W\, dt, q(0) = \epsilon\, W \ , \\ q(\tau) &\leq -\mu\, d + \sqrt{d^2\mu^2 + 2\, m\, \mu\tau\, W} \ , \\ T_0 &= \epsilon \frac{W}{\mu} + d \ , \\ T_1 &= \frac{q(T_0)}{\mu} + d = \frac{1}{\mu} q(\frac{\epsilon}{\mu} W + d) + d \ , \\ &\cdots \ , \\ q_{max} &< -\mu\, d + n\sigma + \sqrt{d^2\mu^2 + 10 d\mu\, W\sigma + 23\sigma^2\, W^2 + 2\sigma^2 W n} \ . \end{aligned} \tag{A.4}$$

If we constrain all control loops such that $\forall i : w_i \geq 1.0$ then $n \leq W$ and Inequality A.4 is bounded by

$$\begin{aligned} q_{max} &< -\mu\, d + n\sigma + \sqrt{d^2\mu^2 + 10 d\mu\, W\sigma + 25\sigma^2\, W^2} \ , \\ q_{max} &< n\,\sigma + 5\,\sigma\, W \ . \end{aligned} \tag{A.5}$$

Inequality A.5 appears in Figure A.20(e) as the upper line. It is straightforward to derive similar bounds for the simulations where there is no increase differentiation, but we do not show it here. Unfortunately, even the larger analytical bound given by Inequality A.5 does not closely follow the simulation results as weight increases and most of the simulation points violate the bound.

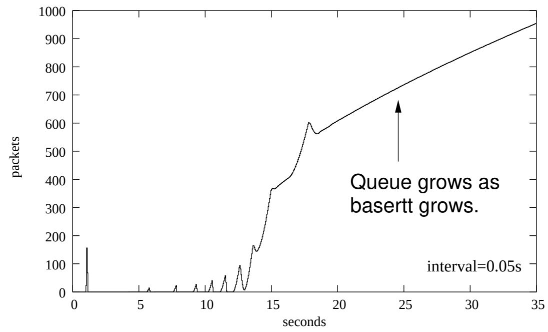
As we increase the weights above around 100, a limitation of Riviera appears. When we estimate the RTPD (i.e., basertt) using the min-of-the-last-k RTT samples, heavy weights cause the queue length to diverge violating Goal 3. The problem arises when the number of RTTs required to reach the maximum threshold on accumulation exceeds k RTTs. When this occurs, the basertt increases alongside additive increase preventing the accumulation estimate from ever reaching the maximum threshold. Since the system never detects congestion, the setting for β becomes irrelevant. We demonstrate this problem in Figure A.21 for the case with a weight of 1000 versus 1 with $k = 30, \beta = 0.4$. An obvious solution is to increase k, but this results in poor response to changes in propagation delay. Furthermore, no matter how large we set k, as we increase the sum of the weights, at some point the time it takes to build enough queue to detect congestion will take longer than k RTTs. A better alternative might be to adapt k according to the weight of the control loop. For the remainder of the simulations in this section we simply disabled min-of-the-last-k in favor of min-seen-so-far (i.e., Vegas's RTPD estimator). We leave the solution to this problem up to later research.

Even with the min-of-the-last-k algorithm disabled, we found that the range of achievable weights is somewhat improved if the max threshold is increased and β decreased. In Figure A.22 we provide extended results for the simulations in Figure 5.3. These results correspond to a $\beta = 0.8$ and $\epsilon_{max} = 6\sigma$. We include the two of the analyses appropriately adjusted for the increased threshold. The threshold adjustment for Inequality A.1 yields $q_{max} \leq (6x + 10)\sigma$, and Inequality A.5 yields

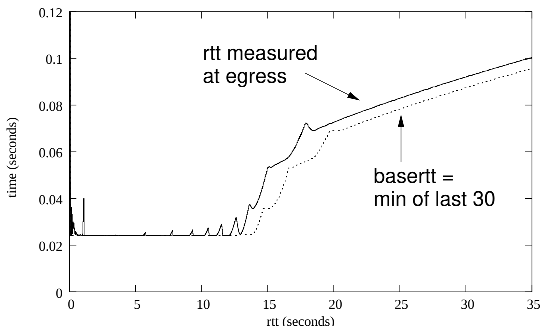
$$\begin{aligned} q_{max} &< -\mu\, d + n\sigma + \sqrt{\mu^2\, d^2 + 16W\sigma\mu d + 62\sigma^2 W^2 + 2nW\sigma^2}\ , \\ q_{max} &< n\sigma + 8\sigma W\ , \\ q_{max} &< (8x + 10)\sigma\ . \end{aligned} \tag{A.6}$$

Unfortunately Inequality A.6 falls below the simulation results. We will have to determine the cause of this error in later work.

In these simulations, we increase the simulation run time to 500 seconds to give the system more time to converge. We collect results starting from 350 seconds into the simulation. We determined the reset time empirically by running a few sample simulations including the extrema and then picking a reset time larger than it took for any of the sample simulations to converge on a final bandwidth allocation. The convergence time with weight proportional additive increase is always less than 20 seconds, whereas without additive increase it is difficult to determine when the system "converges" because the oscillations are much slower. As an example we show the rate limit versus time for the case with 200-to-1 weights in Figure A.23 with and without proportional additive increase.



(a) Bottleneck Queue Length V. Time



(b) RTT and BaseRTT V. Time

Figure A.21: Diverging With Heavy Weights and Min-of-the-last-k

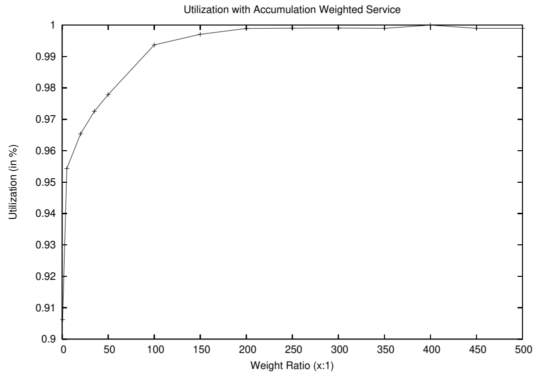
Notice the difference in timescales for rate limit versus time between Figure A.23(a) and (b).

A.14 Guaranteed Bandwidth Building Block

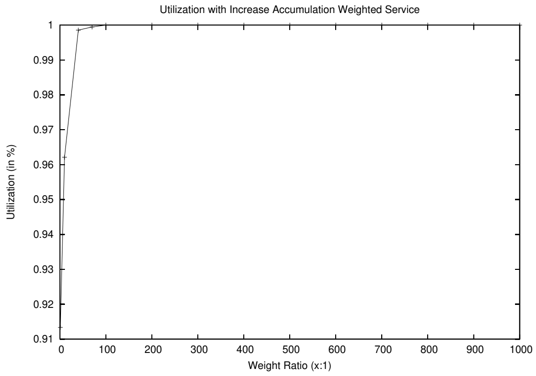
In this appendix, we describe how we derived the graphs appearing in Figure 5.4 and 5.5. As in Section A.13 we turned off the min-of-the-last-k round-trip times because as guarantees become large, the queue begins to diverge.

The queue length bound derived in Section 5.3.3 assumes that the bottleneck is fully utilized by non-guaranteed traffic and then all guarantees passing through the bottleneck suddenly start at their respective rate limits. To emulate this for the results appearing in Figure 5.4 we use two control loops with guarantees $g_0 = 0$ and $g_1 = x$ across the single bottleneck topology in Figure A.2 with sufficient buffer to eliminate loss. Loop zero begins at time zero. Loop one begins at time t, where t is sufficiently large to allow loop zero to fully utilize the bottleneck capacity before loop one begins. Once started, both control loops experience persistent demand. As with prior sections we ran several simulations with different run times to determine proper reset time (85s) and simulation time (100s). However this time, we also determine an appropriate value for $t = 8$ seconds. In Figure A.24, we show a sample simulation for $g = 95$Mbps out of the 100Mbps bottleneck. The bottleneck buffer is large enough to avoid loss.

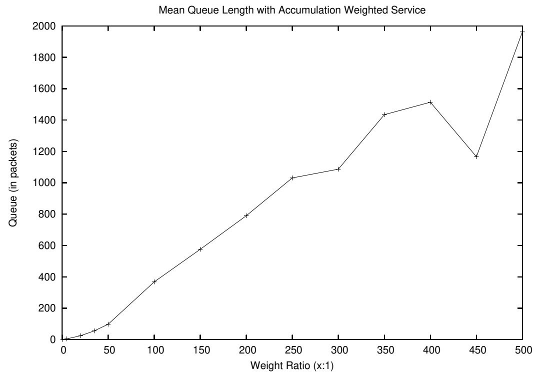
For the multiple bottleneck simulations that appear in Figure 5.5, we use the same linear network topology shown in Figure A.4 with sufficient buffer to avoid loss. We start all control loops at a random time uniformly distributed in $[0, 2]$ seconds. Once started each control loop experiences persistent demand. Running a few simulations each with a



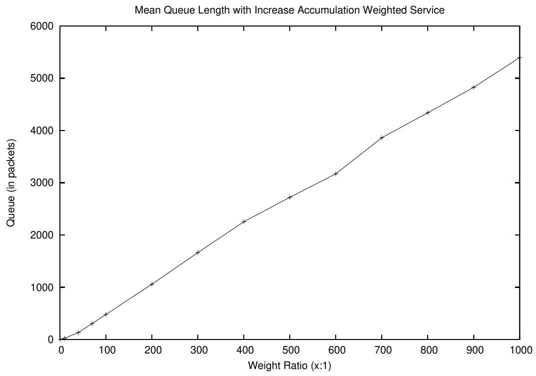
(a) Utilization (ADiff)



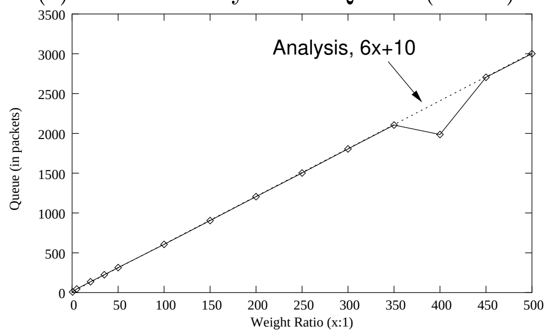
(b) Utilization (AIDiff)



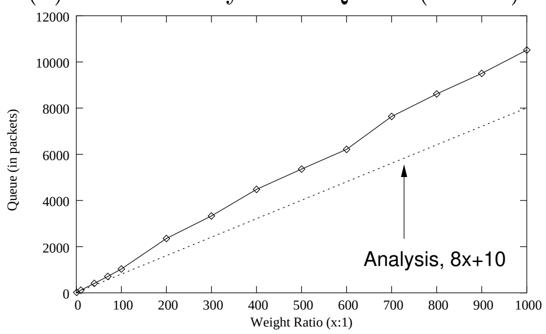
(c) Mean Steady State Queue (ADiff)



(d) Mean Steady State Queue (AIDiff)



(e) Max Steady State Queue (ADiff)



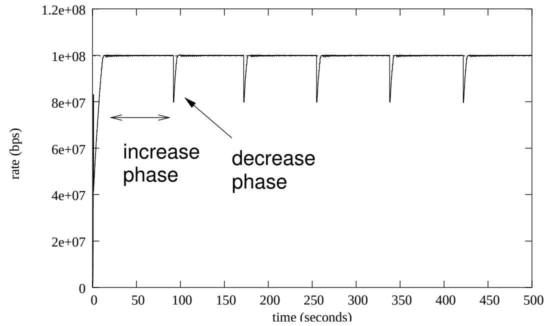
(f) Max Steady State Queue (AIDiff)

Figure A.22: Weighted Building Block Variants with Heavy Weights

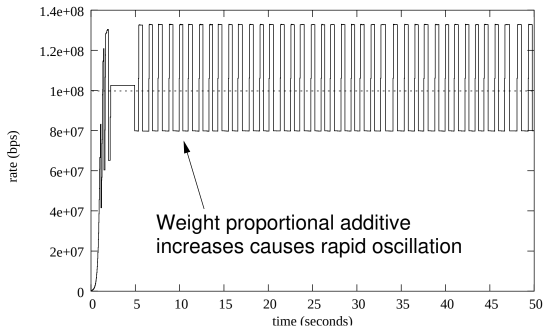
different number of bottlenecks we determine an appropriate run time of 30 seconds with a reset time of 15 seconds. We show the results for these simulations in Figure A.25.

A.15 Assured Bandwidth Building Block

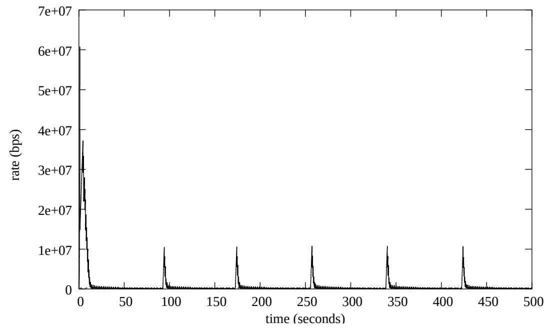
In this appendix, we describe how we derived the graphs appearing in Figure 5.11, 5.12, and 5.13. As in Section A.13 we turned off the min-of-the-last-k round-trip times because as assurances become large, the queue begins to diverge.



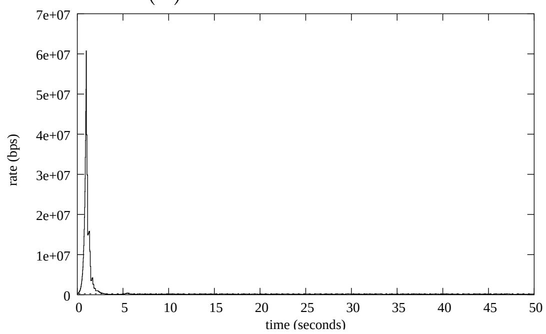
(a) ADiff for $w = 1$



(b) AIDiff for $w = 1$

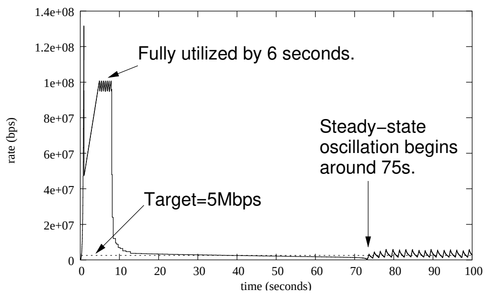


(c) ADiff for $w = 200$

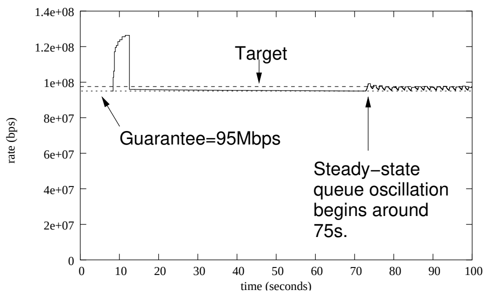


(d) AIDiff for $w = 200$

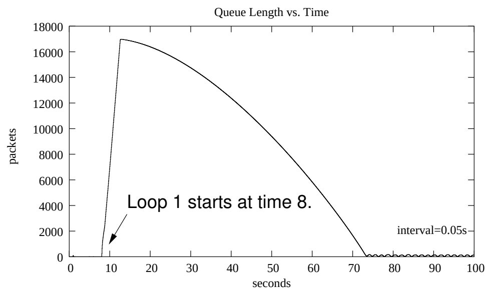
Figure A.23: Rate Limits For Examples with Heavy Weight



(a) Rate limit, no guarantee

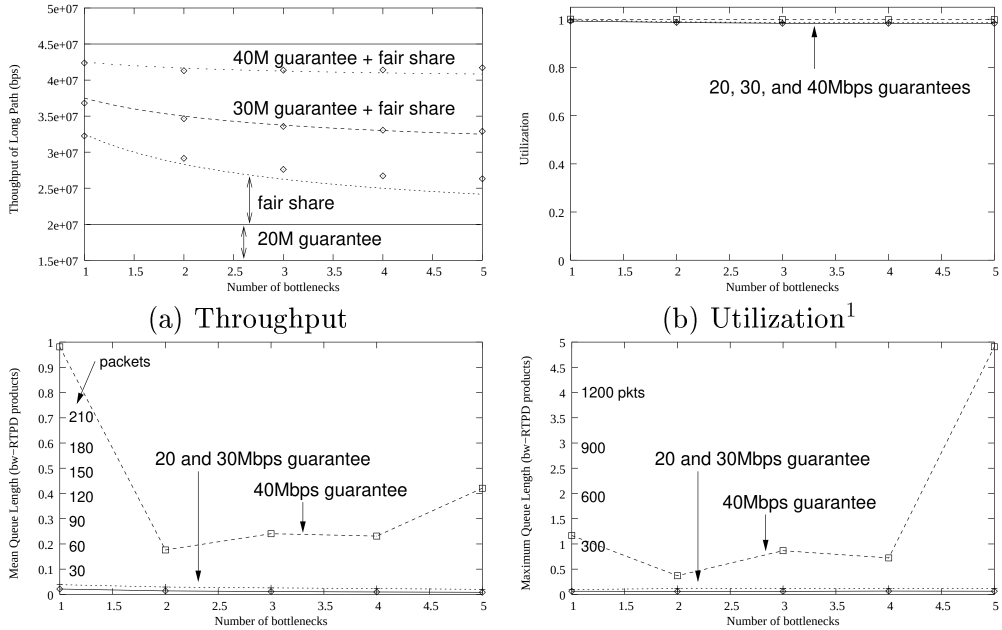


(b) Rate limit, guarantee of 95Mbps



(c) Bottleneck queue length

Figure A.24: Sample Simulation for g=95Mbps



(a) Throughput (b) Utilization[1]

(c) Mean Queue Length[1] (d) Max Queue Length[2]

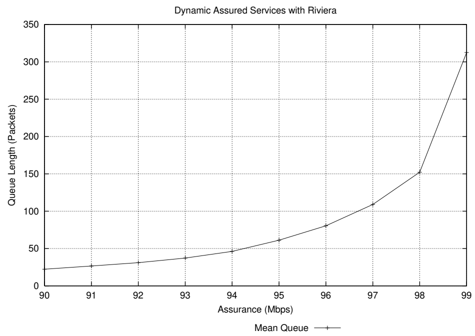
[1] Mean of means across the simulations, [2] Max of maxima across the simulations

Figure A.25: Quasi-Leased Line Multi-bottleneck Results

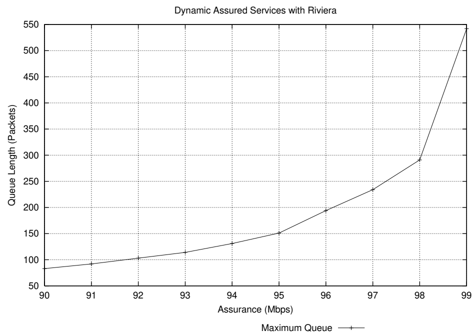
For both Figures 5.11 and 5.13, we use the same single bottleneck topology shown in Figure A.2 with sufficient buffer to eliminate loss and two control loops. The first control loop has a zero assurance while we change the other assurance in each subsequent simulation. Both control loops begin sending at time 0 and experience persistent demand. We tune the run and reset times to 60 and 20 seconds respectively. We present additional results to these simulations in Figure A.26 and A.27 respectively.

As described in Chapter 5, the assured bandwidth allocation imposes a maximum weighted share to prevent a control loop with an assurance from starving other control loops during oversubscription. In order to obtain allocations very close to full capacity, in Figures 5.11 and A.26 we use a maximum weight of 10000. In Figure A.27 to demonstrate behavior of the assured building block in response to oversubscription we use a maximum weight of 10.

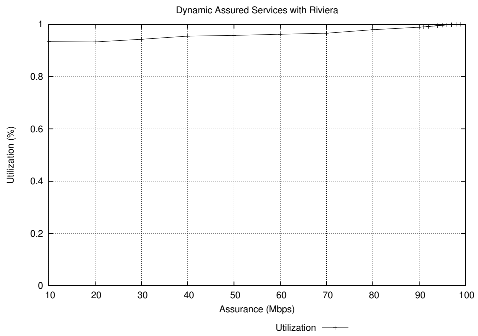
Perhaps the only interesting feature of Figure A.27 is that mean queue and max queue do not change once the control loop with the assurance exceeds a weighted share around 10.



(a) Mean Queue

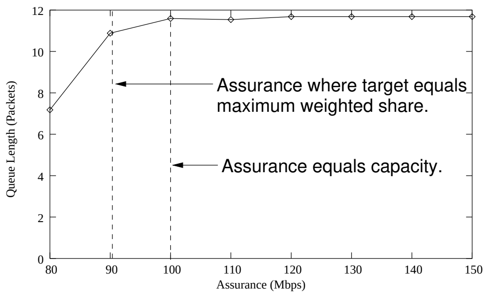


(b) Max Queue

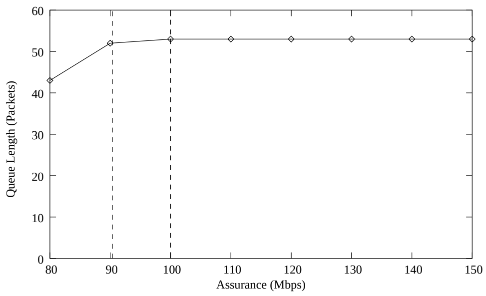


(c) Utilization

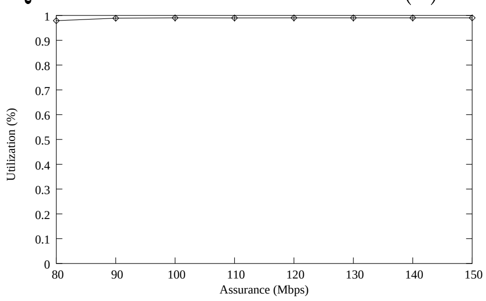
Figure A.26: Single Bottleneck Assured Bandwidth Allocation Results



(a) Mean Queue



(b) Max Queue



(c) Utilization

Figure A.27: Oversubscription for Single Bottleneck Bandwidth Allocation

APPENDIX B
Riviera Pseudocode

In this appendix we provide pseudocode for the Riviera congestion svoidance scheme.

B.1 Ingress

In Tables B.1, and B.3 we present the mapping between Riviera's parameters and their names in the pseudocode. In Tables B.2 and B.4. Default values for the parameters are given in Table A.1.

Table B.1: Ingress Parameters in Equations and Pseudocode

in eqns	in code	description
a_i	`assured_capacity_`	(see Section B.3.2)
β	`beta_`	decrease factor during congestion
β_{loss}	`beta_loss_`	decrease factor when loss
	`cap_fraction_`	initial `rthresh_ = cap_fraction_ * link_capacity_`
	`g_`	gain in RTO algorithm
g_i	`guaranteed capacity_`	(see section B.3.3)
	`increase_weight_`	(see section B.3.1)
	`init_rtt_`	initial value for RTT.
	`link_capacity_`	ingress link capacity
m	`mtu_`	maximum packet size
σ	`sigma_`	maximum burst size (in bytes)
s	`ss_increase_`	slow start increase factor
	`timestamp_window_size_`	number of control packet timestamps remembered

B.1.1 Initialization

```
bdepartures_ = 0;                                      // 1
prev_bdepartures_ = 0;                                 // 2
tripcount_= 0;                                         // 3
first_packet = true;                                   // 4
r_ = (mtu_<<3) / init_rtt_;                            // 5
rthresh_ = cap_fraction_ * link_capacity_;             // 6
srtt_ = -1.0;                                          // 7
```

Table B.2: Ingress Variables

name	description
`arto_`	`now() + rto`
`bdepartures_`	byte departures in trunk lifespan
`departure_rate`	rate of ingress departing traffic
`egress_rate`	rate extracted from reverse control packet
`err`	error RTO algorithm
`first_packet_`	first packet sent on trunk
`insert_etc_in_next_packet_`	insert a control header in next packet
`pdepartures_`	packet departures so far
`prev_bdepartures_`	byte departures at previous control packet departure including control packet
`prev_retc_time_`	time previous RETC arrived
`r_`	rate limit
`r_inc`	rate limit increment
`rthresh_`	rate-based slow-start threshold
`rto_`	retransmission timeout in RTO algorithm
`rtt`	round-trip time sample
`sdev_`	smoothed mean deviation in RTO algorithm
`tripcount_`	number of round-trips made by control packet
`srtt_`	smoothed RTT in RTO algorithm

Steps 1-2 initialize state used in departure rate computations. Step 3 always starts the trip count of a new trunk at zero. There may be a reason to initialize `tripcount_` to something other than zero to avoid ambiguity between incarnations of the same trunk.

We distinguish the first packet (Step 4) from other packets to handle ambiguities in the RTT estimation and egress rate estimation when no prior packet has been received from which to make such estimates.

Because no prior RTT measurement exists, we initialize `r_` with respect to the `init_rtt_` parameter (Step 5).

We initialize the rate-based slow start threshold with respect to the ingress's link capacity (Step 6) because a trunk will never exceed the ingress's own link capacity.

`srtt_ < 0.0` is later used in Section B.1.2.1 to differentiate the arrival of the first Reverse Edge Traffic Control (RETC) packet. Thus in Step 7, we initialize `srtt_` to a negative value.

B.1.2 Receive Edge Control

This function processes a reverse edge traffic control packet `pkt`.

```
recv_edge_control( pkt ) {
  round-trip estimation
  departure rate computation
  loss
  decrease
  discard old control packets
  increase
  update counters
}
```

B.1.2.1 Receive Edge Control: Round-trip Estimation

```
timestamp = get_timestamp_from_tripcount(pkt);           \\ 1
if ( timestamp < 0.0 ) { free( pkt ); return; }          \\ 2

rtt =  now() - timestamp;                                \\ 3

if ( srtt_ < 0.0 ) {                                     \\ 4
  srtt_ = rtt;                                           \\ 5
  sdev_ = srtt_;                                         \\ 6
}
else {
  err = rtt - srtt_;                                     \\ 7
  srtt_ += g_ * err;                                     \\ 8
  if ( err < 0.0 ) err = -err;                           \\ 9
  sdev_ += g_ * ( err - sdev_ );                         \\ 10
}
rto_ = srtt_ + max ( 4 * sdev_, srtt_ );                 \\ 11
arto_ = now() + rto_;                                    \\ 12
```

In Step 1, `get_timestamp_from_tripcount` retrieves the appropriate timestamp in the timestamp window for the given trip count. If the `<trip count,timestamp>` pair does not appear in the timestamp window then `get_timestamp_from_tripcount` returns -1.0. The timestamp window is implemented using a circular buffer thus `get_timestamp_from_tripcount` only returns -1.0 for a packet old enough that its information is of dubious use. When the return value is -1.0, Step 2 discards the control packet and immediately returns.

In Step 3, `now()` returns the system clock's time. Preferably this time will be much higher resolution than occurs with end system clocks.

Steps 4-6 initialize the smoothed round-trip time and smoothed mean. Since only one round-trip sample is known, the smoothed round-trip time is set equal to the single round-trip time sample. The mean deviation of a single sample is always zero. However, a zero mean deviation generates a timeout on the first packet that encounters even the slightest queueing delay or experiences positive estimation error. An initial mean deviation on the order of the round-trip time seems more reasonable (see Step 5).

Steps 7-12 implement Van Jacobson's retransmission timeout algorithm. We use Van Jacobson's algorithm to trigger retransmission of lost control packets. Riviera control packet retransmission differs from TCP retransmission in that the retransmitted control packet has updated trip count and byte count. Step 11 differs from Van Jacobson's algorithm in that we require the retransmission timeout to at least equal two smoothed round-trip times. This handles the situation where after a long period of underutilization and consequently small mean deviation, even a slight delay pertubation of the system would generate a timeout if we set `rto` to 4× mean deviation. Small pertubations do not generate timeouts with TCP becase TCP implements only a coarse-grained timeout mechanism whereby the retransmission timeout takes on the granularity of tenths of a second.

B.1.2.2 Receive Edge Control: Departure Rate Computation

```
r_inc = (mtu_<<3) / rtt;                                \\ 1
departure_rate = ( bdepartures_ - prev_bdepartures_ ) * 8.0 /
  ( now() - prev_retc_time_ );                          \\ 2
departure_rate = min( departure_rate, r_ );             \\ 3
```

Step 1 computes the increase in rate limit as one maximum transmission unit per RTT. Step 1 currently uses the current RTT sample rather than the smoothed round-trip time. This causes `r_inc` to diminish the moment congestion begins regardless of whether congestion is actually detected. Using the curent RTT also introduces variance in `r_inc`. We have not studied `r_inc` variance though the benefit of sublinearity due to `r_inc`'s decrease with increasing RTT is demonstrated in Section 3.2.3.

Step 2 determines the departure rate as the number of bytes that have departed since the most recent reverse edge traffic control packet arrival. This computation does not use RTT because rate limits change with each RETC arrival and because the time since the previous arrival of RETC will typically be longer than the RTT. RTTs are shorter simply because control headers are inserted in the next outgoing packet in the trunk. At some point we may change this mechanism such that a separate control packet rather than a control header is sent.

In a fluid flow with zero burstiness, the departure rate never exceeds the rate limit. However, due to burstiness up to the maximum burst size σ, packetization, and discrete measurement intervals, it is possible for the departure rate to somewhat exceed the rate limit. The effect is exaggerated at low bit rates. Step 3 sets the departure rate equal to the minimum of the departure rate and the rate limit so that the rate limit will never increase by more than `r_inc` from the current rate limit in any RTT when in the congestion avoidance phase.

B.1.2.3 Receive Edge Control: Loss

```
if ( pkt.action == LOSS ) {                              \\ 1
  rthresh_ = egress_rate * beta_loss_;                   \\ 2
  rthresh_ = min( rthresh_, departure_rate );            \\ 3
  r_ = rthresh_;                                         \\ 4
}
```

When the egress detects packet loss, the egress sets the control packet's action field to `LOSS`. The ingress reacts by executing Steps 2-4.

In Step 2, rate-based slow start threshold, i.e., `rthresh_`, is reduced by `beta_loss_` from the egress rate thus ensuring at least in a single bottleneck case that the aggregate arrival rate at the bottleneck reduces below the bottleneck capacity within one RTT.

Step 3 ensures that negative feedback cannot increase the rate limit. Such an increase might occur in a scenario where one trunk backs off before another causing a momentary increase in the share received by the flow that backs off later. An increase could also occur due to noise in the egress rate measurement. For example, limited clock resolution could cause truncation in the denominator of the egress rate computation leading to sudden jumps in the egress rate. This problem is particularly exacerbated on small RTT paths.

Step 4 reduces the rate-based slow-start threshold to match the point when congestion is detected. As a result, if sources feeding the traffic trunk slow down below the `rthresh_` then the traffic trunk is allowed to come up multiplicatively to the last point that was known to not congest the network.

B.1.2.4 Receive Edge Control: Decrease

```
else if ( pkt.action == DECREASE ) {                        \\ 1
  r_ = egress_rate * beta_;                                 \\ 2
  r_ = min( r_, departure_rate );                           \\ 3
  rthresh_ = r_;                                            \\ 4
}
```

Step 2 decreases from egress rate thus ensuring that for a single bottleneck with zero time-delay the backoff immediately leads to queue drain. Step 2 also has the tendency to hold the rate limit steady during congestion until the queue drains. We see no strong reason why holding the rate steady during congestion is preferable except that it allows us to easily provision drain capacity.

Step 3 ensures that negative feedback can never increase the rate limit and serves much the same purpose as Step 4 in section B.1.2.3 and Step 2 in section B.1.2.6.

Step 4 moves the `rthresh_` down to the rate limit so that the system enters linear rather than multiplicative increase when congestion ends. Step 4 mimics TCP's reduction of `ssthresh` in response to loss.

B.1.2.5 Receive Edge Control: Discard Old Control Packets

```
if (tripcount_ != pkt.tripcount){free( pkt ); return;}  \\ 1
```

Control packets reach this step only if they are "moderately" old in the sense that the control packet has arrived after a timeout and consequent control packet retransmission, but before the control packet's timestamp has been flushed from the timestamp window. A control packet delayed by a routing pathology may arrive later than the useful lifespan of the control packet's information; however, a control packet delayed by queuing likely carries useful information even if the delay is sufficient to cause a timeout. Riviera therefore defers discarding "moderately" old control packets until after updating the RTT estimate based on the control packet's timestamp, and decreasing the rate limit if the control packet declares loss or requests an early decrease. However, because an uncongested network is unlikely to delay a packet sufficient to generate a timeout, Riviera discards "moderately" old control packets without allowing an increase.

Defining an appropriate "moderate" lifespan for information is a key factor in determining the size of the `timestamp_window_size_`.

B.1.2.6 Receive Edge Control: Increase

```
if ( pkt.action == INCREASE ) {                          \\ 1
  min_rate = min( departure_rate, egress_rate );         \\ 2
  if ( min_rate >= rthresh_ - r_inc )                    \\ 3
    rthresh_ = min_rate + r_inc;                         \\ 4
  r_ = ss_increase_ * min_rate;                          \\ 5
  r_ = min( r_, rthresh_ );                              \\ 6
  r_ = max( r_, r_inc );                                 \\ 7
}
```

Step 2 ensures that the rate limit never increases by more than one MTU per RTT above the current ingress departure rate, and because of Step 2 in section B.1.2.2 the rate limit never increases by more than one MTU per RTT above the current rate limit either. Step 4 has the same motivation as Step 3 in Section B.1.2.3.

Step 3 increases the rate-based slow-start threshold only if the minimum of the departure rate and the egress rate is within `r_inc_` of the rate-based slow start threshold. Thus `rthresh_` ends the slow-start phase when the input rate reaches within the neighborhood of `rthresh_`. `r_inc` allows for one MTU worth of packetization error in the rate estimate.

Notice that in Step 4, `rthresh_` increases from the minimum of the egress rate and the ingress departure rate. Because the rate limit is bounded by `rthresh_` in Step 6, Step 4 ensures that when the bottleneck arrival rate (λ) exceeds bottleneck capacity (μ), that the rate limit stops increasing even before congestion is actually detected. Instead each RTT approximately 1 MTU per RTT is added to the backlog until congestion is detected. Because the RTT is increasing, queue growth should be slightly sublinear. We demonstrate this property in section 3.2.3.

Step 5 attempts a slow-start increase by the `ss_increase_` factor. As stated, Step 6 bounds this increase so that the rate increases linearly once the rate reaches `rthresh_`.

Step 7 ensures that the increase is at least to 1 MTU per RTT. EEC relies on a slowly adapting interval, thus it is possible for an interval to elapse in which no packets are sent resulting in a zero departure rate. Slow start increases multiplicatively by factor β from the departure rate, but any β times a zero departure rate is still zero. We call this the "stuck at zero" problem. In EEC, we avoid the "stuck at zero" problem by imposing a minimum bound on the rate limit of 1 MTU per interval. The side effect of EEC's solution to the "stuck at zero" problem is unstable queue whenever the number of trunks grow so large that the sum of the minimum rates of the active trunks exceeds the bottleneck capacity. However, in designing EEC, we did not appreciate that the "stuck at zero" problem is solved by merely bounding the minimum rate following an *increase* (i.e., Step 7). By not lower bounding the rate limit during *decrease*, we allow arbitrarily low rates during congestion thus allowing the queue to drain even when there are many trunks.

B.1.2.7 Receive Edge Control: Update Counters

```
prev_bdepartures_ = bdepartures_;                      \\ 1
prev_retc_time_ = now();                               \\ 2
insert_etc_in_next_packet_ = true;                     \\ 3
Packet::free(pkt);                                     \\ 4
```

Steps 1 and 2 update state used in the next departure rate measurement (see Step 2 section B.1.2.2).

Step 3 notifies the `send` function (see section B.1.3) that a control header should be embedded in the next data packet and thereby turn the data packet into a control packet. This also ensures that the rate of control packets never exceeds the data rate.

B.1.3 Send Packet

This function is called when the shaper decides it is time to send the next queued packet.

```
send(pkt) {
  first packet processing
  timeout processing
  insert fetc header
  send packet
}
```

B.1.3.1 Send: First Packet Processing

```
if ( first_packet_ ) {                        // 1
  first_packet_ = false;                      // 2
  prev_retc_time_ = now();                    // 3
  insert_etc_in_next_packet_ = true;          // 4
  rto_ = 2 * init_rtt_;                       // 5
  arto_ = now() + rto_;                       // 6
}
```

In Step 3, `prev_retc_time_` is set to the departure time of the first FETC simply because there has been no prior RETC arrival time.

In Step 5, the control packet retransmission timeout is set with respect to the `init_rtt_` simply because there has been no prior RTT sample. The choice of two times `init_rtt_` has been empirically chosen. Presumably `init_rtt_` has already been chosen conservatively.

B.1.3.2 Send: Timeout Processing

```
if ( now() > arto_ ) {                        // 1
  set_r( r_ * beta_loss_ );                   // 2
  rthresh_ = r_;                              // 3
  tripcount_++;                               // 4
  rto_ *= 2;                                  // 5
  arto_ = now() + rto_;                       // 6
  insert_etc_in_next_packet_ = true;          // 7
}
```

Step 1 evaluates to true when a retransmission timeout occurs. At that time, the rate limit is decreased multiplicatively from its own rate (Step 2). The rate is not decreased with respect to egress rate because no RETC has been received from which to extract an egress rate measurement.

In Step 4, we increase the trip count by one, but it is increased again in send's section B.1.3.3. The resulting increase of the trip count allows the egress to distinguish between a regular control packet and a retransmitted control packet. The egress does not take an RTT measurement at the arrival of a retransmitted control packet.

In Step 5, we exponentially backoff the time until the next retransmission timeout. Exponential increase in the RTO handles the situation when the path RTT far exceeds the current `rto_` (e.g., after a route change).

B.1.3.3 Send: Insert FETC Header

```
if ( insert_etc_in_next_packet_ )
  bdepartures_ += pkt.size;                          // 1
else {
  insert_etc_header(pkt);                            // 2
  bdepartures_ += pkt.size;                          // 3
  insert_etc_in_next_packet_ = false;                // 4
  pkt.bytecount = bdepartures_;                      // 5
  tripcount_++;                                      // 6
  set_timestamp_and_tripcount( pkt, tripcount_ );   // 7
}
```

Steps 1 increases the counts of departures for every data packet. Step 4 does the same except for control packets and includes the control header in the computation. `bdepartures_` is used in departure rate computation, and by the egress to detect loss.

Step 3 encapsulates the transport and application layer headers inside a control header. The control header contains the fields specified for the control packet in section 3.2.5 plus a field denoting the type of the encapsulated transport layer header (i.e., TCP, UDP, ICMP, IGMP). The additional header is later stripped out by the egress. If the `pkt` is already close enough to the MTU that inserting an FETC exceeds the MTU then Riviera must send a separate control packet. Sending separate control packets is not implemented in our pseudocode. Optionally Riviera could generate a separate control packet at all time that Riviera would normally insert an FETC header.

Step 5 inserts the count of departing bytes into the control header's `bytecount` field. Riviera communicates byte counts rather than packet counts to handle possible fragmentation in intervening networks. Since IPv6 does not support fragmentation, in IPv6 deployment Riviera could maintain a separate packet count that is communicated to the egress. Packet counts have the advantage that they can represent a larger amount of traffic than a byte count in the same space in the control packet.

The ingress increments the trip count field in each control header on each departure from the ingress (Steps 6 and 7). Description of trip counts appears in Section 3.2.5.

B.2 Egress

Table B.3: Egress Parameters in Equations and Pseudocode

in eq	in code	description
	`init_base_rtt_`	Initial value for base rtt.
	`min_thresh_`	congestion ends when accumulation goes below
	`max_thresh_`	congestion begins when accumulation goes above
k	`rtt_window_size_`	min rtt over window of this size

Table B.4: Egress Variables in Equations and Pseudocode

in eq	in code	description
q_i	`accumulator_`	accumulation $= in_j - \overline{out}_j$.
out_j	`barrivals_`	total received since beginning.
	`congestion_epoch_`	in congestion epoch?
ν_i	`egress_rate_`	this trunk's egress rate estimate.
$\bar{t}_j^{out}$	`expected_fetc_time_`	all bytes arriving after this time are accumulation.
	`first_packet_`	first control packet?
out_{j-1}	`prev_barrivals_`	bytes received at previous FETC.
t_{j-1}^{out}	`prev_fetc_time_`	arrival time of previous FETC.
	`tripcount_`	trip count.

The accumulator counts the number of bytes that arrive following the expected arrival time of the next FETC.

B.2.1 Initialization

```
accumulator_ = 0;
barrivals_ = 0;
```

```
  prev_barrivals_ = 0;
  prev_fetc_time_ = 0.0;
  expected_fetc_time_ 0.0;
  congestion_epoch_ = false;
  first_packet_ = true;
  tripcount_ = 0;
```

B.2.2 Receive Packet

Called on each packet arrival whether the packet bears a control header.

```
recv()
  barrivals_ += get_packet_size(p);                          // 1
  if ( now() > expected_fetc_time_ )
    accumulator_ += get_packet_size(p);                      // 2
  if ( is_edge_control_packet(p) ) {
    recv_edge_control(p);                                    // 3
    remove_etc_header(p);                                    // 4
  }
  send packet on towards destination
```

Step 1 maintains state for the later computation of egress rate. Every byte arriving after the expected FETC arrival time is counted as accumulation (Step 2).

Step 3 is described in Section B.2.3.

Step 4 removes the FETC header so that end-systems remain oblivious to the edge-to-edge control loop. If the edge-to-edge control loop uses a separate control packet then Step 4 discards the control packet.

B.2.3 Receive Edge Control Packet

```
recv_edge_control( Packet *p ) {
  first packet processing
  ignore old control packets
  retransmitted control packet processing
  rtt window
  loss
  accumulation algorithm
```

```
}
```

B.2.3.1 Receive Edge Control: First Packet Processing

```
if ( first_packet_ ) {
  first_packet_ = false;                          // 1
  prev_fetc_time_ = now() - init_base_rtt_;       // 2
  tripcount_ = p.tripcount;                       // 3
}
```

prev_fetc_time_ stores the arrival time of the previous visit from a Forward Edge Traffic Control (FETC) packet. When the first packet arrives, no such prior visit exists so Step 2 sets the prev_fetc_time_ to some reasonable default. Since the base_rtt approximates the minimum time between arrivals, it seems reasonable to set prev_fetc_time_ based on base_rtt_. Unfortunately the egress has no prior RTTs from which to set base_rtt_ so the egress uses a parameter. prev_fetc_time_ is used in setting the RTT samples in the RTT window and setting egress rate in the Reverse Edge Traffic Control (RETC) packets.

Step 3 synchronizes the ingress and egress trip counts when a traffic trunks first starts. This accomodates the possibility that a starting traffic trunk does not begin with a trip count of zero (See Section B.1.1).

B.2.3.2 Receive Edge Control: Ignore Old Control Packets

```
if ( p.tripcount < tripcount_ ) return;
```

B.2.3.3 Receive Edge Control: Retransmitted Control Packet Processing

```
if ( p.tripcount > tripcount_ ) {                       // 1
  tripcount_ = p.tripcount +1;                          // 2
  flawed_rtt = now() - prev_fetc_time_;                 // 3
  base_rtt = min_rtt_window();                          // 4
  weight = min( 1.0, flawed_rtt / base_rtt );           // 5
  curr_egress_rate = (barrivals_ - prev_barrivals_ ) * 8.0 /
    ( now() - prev_fetc_time_ );                        // 6
  egress_rate_ = (1.0 - weight) * egress_rate_
    + weight * curr_egress_rate;                        // 7
```

```
    send_edge_control( fetc, egress_rate_, DECREASE ); // 8
    expected_fetc_time_ = now() + base_rtt;            // 9
    prev_fetc_time_ = now();                           // 10
    prev_barrivals_ = barrivals_;                      // 11
    accumulator_ = 0;                                  // 12
    return;  // returns from recv_edge_control.
  }
  else tripcount_++;                                   // 13
```

In the absence of loss, each time the control packet visits the ingress, its trip count is incremented. When a control packet is lost or unduly delayed in the network, the ingress generates a new control packet. If the lost or delayed control packet has trip count i then the new control packet has trip count $i+2$. Why? If control packet i is lost before reaching the egress then the egress expects the next control packet to have trip count i. If control packet i is lost on the return trip then the egress expects the next control packet to have trip count $i+1$. By skipping $i+1$, the ingress ensures that the egress can disambiguate a newly generated control packet from the lost or delayed control packet.

In Step 1, `tripcount_` is the next expected trip count. If the packet's trip count exceeds the expected trip count this can only be because the ingress has skipped a trip count (i.e., $i+2$), one or more times. In Step 2, the egress resynchronizes the egress and ingress's trip count.

In Step 3, the egress computes the FETC interarrival time. Because the newly generated packet might be in transit at the same time as an older control packet, the interarrival time can be arbitrarily small. Specifically the interarrival time does not estimate round-trip time. To underscore this fact we call this interarrival time `flawed_rtt`. Because small averaging intervals will likely return wildly erroneous egress rate measurements, in Steps 4 through 7 we average the egress rate measured over the current `flawed_rtt` with the previous RTT's egress rate. We weight the average such that when the `flawed_rtt` is small compared to `base_rtt`, most of the average comes from the previous interval. Conversely when `flawed_rtt` is nearly as large or larger than `base_rtt` then most or all respectively of the average comes from the `flawed_rtt`'s averaging interval.

In Steps 9-12, we update counters so as to start a new RTT synchronized with the retransmitted control packet, and then return from `recv_edge_control`.

Step 13 executes on every packet arrival except when control packet loss occurs (see Steps 2-12). Step 13 increments the trip count i in the expectation that the next control packet will bear trip count $i + 1$.

B.2.3.4 Receive Edge Control: RTT Window

The code in section B.2.3.4 and subsequent sections is reached only when the control packet is *not* a retransmitted control packet.

```
rtt = now() - prev_fetc_time_;                        \\ 1
base_rtt = min_rtt_window( rtt );                      \\ 2
expected_fetc_time_ = now() + base_rtt_;               \\ 3
egress_rate_ =
  ( barrivals_ - prev_barrivals_ ) * 8.0 / rtt;        \\ 4
```

The only notable Step is 4 where we use a straight average over the last RTT to compute the egress rate. This differs from the average over two RTT's done in Step 7 in section B.2.3.2.

B.2.3.5 Receive Edge Control: Loss Window

```
if ( fetc.bcount != barrivals_ ) {                  \\ 1
  send_edge_control( fetc, egress_rate_, LOSS );    \\ 2
  barrivals_ = fetc.bcount;                         \\ 3
  prev_barrivals_ = barrivals_;                     \\ 4
  accumulator_ = 0;                                 \\ 5
  prev_fetc_time_ = now();                          \\ 6
  return;
}
```

If the number of packets sent does not equal the number of packets that arrived (see Step 1) then we assume that some packet since the last control packet arrival has been lost. This would be erroneous if a data packet and the control packet are reordered in the network. Thus we assume that packet reordering occurs only rarely. Once loss has been detected, the egress generates negative feedback with the action field set to LOSS (see Step 2).

In Steps 3, the egress resynchronizes the byte count with the ingress's byte count so that loss is not reported on subsequent control packet arrivals.

Steps 4-6 update counters to begin the next RTT.

B.2.3.6 Receive Edge Control: Accumulation Algorithm

```
if ( accumulator_ > max_thresh_ ) congestion_epoch_ = true;   \\ 1
if ( accumulator_ <= min_thresh_ ) congestion_epoch_ = false; \\ 2
if ( congestion_epoch_ )
  send_edge_control( fetc, egress_rate_, DECREASE );          \\ 3
else send_edge_control( fetc, egress_rate_, INCREASE );       \\ 4
prev_fetc_time_ = now();                                      \\ 5
prev_barrivals_ = barrivals_;                                 \\ 6
accumulator_ = 0;                                             \\ 7
```

`accumulator_` contains a count of the number of bytes that arrived after the expected FETC arrival time. As soon as the accumulation exceeds `max_thresh_` (Step 1) and then remains above `min_thresh` (Step 2), the system remains inside a congestion epoch. As long the system is in a congestion epoch, the egress sends negative feedback each RTT (Step 3). Outside any congestion epoch, the egress sends positive feedback each RTT (Step 4). The min and max thresholds introduce hysteresis. The reason for imposing hysteresis is to allow a large enough begin threshold to minimize false positives and a low enough end threshold to ensure that the queue actually drains completely in each congestion epoch or drains as close to completely as can reasonably be imposed without forcing the system to near zero utilization. As long as the system frequently drains the queue, the system maintains a good estimate of the minimum rtt within the last `rtt_window_size_` rtt samples.

B.3 Service Building Blocks

We present the building blocks one at time noting the changed steps with an asterisk following the step number appearing in the right-hand edge of each line of pseudocode.

B.3.1 Weighted Sharing Pseudocode

In section 5.2 we present three building blocks for providing weighted sharing,

1. increase differentiation

2. decrease differentiation

3. accumulation differentiation

Of the three, only increase differentiation requires modification to the pseudocode. In achieving increase differentiation, the increase step in the ingress pseudocode becomes

```
else if ( pkt.action == INCREASE ) {                    \\ 1
  min_rate = min( departure_rate, egress_rate );        \\ 2
  if ( min_rate >= rthresh_ - r_inc )                   \\ 3
    rthresh_ = min_rate + increase_weight_ * r_inc;     \\ 4*
  r_ = ss_increase_ * min_rate;                         \\ 5
  r_ = min( r_, rthresh_ );                             \\ 6
  r_ = max( r_, r_inc );                                \\ 7
}
```

Notice that only step 4 changes by including an `increase_weight_`. The `increase_weight_` parameter corresponds to w_i in Section 5.2.

To achieve weighted sharing through decrease differentiation, we need only modify each control loop's β according to (**??**).

In achieving accumulation-based weighted differentiation, each egress need only set its accumulation thresholds to w_i times the default accumulation thresholds.

B.3.2 Assured Bandwidth Allocation Pseudocode

In section 5.4 we present four building blocks for constructing an assured service.

1. accumulation differentiation
2. decrease differentiation
3. aberration corrected increase differentiation
4. aberration corrected decrease differentiation

Accumulation turns out to be the workhorse providing most of the differentiation. Accumulation requires only modification to the egress. The last three require only modifications to the ingress while accumulation differentiation requires only modification to the egress.

Accumulation differentiation pseudocode updates Section B.2.3.6 with

```
be_accumulation =
  accumulation_ * ( 1 - assured_capacity_/egress_rate_);      \\ 0*
if ( accumulation_ > weight_ * max_thresh_ ||
      be_accumulation_ > max_thresh_ )
  congestion_epoch_ = true;                                    \\ 1*
if ( accumulation_ <= weight_ * min_thresh_ &&
      be_accumulation_ <= min_thresh_ )
  congestion_epoch_ = false;                                   \\ 2*
if ( congestion_epoch_ )
  send_edge_control( fetc, egress_rate_, DECREASE );           \\ 3
else send_edge_control( fetc, egress_rate_, INCREASE );        \\ 4
prev_fetc_time_ = now();                                       \\ 5
prev_barrivals_ = barrivals_;                                  \\ 6
accumulator_ = 0;                                              \\ 7
```

Step 0 implements approximation (**??**) in Section **??**. `be_accumulation_` is equivalent to q_ib. q_ib denotes the portion of the accumulation attributed to the best-effort component of the egress rate.

Step 1 detects congestion for the best-effort component of the egress rate alone and for the total accumulation (i.e., best-effort plus assured components). When the path is not oversubscribed, the best-effort component of accumulation reaches its threshold first, and thus the total accumulation does not affect the system. When the total accumulation, `accumulation_`, exceeds a maximum weighted share, `weight_*max_thresh_`, Riviera assumes that oversubscription is occurring and backs off to prevent unbounded queue.

Step 2 detects the end of a congestion epoch only when both `accumulation_` and `be_accumulation_` reach their respective thresholds. When the path is not oversubscribed, the first antecedent almost always evaluates to true. Thus prior to oversubscription the system increases and decreases dependent only on the best-effort component of the accumulation (i.e., excluding the assured component). When oversubscription occurs, the total accumulation and the weighted thresholds determine the convergence of the system on a weighted share.

Assured bandwidth allocation through decrease differentiation mandates modification to Section B.1.2.4's pseudocode as given by

```
else if ( pkt.action == DECREASE ) {                          \\ 1
  backoff = min( egress_rate * beta_as_,
    ( egress_rate - assured_capacity_ ) * beta_
    + assured_capacity_ );                                    \\ 2*
  r_ = min( r_, departure_rate );                             \\ 3
  rthresh_ = r_;                                              \\ 4
}
```

Step 2 is equivalent to (5.45) in Section 5.4. `beta_as_` corresponds to β_{as} and `assured_capacity_` corresponds to a_i.

In Section **??** we present mechanisms for correcting aberration in the assured bandwidth allocation increase and decrease steps. The decrease aberration correction modifies Section B.1.2.4's pseudocode with

```
else if ( pkt.action == DECREASE ) {                          \\ 1
  correction = (1 - assured_capacity_/departure_rate);  \\ 2a*
  backoff = min( egress_rate * beta_as_,
    beta_ * egress_rate * correction
    + assured_capacity_ );                                    \\ 2b*
  r_ = min( r_, departure_rate );                             \\ 3
  rthresh_ = r_;                                              \\ 4
}
```

Steps 2a and 2b implement (**??**) in Section **??**.

The increase aberration correction modifies Section B.1.2.6's aberration correction with

```
else if ( pkt.action == INCREASE ) {                          \\ 1
  correction = min( departure_rate, assured_capacity_ ) \\ 2a*
    * ( 1 - egress_rate / departure_rate );
  min_rate = min( departure_rate, egress_rate );        \\ 2b*
  min_rate += correction;                                     \\ 2c*
  if ( min_rate >= rthresh_ - r_inc )                         \\ 3
```

```
    rthresh_ = min_rate + increase_weight_ * r_inc;     \\ 4
  r_ = ss_increase_ * min_rate;                         \\ 5
  r_ = min( r_, rthresh_ );                             \\ 6
  r_ = max( r_, r_inc );                                \\ 7
}
```

Steps 2a, 2b, and 2c above implement (**??**) in Section **??**.

B.3.3 Quasi Leased Line Pseudocode

In section **??** we introduce building blocks for providing traffic trunks that achieve a guaranteed bit rate. Achieving this guarantee requires only modifying the ingress. In this section we combine the pseudocode for both the assured and guaranteed bandwidth allocation building blocks using the increase and decrease assured aberration corrections. We define `guaranteed_capacity_` as the guaranteed bit rate, and define `reserved_capacity_` as the sum of the `assured_capacity_` and the `guaranteed_capacity_`.

The QLL pseudocode at the egress is given as

```
be_accumulation =
  accumulation_ * ( 1 - reserved_capacity_/egress_rate_);    \\ 0*
if ( accumulation_ > weight_ * max_thresh_ ||
      be_accumulation_ > max_thresh_ )
  congestion_epoch_ = true;                                   \\ 1
if ( accumulation_ <= weight_ * min_thresh_ &&
      be_accumulation_ <= min_thresh_ )
  congestion_epoch_ = false;                                  \\ 2
if ( congestion_epoch_ )
  send_edge_control( fetc, egress_rate_, DECREASE );          \\ 3
else send_edge_control( fetc, egress_rate_, INCREASE );       \\ 4
prev_fetc_time_ = now();                                      \\ 5
prev_barrivals_ = barrivals_;                                 \\ 6
accumulator_ = 0;                                             \\ 7
```

Step 0 now sets the `be_accumulation` according to the sum of the assured and guaranteed components.

The QLL mechanisms modify the decrease step (see Section B.1.2.4) with

```
else if ( pkt.action == DECREASE ) {                    \\ 1
  correction = (1 - reserved_capacity_/departure_rate); \\ 2a*
  backoff = min( egress_rate * beta_as_,
    beta_ * egress_rate * correction
    + reserved_capacity_ );                             \\ 2b*
  r_ = min( r_, departure_rate );                       \\ 3a
  r_ = max( r_, guaranteed_capacity_ );                 \\ 3b*
  rthresh_ = r_;                                        \\ 4
}
```

In Steps 2a and 2b, we only change the `assured_capacity_` to `reserved_capacity_`. Thus the aberration correction affects both the guaranteed and assured components of the egress rate.

In Step 3b, we add the condition that prevents the rate limit `r_` from decreasing below its guarantee. If oversubscription occurs then Step 3b can cause unstable queuing. This is why admission control is necessary to prevent the sum of the guarantees from exceeding capacity. Note that oversubscription of the assured component can still occur. The result being that the assurances reduce to a weighted share while the QLLs obtain their respective guarantees.

The QLL mechanisms modify the increase step (see Section B.1.2.6) with

```
else if ( pkt.action == INCREASE ) {                    \\ 1
  correction = min(departure_rate, reserved_capacity_)  \\ 2a*
    * ( 1 - egress_rate / departure_rate );
  min_rate = min( departure_rate, egress_rate );        \\ 2b
  min_rate += correction;                               \\ 2c
  if ( min_rate >= rthresh_ - r_inc )                   \\ 3
    rthresh_ = min_rate + increase_weight_ * r_inc;     \\ 4
  r_ = ss_increase_ * min_rate;                         \\ 5
  r_ = min( r_, rthresh_ );                             \\ 6
  r_ = max( r_, r_inc );                                \\ 7
}
```

Again we swap in `reserved_capacity` for `assured_capacity_`. Thus the aberration correction affects both the guaranteed and assured components of the egress rate.

APPENDIX C
EC's Single Path Requirement: Load Balancing, Tunneling, and NAT

Load balancing reduces congestion by spreading traffic heading in the same general direction across more than one path. If traffic within an EC trunk is spread across multiple paths then congestion points might exist on only a subset of the EC trunk's paths. Under such conditions, feedback for the EC trunk aggregate is ambiguous. For this reason, EC trunks require single-path routing. Single-path routing can be accomplished trivially using Label-Switched Paths (LSPs) or Virtual Circuits (VCs); however, as we now demonstrate in environments where LSPs and VCs are not available, tunneling or Network Address Translation (NAT) can also assure single-path routing.

Load balancing at finer granularities than individual connections adversely affects end-to-end TCP performance. To prevent multi-path routing for an aggregate, one need only make multiple connections appear as a single connection to the router's view. This is accomplished by obfuscating the two features most commonly used by routers in distinguishing individual connections: IP addresses and port numbers.

IP tunnels span between two predefined points on the network. As packets enter one end of an IP tunnel (i.e., through the tunnel's ingress), an IP header is prepended to each packet bearing the destination address of the egress and a protocol number identifying the packet as belonging to a tunnel. As packets depart the egress, the egress identifies packets bearing the tunnel IP protocol number. The egress then removes the outermost IP header for each tunneled packet and performs any other necessary functions to restore the packet to its original appearance. To intermediate routers that only look at the outermost IP header, all tunneled packets appear no different from any other packet passing from the ingress to the egress. If the tunnel uses IPSec [70] then all headers other than the tunnel's IP header may be encrypted thus assuring that intermediate routers will not identify individual connections amongst the packets within the tunnel.

Tunnels introduce significant communication and processing overhead, but since the context of Edge-to-edge Control is VPN QoS, there is a natural side-by-side deployment of IPSec tunneling and Edge-to-edge Control. As we discuss presently, NAT introduces no communication overhead.

NAT maps IP addresses allocated to a private network onto globally unique IP addresses. As packets leave a private network and pass through the NAT gateway, the gateway translates each packet's source IP address to the appropriate globally unique IP address. NAT as described by Egevang [35] assumes a 1-to-1 mapping between local and global IP addresses. NAT with Port Address Translation (PAT) is a method of connecting multiple computers to the Internet using only one globally unique IP address. PAT maps between the (internal IP address, source port) pair and a unique (gateway IP address, PAT allocated source port) pair. We are interested in NAT with PAT.

Unlike tunneling, NAT and PAT do not prepend an additional header and thus impose zero communication overhead. Though of course NAT and PAT impose processing overhead in the NAT gateway. If PAT is performed at both ends of an EC trunk then all communication in the EC trunk bears the IP addresses of the ingress and egress thus obfuscating the end system IP addresses. Though it is unlikely in any provider network, intermediate routers may identify individual connections by looking at port numbers. To prevent load balancing based on port numbers, one can modestly extend NAT such that the ingress maps the IP protocol number of each packet entering the EC trunk onto something other than TCP or UDP. If the intermediate routers do not recognize the packets as being either TCP or UDP then they will not load balance based on port numbers, because port numbers are only defined for TCP and UDP. As each packet leaves the EC trunk, the egress restores the packet's IP protocol number to its original value. This NAT extension assumes that administrators can reconfigure any intervening firewalls to not block the new IP protocol number.

APPENDIX D
ERICA Extensions and Our TCP Rate Controller

In Section 2.8.1 we discuss the basics of TCP Rate Control; however, we do not specify an algorihtm for allocating rates between TCP connections. Our rate controller is derived from the TCP rate controller developed by Karandikar *et al* [67], which is based on Charny's rate allocation algorithm[22]. Our derivative rate controller is based on Kalyanaraman *et al*'s ERICA switch algorihtm[66]. In this appendix we describe our modifications to ERICA to act as the rate allocation algorihtm for TCP Rate Control:

- We extend the rate computation to support *weighted* max-min fairness.
- In ERICA, `interval` is a parameter. Our TCP rate controller uses the time between arrivals of Riviera's control packet.
- We renamed variables that were specific to ATM (see Table D.1).
- We modify ERICA's queue control function to make use of Riviera's RTT estimate and to account for the additive increase behavior of both the end-system and Riviera.

D.1 Achieving Weighted Max-Min Fairness

To achieve *weighted* max-min fairness, we modify ERICA's rate control algorithm as shown in Figure D.1. We number the steps in the psuedocode as they are numbered in the original ERICA psuedocode [66]. We omit ERICA's step (1) since it is specific to ATM. To achieve weighted max-min fairness, the only modifications necessary appear right before step (8) and in steps (8), (10), and (11). Above step (8) our rate controller computes a TCP connection's weighted fair share of the `Trunk Target Capacity` scaled by the inverse of the overload factor. ERICA has no concept of a weighted fair share and thus omits this step. Accdordingly in steps (8) and (11), we constrain each connection to its weighted rather than its unweighted fair share. In step (10), we scale a connection's explicit rate inversely to its weight before performing the `Max` function; otherwise, all connections would have the opportunity in step (9) at the end of the next interval to come up to the explicit rate afforded to the heaviest weight and thus defeat weighted sharing.

Table D.1: Mapping of ERICA ATM Variables onto EC Equivalents

ERICA Variable	EC Equivalent	Description
`Total ABR Cap.`	r	EC traffic trunk rate limit. No ABR concept in EC.
`Link Cap.`		Variable eliminated. `Link Cap` is known to the ingress, and is used to bound the rate limit, but it is not used in the explicit rate allocation.
`ABR Input Rate`	`Trunk Input Rate`	Aggregate arrival rate at trunk's ingress.
`Target ABR Cap.`	`Trunk Target Cap.`	Trunk's rate limit scaled by queue control.
`VCShare`	`Share`	ith TCP connection's share of input rate.
CCR[VC]	$\mathtt{MeasuredRate}_\mathtt{i}$	ith TCP connection's measured rate.
`Number of Active VCs`	`Activity Level`	Count of active trunks each multiplied by it's weight.

D.2 Queue Control

When the ingress queue empties, the departure rate on the edge-to-edge control loop drags behind the rate limit. When the departure rate is not within $\frac{m}{t}$ of $rthresh$, $rthresh$ and the rate limit r do not increase. The more frequently a particular ingress's queue drains compared to other ingress edges, the more unfair the system becomes. So long as the ingress queue does not drain, TCP connections entering one traffic trunk can compete fairly against a UDP flood entering another trunk. Our queue control function attempts to maintain a target backlog that is sufficiently large to prevent the ingress queue from frequently draining.

When end-to-end control loops are substantially larger than the edge-to-edge control loop, the end-to-end control loops adapt more slowly than the edge-to-edge control loops. Thus when a large difference between end-to-end and edge-to-edge control loops exists, no queue control function can maintain a backlog without slowing the adaptation of the edge-to-edge control loop. Fortunately in an access-to-access deployment, the edge-to-edge and end-to-end control loops are probably similar enough that even a single connection adapts sufficiently fast to drive the edge-to-edge control loop at top rate.

ERICA uses a hyperbolic queue control function that does not take into account end-to-end round-trip times in determining how much to adjust rates. ERICA intentionally does not require the ATM switch to measure the round-trip time of every virtual circuit.

Initialization:

$\texttt{MaxAllocPrevious} \leftarrow \texttt{MaxAllocCurrent} \leftarrow \texttt{FairShare} \leftarrow 0$

End of Averaging Interval:

$\texttt{Trunk Target Capacity} \leftarrow \mathit{queue\_control}(r)$ (2)

$z \leftarrow \frac{\texttt{Trunk Input Rate}}{\texttt{Trunk Target Capacity}}$ (3)

$\texttt{FairShare} \leftarrow \frac{\texttt{Trunk Target Capacity}}{\texttt{Activity Level}}$ (4)

$\texttt{MaxAllocPrevious} \leftarrow \texttt{MaxAllocCurrent}$ (5)

$\texttt{MaxAllocCurrent} \leftarrow \texttt{FairShare}$ (6)

$\texttt{FOR EACH}\ i:$

$\texttt{Share} \leftarrow \frac{\texttt{Measured Rate}_\texttt{i}}{z}$ (7)

$\texttt{WeightedFairShare} \leftarrow \texttt{Weight}_\texttt{i} \times \texttt{FairShare}_\texttt{i}$

$\texttt{IF}\ (z > 1 + \delta)\ \texttt{THEN ER}_\texttt{i} \leftarrow \texttt{Max(WeightedFairShare, Share)}$ (8)

$\texttt{ELSE ER}_\texttt{i} \leftarrow \texttt{Max(MaxAllocPrevious, Share)}$ (9)

$\texttt{MaxAllocCurrent} \leftarrow \texttt{Max}(\texttt{MaxAllocCurrent}, \frac{\texttt{ER}_\texttt{i}}{\texttt{Weight}_\texttt{i}})$ (10)

$\texttt{IF (ER}_\texttt{i} > \texttt{WeightedFairShare AND Share} < \texttt{WeightedFairShare)}$ (11)

$\texttt{THEN ER}_\texttt{i} \leftarrow \texttt{WeightedFairShare}$

$\texttt{ENDFOR}$

Figure D.1: ERICA Algorithm Modified for Weighted Max-Min Fairness

However, in the context of EC the RTT is readily available from Riviera and is needed by the rate controller to perform the rate-to-window conversion. Our TCP rate controller thus adjusts the rate limit (see step 2 in Figure D.1) to achieve a target backlog within a number of RTTs. Because the edge-to-edge control loop steps up its rate on every RTT, the rate adjustment undershoots the target backlog. Thus we set the target backlog sufficiently high to account for undershoot when only one persistent connection passes through the trunk. When more than one connection passes through the trunk, the aggregate is more aggressive and is thus less likely to drain the ingress queue. Our queue control function follows:

$$\Delta\texttt{Wnd} \leftarrow q_0 - q \tag{D.1}$$

$$\overline{\Delta\texttt{Wnd}} \leftarrow (1 - \texttt{Dampen}) \times \overline{\Delta\texttt{Wnd}} + \texttt{Dampen} \times \Delta\texttt{Wnd} \tag{D.2}$$

$$\Delta r \leftarrow \overline{\Delta\texttt{Wnd}} \times \texttt{PktSz} \times 8.0/(\texttt{AdjustTime} \times \texttt{MaxRTT}) \tag{D.3}$$

$$\texttt{return}\ r + \Delta r \tag{D.4}$$

$\Delta\texttt{Wnd}$ is the change in the sum of the congestion window sizes necessary to achieve the desired backlog assuming that the queue length corresponds to the sums of the individual TCP connection's window sizes minus the bandwidth-RTPD (Round-Trip Propagation Delay) product. To handle variance in the Δ`Wnd` estimator, we exponentially average $\overline{\Delta\ \texttt{Wnd}}$. Our rate controller then computes the rate adjustment that will adjust the sum of the window sizes by $\overline{\Delta\ \texttt{Wnd}}$ packets in `AdjustTime` `MaxRTT`s. `MaxRTT` is the maximum RTT across the TCP connections passing through the trunk. In the typical access-to-access deployment, the trunk's RTT is a sufficient estimate of the maximum RTT.

D.3 Parameter Tuning

We started with the ERICA parameter settings and then empirically tuned these parameters to achieve the best performance under a scenario with 2 connections with widely differing end-to-end round-trip times, and scenarios with 1, 2, 3, 10, and 100 connections each with 60ms RTT across a traffic trunk bottlenecked at 100Mbps. Our chosen parameters are given in Table D.2. It was necessary to make δ negative because with window-based algorithms and constant bottleneck capacity (e.g., when the ingress is the bottleneck) the ingress arrval rate rarily exceeds the capacity significantly (i.e., due to ack-clocking) even when the queue approaches q_0. Thus with a positive δ, step (8) may rarily and possibly never evaluate to true. If one connection has a weight $w > 1$, steps (9) and (10) may not pull the connection with the larger weight up to its fair share. The result is that positive values for δ allow for a connection with a large weight to never achieve its weighted fair share but instead leave the weighted connection with a share corresponding to a weight somewhere between 1 and w. With $\delta < 0$, an aggregate sending exactly at capacity will be correctly forced toward weighted fairness.

The activity level decay factor is unchanged from ERICA.

The average packet size may be measured. We ran our simulations with a packet size of 1000 bytes.

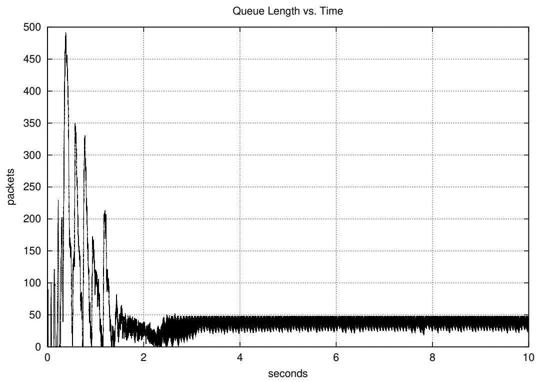
As stated in the previous section, the target backlog is chosen so that the queue usually stays above zero for a single connection and then tested under the cases given above. For the 100 connection scenario, the target was simply set to achieve "reasonable" queue lengths which we deem to be less than 500 packets during transients. We show one such simulation where the ingress is the bottleneck in Figure D.2.

Of course these simulations are simplistic. However, we have substantial anecdotal evidence that these parameter settings are reasonable from the performance results in

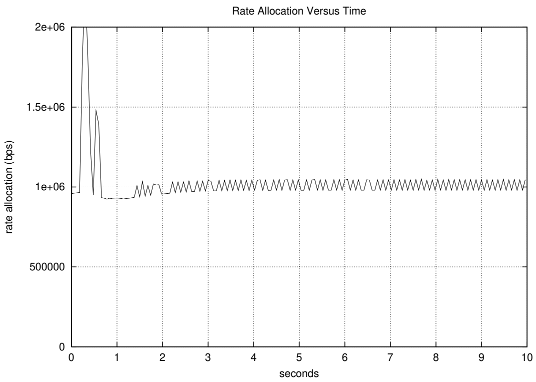
Table D.2: ERICA-based TCP Rate Control Parameters

Parameter	Value
δ	-0.01
activity level `DecayFactor`	0.9
adjustment time, `AdjustTime`	2
average packet size, `PktSz`	1000 bytes
EWMA factor, `Dampen`	0.1
target backlog, q_0	10 packets

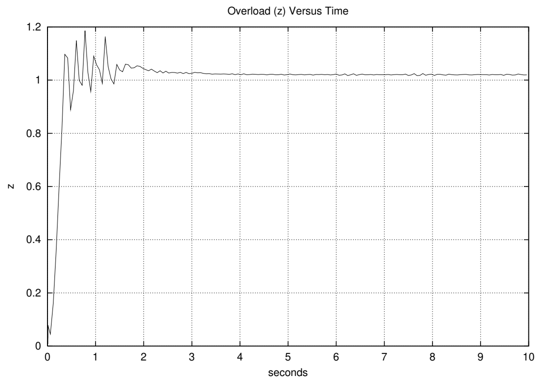
the many complex simulations scattered throughout this dissertation in which our rate controller is used. Still we recognize that a more formal parameter sensitivity analysis of our TCP rate controller is in order.



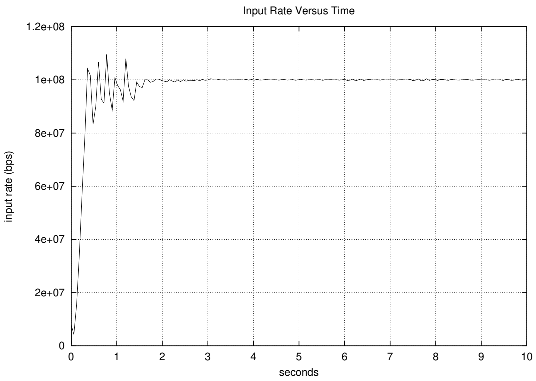
(a) Ingress (also bneck) queue



(b) Connection 0 rate allocation



(c) Overload factor



(d) Input rate

Figure D.2: TCP Rate Control with 100 Connections and 100Mbps Bottleneck