Investigating the Use of Synchronized Clocks in TCP Congestion Control

by
Michele Aylene Clark Weigle

A dissertation submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill
2003

Approved by:

Kevin Jeffay, Advisor

F. Donelson Smith, Reader

Ketan Mayer-Patel, Reader

Bert Dempsey, Reader

Sanjoy Baruah, Reader

Jasleen Kaur, Reader

© 2003
Michele Aylene Clark Weigle
ALL RIGHTS RESERVED

ABSTRACT

MICHELE AYLENE CLARK WEIGLE: Investigating the Use of Synchronized Clocks in TCP Congestion Control.
(Under the direction of Kevin Jeffay.)

In TCP Reno, the most common implementation of TCP, segment loss is the sole indicator of network congestion. TCP Reno only adjusts to congestion when segment loss has been detected in the network, thus TCP Reno's congestion control is tied to its error recovery mechanism.

My dissertation thesis is that precise knowledge of one-way transit times can be used to improve the performance of TCP congestion control. Performance is measured in terms of network-level metrics, including packet loss and average queue sizes at congested links, and in terms of application-level metrics, including HTTP response times and throughput per HTTP response.

A connection's forward path OTT is the amount of time it takes a packet to traverse all links from the sender to the receiver, including both propagation and queuing delays. Queues in routers build up before they overflow, resulting in increased OTTs. If all senders directly measure changes in OTTs and back off when the OTT indicates that congestion is occurring, congestion could be alleviated. I introduce a variant of TCP, called Sync-TCP, which uses synchronized clocks to gather a connection's OTT data. I use Sync-TCP as a platform for investigating techniques for detecting and responding to changes in OTTs.

This dissertation makes the following contributions:

- a method for measuring a flow's OTT and returning this exact timing information to the sender
- comparison of several methods for using OTTs to detect congestion
- Sync-TCP – a family of end-to-end congestion control mechanisms based on using OTTs for congestion detection
- study of standards-track TCP congestion control and error recovery mechanisms in the context of HTTP traffic

I will show that Sync-TCP provides lower packet loss, lower queue sizes, lower HTTP response times, and higher throughput per HTTP response than TCP Reno. Additionally, I will show that Sync-TCP offers performance comparable to that achieved by using router-based congestion control mechanisms. If all flows use Sync-TCP to react to increases in queuing delay, congestion could be alleviated quickly. This would result in overall shorter queues, faster response for interactive applications, and a more efficient use of network resources.

ACKNOWLEDGMENTS

First, I would like to thank my advisor Kevin Jeffay and my "co-advisor" Don Smith for being great mentors and for teaching me how to be a researcher. Thanks to the rest of my dissertation committee for feedback and discussions: Ketan Mayer-Patel, Bert Dempsey, Sanjoy Baruah, and Jasleen Kaur.

I appreciate all of the support, advice, and commiseration from former and current DiRT-sters. I also thank all of the UNC-CS grad students I have known over the years. Thanks for making Sitterson a great place to work!

Thanks to Bill Cleveland, Jin Cao, Don X. Sun, and Dong Lin of Bell Labs for developing the PackMime model and collaborating on getting PackMime into the *ns* network simulator. I thank John Heidemann and the folks at ISI for maintaining (and constantly improving) *ns*. I also wish to acknowledge the role that Jaron Lanier, of Advanced Network & Services, Inc., played in the early stages of this research by helping to formulate the problem of using GPS synchronized clocks for congestion control.

The research reported in this dissertation was supported in parts by the National Science Foundation (grants CDA-9624662, ITR-0082870, CCR-0208924, ANI 03-23648, and EIA 03-03590 and a Graduate Research Fellowship), by a National Institute of Health National Center for Research Resources Award RR02170 on Interactive Graphics for Molecular Studies and Microscopy, by a Department of Computer Science Alumni Fellowship, by the North Carolina Networking Initiative, and by the IBM, Cisco, Intel, Sun Microsystems, Aprisma, Dell, Lucent Technologies, MCNC, and Advanced Network & Services corporations.

Thank you to my parents, Mike and Jean Clark, for always believing in me and for encouraging me my whole life.

Finally, to my husband, Chris – thank you for your love, support, and encouragement. I wouldn't have made it here without you.

TABLE OF CONTENTS

LIST OF TABLES **xv**

LIST OF FIGURES **xvii**

LIST OF ABBREVIATIONS **xxvii**

1 Introduction **1**

1.1 Networking Basics 1

1.1.1 Circuit-Switching vs. Packet-Switching 2

1.1.2 Connection-Oriented vs. Connectionless 3

1.1.3 TCP Transport 4

1.1.4 HTTP 7

1.2 Network Congestion 7

1.3 TCP Congestion Control 8

1.3.1 TCP Tahoe 9

1.3.2 TCP Reno 15

1.3.3 Selective Acknowledgments 17

1.3.4 TCP NewReno 18

1.4 Sync-TCP 19

1.5 Thesis Statement 21

1.6 Evaluation 22

2 Related Work **24**

2.1 Synchronized Clocks 25

2.1.1 Network Time Protocol 25

2.1.2 Global Positioning System 26

2.2 TCP Timestamp Option 26

2.3 TCP Congestion Control 27

2.3.1 TCP Vegas 27

2.3.2 Forward Acknowledgments 28

2.3.3 TCP Santa Cruz 29

2.3.4 TCP Westwood 31

2.3.5 TCP Peach 32

2.3.6 AIMD Modifications 33

2.4 Non-TCP Congestion Control 34

2.4.1 Congestion Control Principles 35

2.4.2 Control-Theoretic Approach 35

2.4.3 General AIMD 35

2.4.4 DECbit 36

2.4.5 Delay-Based Approach 37

2.4.6 Binomial Algorithms 37

2.5 Active Queue Management 38

2.5.1 Random Early Detection 38

2.5.2 Explicit Congestion Notification 40

2.5.3 Adaptive RED 41

2.6 Analysis of Internet Delays 43

2.6.1 Queuing Time Scales 43

2.6.2 Available Bandwidth Estimation 43

2.6.3 Delays for Loss Prediction 44

2.6.4 Delays for Congestion Control 45

2.7 Summary 46

3 Sync-TCP **48**

3.1 Problem and Motivation 49

3.1.1 Network Congestion 50

3.1.2 TCP Congestion Control 50

3.2 Goals 50

3.3 One-Way Transit Times 51

3.4 Sync-TCP Timestamp Option 51

3.5 Congestion Detection 53

3.5.1 SyncPQlen 56

3.5.2 SyncPMinOTT 58

3.5.3 SyncTrend 60

3.5.4 SyncAvg 63

3.5.5 SyncMix 66

3.5.6 Summary 68

3.6 Congestion Reaction 70

3.6.1 SyncPcwnd 71

3.6.2 SyncMixReact 71

3.7 Summary 73

4 Empirical Evaluation **74**

4.1 Methodology 75

4.1.1 Network Configuration 75

4.1.2 Traffic Generation 77

4.1.3 Simulation Running Time 81

4.1.4 Evaluation Metrics 82

4.2 FTP Evaluation 86

4.2.1 Single Bottleneck 86

4.2.2 Multiple Bottlenecks 97

4.2.3 Summary 99

4.3 HTTP Evaluation 101

4.3.1 Single Bottleneck 101

4.3.2 Multiple Bottlenecks 157

4.4 Summary 178

5 Summary and Conclusions **180**

5.1 Acquiring and Communicating One-Way Transit Times 183

5.2 Congestion Detection 184

5.3 Congestion Reaction 184

5.4 Sync-TCP 185

5.4.1 Evaluation 185

5.4.2 Results 186

5.5 Standards-Track TCP Evaluation 187

5.6 Future Work 188

5.6.1 Further Analysis 188

5.6.2 Extensions to Sync-TCP 190

5.6.3 Additional Uses for Synchronized Clocks in TCP 191

5.7 Summary 191

A Methodology **192**

A.1 Network Configuration 193

A.1.1 Evaluating TCP 193

A.1.2 Impact of Two-Way Traffic and Delayed ACKs 195

A.2 HTTP Traffic Generation 196

A.2.1 PackMime 196

A.2.2 Levels of Offered Load 199

A.2.3 Characteristics of Generated HTTP Traffic 202

A.3 Simulation Run Time 203

A.4 Simulator Additions 209

A.4.1 PackMime 209

A.4.2 DelayBox 211

B Standards-Track TCP Evaluation **214**

B.1 Methodology 215

B.1.1 Experimental Setup 216

B.1.2 HTTP Traffic Generation 217

B.1.3 Levels of Offered Load and Data Collection 217

B.1.4 Queue Management 218

B.1.5 Performance Metrics 218

B.2 Results 219

B.2.1 TCP Reno with Drop-Tail Queuing 224

B.2.2 TCP Reno with Adaptive RED Queuing 224

B.2.3 TCP Reno with Adaptive RED Queuing and ECN Marking 231

B.2.4 TCP SACK 236

B.2.5 Drop-Tail vs. Adaptive RED with ECN Marking 236

B.2.6 Performance for Offered Loads Less Than 80% 254

B.2.7 Summary 254

B.3 Tuning Adaptive RED 255

B.4 Summary 270

C Experiment Summary **277**

BIBLIOGRAPHY **285**

LIST OF TABLES

1.1 Adjustments to the Congestion Window Based on the Signal from the Congestion Detection Mechanism 21

3.1 SyncMix Average Queuing Delay Weighting Factors 67

3.2 SyncMix Return Values 68

4.1 HTTP Cross-Traffic for Multiple Bottleneck Experiments 79

4.2 FTP Round-Trip Times 80

4.3 Aggregate FTP Statistics, Single Bottleneck 86

4.4 FTP, Single Bottleneck, Sync-TCP(Pcwnd), Per-Flow Summary 92

4.5 FTP, Single Bottleneck, Sync-TCP(MixReact), Per-Flow Summary 93

4.6 Aggregate FTP Statistics (Utilization and Queue Size) 97

4.7 Aggregate FTP Statistics (Packet Loss) 97

4.8 FTP, Multiple Bottlenecks, Sync-TCP(Pcwnd), Per-Flow Summary 99

4.9 FTP, Multiple Bottlenecks, Sync-TCP(MixReact), Per-Flow Summary 100

4.10 Parameters for “tuned” Adaptive RED 102

4.11 Packet Loss 106

4.12 50% End-to-End Load, 75% Total Load, 2 Bottlenecks 158

4.13 60% End-to-End Load, 75% Total Load, 2 Bottlenecks 158

4.14 70% End-to-End Load, 75% Total Load, 3 Bottlenecks 159

4.15 50% End-to-End Load, 90% Total Load, 2 Bottlenecks 163

4.16 60% End-to-End Load, 90% Total Load, 2 Bottlenecks 163

4.17 70% End-to-End Load, 90% Total Load, 3 Bottlenecks 164

4.18 80% End-to-End Load, 90% Total Load, 3 Bottlenecks 164

4.19 85% End-to-End Load, 90% Total Load, 3 Bottlenecks 164

4.20 50% End-to-End Load, 105% Total Load, 2 Bottlenecks 171

4.21 60% End-to-End Load, 105% Total Load, 2 Bottlenecks 171

4.22 70% End-to-End Load, 105% Total Load, 3 Bottlenecks 172

4.23 80% End-to-End Load, 105% Total Load, 3 Bottlenecks 172

4.24 85% End-to-End Load, 105% Total Load, 3 Bottlenecks 172

4.25 90% End-to-End Load, 105% Total Load, 3 Bottlenecks 173

4.26 95% End-to-End Load, 105% Total Load, 3 Bottlenecks 173

4.27 100% End-to-End Load, 105% Total Load, 3 Bottlenecks 173

A.1 Performance of One-Way FTP Traffic and Two-Way FTP Traffic Without Delayed ACKs 195

A.2 Performance of Two-Way FTP Traffic Without and Without Delayed ACKs 196

A.3 HTTP End-to-end Calibration 202

A.4 HTTP Cross-Traffic Calibration 202

B.1 List of Experiments 216

B.2 Summary of Labels and Abbreviations 219

B.3 Parameters for “tuned” Adaptive RED 269

C.1 List of HTTP Experiments 283

LIST OF FIGURES

1.1 Circuit-Switching Example 2

1.2 Packet-Switching Example 3

1.3 TCP Send Window 5

1.4 TCP Sending Rate 5

1.5 TCP Drop and Recovery 6

1.6 Example Web Page 8

1.7 TCP Timeline 9

1.8 Slow Start 11

1.9 Slow Start with Delayed ACKs 12

1.10 Congestion Avoidance 13

1.11 Congestion Avoidance with Delayed ACKs 13

1.12 TCP Tahoe Fast Retransmit 15

1.13 TCP Reno Fast Retransmit and Fast Recovery 16

1.14 Network Topology 22

2.1 RED States 39

2.2 RED Initial Drop Probability (p_b) 40

2.3 Adaptive RED Initial Drop Probability (p_b) 42

2.4 Adaptive RED States 42

3.1 TCP Header and Sync-TCP Timestamp Option 52

3.2 Sync-TCP Example 53

3.3 Network Topology for FTP Experiments 54

3.4 Queue Size at Bottleneck Router 55

3.5 SyncPQlen Threshold Startup 56

3.6 SyncPQlen Congestion Detection 58

3.7 SyncPMinOTT Congestion Detection 59

3.8 SyncTrend Congestion Detection 62

3.9 SyncTrend Congestion Detection - boxed region 62

3.10 SyncAvg Congestion Detection 64

3.11 SyncAvg Congestion Detection - boxed region 65

3.12 Boxed Regions from SyncTrend and SyncAvg 65

3.13 SyncMix Congestion Detection 69

3.14 SyncMix Congestion Detection - boxed region 69

3.15 Boxed Regions from SyncTrend, SyncAvg, and SyncMix 70

3.16 SyncMixReact Congestion Reactions 72

4.1 Simplified 6Q Parking Lot Topology 76

4.2 FTP Goodput as Running Time Increases 82

4.3 Response Time CDF Example 84

4.4 Response Time CCDF Example 85

4.5 Packet Loss, Single Bottleneck 87

4.6 Queue Size and Queuing Delay, Single Bottleneck 89

4.7 Goodput, Single Bottleneck 91

4.8 Congestion Window and Drops, Flow 2, 97 ms RTT, Single Bottleneck 94

4.9 Congestion Window and Drops, Flow 14, 17 ms RTT, Single Bottleneck . . . 95

4.10 Congestion Window and Drops, Flow 6, 21 ms RTT, Single Bottleneck 96

4.11 Packet Loss and Goodput, Multiple Bottlenecks 98

4.12 HTTP Summary Stats, Single Bottleneck (drops, queue size, average response time) 103

4.13 HTTP Summary Stats, Single Bottleneck (response time, goodput per response, flows with drops) 104

4.14 HTTP Summary Stats, Single Bottleneck (throughput, goodput, utilization) 105

4.15 Summary Response Time CDFs (Uncongested, Reno, Sync-TCP(Pcwnd)) . . 107

4.16 Summary Response Time CDFs (Sync-TCP(MixReact), SACK-ECN-5ms, SACK-ECN-tuned) 108

4.17 Response Time CDFs, Single Bottleneck, Light Congestion 109

4.18 Response Time CDFs, Single Bottleneck, Medium Congestion 110

4.19 Response Time CDFs, Single Bottleneck, Heavy Congestion 111

4.20 Response Time CCDFs, Single Bottleneck, Light Congestion 114

4.21 Response Time CCDFs, Single Bottleneck, Medium Congestion 115

4.22 Response Time CCDFs, Single Bottleneck, Heavy Congestion 116

4.23 Queue Size CDFs, Single Bottleneck, Light Congestion 118

4.24 Queue Size CDFs, Single Bottleneck, Medium Congestion 119

4.25 Queue Size CDFs, Single Bottleneck, Heavy Congestion 120

4.26 Response Size CCDFs, Single Bottleneck, Light Congestion 122

4.27 Response Size CCDFs, Single Bottleneck, Medium Congestion 123

4.28 Response Size CCDFs, Single Bottleneck, Heavy Congestion 124

4.29 Response Time CDFs for Responses Under 25 KB, Light Congestion 126

4.30 Response Time CDFs for Responses Under 25 KB, Medium Congestion . . . 127

4.31 Response Time CDFs for Responses Under 25 KB, Heavy Congestion 128

4.32 Response Time CCDFs for Responses Under 25 KB, Light Congestion 129

4.33 Response Time CCDFs for Responses Under 25 KB, Medium Congestion . . 130

4.34 Response Time CCDFs for Responses Under 25 KB, Heavy Congestion . . . 131

4.35 Response Time CDFs for Responses Over 25 KB, Light Congestion 132

4.36 Response Time CDFs for Responses Over 25 KB, Medium Congestion 133

4.37 Response Time CDFs for Responses Over 25 KB, Heavy Congestion 134

4.38 Response Time CCDFs for Responses Over 25 KB, Light Congestion 135

4.39 Response Time CCDFs for Responses Over 25 KB, Medium Congestion . . . 136

4.40 Response Time CCDFs for Responses Over 25 KB, Heavy Congestion 137

4.41 RTT CCDFs, Light Congestion 139

4.42 RTT CCDFs, Medium Congestion 140

4.43 RTT CCDFs, Heavy Congestion 141

4.44 RTT CCDFs for Responses Under 25 KB, Light Congestion 143

4.45 RTT CCDFs for Responses Under 25 KB, Medium Congestion 144

4.46 RTT CCDFs for Responses Under 25 KB, Heavy Congestion 145

4.47 RTT CCDFs for Responses Over 25 KB, Light Congestion 146

4.48 RTT CCDFs for Responses Over 25 KB, Medium Congestion 147

4.49 RTT CCDFs for Responses Over 25 KB, Heavy Congestion 148

4.50 60% Load, Congestion Window and Unacknowledged Segments 150

4.51 85% Load, Congestion Window and Unacknowledged Segments 151

4.52 Sync-TCP(Pcwnd) - *cwnd* and Queuing Delay at 60% Load 152

4.53 Sync-TCP(MixReact) - *cwnd* and Queuing Delay at 60% Load 153

4.54 Sync-TCP(MixReact) - *cwnd* and Queuing Delay at 85% Load 155

4.55 Difference in Mean Response Duration 156

4.56 Response Time CDFs, Multiple Bottlenecks, 75% Total Load 160

4.57 Response Time CDFs, Total 75% Load 161

4.58 Response Time CCDFs, Total 75% Load 162

4.59 Response Time CDFs, Multiple Bottlenecks, 90% Total Load 165

4.60 Response Time CDFs, Total 90% Load, Light End-to-End Congestion 167

4.61 Response Time CDFs, Total 90% Load, Medium End-to-End Congestion . . . 168

4.62 Response Time CCDFs, Total 90% Load, Light End-to-End Congestion . . . 169

4.63 Response Time CCDFs, Total 90% Load, Medium End-to-End Congestion 170

4.64 Response Time CDFs, Multiple Bottlenecks, 105% Total Load 174

4.65 Response Time CCDFs, Total 105% Load, Light Congestion 175

4.66 Response Time CCDFs, Total 105% Load, Medium Congestion 176

4.67 Response Time CCDFs, Total 105% Load, Heavy Congestion 177

5.1 Simplified 6Q Parking Lot Topology 185

A.1 Simplified 6Q Parking Lot Topology 195

A.2 HTTP Round-Trip Times 197

A.3 CDF of HTTP Request Sizes 198

A.4 CDF of HTTP Response Sizes 198

A.5 CCDF of HTTP Response Sizes 199

A.6 New Connections Started Per Second 200

A.7 Offered Load Per Second 201

A.8 Offered Load Per Second on a Congested Network 201

A.9 Packet Arrivals 203

A.10 Mean Response Size 205

A.11 90th Quantiles of Response Size 206

A.12 Coefficient of Variation of 90th Quantiles of Response Size 206

A.13 99th Quantiles of Response Size 207

A.14 HTTP Warmup Interval 208

B.1 Simulated Network Environment 216

B.2 Summary Statistics for TCP Reno (drops, flows with drops, queue size) 220

B.3 Summary Statistics for TCP Reno (utilization, throughput, response time) 221

B.4 Summary statistics for TCP SACK (drops, flows with drops, queue size) 222

B.5 Summary Statistics for TCP SACK (utilization, throughput, response time) 223

B.6 Response Time CDFs, TCP Reno with Drop-Tail Queuing, Light Congestion 225

B.7 Response Time CDFs, TCP Reno with Drop-Tail Queuing, Medium Congestion 226

B.8 Response Time CDFs, TCP Reno with Drop-Tail Queuing, Heavy Congestion 227

B.9 Response Time CDFs, TCP Reno with Drop-Tail Queuing and with Adaptive RED Queuing, Light Congestion 228

B.10 Response Time CDFs, TCP Reno with Drop-Tail Queuing and with Adaptive RED Queuing, Medium Congestion 229

B.11 Response Time CDFs, TCP Reno with Drop-Tail Queuing and with Adaptive RED Queuing, Heavy Congestion 230

B.12 Response Time CDFs, TCP Reno with “original” RED Queuing and Adaptive RED Queuing 232

B.13 Response Time CDFs, TCP Reno with Adaptive RED Queuing and with Adaptive RED Queuing and ECN Marking, Light Congestion 233

B.14 Response Time CDFs, TCP Reno with Adaptive RED Queuing and with Adaptive RED Queuing and ECN Marking, Medium Congestion 234

B.15 Response Time CDFs, TCP Reno with Adaptive RED Queuing and with Adaptive RED Queuing and ECN Marking, Heavy Congestion 235

B.16 Response Time CDFs, TCP Reno with Adaptive RED Queuing and ECN Marking, One-Way and Two-Way Traffic, Medium Congestion 237

B.17 Response Time CDFs, TCP Reno with Adaptive RED Queuing and ECN Marking, One-Way and Two-Way Traffic, Heavy Congestion 238

B.18 Response Time CDFs, TCP Reno and TCP SACK with Drop-Tail Queuing, Light Congestion 239

B.19 Response Time CDFs, TCP Reno and TCP SACK with Drop-Tail Queuing, Medium Congestion 240

B.20 Response Time CDFs, TCP Reno and TCP SACK with Drop-Tail Queuing, Heavy Congestion 241

B.21 Response Time CDFs, TCP Reno and TCP SACK with Adaptive RED Queuing, Light Congestion 242

B.22 Response Time CDFs, TCP Reno and TCP SACK with Adaptive RED Queuing, Medium Congestion 243

B.23 Response Time CDFs, TCP Reno and TCP SACK with Adaptive RED Queuing, Heavy Congestion 244

B.24 Response Time CDFs, TCP Reno and TCP SACK with Adaptive RED Queuing and ECN Marking, Light Congestion 245

B.25 Response Time CDFs, TCP Reno and TCP SACK with Adaptive RED Queuing and ECN Marking, Medium Congestion 246

B.26 Response Time CDFs, TCP Reno and TCP SACK with Adaptive RED Queuing and ECN Marking, Heavy Congestion 247

B.27 Response Time CDFs, TCP Reno with Drop-Tail Queuing and with Adaptive RED Queuing and ECN Marking, Light Congestion . . . 248

B.28 Response Time CDFs, TCP Reno with Drop-Tail Queuing and with Adaptive RED Queuing and ECN Marking, Medium Congestion . . 249

B.29 Response Time CDFs, TCP Reno with Drop-Tail Queuing and with Adaptive RED Queuing and ECN Marking, Heavy Congestion . . . 250

B.30 Response Time CDFs, TCP Reno with Drop-Tail Queuing and with Adaptive RED Queuing and ECN Marking, Light Congestion . . . 251

B.31 Response Time CDFs, TCP Reno with Drop-Tail Queuing and with Adaptive RED Queuing and ECN Marking, Medium Congestion . . 252

B.32 Response Time CDFs, TCP Reno with Drop-Tail Queuing and with Adaptive RED Queuing and ECN Marking, Heavy Congestion . . . 253

B.33 Response Time CDFs, TCP Reno with Drop-Tail Queuing and with Adaptive RED Queuing, All Levels of Congestion 256

B.34 Response Time CDFs, TCP Reno with Adaptive RED Queuing and with Adaptive RED Queuing and ECN Marking, All Levels of Congestion 257

B.35 Response Time CDFs, TCP SACK with Drop-Tail Queuing and with Adaptive RED Queuing, All Levels of Congestion 258

B.36 Response Time CDFs, TCP SACK with Adaptive RED Queuing and with Adaptive RED Queuing and ECN Marking, All Levels of Congestion 259

B.37 Response Time CDFs, TCP SACK with Drop-Tail Queuing and Adaptive RED Queuing, Light Congestion 260

B.38 Response Time CDFs, TCP SACK with Drop-Tail Queuing and Adaptive RED Queuing, Medium Congestion 261

B.39 Response Time CDFs, TCP SACK with Drop-Tail Queuing and Adaptive RED Queuing, Heavy Congestion 262

B.40 Response Time CDFs, TCP SACK with Adaptive RED Queuing and with Adaptive RED Queuing and ECN Marking, Light Congestion 263

B.41 Response Time CDFs, TCP SACK with Adaptive RED Queuing and with Adaptive RED Queuing and ECN Marking, Medium Congestion 264

B.42 Response Time CDFs, TCP SACK with Adaptive RED Queuing and with Adaptive RED Queuing and ECN Marking, Heavy Congestion 265

B.43 Response Time CDFs, TCP SACK with Drop-Tail Queuing and with Adaptive RED Queuing and ECN Marking, Light Congestion . . . 266

B.44 Response Time CDFs, TCP SACK with Drop-Tail Queuing and with Adaptive RED Queuing and ECN Marking, Medium Congestion . . 267

B.45 Response Time CDFs, TCP SACK with Drop-Tail Queuing and with Adaptive RED Queuing and ECN Marking, Heavy Congestion . . . 268

B.46 Sync-TCP Evaluation Topology 269

B.47 Summary Statistics for TCP SACK with Adaptive RED Queuing and ECN Marking (queue size, drops, response time) 271

B.48 Summary Statistics for TCP SACK with Adaptive RED Queuing and ECN Marking (utilization, throughput, goodput) 272

B.49 Response Time CDFs, TCP SACK with Adaptive RED Queuing and ECN Marking, Medium Congestion 273

B.50 Response Time CDFs, TCP SACK with Adaptive RED Queuing and ECN Marking, Heavy Congestion 274

B.51 Response Time CCDFs, TCP SACK with Adaptive RED Queuing and ECN Marking, Medium Congestion 275

B.52 Response Time CCDFs, TCP SACK with Adaptive RED Queuing and ECN Marking, Heavy Congestion 276

LIST OF ABBREVIATIONS

ACK	acknowledgment
AIMD	additive-increase / multiplicative-decrease
AQM	Active Queue Management
ARED	Adaptive RED
BDP	bandwidth-delay product
CDF	cumulative distribution function
CCDF	complementary cumulative distribution function
cwnd	congestion window
ECN	Explicit Congestion Notification
FIFO	first-in, first-out
GPS	Global Positioning System
NTP	Network Time Protocol
OTT	one-way transit time
RED	Random Early Detection
RTO	retransmission timeout
RTT	round-trip time
SACK	selective acknowledgment
ssthresh	slow start threshold

Chapter 1

Introduction

Computers on the Internet communicate with each other by passing messages divided into units called *packets*. Since each computer does not have wires connecting it to every other computer on the Internet, packets are passed along from source to destination through intermediate computers, called *routers*. Each packet contains the address of the source and destination so that the routers know where to send the packet and so the destination knows who sent the packet. When packets arrive at a router faster than they can be processed and forwarded, the router saves the incoming packets into a queue. *Network congestion* results when the number of packets in this queue remains large for a sustained period of time. The longer the queue, the longer that packets at the end of the queue must wait before being transferred, thus, increasing their delay. Network congestion can cause the queue to completely fill. When this occurs, incoming packets are dropped and never reach their destination. Protocols that control transport of data, such as TCP, must handle these packet loss situations.

This dissertation is concerned with detecting and reacting to network congestion before packets are dropped, and thus, improving performance for Internet applications, such as email, file transfer, and browsing the Web. Since web traffic represents a large portion of the traffic on the Internet, I am especially interested in how congestion affects web traffic. In the rest of this chapter, I will introduce in more detail the workings of the Internet, including TCP, the transport protocol used by the web. I will also introduce Sync-TCP, a delay-based end-to-end congestion control protocol that takes advantage of synchronized clocks on end systems to detect and react to network congestion.

1.1 Networking Basics

The focus of my work is on the performance of data transfer on the Internet in the context of web traffic. The transfer of web pages in the Internet is a *connection-oriented* service that runs over a *packet-switched* network, using *TCP* as its transfer protocol. In the remainder of

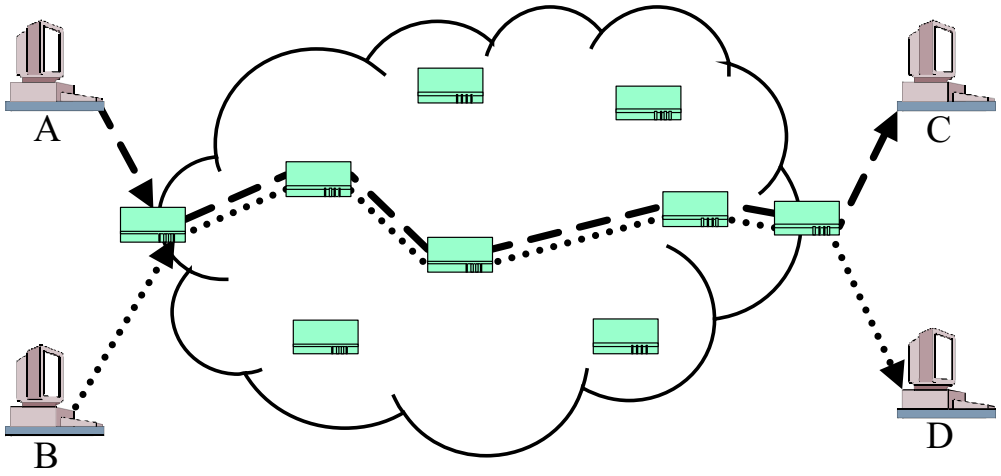


Figure 1.1: Circuit-Switching Example

this section, I will describe a packet-switched network (and also a circuit-switched network), connection-oriented communication (and also connectionless communication), and TCP.

1.1.1 Circuit-Switching vs. Packet-Switching

A communications network can either be based on *circuit-switching* or *packet-switching*. Circuit switching requires that a reservation be made through the network before data transfer can begin. One example of a circuit-switched network is the telephone system. Since there is finite capacity in the network, a circuit-switched network can only support a fixed number of connections simultaneously (depending on the network capacity and the amount of capacity reserved by each connection). If a reservation has been made and a connection has been established, but no data is being transferred, no other connection can use the idle capacity that has been reserved for another connection. An example of circuit-switching is shown in Figure 1.1. Computer A has a circuit reserved to computer C, and computer B has a circuit reserved to computer D. Both of these circuits go through the same routers, but neither reservation is for the entire capacity of the path, so the path can be shared by both. Multiplexing (allowing multiple connections to share the same network resources) in a circuit-switched network is achieved through dividing the capacity into time slots (time-division multiple access) or into frequencies (frequency-division multiple access). No matter the method of multiplexing, if one connection is idle, no other connection can use that idle capacity while the reservation is in effect. Additionally, because resources are pre-allocated and traffic is guaranteed not to exceed the forwarding capacity, no queue will build up and there will be no congestion.

Packet-switching is the method of data transfer that is used in the Internet. In packet-switching, there is no reservation made, but also no guarantee on the timeliness of data delivery. The network will make its "best effort" to transfer data quickly. Data to be trans-

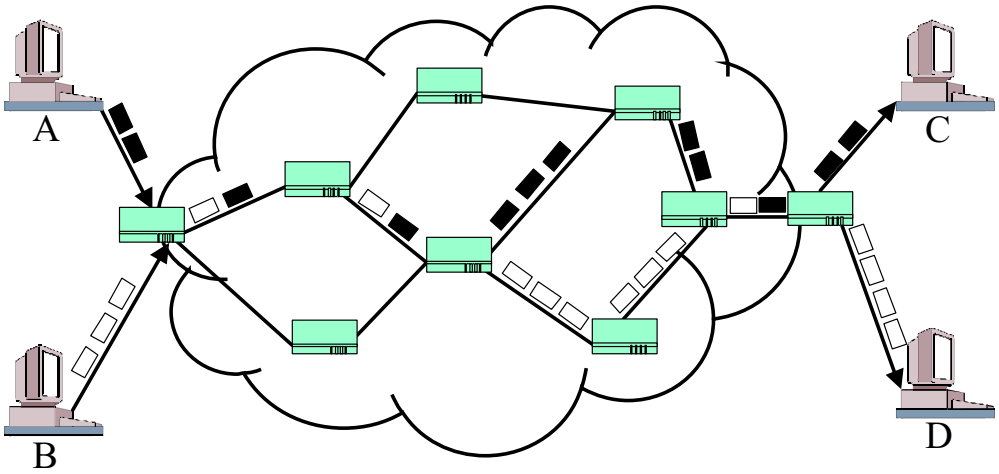


Figure 1.2: Packet-Switching Example

ferred is divided into pieces, called packets. Each packet contains the address of the source and the address of the destination. Intermediate computers in the network, called *routers*, use this information to pass the packets through the network from source to destination. Since there is no reservation, packets from many different sources can be interleaved. This is called *statistical multiplexing*, and it allows an arbitrary number of computers to exchange information at the same time. An example of packet-switching is shown in Figure 1.2. As in Figure 1.1, computer A is sending data to computer C, and computer B is sending data to computer D. The data from computer A is shown in the black packets, and the data from computer B is shown in the white packets. With a packet-switched network, packets from various connections share the capacity of the network. Unlike a circuit-switched network, when data is not being transferred between two computers, packets from other connections can take up that idle capacity. However, if the rate of arrivals exceeds the forwarding capacity at a router, a queue will build up, and congestion and packet drops can occur.

1.1.2 Connection-Oriented vs. Connectionless

There are two major modes by which computers communicate: *connection-oriented* and *connectionless*. Connection-oriented communication can be thought of as making a telephone call, while connectionless communication is more related to sending a letter through the mail. In a connection-oriented system, the sender of the information first initiates communication (or, going along with the telephone analogy, dials a phone number). The receiver of the information acknowledges that the sender has initiated communication (or, answers the phone and says "Hello?"). The sender then responds to the acknowledgment (or, says "Hi, this is Jane Doe"). After this exchange, called handshaking, the information can be transferred from the sender to the receiver.

In a connectionless system, there is no handshaking. Information is sent from the sender to the receiver with no acknowledgment, much like the process of writing a letter and putting it into a mailbox. Like the postal service, in connectionless communication, the sender does not know when, or even if, the information was received by the intended party.

The Internet provides a connectionless service, UDP, and a connection-oriented service, TCP. TCP is used for the transfer of many applications, including email, data files, and web pages. I will focus on this connection-oriented service since my dissertation is directly related to TCP.

1.1.3 TCP Transport

TCP is the Internet's connection-oriented transport service. Since the Internet is packet-switched, TCP divides information to be sent into TCP *segments*, which are then packaged into packets, containing the addresses of the source and destination. TCP promises reliable, in-order delivery from source to destination. To ensure reliable delivery, all segments are acknowledged by the receiver. Every segment from the sender is sent with a sequence number identifying which bytes the segment contains. Acknowledgments (ACKs) in TCP are cumulative and acknowledge the in-order receipt of all bytes through the sequence number returned in the ACK (*i.e.*, the ACK identifies the sequence number of the next in-order byte the receiver expects to receive from the sender). A TCP sender uses these ACKs to compute the *send window*, which roughly keeps track of how much data has been sent but not yet acknowledged. To provide in-order delivery, the TCP receiver must buffer any segments that are received out-of-order until gaps in the sequence number space have been filled.

Messages sent by TCP can be arbitrarily large, but the buffer that a receiver has for holding segments is finite. In each ACK, a TCP receiver includes the amount of space it has left in its buffer, or the *receiver's advertised window*. Upon receiving this window update, a TCP sender will not allow more than that amount of data to be unacknowledged in the network (*i.e.*, if there is no room in the receiver's window, no new data will be sent). This is how TCP performs *flow control*. The goal of flow control is to make sure that the sender does not overrun the receiver's buffer. For flow control, the send window cannot be larger than the receiver's window. If a TCP sender wants to send a message that is divided into 6 segments and the receiver's window is 3 segments, then the sender initially can send segments 1-3, as in Figure 1.3. When the ACK for segment 1 returns (and indicates that the receiver's window is still 3 segments), the sender can send segment 4 (because now segments 2-4 are unacknowledged). This process continues until all 6 segments have been sent.

The size of the send window drives the rate at which a TCP sender transfers data. To illustrate this, the previous example is shown in a different manner in Figure 1.4. The parallel lines represent the sender and receiver, and time advances down the figure. Again, the receiver's window (and thus, the send window) is 3 segments. The time between sending a

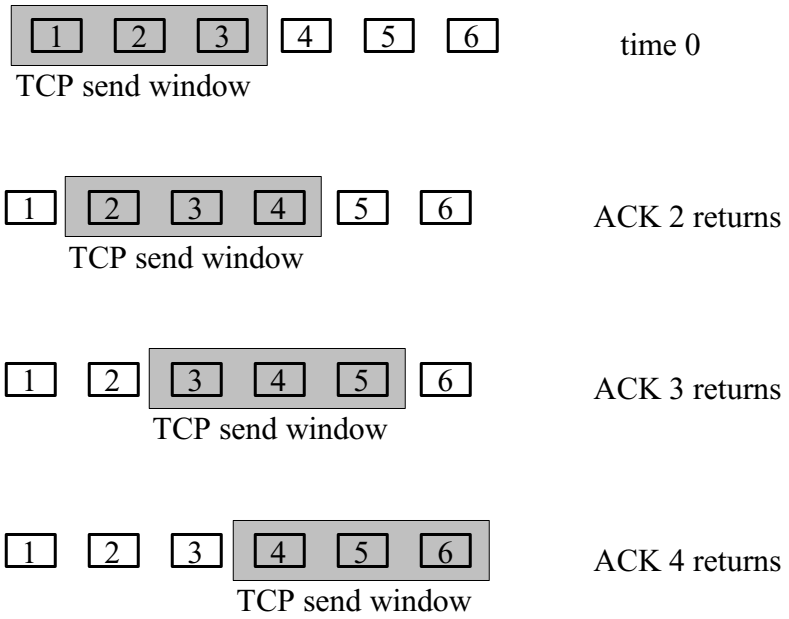


Figure 1.3: TCP Send Window

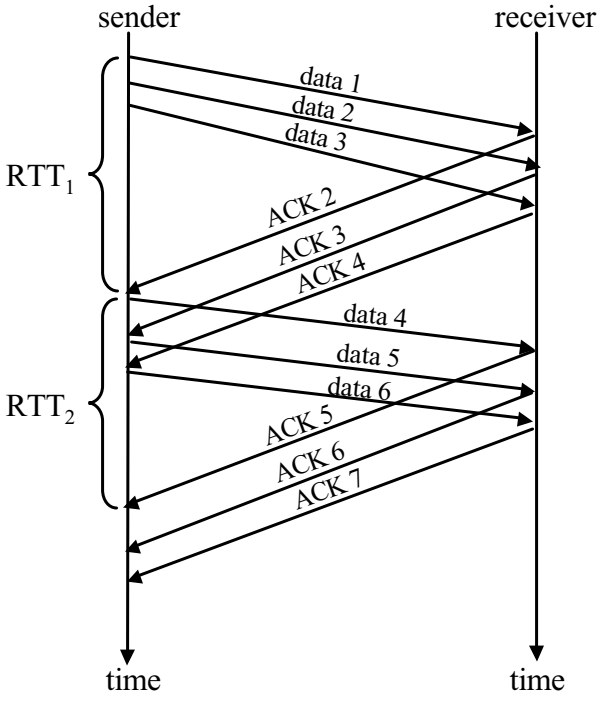


Figure 1.4: TCP Sending Rate

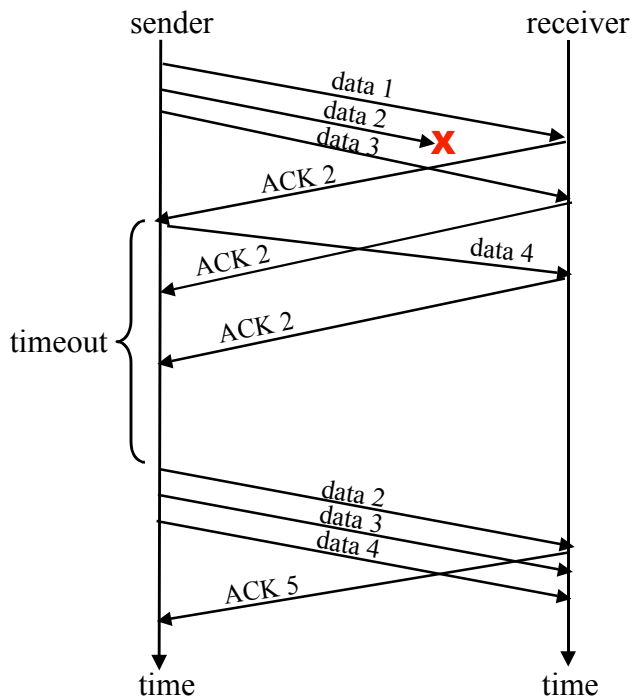


Figure 1.5: TCP Drop and Recovery

segment and receiving an ACK for that segment is called the round-trip time (RTT). With an initial window of 3 segments, TCP will send the first 3 segments out back-to-back. The ACKs for these segments will also arrive closely-spaced. RTT_1 represents the RTT of segment 1, and RTT_2 represents the RTT of segment 4 (sent after the ACK for segment 1 was received). The sending rate of this transfer is 3 segments per RTT, since, on average, 3 segments are sent every RTT. More generally, the rate of a TCP sender can be represented in the following equation:

$$rate = w/RTT,$$

where w is the window size.

Figure 1.5 shows an example of a segment drop and recovery. This example begins like Figure 1.4, with a window of 3 segments. Segment 2 is dropped by the network. The receipt of segment 3 causes the receiver to send a *duplicate ACK* requesting segment 2. When the first ACK requesting segment 2 returns (caused by the receipt of segment 1), the TCP sender sets a timer. In this example, the timer expires before an ACK requesting new data (*i.e.*, something other than segment 2) arrives, so the sender assumes that segment 2 was lost. Since the window size is 3 segments, TCP sends out 3 segments starting at segment 2 (the oldest

unacknowledged segment). This demonstrates the "Go-Back-N" error recovery approach. Once segment 2 has been received, an ACK requesting segment 5 is sent.

Delayed ACKs

Originally, TCP receivers sent an ACK immediately whenever a segment was received. With two-way traffic, delayed ACKs allow a receiver to wait until the receiver has data to send back to the sender and piggy-back the ACK onto the data segment the receiver is transmitting to the sender [Cla82]. If ACKs are delayed for too long, the sender might suspect that segment loss has occurred. To bound this delay, delayed ACKs operate on a timer, usually set to 200 ms. If no data segment is sent from receiver to sender (with the piggy-backed ACK) before the timer expires and there is unacknowledged data, an ACK is sent immediately. There is also an outstanding ACK threshold, usually set to two segments, so that an ACK is sent immediately if there are two unacknowledged segments [Ste94].

1.1.4 HTTP

TCP is used as the transport protocol for many applications, including email, file transfer, and the World Wide Web. Web traffic is transferred over TCP according to the HTTP protocol. My dissertation looks at improving the performance of HTTP by improving the performance of TCP. Here I will give a brief description of how HTTP operates. In HTTP, a web browser makes a request to a web server for a particular file (*e.g.*, the images indicated in the web page in Figure 1.6). The web server receives the request and sends the file to the web client, which then displays the file to the user. The length of this exchange (from sending the request to receiving the response) is called the *HTTP response time*. (Note that this is not the time to download the entire web page, just only one element.) In evaluating the changes I make to TCP, I use HTTP response times as the major metric of performance.

1.2 Network Congestion

Anyone who has used the Internet has noticed delays. For web traffic, delays cause web pages to load slowly. For streaming audio and video, delays can cause gaps or jerkiness during playback. These delays are often caused by *network congestion*, which occurs when the incoming rate at routers exceeds the speed of the outgoing link for a sustained period of time.

Network congestion is a side-effect of the packet-switched design of the Internet. Packet-switching allows data from many different sources to travel along the same paths (*i.e.*, through the same routers). Routers in the Internet contain queues used for buffering packets when the instantaneous arrival rate of packets is greater than the out-bound transmission rate. These queues are first-in/first-out (FIFO) and have a finite capacity. When a packet arrives

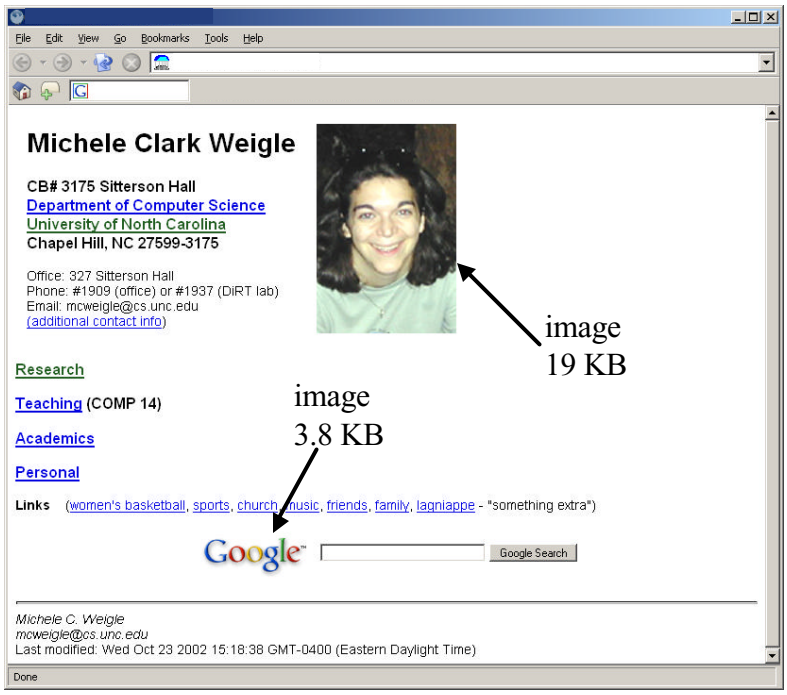


Figure 1.6: Example Web Page

and is buffered at a router, it must wait for all previously queued packets to be transmitted first. The longer the queue (*i.e.*, the more packets in the queue), the longer the *queuing delay* that incoming packets must face before being transferred. Since the queue is finite, incoming packets arriving at a full queue are dropped. Most queues in the Internet are "drop-tail," meaning that incoming packets are only dropped when the queue is full.

Network congestion causes the finite queues in routers to increase in size (thereby increasing queuing delay), and eventually, the queues fill and drop incoming packets. Queuing delays slow down the delivery of data from the sender to the receiver, decreasing the perceived performance of applications by the user. Packet drops are especially problematic for TCP flows. TCP promises in-order and reliable delivery, so if a TCP segment is dropped, subsequently received segments cannot be delivered to the application at the receiver until the dropped segment has been successfully received. When a segment drop occurs, TCP must detect the drop and retransmit the lost segment, both of which take time. Because of TCP's reliability requirements, lost packets result in increased delays for end users.

1.3 TCP Congestion Control

TCP has undergone several changes since it was first specified in 1974 [CK74, CDS74]. Figure 1.7 shows a timeline of significant advances in TCP.

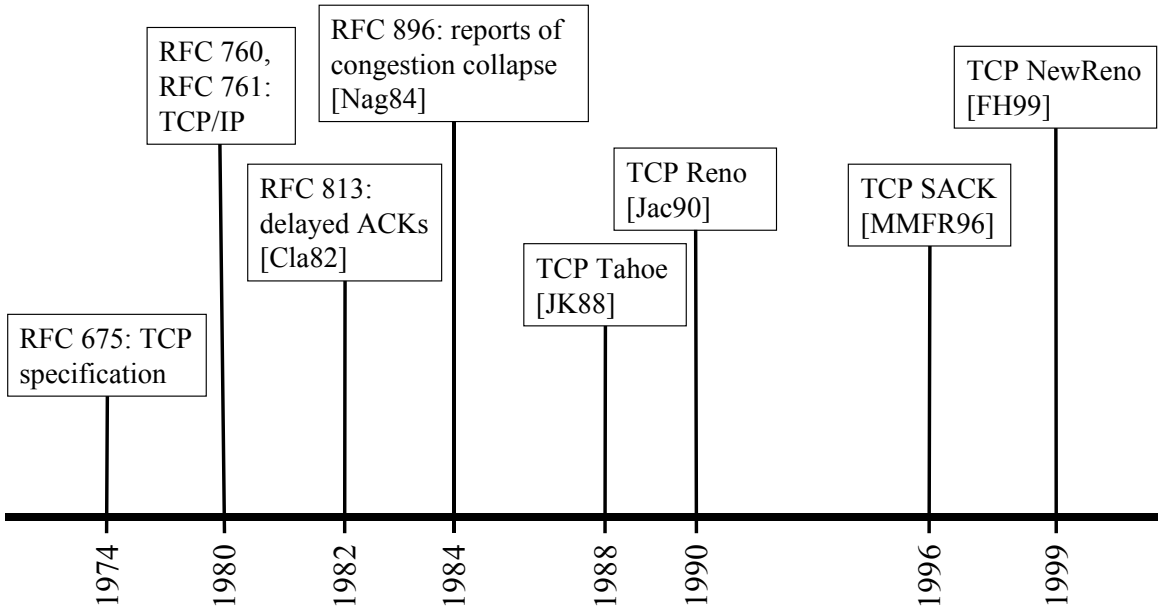


Figure 1.7: TCP Timeline

The driving problem for the development of TCP congestion control was congestion collapse in the mid-1980s [Nag84]. Congestion collapse describes a situation where the network is so overloaded with retransmissions of lost data that no new data can get through. TCP detected segment loss by setting a timer when a segment was sent. If the timer expired before the ACK for that segment was received, the segment was assumed to be lost and all segments starting at that sequence number were retransmitted (this scheme is known as "Go-Back-N"). Prior to this congestion collapse, only flow control was implemented in TCP. Nothing in TCP dictated what should be done when congestion was encountered in the network. In response to the congestion collapse, congestion control algorithms were developed for TCP. The basic idea was that segment loss, which occurs when queues overflow, is a sign of congestion. When segment losses are detected, senders should reduce their sending rates, thereby hopefully eliminating congestion.

1.3.1 TCP Tahoe

TCP Tahoe was the first attempt at TCP congestion control and consists of several changes to the original TCP, including the addition of a slow start phase, a congestion avoidance phase, and a fast retransmit phase [JK88].

Slow Start

TCP follows the idea of *conservation of packets*. A new segment is not sent into the network until a segment has left the network (*i.e.*, an ACK has returned). Before TCP Tahoe was introduced, TCP obeyed conservation of packets except upon startup. At startup, there are no outstanding segments to be acknowledged to release new segments, so TCP senders sent out a full window's worth of segments at once. The TCP senders, though, have no indication of how much data the network can handle at once, so often, these bursts led to packets being dropped at routers.

At startup (and after a packet loss), instead of sending segments as fast as possible, slow start restricts the rate of segments entering the network to twice the rate that ACKs return from the receiver. TCP Tahoe introduced the congestion window, *cwnd*, initially set to one segment. The send window is set to the minimum of *cwnd* and the receiver's advertised window. For every ACK received, *cwnd* is incremented by one segment. The size of the send window controls the sending rate of a TCP sender ($rate = w/RTT$). To increase its sending rate, a TCP sender would increase the value of its send window by increasing *cwnd*, allowing additional data to be transferred into the network before the first segment in the congestion window has been acknowledged. The congestion window can only be increased when ACKs return from the receiver, since ACKs indicate that data has been successfully delivered. A TCP sender will keep increasing *cwnd* until it detects that network congestion is occurring.

Figure 1.8 shows an example of the operation of slow start. The x-axis is time and the y-axis is sequence number. The boxes represent the transmission of a segment and are labeled with their corresponding sequence numbers. In order keep the example simple, I am assuming 1-byte segments. The numbers above the boxes indicate the value of *cwnd* when those segments were sent. Dots represent the arrival of an ACK and are centered along the y-axis of the highest sequence number they acknowledge. The ACKs are also centered along the x-axis with the segments that are released by the arrival of the ACK. The numbers below the ACKs represent the sequence number carried in the ACK (*i.e.*, the sequence number of the next segment expected by the receiver), referred to as the ACK number. For example, the first dot represents the acknowledgment of the receipt of segment 1 and that the receiver expects to next receive segment 2. So, the dot is positioned on the y-axis centered on segment 1 with a 2 below the dot, indicating that the ACK carries the ACK number of 2. In Figure 1.8, the initial value of *cwnd* is 2, so segments 1 and 2 are sent at time 0. When the ACK for segment 1 is received, *cwnd* is incremented to 3, and segments 3 and 4 are sent – segment 3 is released because segment 1 has been acknowledged and is no longer outstanding and segment 4 is sent to fill the congestion window. Thus, during slow start, for each ACK that is received, two new segments are sent.

During slow start, the sender's congestion window grows by one segment each time an ACK for new data is received. If an ACK acknowledges the receipt of two segments (as

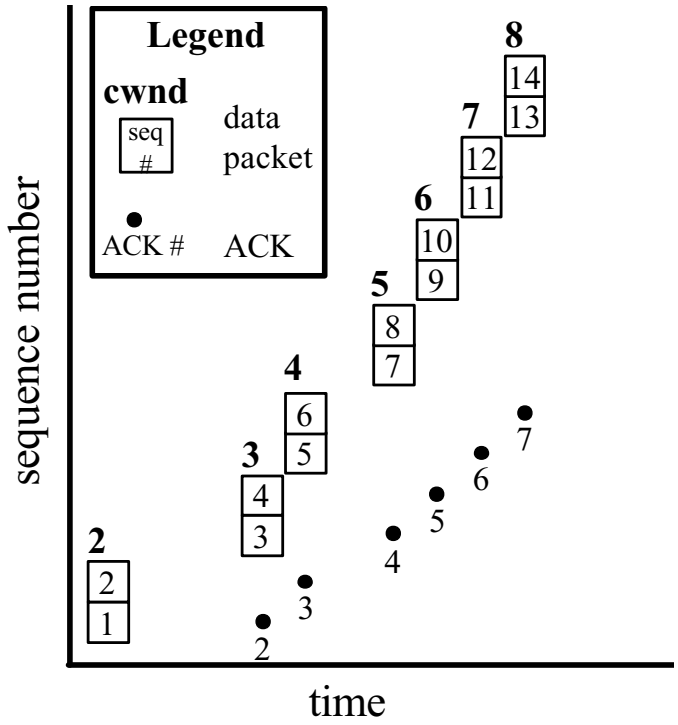


Figure 1.8: Slow Start

in delayed ACKs), then two segments are released by the receipt of the ACK. A sender communicating with a receiver that does not used delayed ACKs bursts two segments each time an ACK is received (one for the acknowledged segment and one for the increasing *cwnd*) as in Figure 1.8. A sender communicating with a receiver that uses delayed ACKs bursts three segments each time an ACK is received (two for the acknowledged segments and one for the increasing *cwnd*). This is demonstrated in Figure 1.9.

Congestion Avoidance

The goal of congestion control in TCP is to make use of all available resources (*i.e.*, buffer space in router queues) while reacting to network congestion. The amount of available bandwidth is not a known quantity and is dynamic as flows enter and leave the network. TCP probes for additional available bandwidth by regularly increasing its sending rate until congestion is detected. Segment loss is the only notification that congestion is occurring in the network. No explicit notification that packets have been dropped is sent by the router, so TCP senders must infer that segment loss has occurred and which segments have been lost.

Congestion avoidance conservatively probes for additional bandwidth by linearly increasing the transmission rate. In TCP Tahoe, a sender begins its transmission in slow start and then transitions to congestion avoidance. TCP Tahoe added a variable, *ssthresh*, which is

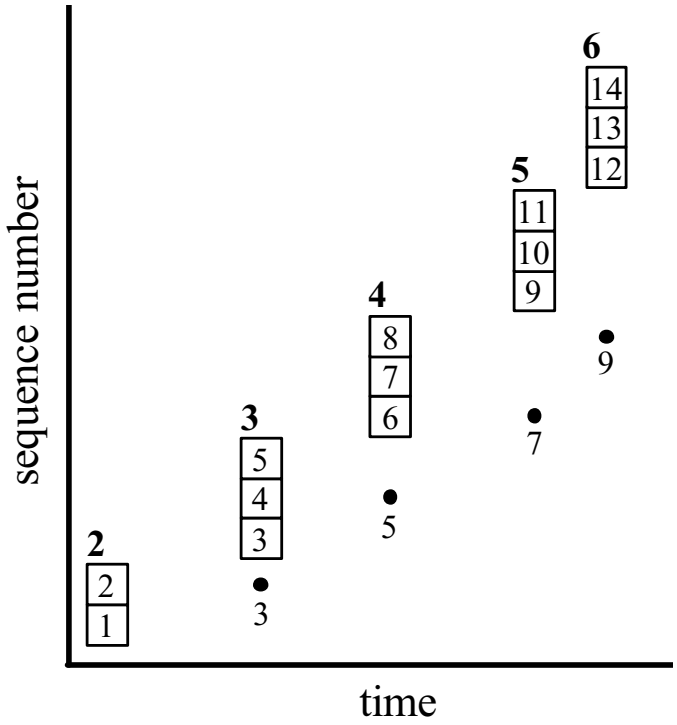


Figure 1.9: Slow Start with Delayed ACKs

the threshold for moving from slow start to congestion avoidance. When *cwnd* > *ssthresh*, slow start ends and congestion avoidance begins[1]. Once in congestion avoidance, *cwnd* is increased by 1/*cwnd* of a segment for each ACK received. The idea is that in one RTT, a TCP sender with a window of size *cwnd* will receive at most *cwnd* ACKs, so this results in an congestion window increase of at most one segment every RTT. This linear increase contrasts with slow start, which is an exponential increase with *cwnd* doubling every RTT.

Figure 1.10 demonstrates the operation of congestion avoidance. Again, the initial value of *cwnd* is 2, so two segments are sent to start the transmission. The initial value of *ssthresh* here is also set to 2. Since *cwnd* ≥ *ssthresh*, congestion avoidance is in effect. When the ACK for segment 1 returns, *cwnd* becomes 2.5 (2 + 1/2) and one segment is sent because segment 1 has left the network. When the ACK for segment 2 is received, *cwnd* becomes 2.9 (2.5 + 1/2.5). Again, only one segment is sent to keep only two outstanding segments in the network. When the ACK for segment 3 arrives, *cwnd* is now greater than 3, so two segments can be sent (one because segment 3 left the network and one because of the additional room in the congestion window).

As a comparison to Figure 1.10, Figure 1.11 shows the operation of congestion avoidance when delayed ACKs are used.

[1]The ideal is for *ssthresh* to be initially set to some reasonable value so that this transition to congestion avoidance from slow start will take place early in a connection.

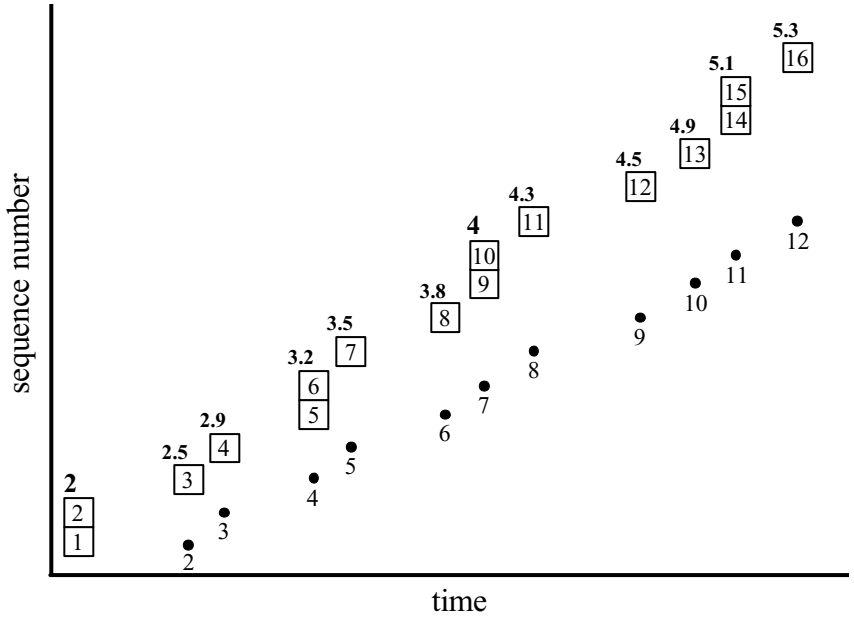


Figure 1.10: Congestion Avoidance

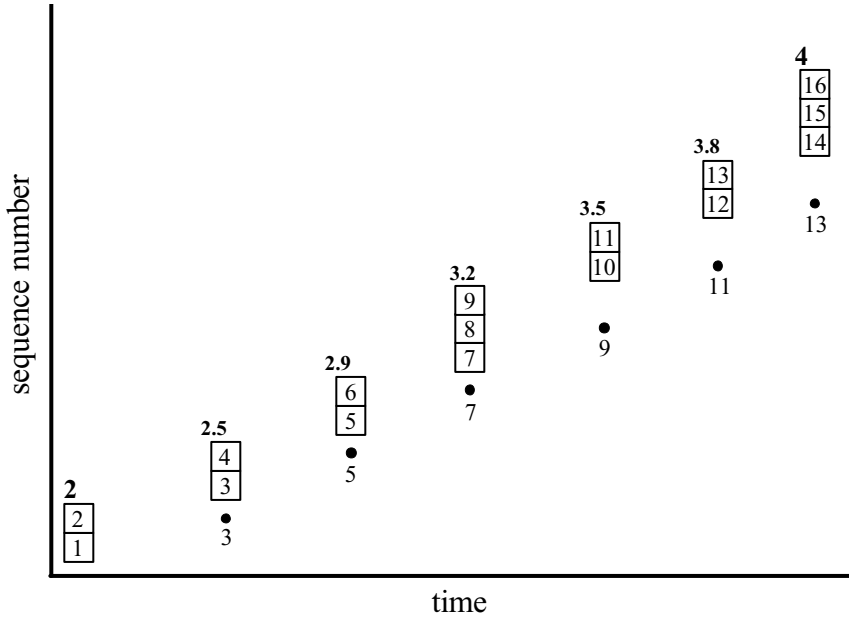


Figure 1.11: Congestion Avoidance with Delayed ACKs

Fast Retransmit

TCP uses a timer to detect when segments can be assumed to be lost and should be retransmitted. This retransmission timeout (RTO) timer is set every time a data segment is sent (if the timer has not already been set). Whenever an ACK for new data is received, the RTO timer is reset. If the RTO timer expires before the next ACK for new data arrives, the oldest unacknowledged segment is assumed to be lost and will be retransmitted. The RTO timer is set to 3-4 times the RTT so that unnecessary retransmissions are not generated by segments being delayed in the network. With the addition of congestion avoidance in TCP Tahoe, when the RTO timer expires (*i.e.*, when a segment is assumed to be lost), 1/2 *cwnd* is saved in *ssthresh*[2] and *cwnd* is set to 1 segment. At this point, $cwnd < ssthresh$, so a timeout marks a return to slow start. As slow start progresses, if no additional segment loss occurs, once *cwnd* reaches *ssthresh* (which is 1/2 of *cwnd* when segment loss was last detected), congestion avoidance is entered.

TCP Tahoe also added a faster way to detect segment loss, called *fast retransmit*. Before TCP Tahoe, the only way to detect segment loss was through the expiration of the RTO timer. TCP Tahoe added the ability to infer segment loss through the receipt of three duplicate ACKs. Whenever a receiver sees an out-of-order segment (*e.g.*, a gap in sequence numbers), it sends an ACK for the last in-order segment it received (which would be a duplicate of the previous ACK sent). The sender uses the receipt of three duplicates of the same ACK to infer that there was segment loss rather just segment re-ordering.

Figure 1.12 shows TCP Tahoe fast retransmit. This figure is a continuation of Figure 1.8 and looks at the situation that would occur if segment 9 was dropped. When the ACK returns for segment 7, segments 15 and 16 are released and *cwnd* is incremented to 9. When the ACK for segment 8 is received, *cwnd* is incremented to 10 and segments 17 and 18 are sent. But, since segment 9 is not received at the destination, the ACK for segment 10 is a duplicate of the ACK for segment 8. The congestion window is not changed when duplicate ACKs are received. Duplicate ACKs continue to return. When the third duplicate (sent by the receipt of segment 12) is received, fast retransmit is entered. Segment 9 is assumed to be lost and is immediately sent. At this point, *cwnd* is reduced to 1 segment. Duplicate ACKs (sent by the receipt of segments 13-18) continue to arrive, but no change is made to *cwnd* and hence no data segments are sent. When the ACK returns that acknowledges that the lost segment has been successfully received (this ACK specifies that all segments through segment 18 have been received), *cwnd* is incremented to 2 and segments 19 and 20 are sent. From this point, slow start continues as normal.

[2]Actually, *ssthresh* is set to the maximum of 1/2 the current flight size (the amount of data actually in the network but not yet acknowledged) and twice the maximum segment size. The value of *cwnd* is allowed to grow past the receiver's window, but the flight size is not.

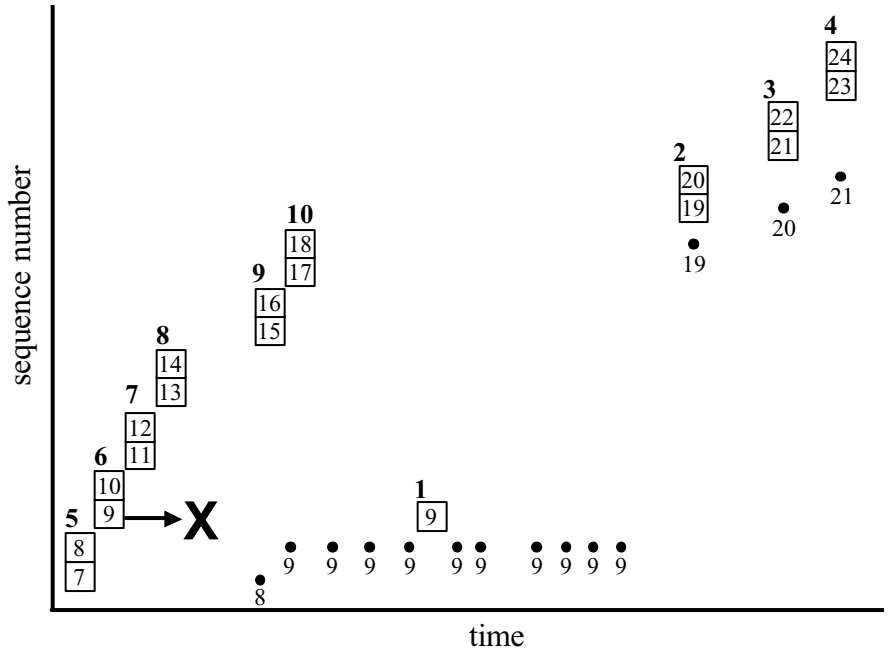


Figure 1.12: TCP Tahoe Fast Retransmit

The advantage of using fast retransmit is that it reduces the amount of time needed to detect a segment loss. Without fast retransmit, the expiration of the RTO timer is needed to detect loss. For flows that have large congestion windows, multiple ACKs will typically arrive in one RTT, so with a trigger of three duplicate ACKs, a segment loss could be detected in less than one RTT. In this way, fast retransmit allows TCP Tahoe to avoid a lengthy timeout during which no data can be sent.

AIMD

The biggest contribution of TCP Tahoe was the introduction of the additive-increase / multiplicative-decrease (AIMD) window adjustment algorithm to TCP. Additive increase is defined as $w(t+1) = \alpha + w(t)$, where $w(t)$ is the size of the current congestion window in segments at time t and the time unit is one RTT. In congestion avoidance, $\alpha = 1$. For every ACK received, the congestion window is increased by $1/w(t)$, which results in an increase of at most one segment in one RTT. Multiplicative decrease is defined as $w(t+1) = \beta w(t)$. In TCP Tahoe, $\beta = 0.5$, with *ssthresh* being set to 1/2 *cwnd* during fast retransmit.

1.3.2 TCP Reno

In 1990, Van Jacobson added another feature to TCP Tahoe, called *fast recovery*. TCP Tahoe with fast recovery is known as TCP Reno and is the *de facto* standard version of TCP on the Internet [APS99].

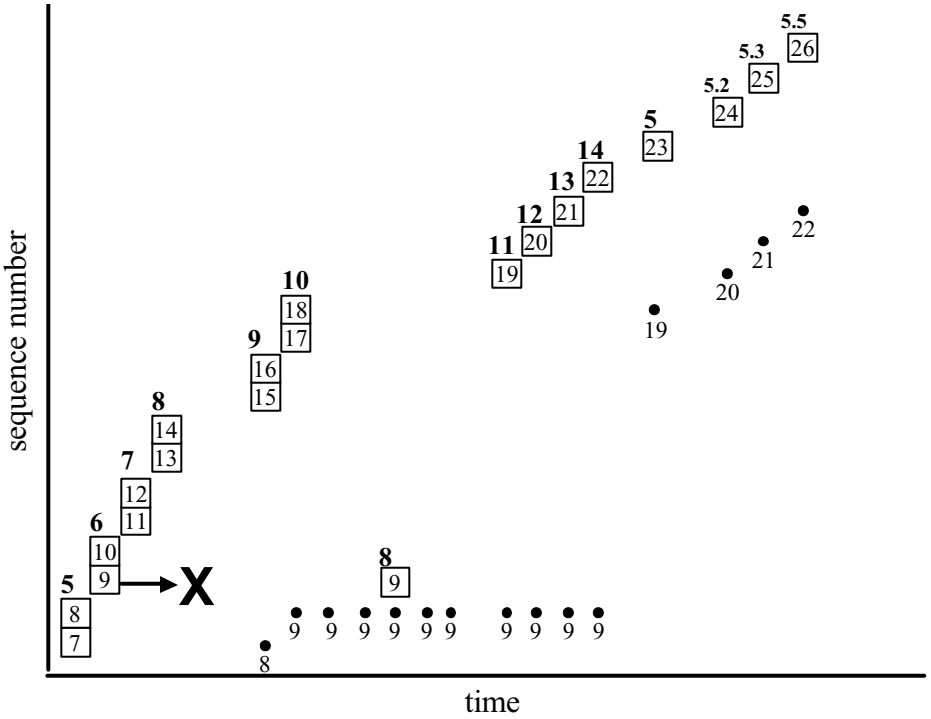


Figure 1.13: TCP Reno Fast Retransmit and Fast Recovery

Fast Recovery

As in TCP Tahoe, a segment can be assumed lost if three duplicate ACKs are received. When segment loss is detected via three duplicate ACKs, fast recovery is entered, instead of slow start. In fast recovery, *ssthresh* is set to 1/2 *cwnd* (according to the AIMD algorithm), and *cwnd* is set to *ssthresh* + 3 (one for each of the three duplicate ACKs, which imply that segments have left the network). For each additional duplicate ACK received, *cwnd* is incremented by one segment, as in slow start. New segments can be sent as long as *cwnd* allows. When the first ACK arrives for the retransmitted segment, *cwnd* is set back to *ssthresh*. Once the lost segment has been acknowledged, TCP leaves fast recovery and returns to congestion avoidance. If *cwnd* had previously been large, cutting *cwnd* in half and entering congestion avoidance instead of reducing *cwnd* to 1 segment and returning to slow start allows the sender to probe for available bandwidth conservatively and approach the previous value of *cwnd* with less chance of overflowing network queues than with using slow start.

Figure 1.13 shows fast retransmit and fast recovery. This situation is much like that in Figure 1.12. Segment 9 is lost. When the third duplicate ACK is received for segment 8, the lost segment, segment 9, is retransmitted, as specified in fast retransmit. According to fast recovery, *cwnd* is not reduced to 1, but is changed to 8 (*cwnd* is cut in half to 5, but then incremented by 3 for each of the duplicate ACKs received). Two more duplicate ACKs arrive

after the retransmission and, for each of these, *cwnd* is incremented. So, *cwnd* is 10, but no additional segments can be sent because as far as the TCP sender knows, 10 segments are outstanding (segments 9-18). When the ACK sent by the receipt of segment 15 is received, *cwnd* is again incremented. Since *cwnd* is now 11, there is room for one more segment to be sent, and so, segment 19 is released. This continues until the retransmission of segment 9 is acknowledged. When this occurs, *cwnd* is returned to 5 (one-half of the congestion window when loss was detected) and congestion avoidance is entered and progresses as normal.

As seen in the example above, fast recovery also provides an additional transition from slow start to congestion avoidance. If a sender is in slow start and detects segment loss through three duplicate ACKs, after the loss has been recovered, congestion avoidance is entered. As in TCP Tahoe, congestion avoidance is also entered whenever $cwnd > ssthresh$. In many cases, though, the initial value of *ssthresh* is set to a very large value (*e.g.*, 1 MB in FreeBSD 5.0), so segment loss is often the only trigger to enter congestion avoidance [Hoe96].

TCP Reno is limited to recovering from only one segment loss during a single fast retransmit and fast recovery phase. Additional segment losses in the same window may require that the RTO timer expire before the segments can be retransmitted. The exception is when *cwnd* is greater than 10 segments upon entering fast recovery, allowing two segment losses to be recovered without the sender experiencing a timeout. During fast recovery, one of the new segments to be sent (after the retransmission) could be lost and detected with three duplicate ACKs after the first fast recovery has finished (with the receipt of the ACK for the retransmitted packet). In this case, TCP Reno could recover from two segment losses by entering fast recovery twice in succession. This causes *cwnd* to effectively be reduced by 75% in two RTTs. This situation is demonstrated by Fall and Floyd when two segments are dropped [FF96].

1.3.3 Selective Acknowledgments

A recent addition to the standard TCP implementation is the selective acknowledgment option (SACK) [MMFR96]. SACK is used in loss recovery and helps the sender determine which segments have been dropped in the case of multiple losses in a window. The SACK option contains up to four (or three, if the RFC 1323 timestamp option is used) SACK blocks, which specify contiguous blocks of the most recently received data. Each SACK block consists of two sequence numbers which delimit the range of data the receiver holds. A receiver can add the SACK option to ACKs it sends back to a SACK-enabled sender. Using the information in the SACK blocks, the sender can infer which segments have been lost (up to three non-contiguous blocks of lost segments). The SACK option is sent only on ACKs that occur after out-of-order segments have been received. Allowing up to three SACK blocks per SACK option ensures that each SACK block is transmitted in at least three ACKs, providing some amount of robustness in the face of ACK loss.

The SACK RFC specifies only the SACK option and not how a sender should use the information given in the SACK option. Blanton *et al.* present a method for using the SACK option for efficient data recovery [MMFR96]. This method was based on an implementation of SACK by Fall and Floyd [FF96]. The SACK recovery algorithm only operates once fast recovery has been entered via the receipt of three duplicate ACKs. The use of SACK allows TCP to decouple the issues of when to retransmit a segment from which segment to send. To do this, SACK adds two variables to TCP: *scoreboard* (which segments to send) and *pipe* (when to send segments).

The scoreboard records which segments the sender infers to be lost based on information from SACKs. The segments in the scoreboard all have sequence numbers past the current value of the highest cumulative ACK. During fast recovery, *pipe* is the estimate of the amount of unacknowledged data "in the pipe." Each time a segment is sent, *pipe* is incremented. The value of *pipe* is decremented whenever a duplicate ACK arrives with a SACK block indicating that new data was received. When $cwnd - pipe \geq 1$, the sender can either send a retransmission or transmit new data. When the sender is allowed to send data, it first looks at *scoreboard* and sends any segments needed to fill gaps at the receiver. If there are no such segments, then the sender can transmit new data. The sender leaves fast recovery when all of the data that was unacknowledged at the beginning of fast recovery has been acknowledged.

1.3.4 TCP NewReno

During fast recovery, TCP Reno can only recover from one segment loss without suffering a timeout[3]. As long as duplicate ACKs are returning, the sender can send new segments into the network, but fast recovery is not over until an ACK for the lost segment is received. Only one retransmission is sent during each fast recovery period, though multiple retransmissions can be triggered by the expiration of the RTO timer.

TCP NewReno is a change to non-SACK-enabled TCP Reno where the sender does not leave fast recovery after a partial ACK is received [Hoe95, FH99]. A partial ACK acknowledges some, but not all, of the data sent before the segment loss was detected. With the receipt of a partial ACK, the sender can infer that the next segment the receiver expects has also been lost. TCP NewReno allows the sender to retransmit more than one segment during a single fast recovery, but only one lost segment may be retransmitted each RTT.

One recent study has reported that TCP NewReno is the most popular TCP version for a sample of Internet web servers [PF01].

[3]Except for in the situation described in section 1.3.2.

1.4 Sync-TCP

In TCP Reno, the most common implementation of TCP, segment loss is the sole indicator of network congestion. TCP hosts are not explicitly notified of lost segments by the routers, but must rely on timeouts and duplicate acknowledgments to indicate loss. One problem with TCP congestion control is that it only reduces its sending rate after segment loss has occurred. With its probing mechanism of increasing *cwnd* until loss occurs, TCP tends to cause queues to overflow as it searches for additional bandwidth. Additionally, TCP's congestion signal is binary – either an ACK returns and TCP increases its *cwnd*, or segment loss is detected and *cwnd* is reduced drastically. So, TCP's congestion control is tied to its mechanism for data recovery. An ideal congestion control algorithm would be able to detect congestion in the network and react to it before segment loss occurred.

There have been two main approaches to detecting congestion before router buffers overflow: using end-to-end methods and using router-based mechanisms. Router-based mechanisms, such as active queue management (AQM), come from the idea that drop-tail routers are the problem. These mechanisms make changes to the routers so that they notify senders when congestion is occurring but before packets are dropped. End-to-end approaches are focused on making changes to TCP Reno, rather than the drop-tail queuing mechanism. Most of these approaches try to detect and react to congestion earlier than TCP Reno (*i.e.* before segment loss occurs) by monitoring the network using end-to-end measurements. AQM, in theory, gives the best performance because congested routers are in the best position to know when congestion is occurring. Drawbacks to using AQM methods include the complexity involved and the need to change routers in the network (and, as explained in Chapter 2, possibly the end system). My approach is to look at end-to-end methods that try to approximate the performance benefit of router-based mechanisms.

My claim is that the knowledge of a flow's one-way transit times (OTTs) can be used to improve TCP congestion control by detecting congestion early and avoiding segment losses. A connection's forward path OTT is the amount of time it takes a segment to traverse all links from the sender to the receiver and includes both propagation and queuing delays. Queues in routers build up before they overflow, resulting in increased OTTs. If all senders directly measure changes in OTTs and back off when the OTT indicates that congestion is occurring, congestion could be alleviated.

In this dissertation, I will introduce a family of congestion control mechanisms, called Sync-TCP, which uses synchronized clocks to gather a connection's OTT data. I will use Sync-TCP as a platform for investigating techniques for detecting and responding to changes in OTTs. I will demonstrate that, from a network standpoint, using Sync-TCP results in fewer instances of packet loss and lower queue sizes at bottleneck routers than TCP Reno. In the context of HTTP traffic, flows will see improved response time performance when all flows use Sync-TCP as opposed to TCP Reno.

The goal of my work is to determine if providing exact transmission timing information to TCP end systems could be used to improve TCP congestion control. Exact timing information could be obtained by the use of synchronized clocks on end systems. If timestamps from computers with synchronized clocks are exchanged, the computers can calculate the OTT between them. The OTT is the amount of time that it takes a packet from a sender to reach its destination and includes both the propagation delay and the queuing delay experienced by the packet. Assuming that the propagation delay for packets in a flow remains constant, any variation in a flow's set of OTTs reflects changes in the queuing delays for packets in that flow. Increases in queuing delays could be a good indicator of incipient network congestion and hence be a good basis for a congestion control algorithm. To investigate this idea, I developed a family of congestion control mechanisms called Sync-TCP. These mechanisms are based on TCP Reno, but use synchronized clocks to monitor OTTs for early congestion detection and reaction.

For a receiver to compute the OTT of a segment, it must have the time that the segment was sent. I add a Sync-TCP timestamp option to the TCP header that includes, in each segment, the time the segment was sent, the time the most recently-received segment arrived, and the OTT of the most recently-received segment. When a receiver gets a segment with the Sync-TCP timestamp option, it computes the OTT by subtracting the time the segment was received from the time the segment was sent (found in the Sync-TCP timestamp option). The receiver then inserts the OTT into the header of the next segment going back to the sender.

In the abstract, congestion control is a two-step process of congestion detection and reaction to congestion. Sync-TCP describes a family of methods using OTTs for congestion control. The individual congestion control algorithms in Sync-TCP are combinations of different congestion detection and congestion reaction mechanisms. I will look at five congestion detection mechanisms that use OTTs to monitor the status of the network. All of these congestion detection mechanisms calculate the latest queuing delay, which is the latest OTT subtracted by the minimum-observed OTT. I look at the following congestion detection mechanisms:

1. Percentage of the maximum queuing delay – congestion is detected if the current computed queuing delay is greater than 50% of the maximum-observed queuing delay, which represents an estimate of the maximum amount of queuing available in the network.

2. Percentage of the minimum OTT – congestion is detected if the current computed queuing delay is greater than 50% of the flow's minimum OTT.

3. Average queuing delay – congestion is detected when the average computed queuing delay is greater than a predefined threshold.

Trend Direction	**Average Queuing Delay (as a % of the maximum)**	**Adjustment to *cwnd***
increasing	75-100%	decrease *cwnd* 50%
increasing	50-75%	decrease *cwnd* 25%
increasing	25-50%	decrease *cwnd* 10%
increasing	0-25%	increase *cwnd* 1 segment per RTT
decreasing	75-100%	no change to *cwnd*
decreasing	50-75%	increase *cwnd* 10% per RTT
decreasing	25-50%	increase *cwnd* 25% per RTT
decreasing	0-25%	increase *cwnd* 50% per RTT

Table 1.1: Adjustments to the Congestion Window Based on the Signal from the Congestion Detection Mechanism

4. Trend analysis of queuing delays – congestion is detected when the trend of nine queuing delay samples is increasing.

5. Trend analysis of average queuing delay – provides a congestion signal that uses the direction of the trend of the average computed queuing delay and the value of the average computed queuing delay as a percentage of the maximum-observed queuing delay.

I will show that trend analysis of the average queuing delay (mechanism 5), which is a combination of the best parts of mechanisms 1, 3, and 4, offers the best congestion detection.

I look at two congestion reaction mechanisms. The first method is the same as TCP Reno's reaction to the receipt of three duplicate ACKs, where the sender reduces *cwnd* by 50%. This mechanism can operate with any congestion detection mechanism that provides a binary signal of congestion. The second method is based on using the trend analysis of the average computed queuing delay as the congestion detection mechanism. Table 1.1 lists the adjustments that this mechanism makes to *cwnd* based on the result of the congestion detection.

1.5 Thesis Statement

My thesis for this work is as follows: Precise knowledge of one-way transit times can be used to improve the performance of TCP congestion control. Performance is measured in terms of network-level metrics, including packet loss and average queue sizes at congested links, and in terms of application-level metrics, including HTTP response times and throughput per HTTP response. I will show that Sync-TCP provides lower packet loss, lower queue sizes, lower HTTP response times, and higher throughput per HTTP response than TCP Reno. Additionally, I will show that Sync-TCP offers performance comparable to that achieved by using router-based congestion control mechanisms.

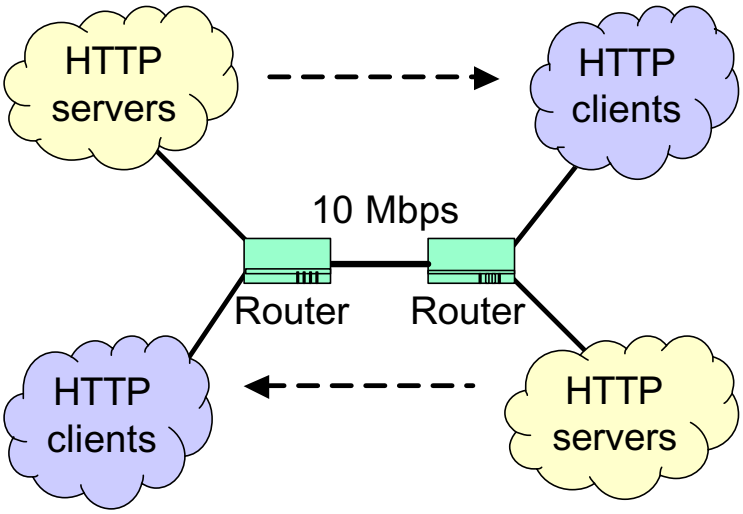


Figure 1.14: Network Topology

1.6 Evaluation

I will present the study of the performance of the congestion detection and reaction mechanism described in Table 1.1, which I will refer to as Sync-TCP for the remainder of this section.

In Chapter 4, I will present the evaluation of Sync-TCP in the context of HTTP traffic by performing simulations of web browsing. A simplified diagram of the topology used is in Figure 1.14. Clouds of HTTP servers and clients feed traffic into a network that is constrained by a 10 Mbps bottleneck link. This bottleneck link will be congested by the traffic from the HTTP servers. Traffic levels of 50-105% of the bottleneck link capacity (of 10 Mbps) were generated.

I will look at network-level metrics, such as packet loss rates at the bottleneck router and the average queue size at the bottleneck router, and application-level metrics, including the average goodput per HTTP response (bytes received per second at each HTTP client) and the distribution of HTTP response times (the time between the sending of an HTTP request and receipt of its corresponding HTTP response).

I will show that Sync-TCP is better than TCP Reno for HTTP traffic in terms of the following metrics:

- packet loss rates at the bottleneck link
- average queue size at the bottleneck link
- goodput per HTTP response
- HTTP response times

In particular, at load levels of 50% and 60% of the capacity of the bottleneck link, I will show that using Sync-TCP for all HTTP flows results in 0 drops at the bottleneck link. In comparison, TCP Reno had over 3500 and 8000 drops, respectively. At all load levels, Sync-TCP provides less packet loss and lower queue sizes, which in turn, resulted in lower HTTP response times. If all flows use Sync-TCP to react to increases in queuing delay, congestion could be alleviated quickly. This would result in overall shorter queues, faster response for interactive applications, and a more efficient use of network resources.

By showing that Sync-TCP can provide better network performance, I will show that synchronized clocks can be used to improve TCP congestion control. In this dissertation, I make the following contributions:

- a method for measuring a flow's OTT and returning this exact timing information to the sender
- a comparison of several methods for using OTTs to detect congestion
- a family of end-to-end congestion control mechanisms, Sync-TCP, based on using OTTs for congestion detection
- a study of standards-track[4] TCP congestion control and error recovery mechanisms in the context of HTTP traffic, used as a basis for comparison to Sync-TCP

In the remainder of this dissertation, I will first discuss related work in Chapter 2, including previous approaches to the congestion control problem. Then, in Chapter 3, I will discuss the various Sync-TCP congestion detection and reaction mechanisms. Chapter 4, presents results from a comparison of the best version of Sync-TCP to the best standards-track TCP protocol in the context of HTTP traffic. A summary of this work and ideas for future work are given in Chapter 5. Appendix A covers the experimental methodology, including details of the network configuration, HTTP traffic model, and components I added to the *ns* network simulator for running the experiments. Finally, Appendix B describes experiments performed to choose the best standards-track TCP protocol for HTTP traffic.

[4]By standards-track congestion control, I mean TCP Reno and Adaptive RED, a form of router-based congestion control. By standards-track error recovery, I mean TCP Reno and TCP SACK. TCP Reno is an IETF standard, TCP SACK is on the IETF standards track, and Adaptive RED, a modification to the IETF standard RED, is widely encouraged to be used in Internet routers.

Chapter 2

Related Work

In this chapter, I will describe previous work related to my thesis. First, I will briefly discuss computer clock synchronization, including the Global Positioning System. Then, I will give an overview of several recent TCP congestion control algorithms. I will also discuss several congestion control algorithms that are not tied to TCP. In addition to end system congestion control mechanisms, I will describe variants of Random Early Detection, an active queue management algorithm that assists end systems in congestion control. Finally, I will look at network measurement and analysis research that pertains to detecting network congestion.

2.1 Synchronized Clocks

Mills defines synchronizing frequency as adjusting clocks to run at the same frequency and synchronizing time as setting clocks to have the same time relative to Coordinated Universal Time (UTC), the international time standard [Mil90]. Synchronizing clocks is defined as synchronizing both frequency and time. In a computer network, there are two popular methods of synchronizing clocks: the Network Time Protocol (NTP) and the Global Positioning System (GPS). NTP is a network protocol that can run on computers to synchronize clocks to a specific computer that serves as a time source. The GPS satellite network allows computers that have GPS receiver hardware to synchronize their clocks to the GPS satellites.

2.1.1 Network Time Protocol

The Network Time Protocol (NTP) is a distributed clock synchronization protocol currently used to provide accurate time to computers around the world [Mil92]. Primary NTP servers are synchronized to standard reference clocks via GPS, modem, or radio. Secondary NTP servers and clients synchronize to the primary servers. When a client requests a time update, the server responds with its current time. The client notes when the message was received and uses one-half of the round-trip time as an estimate of the offset between the

server and client. NTP is accurate to about one millisecond on a LAN and a small number of tens of milliseconds on a WAN.

2.1.2 Global Positioning System

The Global Positioning System (GPS) can provide accurate timing information as well as position measurements to receivers on Earth [DP90]. GPS satellites are on a fixed orbit around the earth, so that at all times, the position of any satellite is known. For a stationary GPS receiver on Earth, information from four satellites is needed to determine both position and time. The GPS satellites transmit signals to Earth, and a GPS receiver locks on to these signals and decodes the position of the satellites. A special code is transmitted every millisecond that helps the receivers keep accurate time. The time computed from GPS can fall to within one microsecond of UTC.

Currently, GPS is limited by the fact that receivers need line of sight to receive signals from the satellites. For computer clock synchronization via GPS, there are methods, such as NTP, for propagating the GPS information to many machines that do not have line of sight to the satellites or do not have GPS receivers. Using NTP as a propagation method, though, reduces the accuracy of clock synchronization to that of NTP. It is reasonable to expect that clocks of computers on a WAN could be synchronized to within 1-2 ms if there were GPS time sources that could be used as NTP servers located on managed networks with low delay variability.

2.2 TCP Timestamp Option

RFC 1323 describes extensions to TCP to improve performance on large bandwidth-delay product network paths [JBB92]. The document introduces two new TCP options: window scale and timestamp. Both of these options are additions to the TCP header. Without the window scale option, the maximum value of the receiver's advertised window (and thus, the maximum value of the send window) in a TCP connection is 65 KB (2^{16}). Using the window scale option allows the receiver window to grow to up to 1 GB (2^{30}). The timestamp option can be used to protect against wrapped sequence numbers in a single connection and to compute more accurate RTTs. The timestamp option adds 10 bytes to the TCP header and contains the time the segment was sent and, if the segment is an ACK, the timestamp contained in the last data segment received. This TCP option is the basis for the Sync-TCP timestamp option.

2.3 TCP Congestion Control

TCP's congestion control has evolved through a process of iterative refinement. Current standards include TCP Tahoe, TCP Reno, TCP NewReno, and Selective Acknowledgments. In the following sections, I will describe developments of TCP congestion control algorithms and mechanisms past the standards-track improvements discussed in Chapter 1. TCP Tahoe provides the basis for the TCP variants that I discuss. The TCP modifications are grouped here according to changes they make to TCP Tahoe:

- **Variations on Slow Start:** TCP Vegas, TCP Santa Cruz, TCP Peach.
- **Variations on Congestion Avoidance:** TCP Vegas, TCP Santa Cruz, TCP Peach, AIMD Modifications.
- **Loss Detection:** TCP Vegas, Forward Acknowledgments, TCP Santa Cruz.
- **Data Recovery:** Forward Acknowledgments, TCP Santa Cruz, TCP Westwood, TCP Peach.

2.3.1 TCP Vegas

To address the sometimes large amounts of segment losses and retransmissions in TCP Reno, Brakmo *et al.* proposed several modifications to its congestion control algorithm [BOP94]. The resulting protocol, TCP Vegas, includes three main modifications: a fine-grained RTT timer, congestion avoidance using expected throughput, and congestion control during TCP's slow start phase. With these modifications, Brakmo reported a 40-70% increase in TCP Vegas' throughput over TCP Reno. As a result, TCP Vegas has been studied and debated extensively in the networking community [Jac94, ADLY95, MLAW99, HBG00, LPW01].

Congestion Avoidance

TCP Vegas uses a decrease in throughput as an early indication of network congestion. This is done by keeping track of the expected throughput, which corresponds to the amount of data in transit. As the congestion window increases, the expected throughput should increase accordingly. The basis of TCP Vegas' congestion avoidance algorithm is to keep just the right amount of extra data in the network. Vegas defines "extra data" as the data that would not have been sent if the flow's rate exactly matched the bandwidth available in the network. This extra data will be queued at the bottleneck link. If there is too much extra data, the connection could become a cause of congestion. If there is not enough extra data in the network, the connection does not receive feedback quickly enough to adjust to congestion. Feedback is in the form of RTT estimates which only return if data continues to be sent and acknowledged. By monitoring changes in throughput and reacting to conditions other than segment loss, TCP Vegas pro-actively handles network congestion. Unfortunately, study of

this portion of TCP Vegas revealed that it contributed the least to the authors' claims of higher throughput than TCP Reno [HBG00].

Slow Start

TCP Vegas added its congestion detection algorithm to slow start. During slow start, the expected throughput and actual throughput are monitored as in congestion avoidance. When the actual throughput is less than the expected throughput by a certain threshold amount, TCP Vegas transitions from slow start to congestion avoidance. According to one study, the congestion detection in slow start accounted for a 25% increase in throughput over TCP Reno, making it the most beneficial of all of TCP Vegas' modifications [HBG00].

Data Recovery

TCP Reno sends retransmissions after receiving three duplicate ACKs or after the RTO timer has expired. TCP Vegas tries to improve upon this by using a fine-grained measurement of the RTT as the RTO and by recording the time each segment is sent. Upon receiving a duplicate ACK, if the time since sending the potentially lost segment is greater than the fine-grained RTO, TCP Vegas retransmits the assumed lost segment. This modification met with criticism from prominent members of the networking community. In a note to the end2end-tf mailing list, Van Jacobson argued that the RTO should never be set to a segment's RTT because of normal fluctuations in RTT due to competing flows [Jac94]. In particular, Jacobson said that the RTO should never be set less than 100-200 ms.

TCP Vegas has not yet been accepted as a standard part of TCP. Recent work [LPW01], though, has focused on reinterpreting TCP Vegas and casting it as a complement to Random Early Marking [LL99], an active queue management technique.

2.3.2 Forward Acknowledgments

The forward acknowledgment congestion control algorithm (FACK) was developed as a method of using the information in the SACK option for congestion control [MM96]. The term "forward acknowledgment" comes from using information about the forward-most data known to the receiver, which is the highest sequence number the receiver has seen. The idea behind FACK is to use the information in the SACK option to keep an accurate count of the amount of unacknowledged data in the network. This allows the sender to more intelligently recover from segment losses. FACK incorporates congestion control into fast recovery but makes no changes to TCP Reno's congestion control algorithms outside of fast recovery.

Data Recovery

FACK adds two state variables: $snd.fack$, the largest sequence number held by the receiver (also called the "forward-most" sequence number) and $retran_data$, the amount of unacknowledged retransmissions in the network. FACK uses information from the SACK option returned in each ACK to update $snd.fack$. The amount of unacknowledged data is represented by $awnd$, which is $snd.nxt - snd.fack + retran_data$, where $snd.nxt$ is the sequence number of the next new data segment to be sent. During recovery, while $awnd < cwnd$, data can be sent. This regulates how fast lost segments can be retransmitted and forces the recovery mechanism to follow TCP Reno's congestion window mechanism instead of rapidly sending a series of retransmissions of lost segments.

Loss Detection

FACK also changes the trigger for entering fast recovery. In TCP Reno, fast recovery is entered whenever three duplicate ACKs are received. FACK uses $snd.fack$ to calculate how much data the receiver has buffered in its reassembly queue (data waiting for a hole to be filled before delivery). If the sender determines that the receiver's reassembly queue is larger than three segments, it enters fast recovery. The receipt of three duplicate ACKs still triggers an entry into fast recovery. This modification to the fast recovery trigger is useful if several segments are lost prior to receiving three duplicate ACKs. If only one segment is lost, then this method would trigger fast recovery at the same time as TCP Reno.

2.3.3 TCP Santa Cruz

TCP Santa Cruz offers changes to TCP Reno's congestion avoidance and error recovery mechanisms [PGLA99]. The congestion avoidance algorithm in TCP Santa Cruz uses changes in delay in addition to segment loss to detect congestion. Modifications to TCP Reno's error recovery mechanisms utilize a SACK-like ACK window to more efficiently retransmit lost segments. TCP Santa Cruz also includes changes to the RTT estimate and changes to the retransmission policy of waiting for three duplicate ACKs before retransmitting.

Congestion Avoidance

TCP Santa Cruz attempts to decouple the reaction to congestion on the data path from congestion detected on the ACK path. This is done through the use of a new TCP option, which includes the time the segment generating the ACK was received and its sequence number. TCP Santa Cruz uses *relative delay*, or change in forward delay, to detect congestion. The relative delay for two segments is calculated by subtracting the difference in the receipt times of the segments from the difference in the sending times of the segments. These relative delays are summed from the beginning of the connection and updated every RTT. If the

sum of relative delays is 0, then the amount of data queued in the network is steady over the interval. The relative delay is then translated into an estimate of the number of packets queued at the bottleneck using a packet-pair-like method (as described in section 2.4.2).

The goal of the algorithm is to keep a certain number of packets, N_{op}, queued at the bottleneck. If the current number of packets queued, n_t, is within δ of N_{op}, the current window is unchanged. If $n_t < N_{op} - \delta$, the window is increased linearly. If $n_t > N_{op} + \delta$, the window is decreased linearly. This approach is much like the window adjustment in TCP Vegas.

In their experiments, the authors varied N_{op} between 1-5 packets. They found that using between 3-5 packets gave high utilization and lower delay than TCP Reno. These experiments were performed with a single source and sink over a single bottleneck. In a more complex environment, the best value for N_{op} would have to be determined experimentally.

Slow Start

TCP Santa Cruz makes a small change to TCP Reno's slow start algorithm. The congestion window is initially set to two segments. This allows for the calculation of relative delay during slow start. TCP Santa Cruz can force the transition from slow start to congestion avoidance before $cwnd > ssthresh$ whenever a relative delay measurement or n_t is greater than $N_{op}/2$. The idea is to end slow start once any build-up in the queue is detected.

Data Recovery

TCP Santa Cruz includes in each ACK the sequence number of the segment generating the ACK and that segment's retransmission number (*e.g.*, if the segment was the second retransmission of a segment, its retransmission number would be 2). With this change, the RTT estimate (using the same algorithm as TCP Reno) can be calculated for every ACK including those that acknowledge retransmissions. This also removes the need for an exponential backoff in setting the RTO for successive retransmissions of the same segment because the RTT of the individual retransmissions can now be tracked. Additionally, the RTT estimate can be updated for each ACK received instead of once per RTT as in TCP Reno.

An *ACK Window* field is also included in every ACK. The ACK Window can indicate the status (received or not received) of each segment in a window. Approaches like SACK are limited to indicating only 3-4 missing segments. The ACK Window is represented by a bit-vector, where each bit represents a certain number of bytes. The number of bytes represented by each bit is determined by the receiver and indicated in the first byte of the ACK Window option. With a maximum of 19 bytes for the option, the ACK Window indicates the status of a 64 KB window with each bit representing 450 bytes. Normally, each bit represents the

number of bytes in the network's maximum segment size. If there are no gaps in the sequence space, the ACK Window is not included.

Loss Detection

TCP Santa Cruz also makes changes to TCP Reno's retransmission policy. A sender does not have to wait for three duplicate ACKs. If a segment is lost (as noted in the ACK Window), it can be retransmitted when the first ACK for a later segment is received after a RTT has passed since sending the lost segment. This mechanism is also much like the loss detection used in TCP Vegas.

2.3.4 TCP Westwood

TCP Westwood consists of a change to fast recovery that improves the performance of flows that experience link error losses [MCG+01]. This is useful on physical links where losses due to link errors are common, such as on a wireless link. To assist in this "faster recovery," the bandwidth used by the flow is estimated each time an ACK is returned.

The key insight to bandwidth estimation is that as soon as segments are dropped, the subsequent return rate of ACKs represents the amount of bandwidth available to the connection. TCP Westwood uses a low-pass filter on the bandwidth estimates to ensure that the most recent information is being used without reacting to noise. This low-pass filter needs to be given samples at a uniform rate. Since ACKs arrive at a non-uniform rate, the filtering algorithm in TCP Westwood assumes that virtual ACKs arrive at regular intervals. These virtual ACKs are counted as acknowledging 0 bytes of data, and so, do not affect the bandwidth estimate.

Data Recovery

TCP Westwood's changes TCP Reno's data recovery mechanisms allow a flow that experiences a wireless link error to recover from the segment loss without reacting as if network congestion caused the loss. Instead of backing off by 50% after the receipt of three duplicate ACKs, a TCP Westwood sender sets *cwnd* and *ssthresh* to appropriate values given the estimated bandwidth consumed by the connection at the time of congestion. In particular, at the beginning of fast recovery, *ssthresh* is set to the bandwidth-delay product (BDP), where the bandwidth is the current estimate (as described above) and the delay is the minimum RTT observed. If $cwnd > ssthresh$, then *cwnd* is set to *ssthresh*, otherwise *cwnd* is set as in TCP Reno (*i.e.*, *cwnd* is reduced by 50%). If a loss is detected via the expiration of the RTO timer, *ssthresh* is set to the BDP and *cwnd* is set to 1. slow start will be entered, and the flow will quickly increase its congestion window to *ssthresh* if the bandwidth is available.

2.3.5 TCP Peach

TCP Peach is a set of modifications to TCP Reno to improve communication over satellite networks in a QoS-enabled Internet [AMP01]. The two main changes are the replacement of slow start with "sudden start" and the replacement of fast recovery with "rapid recovery." TCP Peach also includes a small change to congestion avoidance to guarantee TCP-friendliness.

TCP Peach uses the transfer of dummy segments to probe the network for additional bandwidth. These dummy segments have a low-priority bit set in the type-of-service (TOS) field in the IP header. QoS-enabled routers would process the TOS bits and know that in times of congestion they should drop dummy segments first. When a sender receives ACKs for dummy segments, it knows that there is more bandwidth available in the network.

Slow Start

In sudden start, *cwnd* is initialized to 1. Every $RTT/rwnd$ seconds, a dummy segment is sent, where *rwnd* is the receiver's advertised window. After sending $rwnd - 1$ dummy segments, congestion avoidance is entered, so slow start lasts for only one RTT. For each dummy segment that is acknowledged (which would occur during congestion avoidance), *cwnd* is incremented.

Congestion Avoidance

The modification that TCP Peach makes to congestion avoidance controls when to open the congestion window. The variable *wdsn* is initialized to 0 and is used to make sure that during congestion periods, TCP Peach exhibits the same behavior as TCP Reno. When an ACK for a dummy segment is received and $wdsn = 0$, *cwnd* is incremented. If $wdsn \neq 0$, *wdsn* is decremented and *cwnd* is unchanged. No further changes are made to congestion avoidance, and *cwnd* is increased as in TCP Reno.

Data Recovery

Rapid recovery, which replaces fast recovery, lasts for one RTT and is triggered by the receipt of three duplicate ACKs, as in TCP Reno. The congestion window is halved, just like in fast recovery, so $cwnd = cwnd_0/2$, where $cwnd_0$ is the congestion window when loss was detected. The source then sends *cwnd* segments along with $cwnd_0$ dummy segments. The first *cwnd* dummy segments sent during rapid recovery that are acknowledged represent the amount of bandwidth that was available before the loss occurred. The congestion window should only be incremented for each dummy segment acknowledged after the first *cwnd* dummy segments have been acknowledged. To avoid increasing the congestion window for the first *cwnd* dummy ACKs received, *wdsn* is initially set to *cwnd*. The rapid recovery

phase, like fast recovery, ends when an ACK is received for the retransmitted segment. If the retransmitted segment is not dropped, its ACK will return before ACKs for any dummy segments.

If the segment drop was due to congestion, then either none or very few of the second *cwnd* dummy segments will be acknowledged. In this way, the congestion window has been halved and congestion avoidance is entered, just like in TCP Reno. If the segment drop is due to a link error, then most of the second *cwnd* dummy segments will be acknowledged and the congestion window will quickly grow back to $cwnd_0$, the level it was before the segment was dropped. TCP Peach does not explicitly distinguish between link errors and congestion drops, but it allows flows that suffer link errors to recover quickly.

2.3.6 AIMD Modifications

Several modifications to the parameters of TCP's AIMD congestion window adjustment during congestion avoidance have been proposed, including Constant Rate and Increase-By-K. These changes were made to address TCP's unfairness to flows with long RTTs [FP92]. This unfairness stems from the fact that congestion window increases are based on the rate of returning ACKs. Flows with long RTTs will see fewer increases of *cwnd* than a flow with a shorter RTT in the same amount of time.

Congestion Avoidance

The Constant Rate algorithm was introduced by Floyd [FP92, Flo91a]. Constant Rate changes the additive increase of TCP's congestion window to

$$w(t+1) = w(t) + (c * rtt * rtt)/w(t),$$

where $w(t)$ is the size of the congestion window at time t, c is a constant controlling the rate of increase, and rtt is the average RTT in seconds. This change is equivalent to allowing each flow increase *cwnd* each second by c segments.

Henderson *et al.* modified the Constant Rate algorithm to address the problem of very bursty transmission if each ACK increased the congestion window by several segments [HSMK98]:

$$w(t+1) = w(t) + min((c * rtt * rtt)/w(t), 1),$$

where $w(t)$, t, c, and rtt are as before.

An appropriate value of c is not easy to determine. The value c can be thought of as regulating the aggressiveness of the flow to be the same as a flow with a RTT of rtt. If $c = 25$, a RTT of 200 ms makes the numerator equal to 1. So, the flows with $c = 25$ would be on average as aggressive as standard TCP Reno flows with a base RTT of 200 ms. Henderson

recommends a value of c less than 100. Additionally, Henderson found that if only some flows used the Constant Rate (or Modified Constant Rate) algorithm while other flows used TCP Reno, the benefit of Constant Rate was diminished.

Henderson *et al.* also proposed the Increase-By-K algorithm as an improvement to Constant Rate [HSMK98]. This algorithm allows flows with long base RTTs to improve their performance without requiring all flows to make the change. Whenever the congestion detection mechanism signals that the network is not congested, *cwnd* is increased in the following manner:

$$w(t+1) = w(t) + min((K/w(t)), 1).$$

Suggested values of K are 2-4. In TCP Reno, for example, $K = 1$. As with the Modified Constant Rate algorithm, the minimum term is added to ensure that the congestion window does not grow faster than during TCP Reno slow start.

2.4 Non-TCP Congestion Control

Many network applications, such as streaming media and networked games, do not require reliability and use UDP rather than TCP for data transfer. These applications may employ either no congestion control strategy or a congestion control strategy that is far different than TCP. Flows that do not respond to network congestion are called *unresponsive* and could harm the performance of TCP flows. Developers of applications that use these non-TCP flows are encouraged to include congestion control mechanisms, especially those that are *TCP-friendly*, meaning that they receive the same throughput as a TCP flow would have given the same network characteristics (RTT and drop probability).

Bansal *et al.* classify non-TCP congestion control algorithms based on their steady-state and transient state behaviors [BBFS01]. In steady state, an algorithm can be either *TCP-equivalent*, *TCP-compatible*, or *not TCP-compatible*. In a transient state, an algorithm can be *TCP-equivalent*, *slowly-responsive*, or *responding faster than TCP*. TCP-equivalent algorithms use a transmission window that grows according to an AIMD algorithm with the same parameters as TCP[1]. A congestion control algorithm is TCP-compatible if it obtains close to the same throughput as TCP in steady-state. TCP-compatible algorithms see the same performance as TCP given the same loss rate and RTT. TCP-equivalent algorithms are also considered TCP-compatible. Slowly-responsive algorithms reduce their sending rate less than TCP in reaction to a single segment loss or congestion notification (*i.e.*, these also transmit faster than TCP). An AIMD algorithm with different parameters than TCP is an example of a slowly-responsive algorithm. Slowly-responsive algorithms are not guaranteed to be TCP-compatible.

[1]From here on, TCP refers to TCP Reno.

2.4.1 Congestion Control Principles

In RFC 2914, Sally Floyd explains why congestion control is needed and what is required for correct congestion control [Flo00a]. Non-TCP protocols, such as streaming media protocols, which are unresponsive to network congestion, have the potential to be unfair to TCP flows. If flows are unresponsive to network congestion, they will continue to send packets at a rate that is unsustainable by the network. This could cause many packets to be dropped, including those from TCP flows that would then reduce their sending rates. Flows that are unresponsive may not cause congestion collapse, but they could cause starvation among TCP flows. TCP is designed such that flows with similar RTTs should see similar performance. Non-TCP flows are encouraged to be TCP-compatible in order to be fair to TCP flows.

2.4.2 Control-Theoretic Approach

Keshav describes a control-theoretic approach to flow control [Kes91]. This approach requires that a sender can observe changes in the state of the network. This can be achieved if routers use a round-robin queuing and scheduling discipline for packets of different flows and a "packet-pair" probing technique. A round-robin queuing mechanism reserves a separate FIFO queue for each flow and services one packet from each queue in a round-robin manner. Under round-robin queuing, the service rate of a single flow will depend upon the number of other flows on the link rather than the incoming rates of other flows on the link, as with FIFO queuing. Packet-pair probing is a method of determining the bottleneck bandwidth of a path. A source sends out two packets back-to-back. If the queuing scheme is round-robin, it is guaranteed that the bottleneck rate is $1/\alpha$, where α is the duration between the receipt of the packets at the destination. When the sender then receives an ACK for each packet in the packet pair, the sender estimates the bottleneck link speed from the resulting inter-ACK gap. It is assumed that the original packet spacing is preserved in the ACKs. ACKs are sent as data packets arrive and, with round-robin scheduling, their spacing does not change.

Keshav points out that such a control-theoretic approach would be very hard to implement on a network where all routers use FIFO queuing because there is no easy method for monitoring the network state. With round-robin queuing, a flow could determine how many other flows shared the link by observing the time between two of its packets being sent.

The goal of this flow control algorithm is to maintain a certain number of packets queued at the bottleneck router. For example, for maximum link utilization, there should always be at least one packet in the queue to ensure there is data to send when resources are available.

2.4.3 General AIMD

Chiu and Jain analyzed AIMD algorithms and showed that, for fairness and stability, α should be greater than 0 and β should be between 0 and 1 [CJ89]. Yang and Lam extend this

to show that to be considered TCP-friendly [YL00],

$$\alpha = \frac{4(1-\beta^2)}{3}.$$

Yang and Lam further found that $\alpha = 0.31$ and $\beta = 0.875$ was a TCP-friendly setting that lowered the number of rate fluctuations. The authors derive a general model for the sending rate of flow using general AIMD based on α, β, the loss rate, RTT, timeouts, and the number of segments acknowledged by each ACK[2].

2.4.4 DECbit

Ramakrishnan and Jain propose a scheme called DECbit where a router indicates congestion to a source by setting a bit in the packet header during times of congestion [RJ90]. The router indicates congestion when the average queue size is non-zero. If a receiver sees congestion bits set in an incoming packet, it echos those bits in the ACK to notify the sender of congestion. If a majority of ACKs from the latest window have congestion indication bits set, *cwnd* is decreased. Otherwise, *cwnd* is increased. The authors follow the AIMD congestion window adjustment policy and recommend setting $\alpha = 1$ and $\beta = 0.875$.

The average queue size is based upon the queue regeneration cycle, which is the time between busy and idle periods (build up and drain of the queue). The average queue size during a regeneration cycle is given by the sum of the queue size during the cycle divided by the time of the cycle. For making congestion decisions, the router computes the average queue size based on both the current regeneration cycle and the previous regeneration cycle. Since this computation is performed for every packet entering the queue, as the current regeneration cycle grows, the contribution of the previous regeneration cycle diminishes. This mechanism allows for the congestion signal to take into account long regeneration cycles and use more current information in this case.

The sender waits for a window's worth of packets to be acknowledged after a change to *cwnd* before making another change to *cwnd*. When a window update is made, the first packet to reflect the change in window size is the ACK of the first packet of the next window. The sender waits until the entire next window has been acknowledged before deciding how to update *cwnd*. For example, if W_p is the window size before the change and W_c is the window size after the change, a new change to the window is not made until at least $(W_p + W_c)$ packets have been acknowledged (which requires waiting approximately two RTTs). The sender only uses the information from the last W_c ACKs to make its window change decision. This reduces the oscillation of the window and keeps the sender from reacting prematurely to transient increases in delay. The authors also suggest that all history about the state of the network should be cleared after a window update has been made. Thus, after a window

[2]This takes into account delayed ACKs, where two segments can be acknowledged by each ACK.

update, only the congestion bits from the current window are used in deciding how to adjust the next window. There is a balance to be struck between the percentage of ACKs with congestion indications required for making a decision and the router's threshold for detecting congestion. The authors use a router threshold of 1 packet and a 50% percent threshold of ACKs returning with congestion indications. So, routers will mark packets with congestion indications if the average queue size is greater than 1 packet. If 50% or more of the ACKs in a window return with congestion indications, the sender will reduce *cwnd*.

2.4.5 Delay-Based Approach

Jain presents a delay-based approach to congestion avoidance where the RTT is used as a congestion signal [Jai89]. Jain also distinguishes between selfish optimum and social optimum strategies and discusses methods for deciding whether to increase or decrease the window, what increase/decrease algorithm to use, and how often to decide to change the window.

When multiple flows are competing for network resources, the flows can either choose to find the social optimum or the selfish optimum. A selfish optimum leads to packet loss because flows try to keep increasing their windows until there is no longer buffer space available. In a socially optimum scheme, some flows back off while other flows increase their windows in order for there to be fair sharing of network resources. To determine the social optimum with non-uniform packet service times, flows may need to know the number of the other flows in the network and their window sizes.

To determine how to change the window in a simplified system where there are no other flows, the *normalized delay gradient* (NDG) is computed:

$$NDG = (\frac{RTT - RTT_{old}}{RTT + RTT_{old}})(\frac{W + W_{old}}{W - W_{old}}),$$

where RTT is the RTT at window size W and RTT_{old} is the RTT at window size W_{old}. NDG is proportional to the load in the system. If $NDG > 0$, then the window is decreased. If $NDG \leq 0$, then the window is increased. As in DECbit [RJ90], this algorithm uses an AIMD adjustment with $\alpha = 1$ and $\beta = 0.875$. The window is also updated every other RTT.

Jain suggests that if the minimum delay is known, then the socially optimum window size can be determined without knowing the windows of the other flows. This is done by estimating the load put on the network by the other flows, which is proportional to the difference in the delay at startup (when $W = 1$) and the minimum delay.

2.4.6 Binomial Algorithms

Bansal and Balakrishan present binomial algorithms, where the window increase/decrease factors are proportional to a power of the current congestion window [BB01]. In general, the increase algorithm is $w(t + 1) = w(t) + \alpha/w(t)^k$, and the decrease algorithm is

$w(t+1) = w(t) - \beta w(t)^l$. AIMD is represented by $k = 0$ and $l = 1$. Multiplicative-increase/multiplicative-decrease (MIMD), used in TCP slow start, is $k = -1$ and $l = 1$. The authors show that as long as $k + l = 1$ and $l \leq 1$, the binomial algorithm is TCP-compatible. Bansal and Balakrishnan look particularly at two examples of the binomial algorithm: inverse increase/additive decrease (IIAD) where $k = 1$ and $l = 0$ and SQRT where $k = 1/2$ and $l = 1/2$. TCP flows using binomial algorithms can see higher throughput than competing TCP AIMD flows when drop-tail routers are used. This unfairness to TCP AIMD flows would be remedied if routers with active queue management (specifically RED, see section 2.5.1) were used instead of drop-tail routers. The authors argue that this unfairness is not a problem for binomial algorithms, but rather evidence that drop-tail queuing should not be used.

2.5 Active Queue Management

Internet routers today employ traditional FIFO queuing (called "drop-tail" queue management). Active queue management (AQM) is a router-based congestion control mechanism where a router monitors its queue size and makes decisions on how to admit packets to the queue. Traditional routers use a drop-tail policy, where packets are admitted whenever the queue is not full. A drop-tail router thus only monitors whether or not the queue is filled. AQM routers potentially drop packets before the queue is full. These actions are based on the fundamental assumption that most of the traffic in the network employs a congestion control mechanism, such as TCP Reno, where the sending rate is reduced when packets are dropped. Many AQM algorithms are designed to maintain a relatively small queue but allow short bursts of packets to be enqueued without dropping them. In order to keep a small queue, often many packets are dropped "early," *i.e.*, before the queue is full. A small queue results in lower delays for packets that are not dropped. The low delay resulting from a small queue potentially comes at the cost of higher packet loss due to early drops than would be seen with losses only due to an overflowing queue as with drop-tail routers.

2.5.1 Random Early Detection

Random Early Detection (RED) is an AQM mechanism that seeks to reduce the long-term average queue length in routers [FJ93]. Under RED, as each packet arrives, routers compute a weighted average queue length that is used to determine when to notify end-systems of incipient congestion. Congestion notification in RED is performed by "marking" a packet. If, when a packet arrives, congestion is deemed to be occurring, the arriving packet is marked. For standard TCP end-systems, a RED router drops marked packets to signal congestion through packet loss. If the TCP end-system understands packet-marking, a RED router marks and then forwards the marked packet towards its destination.

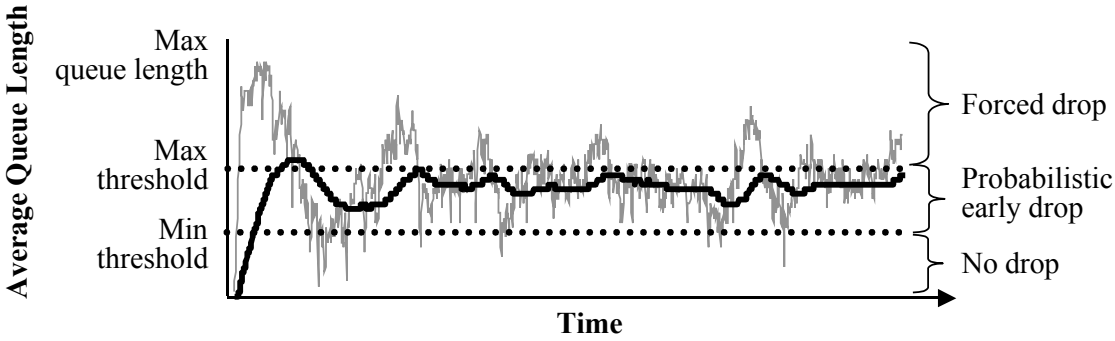


Figure 2.1: RED States. The gray line is the instantaneous queue size, and the black line is the weighted average queue size.

RED's congestion detection algorithm depends on the average queue size and two thresholds, min_{th} and max_{th}. When the weighted average queue size is below min_{th}, RED acts like a drop-tail router and forwards all packets with no modifications. When the average queue size is between min_{th} and max_{th}, RED probabilistically marks incoming packets. When the average queue size is greater than max_{th}, all incoming packets are dropped. These states are shown in Figure 2.1.

The more packets a flow sends, the higher the probability that its packets will be marked. In this way, RED spreads out congestion notifications proportionally to the amount of space in the queue that a flow occupies. The probability that an incoming packet will be dropped is varied linearly from 0 to max_p when the average queue size is between min_{th} and max_{th}. This is illustrated in Figure 2.2.

The following equations are used in computing the average queue size and drop probability in RED:

- exponential weighted moving average:
  $avg_{new} = (1 - w_q)avg_{old} + w_q *$ current queue length
- drop probability: $p_a = p_b/(1 - count * p_b)$, where
  $p_b = (max_p(avg - min_{th}))/(max_{th} - min_{th})$ and $count$ is the number of packets that have been admitted to the queue since the last packet was marked

The RED thresholds min_{th} and max_{th}, the maximum drop probability max_p, and the weight given to new queue size measurements w_q, play a large role in how the queue is managed. Recommendations on setting these RED parameters specify that max_{th} should be set to three times min_{th}, w_q should be set to 0.002, or 1/512, and max_p should be 10% [Flo97].

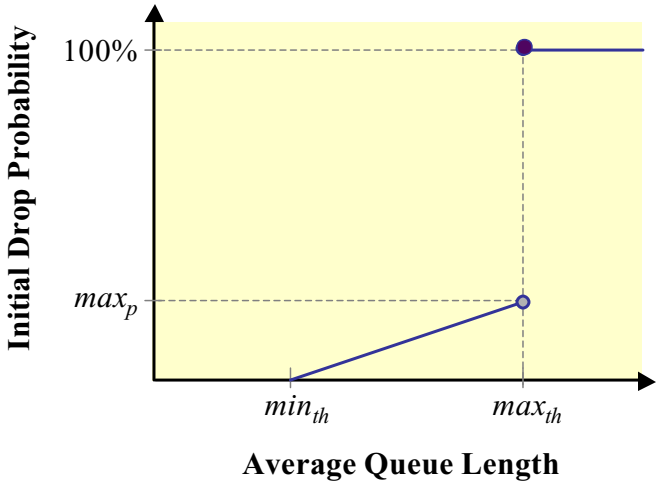


Figure 2.2: RED Initial Drop Probability (p_b)

There have been several studies [CJOS01, BMB00] of network performance with RED routers. Christiansen *et al.*, in particular, looked at the performance of HTTP traffic over RED routers [CJOS01]. Their original goal was to find RED parameter settings that were optimal for links that carried a large majority of HTTP traffic. What they found was that end-user metrics of performance were very sensitive to the parameter settings, the recommended settings performed worse than if the network consisted of all drop-tail routers, and determining optimal parameters for RED was non-intuitive.

2.5.2 Explicit Congestion Notification

Explicit Congestion Notification (ECN) is a form of AQM that allows routers to notify end systems when congestion is present in the network by setting a bit in the packet, as in DECbit [Flo94, RF99]. Ramakrishnan and Floyd set down the following design assumptions (or limitations) of ECN, which are applicable to any early congestion detection method [RF99]:

- ECN can only help alleviate congestion events that last longer than a RTT.
- ECN can only help alleviate congestion caused by flows that carry enough segments to be able to respond to feedback.
- It is likely that the paths followed by data and ACKs are asymmetric.

ECN is recommended for use in routers that monitor their average queue lengths over time (*e.g.*, routers running RED), rather than those that can only measure instantaneous queue lengths. This allows for short bursts of packets without triggering congestion notifications. Most studies of ECN have assumed that RED is the AQM scheme used.

When an ECN-capable RED router detects that its average queue length has reached a min_{th}, it marks packets by setting the CE ("congestion experienced") bit in the packets' IP

headers. When an ECN-capable receiver sees a packet with its CE bit set, an ACK with its ECN-Echo bit set is returned to the sender. Upon receiving an ACK with the ECN-Echo bit set, the sender reacts in the same way as it would react to a packet loss (*i.e.*, by halving the congestion window). Ramakrishnan and Floyd recommend that since an ECN notification is not an indication of packet loss, the congestion window should only be decreased once per RTT, unless packet loss does occur. Thus, a TCP sender implementing ECN receives two different notifications of congestion, ECN and packet loss. This allows senders to be more adaptive to changing network conditions.

ECN has the same feedback constraints as any closed-feedback loop scheme. It takes at least 1/2 RTT for the ECN notification to reach the sender after congestion has been detected by the router. Since congestion can shift to different parts of the network, ECN works optimally when both end systems and all intermediate routers are ECN-capable.

Studies of network performance of ECN-capable RED routers have shown that using ECN is much more preferable than dropping packets upon congestion indications [SA00, ZQ00]. Packet loss rates are lower and throughput is higher. Note that performance of ECN-capable flows is still closely tied to the parameters of the AQM scheme used.

2.5.3 Adaptive RED

Adaptive RED is a modification to RED which addresses the difficulty of setting appropriate RED parameters [FGS01]. Studies have found that the "best" parameter settings depend upon the traffic mix flowing through the router, which changes over time. RED performance depends upon how the four parameters min_{th} (minimum threshold), max_{th} (maximum threshold), max_p (maximum drop probability) and w_q (weight given to new queue size measurements) are set. Adaptive RED adapts the value of max_p so that the average queue size is halfway between min_{th} and max_{th}. The maximum drop probability, max_p is kept between 1-50% and is adapted gradually. Adaptive RED includes another modification to RED, called "gentle RED" [Flo00b]. In gentle RED, when the average queue size is between max_{th} and $2 * max_{th}$, the drop probability is varied linearly from max_p to 1, instead of being set to 1 as soon as the average is greater than max_{th}. These modifications to max_p and the drop probability are shown in Figure 2.3. Additionally, when the average queue size is between max_{th} and $2 * max_{th}$, selected packets are no longer marked, but always dropped. These states are pictured in Figure 2.4.

Adaptive RED adds two parameters to RED, α and β. α controls the increase rate of max_p, and β is the decrease factor of max_p. The authors suggest that $\alpha < \frac{max_p}{4}$ so that, in each interval, the average queue size will not fluctuate too wildly. The default setting of α is $min(0.01, \frac{max_p}{4})$. The same consideration should be made in setting β. The authors suggest that $\beta > 0.83$.

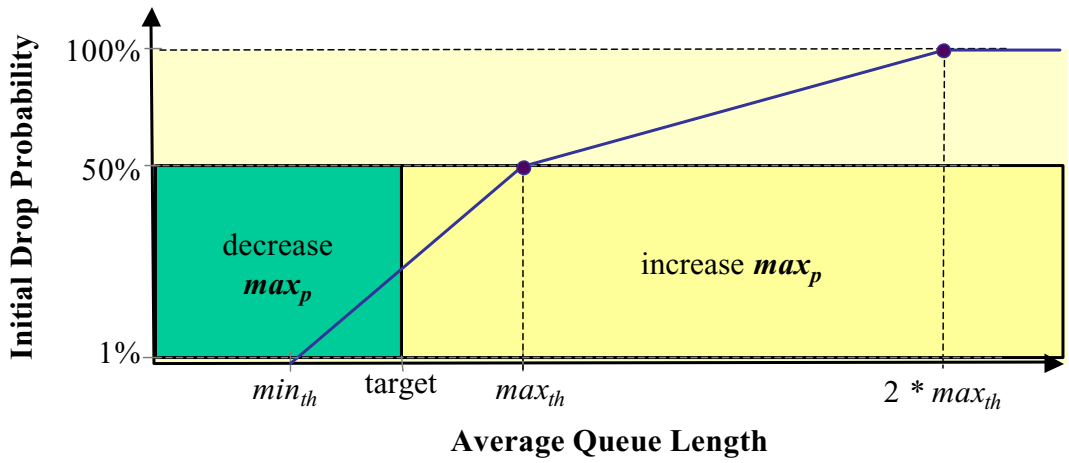


Figure 2.3: Adaptive RED Initial Drop Probability (p_b)

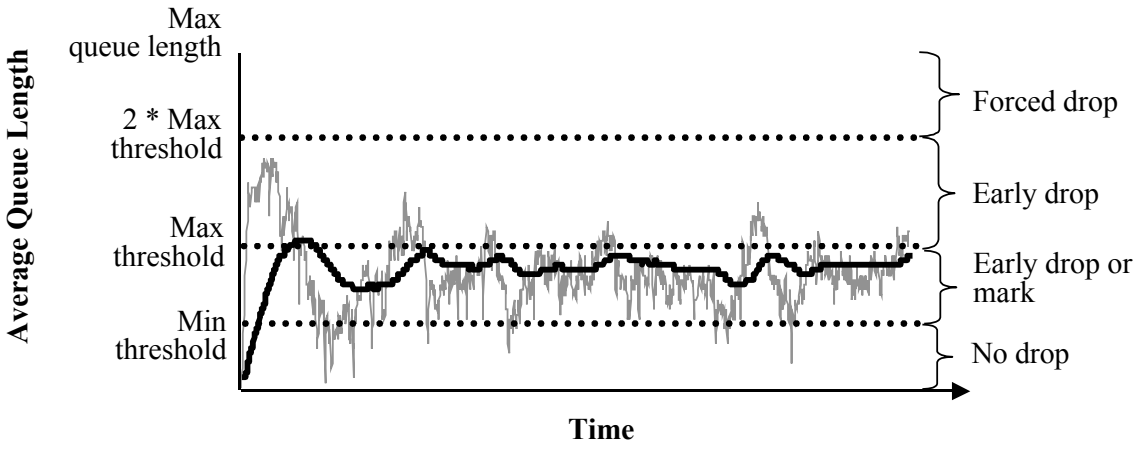


Figure 2.4: Adaptive RED States. The gray line is the instantaneous queue size, and the black line is the weighted average queue size.

Adaptive RED's developers provide guidelines for the automatic setting of min_{th}, max_{th}, and w_q. Setting min_{th} results in a tradeoff between throughput and delay. Larger queues increase throughput, but at the cost of higher delays. The rule of thumb suggested by the authors is that the average queuing delay should only be a fraction of the RTT. If the target average queuing delay is $delay_{target}$ and C is the link capacity in packets per second, then min_{th} should be set to $delay_{target} * C/2$. The guideline for setting max_{th} is that it should be $3 * min_{th}$, resulting in a target average queue size of $2 * min_{th}$. The weighting factor w_q controls how fast new measurements of the queue affect the average queue size and should be smaller for higher speed links. This is because a given number of packet arrivals on a fast link represents a smaller fraction of the RTT than for a slower link. It is suggested that w_q be set as a function of the link bandwidth, specifically, $1 - e^{-1/C}$.

2.6 Analysis of Internet Delays

Sync-TCP is based on the idea that having exact information about the delays that packets from a flow encounter can be used to improve congestion control. This section highlights previous work on how delays can be used to determine certain characteristics of a network.

2.6.1 Queuing Time Scales

Paxson looked at estimating the time scales over which queuing varies [Pax99]. The time scale of queuing delay variation is important because if queues fluctuate over a time scale less than a RTT, then rate adaptations based on queuing delay would not have time to acquire feedback about the effect of the adaptation, and therefore would be useless. Paxson obtained one-way transit time (OTT) data from off-line analysis of packet traces gathered from 20,000 TCP bulk transfers. Clock errors in the traces were detected using algorithms developed for finding clock adjustments and skew [Pax98].

Paxson found that, in the traces, flows typically experience the greatest variation in queuing delay on time scales between 0.1 - 1 seconds. In other words, if a connection was divided into intervals of time, the largest queuing delay variation would be seen in intervals whose length was between 0.1 - 1 seconds. This is encouraging because it indicates that the majority of congestion periods (where queuing delays are high) occur on time scales likely to be longer than most RTT values (typically 20-200 ms), so adaptations have the potential to have a positive effect on congestion.

2.6.2 Available Bandwidth Estimation

Paxson also looked at using OTTs to estimate the amount of available bandwidth in the network [Pax99]. Let λ_i be the amount of time spent by packet i at the bottleneck for transmission and queuing behind packets in the same flow as packet i. The service time of

a b-byte packet, Q_b, is b/B, where B is the bottleneck bandwidth in bytes per second. For all packets with equal b, variation in OTT will reflect the queuing delay of a packet behind packets in its own connection. Variation in OTT, ψ_i, is given by $\lambda_i - Q_b$. The packet's total queuing delay, γ_i, is calculated by subtracting the packet's OTT from the connection's minimum observed OTT. If the connection is the only traffic on the path, $\psi_i = \gamma_i$. All queuing delay is due to other packets of the same connection. Let

$$\beta = \frac{\sum_i (\psi_i + Q_b)}{\sum_j (\gamma_j + Q_b)},$$

where the summation range is over all packets in a connection. β is the proportion of the packet's delay due to packets from its own connection and reflects the amount of available bandwidth in the network. If $\beta \approx 1$, then the connection is receiving all of the bandwidth that it is requesting. All of the queuing delay experienced by the packets is due to that connection's own packets. If $\beta \approx 0$, then this connection's packets are spending the majority of their queuing time behind packets from other connections. So, the load contributed by this connection is relatively insignificant. $\beta \approx 1$ does not mean that the connection is consuming all of the network resources, just that any time it tried to consume more resources, they were available. By computing β from actual traces, Paxson reported that paths with higher bottleneck bandwidths usually carry more traffic (and more connections) and have lower values of β. Also, β is a fairly good predictor of future levels of available bandwidth. Paxson observed that for the same path, one observation of β will be within 0.1 of later observations for time periods up to several hours. Allman and Paxson mention the possibility of using a measure of the available bandwidth in the network to allow connections to ramp up to their fair share more quickly than slow start [AP99].

2.6.3 Delays for Loss Prediction

One of the problems with current congestion control is that on a wireless link it is difficult to distinguish between losses due to wireless link errors and losses due to congestion. The detection of congestion losses should trigger a reduction in the sending rate, while link error losses should not cause a reduction in the sending rate.

Samaraweera and Fairhurst looked at the feasibility of using changes in RTTs as a network congestion detector [SF98]. This would allow wireless senders to distinguish between congestion losses (those preceded by an increase in RTT) from link error losses (those that follow no discernible increase in RTT). Using changes in the RTT as an estimate of queuing delay requires obtaining a minimum RTT measurement close to the actual round-trip propagation delay of the flow. Subsequent increases in RTT can be compared to the minimum RTT to determine the degree of congestion. Samaraweera and Fairhurst noted three problems with such an end-system-based method of inferring the type of loss:

- persistent congestion will cause an incorrect estimate of the minimum delay,
- AQM techniques will cause packet drops without a large build-up of the queue, and
- routing changes will make consecutive delay estimates unrelated.

Biaz and Vaidya looked at how well congestion avoidance algorithms predict packet loss [BV98]. The idea was that a congestion avoidance algorithm was good at predicting loss if, when it indicated that the congestion window should be increased, no packet loss occurred. For a predictor to be accurate, the authors suggested that the following requirements should be met:

- packet losses follow a large build-up in the queue,
- a build-up in the queue usually results in packet loss, and
- the predictor correctly diagnoses a large build-up in the queue.

Through live Internet tests and *ns* simulations, they found that the level of aggregation prevented a single flow's congestion avoidance algorithm from accurately predicting packet loss. The changes in the congestion window of a single flow did not have a large enough effect on the size of the queue at the bottleneck link to prevent packet loss.

Bolot found that unless a flow's traffic comprised a large fraction of the bottleneck link traffic, packet losses appeared to occur randomly (*i.e.*, without a corresponding build-up in delay) [Bol93]. If a flow is a small fraction of the traffic, only a few of its packets would be in the congested router's queue at any one time. With so few samples of the queuing delay during a time of congestion, it would be difficult to observe a significant increase in the queuing delay before packet loss occurred.

2.6.4 Delays for Congestion Control

Martin *et al.* looked at how well TCP flows with delay-based congestion avoidance mechanisms (DCAs), such as TCP Vegas, could compete with TCP Reno flows [MNR00]. Martin showed that small numbers of DCA flows cannot co-exist in the same network with TCP Reno flows without receiving less bandwidth than if there were no TCP Reno flows. The problem is that DCAs and TCP Reno have opposite goals. DCAs react to increased delays by slowing down their sending rates. TCP Reno flows try to overflow the queue in order to determine the available bandwidth of the network. Any bandwidth that DCA flows give up will be taken by TCP Reno flows, which then would cause DCA flows to back off even further. This study looked at the impact that competing with many TCP Reno flows would have on a single DCA flow. It did indicate, though, that if DCA flows represented a significant percentage of traffic, overall packet loss rates decreased.

2.7 Summary

Researchers have approached TCP congestion control either by adding mechanisms to the queuing algorithm in routers or by changing the TCP Reno congestion control protocol itself.

Those who look at the queuing algorithm in routers come from the point-of-view that the main problem is found in drop-tail queuing rather than TCP Reno. Since packet drops indicate to TCP Reno that congestion is occurring, the queuing algorithm should drop some packets before the queue overflows, whenever the router detects that congestion is occurring. The router monitors its queue and makes a decision of which packets to drop at what point in time. This approach adapts the queuing mechanism according to the behavior of the prevalent transfer protocol (*i.e.*, TCP). A change is made at the router in order to produce the desirable behavior in the end systems. If, in the future, a new congestion control protocol were released that did not depend on packet loss for congestion signals, these "early drop" queuing mechanisms might be a hindrance.

The opposite approach is to look at what the congestion control algorithm on the end system can do to detect congestion before it results in packet loss. Researchers have looked at using RTTs and throughput estimates as indicators of congestion [BOP94, PGLA99, Jai89]. This previous work is based on the assumption that the flow's RTT is composed equally of the data path delay and the ACK path delay. There is no guarantee that this assumption would be true.

Sync-TCP, an end-system approach to TCP congestion control, is built on the foundation of TCP Reno and uses TCP Reno's loss detection and error recovery mechanisms. Sync-TCP is also built on the work of researchers who looked at changes in RTTs and throughput as indications of congestion. I have added the assumption that all computers have synchronized clocks (*e.g.* via GPS) in order to investigate the benefit that could come from being able to accurately measure OTTs and separate the detection of data-path congestion from ACK-path congestion. As will be shown in detail in Chapter 3, the TCP timestamp option is the basis for the format of the Sync-TCP timestamp option that will be used to communicate synchronized timestamps between computers.

The idea behind Sync-TCP is that flows can be good network citizens and strive to keep queues small while still achieving high goodput. Jain's work on delay-based congestion control introduced the idea of the selfish optimum vs. the social optimum congestion control algorithm. The social optimum allows for some flows to slow down their sending rates while other flows increase their sending rates in order to maximize the efficiency of the network without overflowing network buffers. What has resulted with Sync-TCP, and will be shown in Chapter 4, is that some longer flows will voluntarily back off when congestion is detected so that short flows (those that are too short to run the Sync-TCP congestion control algorithm) see short queues and complete quickly. This does not mean that the long flows are starved (many even see better performance than with TCP Reno), but that all of the flows in the

network are cooperating to provide a better network environment featuring less packet loss and shorter queue sizes.

With Martin's work on the deployability of DCAs in mind, Sync-TCP will be evaluated in a homogeneous environment, where all flows will use the same congestion control protocol. The goal of my study is to look at the usefulness of synchronized clocks and exact timing information in TCP congestion control. If synchronized clocks are found to be useful in an ideal environment (all clocks are perfectly synchronized and no competing TCP Reno traffic), then techniques for incorporating these changes into the Internet can be developed later.

Chapter 3

Sync-TCP

Sync-TCP is a family of end-to-end congestion control mechanisms that are based on TCP Reno, but take advantage of the knowledge of one-way transit times (OTTs) in detecting network congestion. I use Sync-TCP as a platform for studying the usefulness of adding exact timing information to TCP.

This chapter describes the development of Sync-TCP, which consists of the following:

- identifying parts of TCP congestion control that could be improved,
- developing a mechanism so that computers with synchronized clocks can exchange OTTs,
- developing mechanisms that can detect network congestion using OTTs, and
- developing mechanisms to react appropriately to network congestion.

Qualitative evaluations of the congestion detection mechanisms are provided in this chapter, while evaluations of selected congestion detection mechanisms coupled with appropriate congestion reaction mechanisms are described in Chapter 4.

3.1 Problem and Motivation

On the Internet today, there is no deployed end-to-end solution for detecting network congestion before packet loss occurs. Detecting network congestion before packet loss occurs is the goal of active queue management (AQM), in general, and Random Early Detection (RED), in particular. These solutions, though, require changes to routers and are not universally deployed. The emergence of cheap Global Positioning System (GPS) receiver hardware motivates the study of how synchronized clocks at end systems could be used to address the problem of early congestion detection. In this section, I will discuss network congestion and problems with TCP congestion control.

3.1.1 Network Congestion

Network congestion occurs when the rate of data entering a router's queue is greater than the queue's service rate for a sustained amount of time. The longer a queue, the longer the queuing delay for packets at the end of the queue. It is desirable to keep queues small in order to minimize the queuing delay for a majority of packets. Because of the large numbers of flows that can be aggregated in a queue (and because of protocol effects such as TCP slow start), transient spikes in the incoming rate (and, therefore, in the router queue size) are expected and should be accommodated. In order for these spikes to be handled, the average queue size at a router should be small relative to the total queuing capacity of the router.

3.1.2 TCP Congestion Control

TCP's approach to congestion control is to conservatively probe the network for additional bandwidth and react when congestion occurs. End-to-end protocols, such as TCP, treat the network as a black box, where senders must infer the state of the network using information obtained during the transfer of data to a receiver. A TCP sender infers that additional bandwidth is available in the network by the fact that acknowledgments (ACKs) return for data it has sent. The sender infers that segments have been dropped by the expiration of the retransmission timer (reset whenever an ACK for new data is received) or by the receipt of three duplicate ACKs. TCP senders infer that network congestion is occurring when segments have been dropped. One problem with this inference is that TCP has no other signal of congestion. The only time a TCP sender knows to decrease its sending rate is in response to a segment drop. Thus, TCP's congestion control mechanism is tied to its error recovery mechanism. By using segment loss as its only signal of congestion and the return of ACKs as a signal of additional bandwidth, TCP drives the network to overflow, causing drops, retransmissions, and inefficiency. This inefficiency results from some segments being carried in the network only to be dropped before reaching their destination.

3.2 Goals

The main goal of Sync-TCP is to show that information from computers with synchronized clocks could be used to improve TCP congestion control. One of the ways that TCP congestion control could be improved is to provide more efficient transfer of data by reducing segment loss. If Sync-TCP can accurately detect and react to congestion without having to wait for segment loss, a network of Sync-TCP flows should see lower loss rates than a network of TCP Reno flows. Additionally, the information from synchronized clocks could be used to provide a richer congestion signal than the binary (congestion or no congestion) signal of TCP Reno. Given a richer congestion signal, a Sync-TCP sender could react in different ways to different degrees of congestion.

3.3 One-Way Transit Times

If two end systems have synchronized clocks, they can accurately determine the OTTs between them. If the OTTs are known, the queuing delay of the path can be computed. The difference between the minimum OTT and the current OTT is the forward-path queuing delay. As queues in the network increase, the queuing delay will also increase and could be used as an indicator of congestion.

Using queuing delays as a signal of congestion would allow a sender to reduce its sending rate before queues in routers overflow, avoiding losses and costly retransmissions. Monitoring OTTs for indications of congestion also has the benefit of allowing senders to quickly realize when periods of congestion have ended, where they could increase their sending rates more aggressively than TCP Reno.

Sync-TCP, an end-to-end approach to congestion control, uses OTTs to determine congestion, because OTTs can more accurately reflect queuing delay caused by network congestion than round-trip times (RTTs). Using RTTs there is no way to accurately compute the forward-path queuing delay. If there is an increase in the RTT, the sender cannot distinguish between the cause being congestion on the forward path or congestion on the reverse path.

There are a few limitations to using OTTs to compute queuing delays. If there is congestion in the network that exists when a flow begins and this congestion lasts for the entire duration of the flow, then the flow's minimum-observed OTT would not represent the propagation delay of the path. In this case, the difference between the current OTT and the minimum-observed OTT would not be a clear picture of the actual queuing delay.

Assuming, though, that routes are stable[1] and persistent congestion is not occurring, the minimum-observed OTT is (or, is close to) the propagation delay. This minimum-observed OTT may decrease if the flow began when the network was already congested. As the queues drain, the minimum-observed OTT will decrease. Because of the fluctuations in most queues, the minimum-observed OTT probably will be determined fairly early and remain stable for the remainder of the flow's duration.

3.4 Sync-TCP Timestamp Option

If two computers have synchronized clocks, they can determine the OTTs between them by exchanging timestamps. For Sync-TCP, I added an option to the TCP header, which is based on the RFC 1323 timestamp option [JBB92]. Figure 3.1 pictures the full TCP header along with a cut-out of the Sync-TCP timestamp option[2]. This TCP header option includes the OTT in μs calculated for the last segment received (OTT), the current segment's sending

[1] Paxson reported that in 1995 about 2/3 of the Internet had routes that persisted on the time scale of days or weeks [Pax97].

[2] See [Ste94] for a description of the header fields.

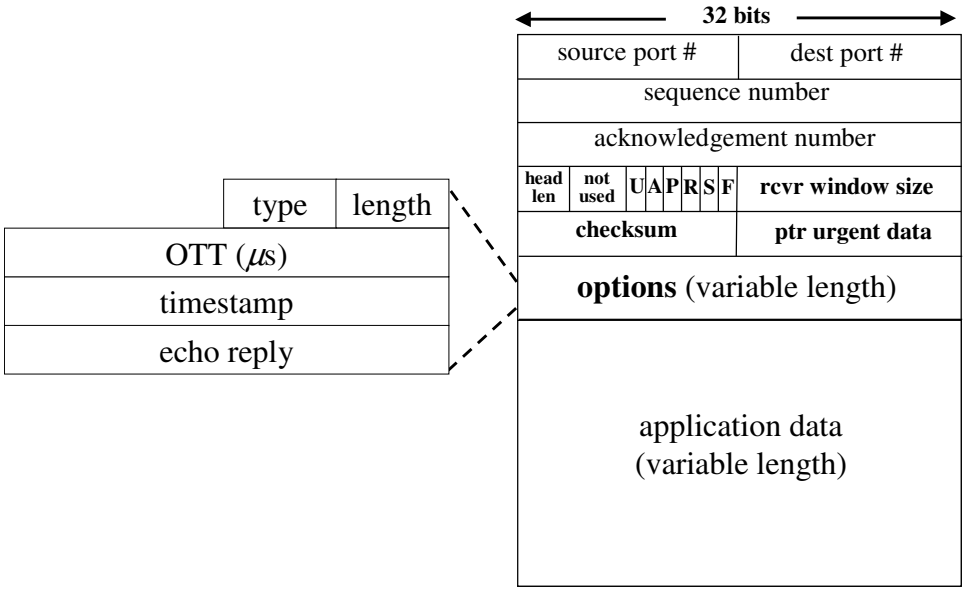


Figure 3.1: TCP Header and Sync-TCP Timestamp Option

time (*timestamp*), and the sending time of the last segment received (*echo reply*).

When the receiver sees any segment with the Sync-TCP timestamp option, it calculates the segment's OTT by subtracting the time the segment was sent from the time the segment was received. The OTT is then inserted into the next segment the receiver sends to the sender (generally an ACK). Upon receiving the ACK, the sender can calculate the following metrics:

- OTT of the data segment being acknowledged – This is provided in the OTT field of the Sync-TCP timestamp option.
- OTT of the ACK – This is the receive time minus the time the ACK was sent.
- The time the data segment was received – This is the time the data segment was sent plus the OTT.
- The time the ACK was delayed at the receiver – This is the time the ACK was sent minus the time the data segment was received.
- The current queuing delay – This is the current OTT minus the minimum-observed OTT.

Figure 3.2 provides an example of Sync-TCP timestamp option exchange and the computation of the OTTs. The first data segment is sent at time 2 and, for initialization, contains -1 as the entries for the OTT and the echo reply fields. When the data segment is received at time 3, the OTT is calculated as 1 time unit. When the ACK is generated and sent at time 4, the OTT of the data segment, the timestamp of the ACK, and the timestamp found in the data segment's option are sent in back in the Sync-TCP timestamp option to the sender. At time 5, when the ACK is received, the sender can calculate the OTT of the data segment it

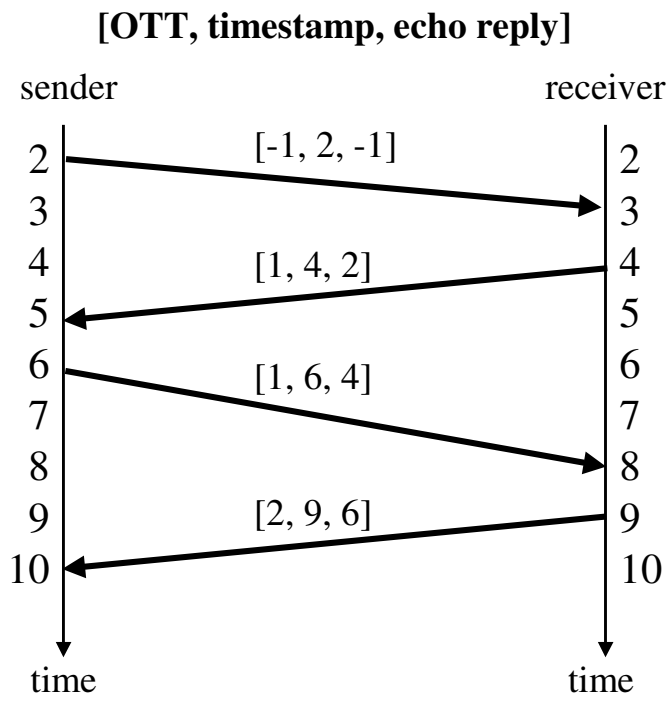


Figure 3.2: Sync-TCP Example

sent at time 2 (1 time unit), the OTT of the ACK (1 time unit), the time the data segment was received (time 3), and the amount of time the ACK was delayed (1 time unit). At time 10 when the second ACK is received, the OTT of the data segment is 2. Since the sender previously saw an OTT of 1, it can determine that the current queuing delay is 1 time unit.

3.5 Congestion Detection

In all of the congestion detection mechanisms I discuss, queuing delays are calculated from the exchange of OTTs in order to detect congestion. I began my study by looking at four different methods for detecting congestion using OTTs and queuing delays (each of these will be discussed in greater detail in the following sections):

- SyncPQlen (percentage of the maximum queuing delay) – Congestion is detected when the estimated queuing delay exceeds 50% of the maximum-observed queuing delay (which is an estimate of the maximum amount of queuing in the network).
- SyncPMinOTT (percentage of the minimum OTT) – Congestion is detected when the estimated queuing delay exceeds 50% of the minimum-observed OTT
- SyncAvg (average queuing delay) – Congestion is detected when the weighted average of the estimated queuing delays crosses a threshold, by default set to 30 ms.

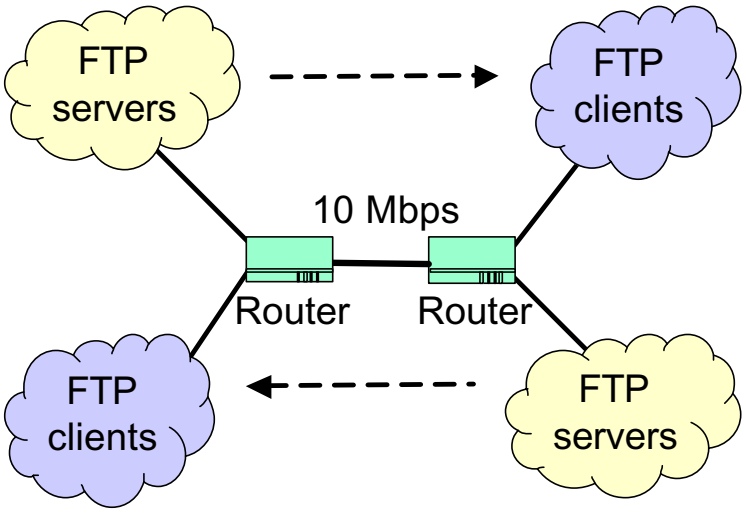


Figure 3.3: Network Topology for FTP Experiments

- SyncTrend (trend analysis of queuing delays) – Congestion is detected when the trend of the estimated queuing delays is increasing.

After evaluating these methods, I developed a congestion detection mechanism called SyncMix that is a combination of SyncPQlen, SyncAvg, and SyncTrend. In this section, I will describe each of the five congestion detection mechanisms I studied and the shortcomings of the first four mechanisms that led me to develop SyncMix.

To aid in the discussion of each of the congestion detection mechanisms, I will use a graph of the actual queuing delay over time at a bottleneck router as an example to illustrate the mechanism at work. All of the graphs are from simulations run with one 10 Mbps bottleneck link and 20 long-lived FTP flows transferred in each direction. Figure 3.3 shows the topology for these experiments. Different base (minimum) RTTs were used for the FTP flows. The motivation and source of these RTTs is explained in Chapter 4. The minimum RTT was 16 ms, the maximum RTT was 690 ms, and the median RTT was 53 ms. All of the FTP flows were run with TCP Reno as the underlying protocol and drop-tail queues at routers on each end of the bottleneck link. The congestion detection mechanisms only reported congestion but did nothing to react to those congestion indications. TCP Reno controlled all congestion window adjustments. For all of the experiments, the queue size seen at the bottleneck router was the same. Additionally, the estimated queuing delay for all of the experiments was the same (since the only variation among experiments was the congestion detection mechanism). Figure 3.4 shows the queuing delay as seen by packets entering the bottleneck router over a particular 5-second interval. The queuing delay at the router is calculated by dividing the number of bytes in the queue when the packet arrives by the speed of the outgoing link. Since

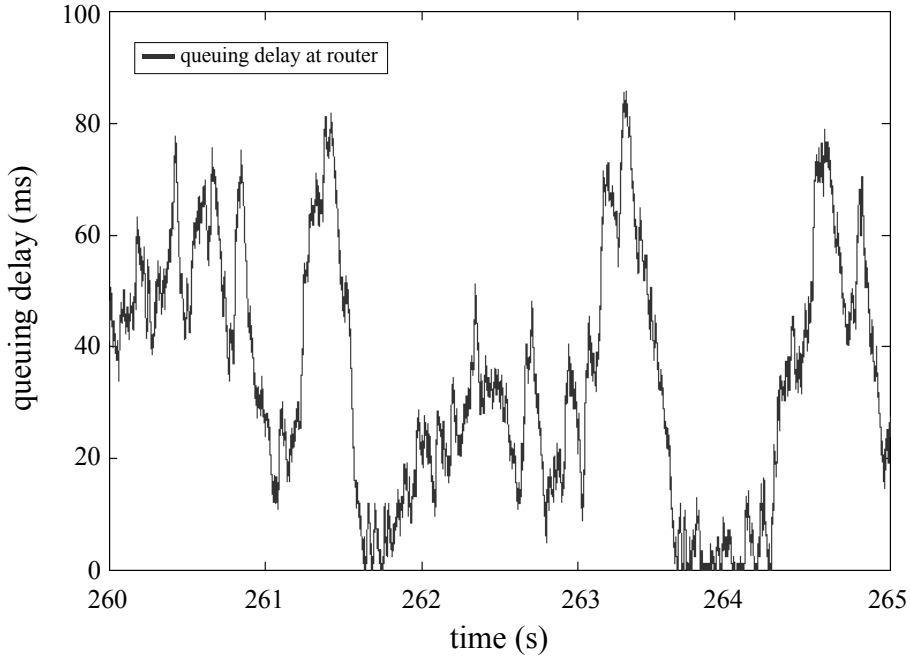


Figure 3.4: Queue Size at Bottleneck Router

these experiments were performed in simulation, the queue size (and queuing delay) can be monitored each time a new packet enters the queue. As can be seen even with a small number of long-lived flows, congestion and queuing delay varies widely. When data from a single FTP flow is shown on the graph (as will be done in the discussions of the individual congestion detection mechanisms), it is from the flow with the median RTT (53 ms).

Since Sync-TCP uses ACKs to report forward-path OTTs to the sender, the delay that a segment experiences will not be reported to the sender until its ACK is received. All of the FTP experiments were performed with delayed ACKs turned on, so in most cases, one ACK is sent to acknowledge the receipt of two data segments. If the congested link is close to the sender and the two data segments to be acknowledged arrive more than 100 ms apart, then it may take almost one RTT + 100 ms from the time congestion was observed to the time the sender knows about it. This is the worst-case delay notification. The best-case delay notification is the reverse-path OTT and would occur if the congested link was close to the receiver. If the paths are symmetric, this best-case delay is approximately 1/2 of the RTT.

Note that although the evaluation of the congestion detection mechanisms presented here considers only one congested link, experiments were run with two congested links and the same conclusions were reached. Even when there are multiple congested links, the difference between the OTT and the minimum-observed OTT estimates the total amount of queuing delay that a segment experiences. If the total amount of queuing delay increases over time,

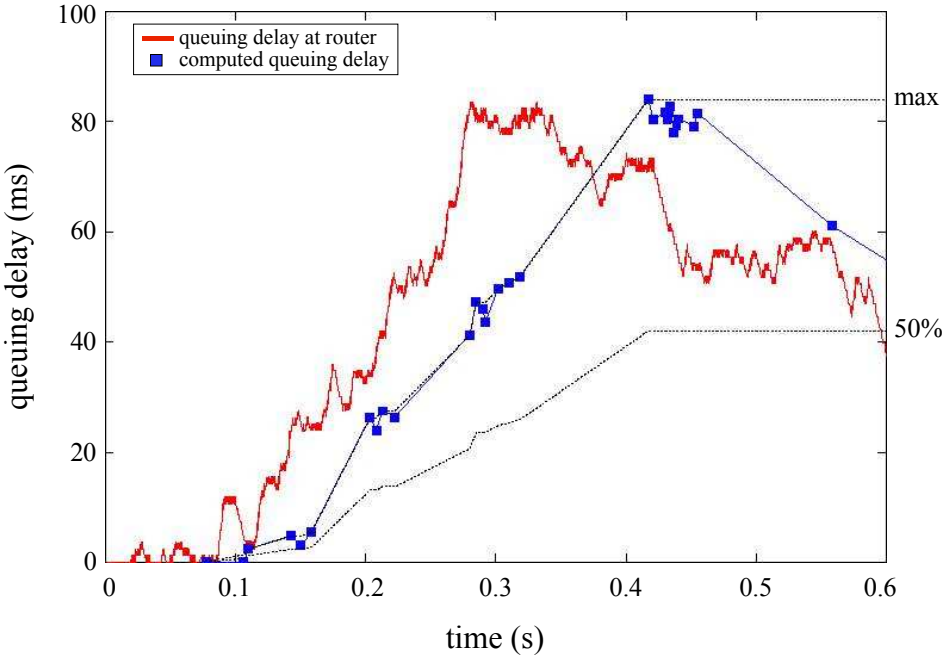


Figure 3.5: SyncPQlen Threshold Startup

then at least one of the intermediate queues is building up, signaling that congestion is occurring.

3.5.1 SyncPQlen

SyncPQlen records the minimum and maximum observed OTTs and uses them to estimate both the maximum and current queuing delay. SyncPQlen does not make congestion decisions until a segment has been dropped, so that the algorithm has a good estimate of the maximum amount of queuing in the network. The conjecture is that once a segment has been dropped, there is some segment in the flow (*i.e.*, the segment preceding or following the dropped segment) that will have seen a computed queuing delay close to the maximum amount of delay on the path, assuming no route changes. Each time a new OTT is received (*i.e.*, each time an ACK is received), SyncPQlen estimates the current queuing delay by subtracting the most recent OTT from the minimum-observed OTT. When this queuing delay exceeds 50% of the maximum-observed queuing delay, SyncPQlen signals that congestion is occurring. Frequently, both the maximum and minimum OTTs are observed fairly early in a flow's transfer. The minimum is often observed for one of the first segments to be transferred, and the maximum is often observed at the end of slow start when segments are dropped.

Figure 3.5 shows how the 50% of maximum-observed queuing delay threshold value changes with the maximum-observed OTT before segment loss occurs. The gray line is the queuing

delay as observed at the bottleneck link. The black squares are times at which ACKs were received and queuing delays were calculated at the sender for the corresponding ACKs. (The line between the squares is just a visualization aid. The queuing delay is only computed at discrete points.) The upper dotted line is the maximum-observed queuing delay. The lower dotted line is the SyncPQlen threshold of 50% of the maximum-observed queuing delay. A segment drop occurred just before time 0.3 (though it is not pictured). Around time 0.4 the sender receives a queuing delay sample reflecting the high queuing delay associated with the full queue. Until the segment loss is detected, the maximum-observed queuing delay is close to the current computed queuing delay as the queue builds up. After loss occurs, the computed and actual delays decrease but the maximum stays constant and will never decrease. Once segment loss has occurred, the assumption is that if the computed queuing delay gets close to the maximum-observed delay, the potential exists for additional segment loss to occur. Note the lag between the time the segment experienced a delay (on the gray line) and the time the sender was notified of that delay (at the black squares), *i.e.*, the horizontal distance between the gray and black lines at a given queuing delay. This lag is a function of the RTT and the actual queuing delay and will have a great impact on the algorithm's ability to sense congestion and react in time to avoid loss.

Discussion

Figure 3.6 shows the SyncPQlen congestion detection for one long-lived FTP flow with a 53 ms base RTT. The gray line is the queuing delay experienced by packets entering the router and is the same as in Figure 3.4. The black squares are the queuing delays computed by the sender. The top dotted line is the maximum-observed queuing delay, which occurred at some point previously in the flow's lifetime. The bottom dotted line is the 50% of maximum-observed queuing delay threshold. Whenever the most recent computed queuing delay is greater than the threshold, congestion is indicated. Whenever the most recent computed queuing delay value is lower than the threshold, the network is determined to be uncongested. By time 260, the flow has experienced at least one dropped segment, so the flow has a good estimate of the maximum-observed queuing delay. This is also evidenced by the fact that the maximum-observed queuing delay remains constant.

If there is a good estimate of the maximum amount of queuing available in the network, 50% of this value should be a good threshold for congestion. The goal is to react to congestion before packets are dropped. Using 50% of the maximum allows for there to be some amount of queuing, but allows flows to react before there is danger of packet loss. Using a threshold based on the amount of queuing delay in the network allows for multiple congested links. There is no predetermined threshold value, so the threshold can adapt to the total amount of queuing capacity in the path. As the number of bottlenecks increases (*i.e.*, between different paths) the queuing capacity in the network also increases. There is no way to pick a constant

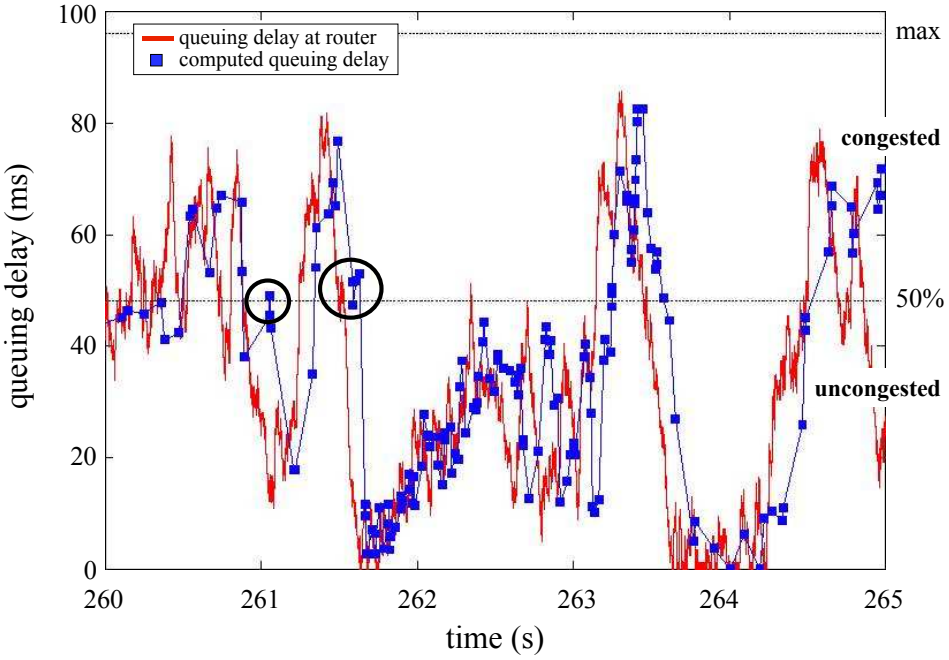


Figure 3.6: SyncPQlen Congestion Detection

queuing delay threshold without knowing the characteristics of the path ahead of time. Basing the threshold on a percentage of the maximum-observed queuing delay allows SyncPQlen to use a threshold without requiring *a priori* knowledge of the path.

One disadvantage of this algorithm is that until a segment is lost, SyncPQlen does not have a good estimate of the maximum queue size. Because of this limitation, this algorithm cannot operate during a flow's initial slow start phase. This may prevent the algorithm from operating on many short-lived flows, such as those prevalent in HTTP traffic.

Another disadvantage is that SyncPQlen operates on instantaneous values of the computed queuing delay. The circled areas in Figure 3.6 point out situations where the algorithm fails because of the noise in the instantaneous queuing delays. Just one sample above the threshold (*e.g.*, in the areas circled on the graph) would cause a congestion notification and could cause a reduction in the flow's sending rate, depending on the reaction mechanism. The algorithm is reacting to transient spikes in computed queuing delay due to using instantaneous OTTs. These spikes are a problem because they are not indicators of long-term congestion but would cause congestion to be detected.

3.5.2 SyncPMinOTT

SyncPMinOTT records the minimum-observed OTT for each flow. Anytime a new computed queuing delay is greater than 50% of the minimum-observed OTT, congestion is indi-

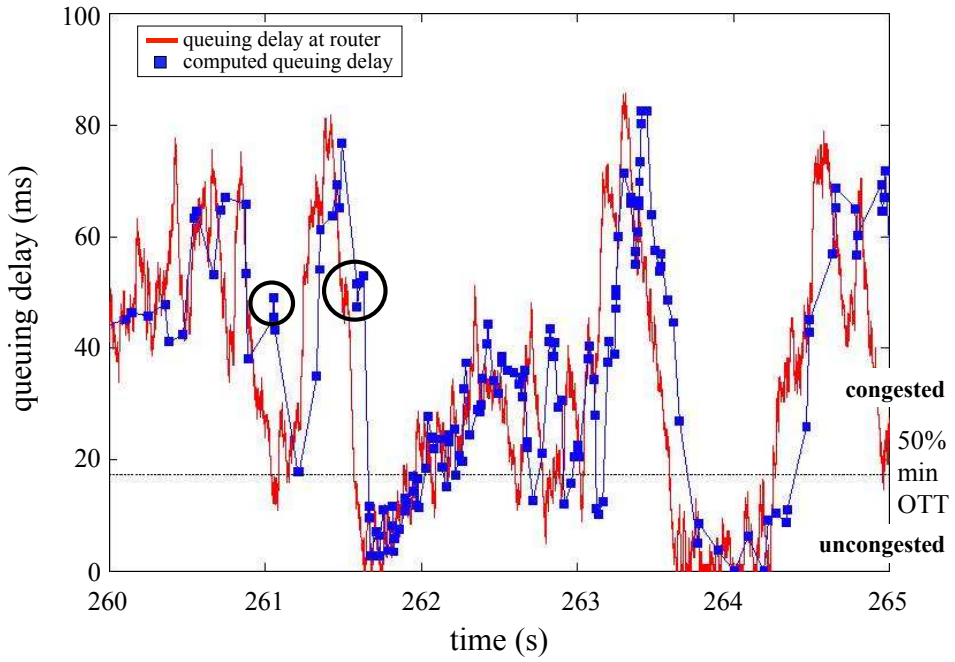


Figure 3.7: SyncPMinOTT Congestion Detection

cated. This algorithm was proposed in order to avoid waiting for a segment loss before having a good indicator of congestion. Basing the congestion detection mechanism on the minimum-observed OTT allows the congestion window of flows with large propagation delays to grow larger before SyncPMinOTT signals congestion than flows with short propagation delays. For example, if a flow had a 80 ms minimum-observed OTT, SyncPMinOTT would allow it to see computed queuing delays of up to 40 ms before signaling congestion. On the other hand, if a flow had a 20 ms minimum-observed OTT, SyncPMinOTT would signal congestion if the computed queuing delay ever reached above 10 ms. This could improve fairness between flows sharing the same path that have different RTTs, since it takes longer for flows with long base RTTs to recover from sending rate reductions than flows with short base RTTs. A flow with a long RTT would have a higher congestion threshold and hence would signal congestion less frequently than a flow with a shorter RTT.

Discussion

Figure 3.7 shows the SyncPMinOTT algorithm at work for one FTP flow with a 53 ms base RTT. As with the example in the previous section, the gray line is the queuing delay as seen by packets entering the router, and the black squares are the queuing delays as computed by the sender. (Note that since there is no congestion reaction, both the queuing delay at the router and the computed queuing delays here are exactly the same as that in Figure 3.6.) The

dotted line here is the 50% of minimum-observed OTT threshold. The minimum-observed OTT for this flow was 35 ms, so the threshold is 17.5 ms. If the most recent computed queuing delay is over the threshold, congestion is detected, otherwise the network is considered to be uncongested.

Unlike SyncPQlen, the threshold in SyncPMinOTT is not based on the amount of queuing capacity in the network, but rather on the propagation delay of the flow. No matter how many queues the flow passes through, the threshold remains the same. There could be small queues at each router, but congestion could be signaled because the sum of the queuing delays at each bottleneck would be large.

SyncPMinOTT also shares a potential disadvantage with SyncPQlen. Both algorithms react to transient spikes in computed queuing delay due to using instantaneous OTTs. These spikes are a problem because they are not indicators of long-term congestion but would cause congestion to be detected. In Figure 3.7, the circled areas are the same as in Figure 3.6. Even though the spikes in computed queuing delay are present, they are less critical because they occur well above the 50% minimum-observed OTT threshold.

Another potential disadvantage to SyncPMinOTT is how flows with asymmetric paths would be treated. For example, a flow with a large base RTT but small forward path OTT would be treated like a flow with a small base RTT, *i.e.*, one with a RTT of twice the small forward path OTT. The expectation is that flows with short forward path OTTs would have more frequent congestion indications than flows with larger forward path OTTs. This is because the flows with shorter RTTs can recover from rate reductions faster than flows with longer RTTs. A flow with a short forward path OTT but a large base RTT would then see these frequent congestion indications, but it would also take longer than a flow with a short RTT to recover. So, the flow would be unfairly penalized because of the asymmetry in its path and the large reverse path OTT.

3.5.3 SyncTrend

SyncTrend uses trend analysis to determine when computed queuing delays are increasing or decreasing. The trend analysis algorithm is taken from Jain *et al.*, which uses the trend of one-way delays to measure available bandwidth [JD02]. SyncTrend uses this trend analysis algorithm on the queuing delay samples that are computed by the sender.

Jain's trend analysis requires that samples be gathered in squares (*e.g.*, 100 samples gathered and divided into 10 groups of 10 samples each). I chose to use nine queuing delay samples for trend analysis in SyncTrend. This decision marks a tradeoff between detecting longer term trends and quickly making decisions. SyncTrend first gathers nine queuing delay samples and splits them into three groups of three samples, in the order of their arrival. The median of each of the three groups is computed, and the three medians are used to compute the trend. There are three tests that are performed on the medians:

- Pairwise Increasing Comparison Test (PCT_I).
- Pairwise Decreasing Comparison Test (PCT_D).
- Pairwise Difference Test (PDT).

Let the median computed queuing delays be $\{m_k, k = 1, 2, 3\}$. Let $I(X)$ be 1 if X is true and 0 if X is false. The PCT and PDT equations are as follows:

$$PCT_I = \frac{I(m_2 > m_1) + I(m_3 > m_2)}{2}$$

$$PCT_D = \frac{I(m_2 < m_1) + I(m_3 < m_2)}{2}$$

$$PDT = \frac{m_3 - m_1}{|m_2 - m_1| + |m_3 - m_2|}$$

PCT_I determines what percentage of the pairs of medians are increasing. With only three medians (and therefore, only two pairs of medians), there are only three possible values: 0, 0.5, and 1. Likewise, PCT_D determines what percentage of the pairs of medians are decreasing and can have the values 0, 0.5, or 1. PDT indicates the degree of the trend. If both pairs of medians are increasing, $PDT = 1$. If both pairs are decreasing, $PDT = -1$. Jain *et al.* performed an empirical study to determine that a strong increasing trend is indicated if $PCT_I > 0.55$ or $PDT > 0.4$ (in Jain's study PCT and PDT took on a wider range of values). I used these tests to also detect decreasing trends, and I make the claim that there is a strong decreasing trend if $PCT_D > 0.55$ or $PDT < -0.4$. If the medians have neither a strong increasing nor a strong decreasing trend, the trend is reported as undetermined.

SyncTrend computes a new trend every three queuing delay samples by replacing the three oldest samples with the three newest samples. SyncTrend returns the direction of the trend to the congestion reaction mechanism. The reaction mechanism could then choose to increase the flow's sending rate more aggressively than TCP Reno when the trend was decreasing.

Discussion

Figure 3.8 shows the SyncTrend congestion detection for one FTP flow with 53 ms base RTT. This figure is much like Figures 3.6 and 3.7, with the addition of the trend indications. (Note that since there is no congestion reaction, both the queuing delay at the router and the computed queuing delays here are exactly the same as that in Figures 3.6 and 3.7. In particular, some number of points are now marked to show congestion indications.) Whenever an increasing trend is detected, the computed queuing delay point is a circle. If a decreasing trend is detected, the computed queuing delay point is a star. If no trend could be determined or if there are not enough samples gathered to calculate the trend, the point remains a square. The boxed region on the graph is enlarged and pictured in Figure 3.9.

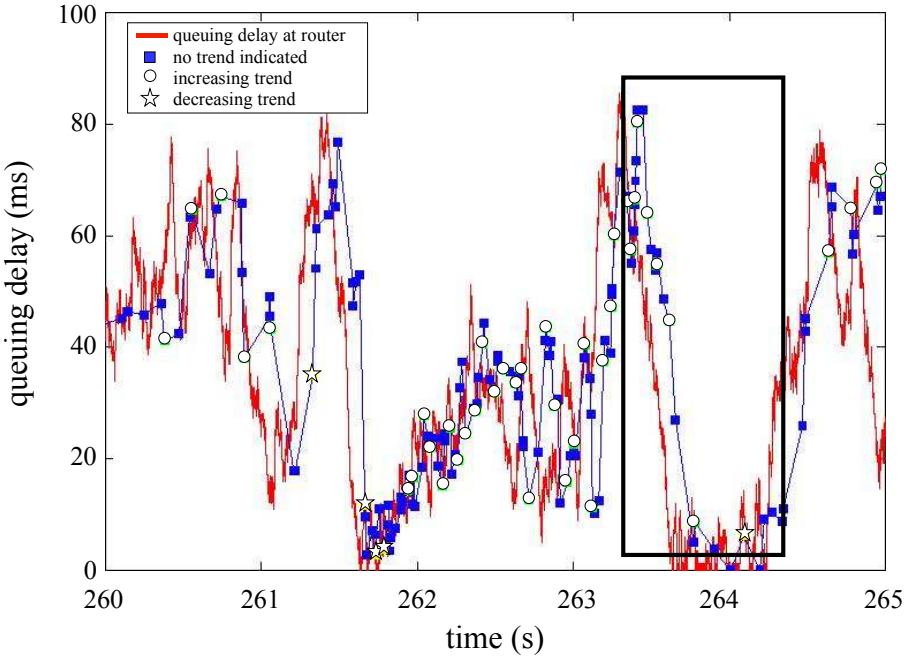


Figure 3.8: SyncTrend Congestion Detection

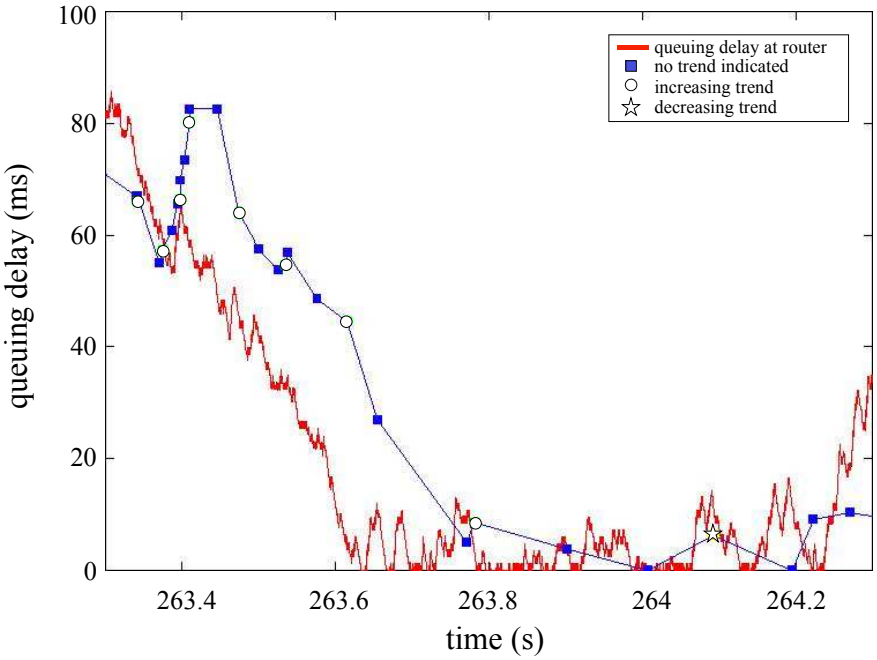


Figure 3.9: SyncTrend Congestion Detection - boxed region

Jain *et al.* used ten groups of ten one-way delay samples in their analysis. SyncTrend collects nine queuing delay samples (three groups of three) before evaluating the trend. This allows the algorithm to make decisions relatively quickly: at startup, after nine ACKs, and from there on, every three ACKs. This also results in the detection of small-scale trends, rather than longer-scale events. The algorithm might indicate small-scale trends where trend measurements fluctuate back and forth between increasing and decreasing. In Figure 3.9, the computed queuing delays are clearly decreasing, but because of the noise in the computed queuing delay signal, the trend analysis algorithm incorrectly detects that the trend is increasing. This could cause a congestion indication when congestion is actually subsiding. A tradeoff exists between collecting a larger number of samples to detect longer-scale trends and collecting few samples to be able to quickly make trend decisions. I chose to design the algorithm for quick detection.

3.5.4 SyncAvg

SyncAvg calculates the weighted average of the computed queuing delays, which it uses to detect congestion. Each time the average computed queuing delay exceeds a certain threshold, SyncAvg indicates that congestion is occurring. The averaging algorithm used by SyncAvg is based on the averaging algorithm used in RED. The weighted average queue length in RED is based on a single queue aggregating traffic from many flows. In contrast, SyncAvg smoothes the queuing delay estimates from only one flow. Therefore, the weighting factor SyncAvg uses should be much larger than RED's default of 1/512. SyncAvg uses the same smoothing factor as TCP's RTO estimator, which is 1/8. Both of these metrics (average queuing delay and RTT) are updated every time an ACK returns. The threshold set in SyncAvg is also based on RED. Christiansen *et al.* showed that using a minimum threshold of 30 ms was a good setting for RED [CJOS01]. In RED, the goal of the threshold is to keep the average queue size no larger than twice the threshold (*i.e.*, a target queuing delay of 60 ms). In SyncAvg, I set the default threshold to 30 ms so that, like RED, congestion will be indicated when the average queue size is greater than 30 ms.

Discussion

Figure 3.10 shows the SyncAvg congestion detection for one FTP flow with a 53 ms base RTT. In this figure, the black squares are the average queuing delays as computed at the sender. Previously all of the black squares of computed queuing delays were the same. Now the values are different (since they represent the average), but they are still computed at the same times as before. The average computed queuing delay tracks pretty well with the queuing delays reported at the router (the gray line) but with a noticeable lag.

In the boxed region (pictured in Figure 3.11), the lag between the actual queuing delay at the router and the average queuing delay computed at the sender is particularly striking.

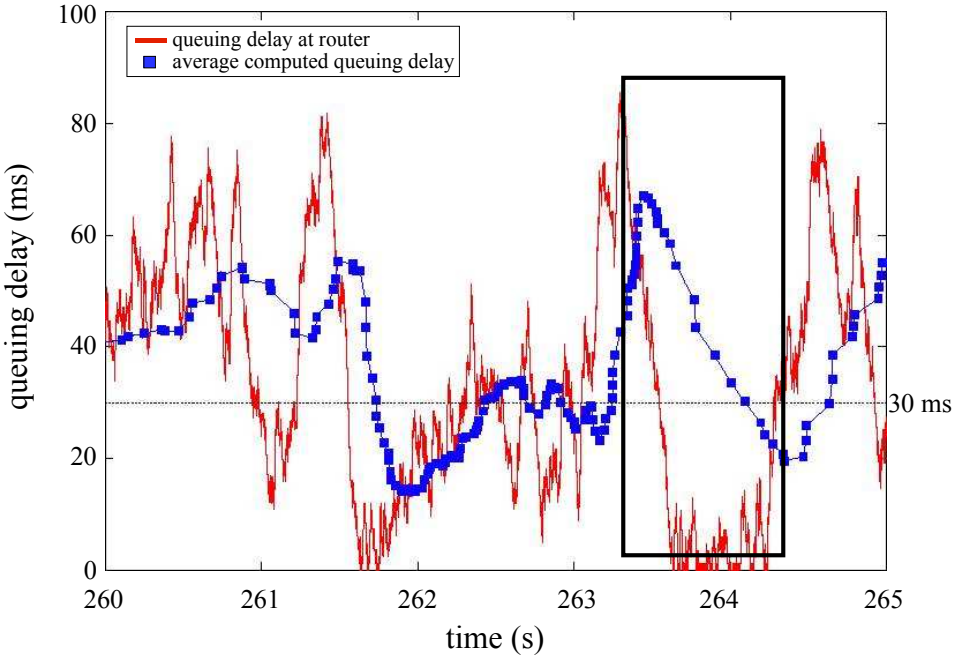


Figure 3.10: SyncAvg Congestion Detection

By the time the average computed queuing delay decreases below the threshold, it has been steadily decreasing for a significant amount of time and the actual queuing delay at the router is very low. Each black square represents the return of an ACK. Increases in the gaps between points are directly related to the gaps between the receipt of ACKs. The averaging weight is 1/8, so the last eight queuing delay samples have a large impact on the average. In the boxed region, there were a large number of high queuing delay samples. This can be seen when comparing the boxed region of SyncTrend (which shows computed queuing delays) and SyncAvg (which shows the average queuing delay) in Figure 3.12. When averaged, the large queuing delay samples, pictured on the left side of the SyncTrend graph, keep the average queuing delay high and increase the lag between the delay at the router and the computed delay. In fact, the average queuing delay value at a certain height on the y-axis does not directly correspond to values of queuing delay seen by the router as the computed queuing delay samples do in Figure 3.12a. When looking at the distance between the gray line and the black squares points in Figure 3.12a, the lag is directly related to the current RTT of the flow. When looking at the same distance in Figure 3.12b, the lag is related to the value of the previous computed queuing delays.

Another disadvantage of SyncAvg is that the fixed threshold value cannot adjust with increases in the total amount of queuing in the network (*e.g.*, with additional congested

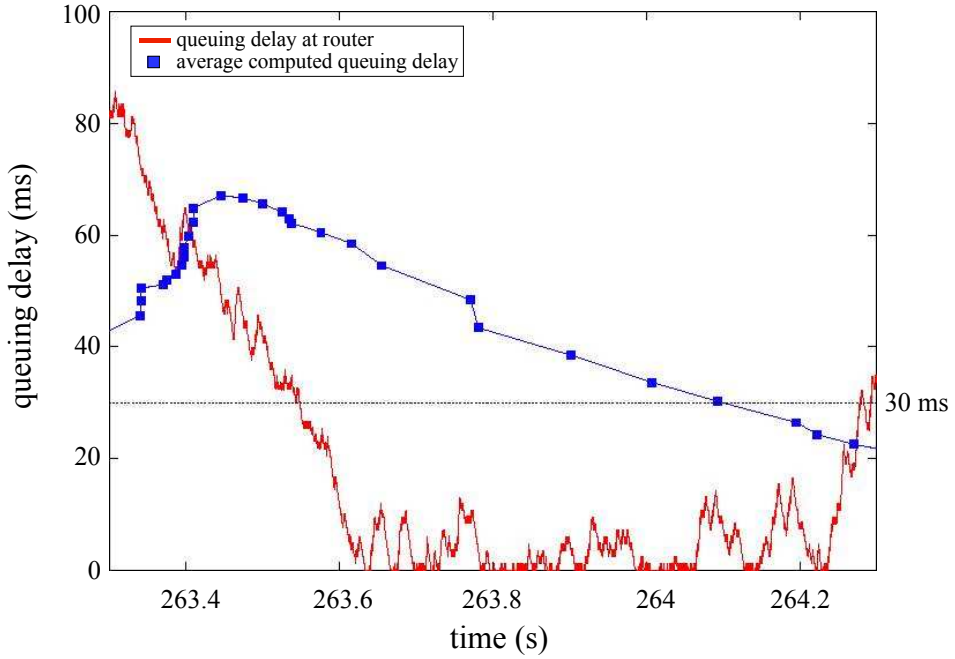


Figure 3.11: SyncAvg Congestion Detection - boxed region

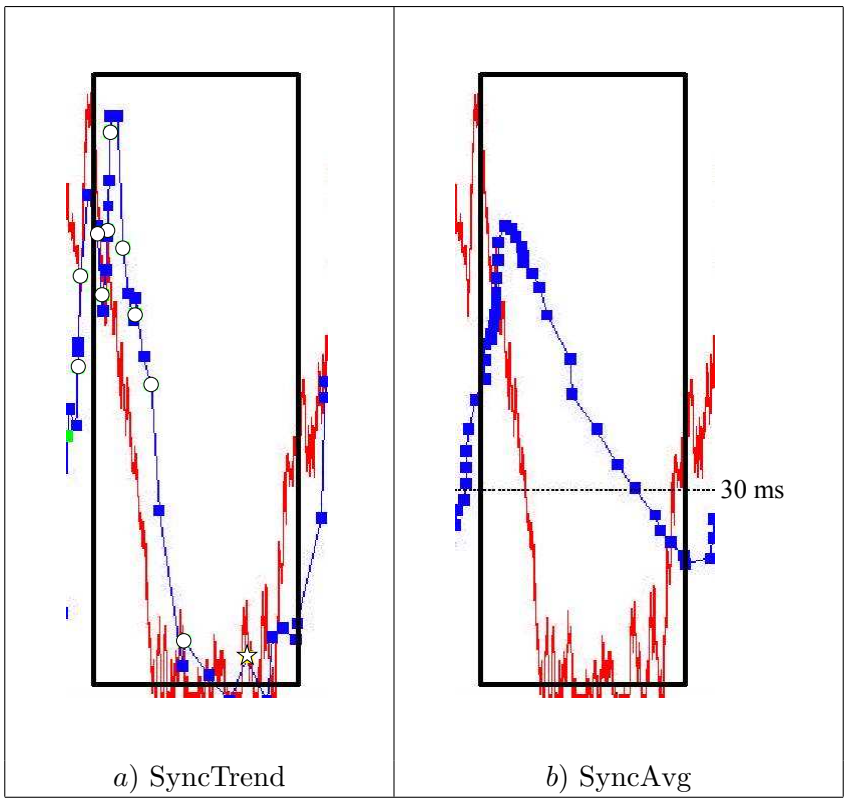


Figure 3.12: Boxed Regions from SyncTrend and SyncAvg

links). Since this method deals with end-to-end queuing delays, there is no one value that will be suitable for a threshold. This limitation is similar to that of SyncPMinOTT.

3.5.5 SyncMix

There were several problems discovered with the previous approaches to detecting congestion:

- SyncPQlen – The instantaneous computed queuing delay is too noisy.
- SyncPMinOTT – The instantaneous computed queuing delay is too noisy, and the threshold is not based on total amount of computed queuing delay in the network.
- SyncTrend – The instantaneous computed queuing delay is too noisy.
- SyncAvg – The average computed queuing delay introduces additional lag, and the threshold is not based on total amount of queuing delay in the network.

SyncMix uses a mixture of the previously described algorithms for congestion detection. SyncMix calculates the weighted average of the computed queuing delay and performs trend analysis on this average. SyncMix also compares the weighted average to a threshold based on an estimate of the maximum amount of queuing delay in the network to report the trend and where the average computed queuing delay lies.

Threshold

SyncMix uses a variation of the SyncPQlen threshold. Instead of the one 50% threshold, SyncMix divides the average computed queuing delay into four regions: 0-25%, 25-50%, 50-75%, and 75-100% of the maximum-observed queuing delay. Since SyncMix does not have a good estimate of the maximum queuing delay until a segment loss occurs, a different metric is used for congestion detection before the first segment loss. If there has been no segment loss and the maximum-observed queuing delay is less than 10 ms, SyncMix uses the absolute value of the percentage difference between the last two estimated queuing delays to determine the region. The 10 ms threshold is used to provide a transition to the original method of computing the region when there is no segment loss for an extended amount of time. If the maximum-observed queuing delay is at least 10 ms, then the four regions are at least 2.5 ms long. For example, if the maximum-observed queuing delay was 10 ms, then the 0-25% region would capture average queuing delays between 0-2.5 ms, the 25-50% region would capture average queuing delays between 2.5-5 ms, and so on. Letting maximum-observed queuing delays under 10 ms be the basis for setting the regions would cause very small regions at the beginning of a flow. Looking at the percentage difference between two consecutive queuing delay samples allows the algorithm to have an idea of the magnitude of change, while letting the trend analysis algorithm decide the direction of the change.

Number of Queuing Delay Samples	Weighting Factor
1	1
2	1
3	3/4
4	1/2
5	1/4
6 +	1/8

Table 3.1: SyncMix Average Queuing Delay Weighting Factors

Trend Analysis

SyncMix uses a simplified version of the trend analysis used in SyncTrend. Instead of computing the trend of the instantaneous queuing delay estimates, SyncMix computes the trend analysis on the weighted average of the queuing delays. SyncMix first gathers nine average queuing delay samples and splits them into three groups of three samples, in the order of their computation. The median, m_i, of each of the three groups is computed. Since there are only three medians, the trend is determined to be increasing if $m_0 < m_2$. Likewise, the trend is determined to be decreasing if $m_0 > m_2$. SyncMix computes a new trend every three ACKs by replacing the three oldest samples with the three newest samples for trend analysis. Due to the nine-sample trend analysis, no early congestion detection will occur if the server receives fewer than nine ACKs (*i.e.*, sends fewer than eighteen segments in the HTTP response, if delayed ACKs are turned on). More precisely, since OTTs are computed for every segment (including ACKs), each data segment from the client (*i.e.*, HTTP request segments) will carry an OTT sample that will be used in the server's trend analysis. So, the trend could be computed if the HTTP response were smaller than eighteen segments, but the early congestion detection would not have much of an effect.

Weighting Factor

SyncAvg used a weighting factor of 1/8 for the average computed queuing delay. This weighted averaging was inspired by RED. When RED averages its queue size for smoothing, it can assume that the process begins with an empty and idle queue. Therefore, the queue average is initialized to 0. Subsequent samples are weighted with a constant weight. Since SyncMix is not looking directly at a queue and a flow may begin when the network is in various states, it cannot assume that zero should be the basis for the average. SyncMix bootstraps the average with queuing delay samples rather than averaging them. Table 3.1 shows the weighting factors for the first few queuing delay samples. Each time a new queuing delay value is computed, the weighted average is calculated: $avg = (1 - w_q)avg + w_q(qdelay)$, where avg is the average and $qdelay$ is the newly-computed queuing delay. The weighting factors

Average Queuing Delay Trend	Average Queuing Delay as a % of the Maximum-Observed Queuing Delay
increasing	75-100%
increasing	50-75%
increasing	25-50%
increasing	0-25%
decreasing	75-100%
decreasing	50-75%
decreasing	25-50%
decreasing	0-25%

Table 3.2: SyncMix Return Values

given in Table 3.1 are used both at startup and whenever a timeout has occurred, which are times when the previous computed queuing delay average is considered to be out-of-date.

Congestion Indications

Each time an ACK is received, SyncMix either reports one of eight indications or that not enough samples have been gathered to compute the trend. The congestion indications are a combination of the trend direction and the region of the average computed queuing delay (Table 3.2).

Discussion

Figure 3.13 shows the congestion indications for SyncMix. The black squares are the average computed queuing delays. An increasing trend is indicated by a white circle, and a decreasing trend is indicated by a white star. The four average computed queuing delay regions are outlined with dotted lines at 25%, 50%, 75%, and 100% of the maximum-observed queuing delay.

Notice that the trend analysis works rather well for detecting trends in the average computed queuing delay. Particularly, in the boxed region of Figure 3.13 (enlarged in Figure 3.14), the detection of a decreasing trend indicates that congestion is abating even though there is still a large lag between the average queuing delay and the actual queue size.

3.5.6 Summary

Figure 3.15 shows the boxed regions from SyncTrend, SyncAvg, and SyncMix for comparison. SyncMix detects that congestion is abating (the trend is decreasing) earlier than either SyncTrend or SyncAvg. Due to noise, SyncTrend does not detect the trend decreasing until the computed queuing delay has been low for some time. SyncAvg detects that congestion is subsiding about the same time as SyncTrend when the average queuing delay passes below the

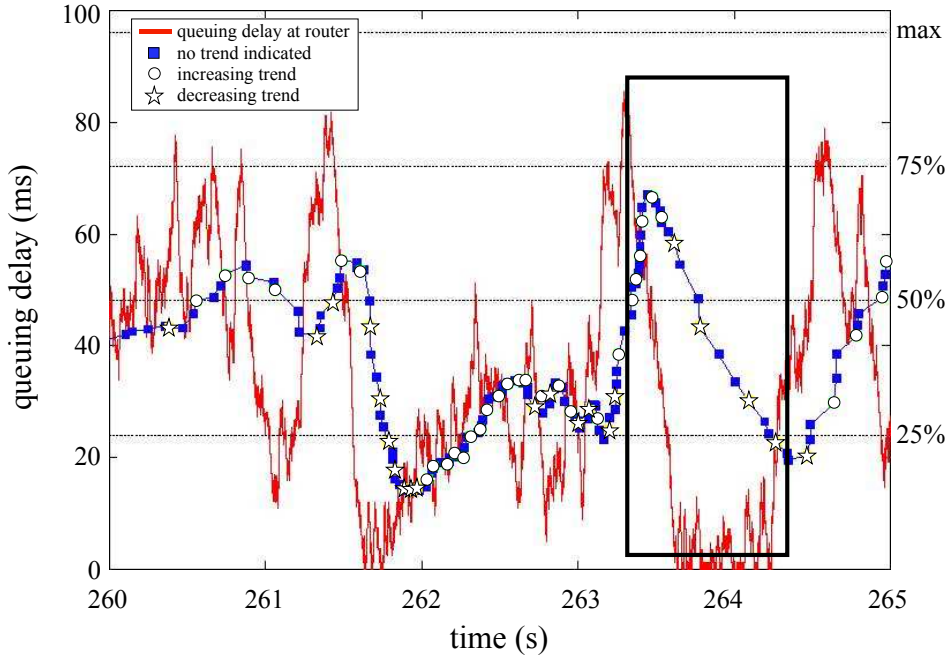


Figure 3.13: SyncMix Congestion Detection

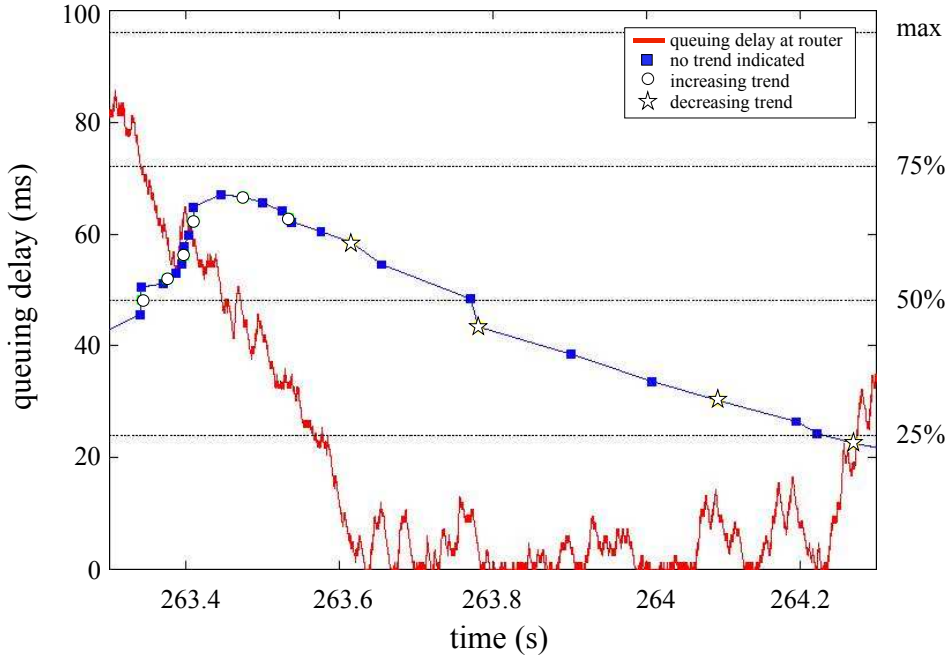


Figure 3.14: SyncMix Congestion Detection - boxed region

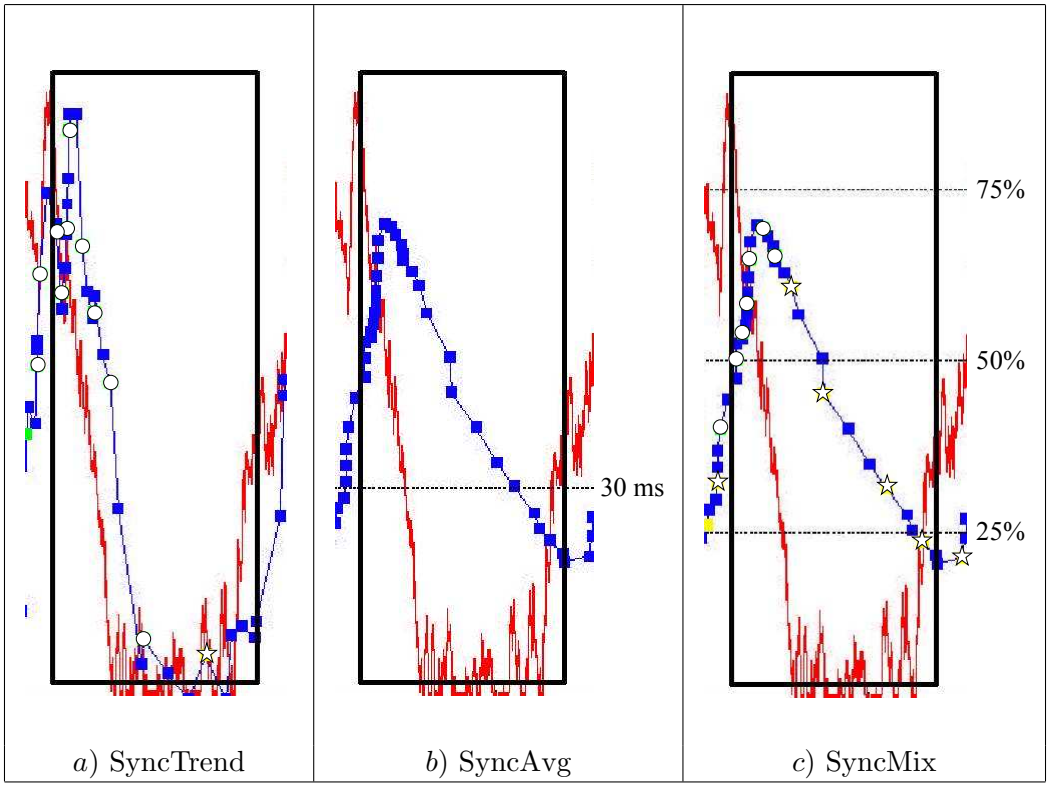


Figure 3.15: Boxed Regions from SyncTrend, SyncAvg, and SyncMix

30 ms threshold. For this reason, one of the congestion reaction mechanisms was specifically designed to be used with SyncMix and no congestion reaction mechanism will be evaluated with SyncTrend or SyncAvg as the congestion detection mechanism.

3.6 Congestion Reaction

Congestion detection is an important part of any congestion control algorithm. It is essential to be able to know when congestion is occurring so that the sender can react appropriately. The sender's reaction to congestion notification is also important. The reaction should be appropriate and should have the effect of reducing congestion and maintaining a low computed queuing delay. A Sync-TCP protocol is made up of a congestion detection mechanism and a congestion reaction mechanism. Since the congestion reaction mechanism cannot be evaluated apart from the congestion detection mechanism, I will present the evaluation of these methods in Chapter 4. I will consider two mechanisms for reaction to congestion indications, SyncPcwnd and SyncMixReact. SyncPcwnd is tuned for a binary congestion indicator and SyncMixReact for a more continuous signal. These algorithms will also be discussed in more detail in Chapter 4.

3.6.1 SyncPcwnd

SyncPcwnd reduces the congestion window by 50% whenever congestion is detected. Like the congestion window reduction policy in ECN, the reduction of *cwnd* is limited to at most once per RTT. The amount of the reduction is like that when an ECN notification is received or when TCP Reno detects segment loss through the receipt of three duplicate ACKs. When there is no congestion signal, SyncPcwnd increases *cwnd* exactly as TCP Reno does. SyncPcwnd can be used with any detection mechanism that gives a binary signal of congestion, such as SyncPQlen, SyncPMinOTT, SyncTrend, or SyncAvg.

3.6.2 SyncMixReact

SyncMixReact is specifically designed to operate with SyncMix congestion detection. SyncMixReact uses the additive-increase, multiplicative decrease (AIMD) congestion window adjustment algorithm but adjusts AIMD's α and β parameters according to the congestion state reported by SyncMix. Additive increase is defined as $w(t+1) = \alpha + w(t)$, where $w(t)$ is the size of the current congestion window in segments at time t and the time unit is one RTT. In TCP Reno's congestion avoidance, $\alpha = 1$. Multiplicative decrease is defined as $w(t+1) = \beta w(t)$. In TCP Reno, $\beta = 0.5$, with *cwnd* being set to 1/2 *cwnd* during fast retransmit. Following AIMD, when *cwnd* is to be increased, SyncMixReact incrementally increases *cwnd* for every ACK returned, and when *cwnd* is to be decreased, SyncMixReact makes the decrease immediately. Figure 3.16 shows the changes SyncMixReact makes to α and β based upon the result from the detection mechanism. The adjustments made to *cwnd* if the trend of average queuing delays is **increasing** are described below:

- If the average queuing delay is less than 25% of the maximum-observed queuing delay, *cwnd* is increased by one segment in one RTT (α is set to 1). This is the same increase that TCP Reno uses during congestion avoidance.
- If the average queuing delay is between 25-50% of the maximum-observed queuing delay, *cwnd* is decreased immediately by 10% (β is set to 0.9).
- If the average queuing delay is between 50-75% of the maximum-observed queuing delay, *cwnd* is decreased immediately by 25% (β is set to 0.75).
- If the average queuing delay is above 75% of the maximum-observed queuing delay, *cwnd* is decreased immediately by 50% (β is set to 0.5). This is the same decrease that TCP Reno would make if a segment loss were detected by the receipt of three duplicate ACKs and that an ECN-enabled TCP sender would make if an ACK were returned with the congestion-experienced bit set.

The adjustments made to *cwnd* if the trend of average queuing delays is **decreasing** are described below:

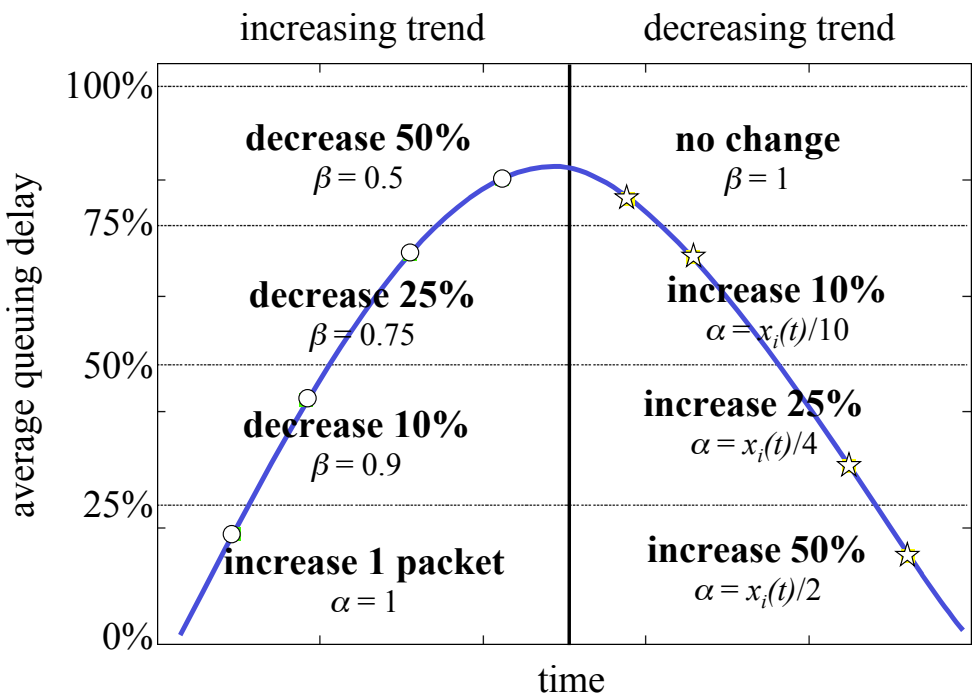


Figure 3.16: SyncMixReact Congestion Reactions

- If the average queuing delay is less than 25% of the maximum-observed queuing delay, *cwnd* is increased by 50% in one RTT (α is set to $x_i(t)/2$, where $x_i(t)$ is the value of *cwnd* at time t). If the congestion detection signal remained in this region, *cwnd* would be doubled in two RTTs. This increase is slower than TCP Reno's increase during slow start, but more aggressive than during congestion avoidance.
- If the average queuing delay is between 25-50% of the maximum-observed queuing delay, *cwnd* is increased by 25% in one RTT (α is set to $x_i(t)/4$).
- If the average queuing delay is between 50-75% of the maximum-observed queuing delay, *cwnd* is increased by 10% in one RTT (α is set to $x_i(t)/10$). This would typically be a very slow increase in the congestion window.
- If the average queuing delay is above 75% of the maximum-observed queuing delay, *cwnd* not adjusted (β is set to 1). The congestion window is not modified in this region, because if the trend begins to increase and the average queuing delay remains above 75% of the maximum-observed queuing delay, then *cwnd* would be reduced by 50%, a large decrease. If the average queuing delay decreased and the trend remained decreasing, then congestion is subsiding.

The additional congestion window decreases in SyncMixReact (as compared to TCP Reno) when the average queuing delay trend is increasing are balanced by the more aggressive congestion window increases when the average queuing delay trend is decreasing.

The congestion detection mechanism in SyncMix will return a new congestion detection result every three ACKs (according to the trend analysis algorithm), causing SyncMixReact to adjust the congestion window.

3.7 Summary

In this chapter, I described the development of Sync-TCP, a family of congestion detection and reaction mechanisms based on the use of OTTs. The OTTs are obtained via the exchange of timestamps between computers with synchronized clocks. I compared five different methods of congestion detection that use forward-path OTTs and computed queuing delays to detect when network queues are increasing and congestion is occurring. I also looked at two methods for reacting to the congestion indications provided by the congestion detection methods.

In Chapter 4, I will evaluate the performance of two Sync-TCP congestion control algorithms as compared with standards-track TCP congestion control algorithms. I chose to look at the combination of the SyncPQlen congestion detection mechanism and the SyncPcwnd congestion reaction mechanism. I will refer to this combination as Sync-TCP(Pcwnd). The second Sync-TCP congestion control mechanism, called Sync-TCP(MixReact), is a combination of the SyncMix congestion detection mechanism and the SyncMixReact congestion reaction mechanism.

Chapter 4

Empirical Evaluation

This chapter describes the evaluation of Sync-TCP. The first part of the chapter describes the experimental methodology used in running the experiments. The goal was to create a non-trivial evaluation of two Sync-TCP congestion control mechanisms, Sync-TCP(Pcwnd) and Sync-TCP(MixReact), as compared with TCP Reno and other standards-track TCP congestion control variants discussed in Chapters 1 and 2. I ran experiments, described fully in Appendix B, to determine the best use of the standards-track TCP versions for HTTP traffic. I found that in order to do a thorough evaluation, both TCP Reno over drop-tail routers and ECN-enabled TCP SACK over Adaptive RED routers were needed. Both of these TCP variants will be compared to the Sync-TCP mechanisms.

4.1 Methodology

This section describes the experimental setup and methodology used to evaluate the performance of Sync-TCP, including the network and traffic types used in the evaluation. A more detailed description of the HTTP traffic model, network calibration experiments, and analysis of issues such as simulation run time are presented in Appendix A.

4.1.1 Network Configuration

I ran simulations using the *ns* network simulator [BEF+00]. The goal was to create a network, congest at least one link with two-way traffic, and take end-to-end and router-based measurements. The two-way traffic loads were designed to provide roughly equal levels of congestion on both the forward path and reverse path in the network.

Figure 4.1 is a simplified depiction of the topology used. Each circle and hexagon in the figure represent five *ns* nodes in the actual experimental topology. In the FTP experiments, each *ns* node represents four end systems, thus, each circle and hexagon represent 20 end systems. In the HTTP experiments, each *ns* node represents a "cloud" (*i.e.*, end systems that share an aggregation point) of HTTP clients or servers. In the HTTP experiments, each

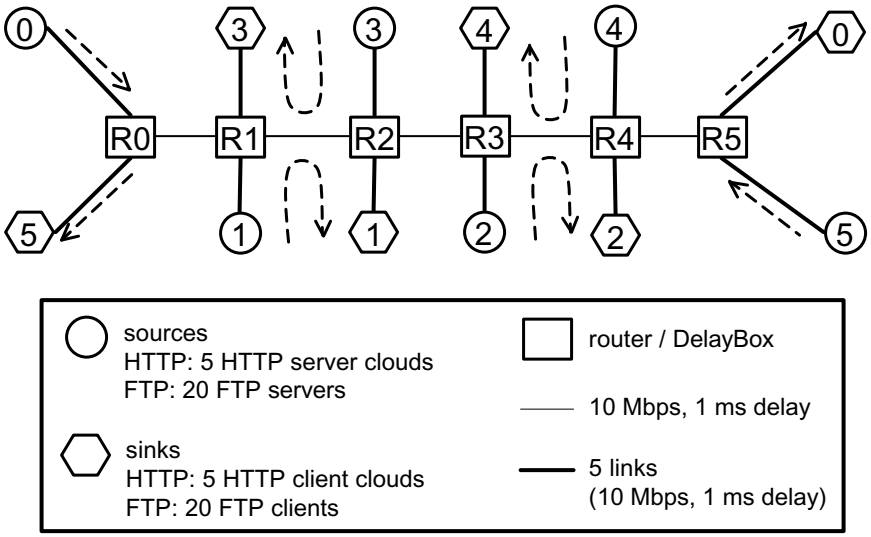


Figure 4.1: Simplified 6Q Parking Lot Topology

circle and hexagon represent hundreds of end systems, and the circles generate thousands of TCP connections. The interior squares are routers. The dashed lines represent the direction of the flow of data traffic in the network. Traffic generated by circles 1-4 does not leave the "middle" of the network. Traffic generated by circles 0 and 5 traverses all routers. The links between the routers carry two-way traffic, and the router queues buffer both data and ACKs. This topology is referred to here as the "6Q parking lot" topology, because of the six routers used and the topology's resemblance to a row in a parking lot. This network setup was first proposed by Floyd to allow testing of multiple congested routers using cross-traffic on the interior links [Flo91b].

In Figure 4.1, I distinguish between traffic on the network as follows:

- Forward-path end-to-end traffic – data from source 0 to sink 0 and ACKs from sink 5 to source 5.
- Reverse-path end-to-end traffic – data from source 5 to sink 5 and ACKs from sink 0 to source 0.
- Forward-path cross-traffic – data from source 1 to sink 1 and from source 2 to sink 2 and ACKs from sink 3 to source 3 and from sink 4 to source 4.
- Reverse-path cross-traffic – data from source 3 to sink 3 and from source 4 to sink 4 and ACKs from sink 1 to source 1 and from sink 2 to source 2.

All links have a 10 Mbps capacity and a 1 ms propagation delay. The 1 ms propagation delay on each link results in a minimum RTT of 14 ms for any end-to-end flow (each end-to-end flow traverses seven links in both directions). Since there are actually five *ns* nodes feeding into each router on each interface (each *ns* node logically has its own interface to the router), there is a maximum incoming rate of 50 Mbps (from each circle shown in Figure 4.1) to each

router. The data presented herein comes from measurements of the end-to-end traffic on the forward path in the network.

Three different high-level scenarios will be used to evaluate Sync-TCP:

- One Bottleneck - Congestion is caused by the aggregation of traffic between routers R0 and R1 on the forward path and between routers R5 and R4 on the reverse path. There is a maximum of 50 Mbps of traffic entering routers R0 and R5. These routers can forward at a maximum of 10 Mbps, so congestion will result.
- Two Bottlenecks - Congestion is caused by two-way cross-traffic between routers R1 and R2 and routers R3 and R4. For example, router R1 will have a maximum of 10 Mbps of traffic coming from router R0 and a maximum of 50 Mbps of additional traffic coming from source 1 (along with the ACKs generated by sink 3) to forward over a 10 Mbps link.
- Three Bottlenecks - Congestion is caused by the aggregation present in the one bottleneck case plus two-way cross-traffic between routers R1 and R2 and routers R3 and R4.

4.1.2 Traffic Generation

I will evaluate the performance of Sync-TCP first with long-lived FTP traffic and then with HTTP 1.0 traffic. The FTP tests were performed to examine how the congestion control algorithm would operate in a relatively simple environment of long-lived TCP flows. A small number of long-lived flows are useful in observing the long-term behavior of a protocol. These flows transfer large amounts of data and so they must react to congestion events in the network. The small number of flows allows flows to be analyzed individually over relatively stable levels of congestion.

Initial evaluations of network protocols often use a simple traffic model: a small number of long-lived flows with equal RTTs, with data flowing in one direction, ACKs flowing in the opposite direction, and every segment immediately acknowledged. These scenarios may be useful in understanding and illustrating the basic operation of a protocol, but they are not sufficiently complex to be used in making evaluations of one protocol versus another. HTTP traffic was used as a more realistic model of Internet traffic and represents a mix of long and short flows.

The goal of this study is to evaluate Sync-TCP in a complex environment that contains some of the variety found in the Internet. Networking researchers cannot directly model the Internet, but should use reasonable approximations when designing evaluations of network protocols [PF97]. As discussed next, there are several ways to mimic a small part of Internet behavior, including using two-way traffic, delayed ACKs, varied RTTs, multiple congested routers, and modern traffic models.

Two-Way Traffic

Most traffic in the Internet is bi-directional. In the case of TCP, data segments in one direction flow alongside ACKs for data flowing in the opposite direction. So, ACKs flow on the same paths and share buffer space in the same queues as data segments. This has several implications [Flo91b, ZSC91]. First, since buffers in routers are typically allocated by packets, rather than bytes, an ACK occupies the same amount of buffer space as a data segment. Second, since ACKs can be queued at routers, several ACKs could arrive at a router separated by time, be buffered behind other packets, and then be sent out in a burst. The ACKs would then arrive at their destination with a smaller inter-arrival time than they were sent. This phenomena is called ACK compression. ACK compression could lead to a sudden increase in a sender's congestion window and a bursty sending rate. Third, there could be congestion on a flow's ACK path and ACKs could be dropped. Depending on the size of the sender's congestion window, these ACK drops could be misinterpreted as data path drops and retransmissions would be generated. Because of the added complexity caused by two-way traffic, I use it in the evaluation of Sync-TCP.

Delayed Acknowledgments

Delayed ACKs are used mainly in order to allow receivers to piggy-back ACKs with data destined for the sender. Because timers are used to ensure that ACKs are sent even if no data segments are transmitted to the sender, the use of delayed ACKs often results in an ACK being sent every other segment. Most TCP stacks include the use of delayed ACKs by default. Therefore, I use delayed ACKs in all of the evaluation experiments. The results of using delayed ACKs and two-way traffic is a reduction in the number of ACKs competing with data segments for buffer space, which reduces loss when compared to not using delayed ACKs. For Sync-TCP, using delayed ACKs reduces the frequency of OTT information being reported back to the sender. This is clearly not the best case for Sync-TCP, but since it is a reality in the Internet, delayed ACKs are used in this evaluation.

In all experiments, TCP protocols are never mixed in an experiment. To compare the performance of Sync-TCP vs. TCP Reno, for example, one experiment was run with all flows (including cross-traffic) using Sync-TCP and a second experiment (with the same traffic characteristics and random number seeds) was run with all flows using TCP Reno.

Traffic Intensity

The FTP traffic consists of 20 infinite bulk transfer flows in each direction. For FTP traffic, cross-traffic consists of 10 additional infinite bulk transfer FTP flows in both directions over the two congested links. HTTP traffic comes from the PackMime traffic model developed at Bell Labs [CCLS01b]. This model is based on the analysis of HTTP connections in a trace of a

End-to-End Load	Cross-Traffic Offered Load 75% Total Load	90% Total Load	105% Total Load	Number of Bottlenecks
50%	25%	45%	55%	2
60%	15%	30%	45%	2
70%	5%	20%	35%	3
80%	-	10%	25%	3
85%	-	5%	20%	3
90%	-	-	15%	3
95%	-	-	10%	3
100%	-	-	5%	3

Table 4.1: HTTP Cross-Traffic for Multiple Bottleneck Experiments

100 Mbps Ethernet link connecting an enterprise network of approximately 3,000 hosts to the Internet [CLS00, CCLS01a]. The fundamental parameter of PackMime is the TCP/HTTP connection initiation rate (a parameter of the distribution of connection inter-arrival times). The model also includes distributions of base RTTs, the size of HTTP requests, and the size of HTTP responses.

For HTTP, the levels of offered load used in these experiments are expressed as a percentage of the capacity of a 10 Mbps link. I initially ran the network at 100 Mbps and determined the PackMime connection rates (essentially the HTTP request rates) that will result in average link utilizations (in both forward and reverse directions) of 5, 6, 7, 8, 8.5, 9, 9.5, 10, and 10.5 Mbps. These rates are then used to represent a percentage of the capacity of a 10 Mbps link. For example, the connection rate that results in an average utilization of 8.5 Mbps, or 8.5% of the (clearly uncongested) 100 Mbps link, will be used to generate an offered load on the 10 Mbps link of 8.5 Mbps, or 85% of the 10 Mbps link. Note that this "85% load" (*i.e.*, the connection rate that results in 8.5 Mbps of traffic on the 100 Mbps link) will not actually result in 8.5 Mbps of traffic on the 10 Mbps link. The bursty HTTP sources will cause congestion on the 10 Mbps link and the actual utilization of the link will be a function of the protocol and router queue management scheme used.

For HTTP traffic, there will be three levels of cross-traffic depending on the total load at the two links carrying cross-traffic as shown in Table 4.1. Calibration experiments were performed to find PackMime connection rates for cross-traffic that, when added to end-to-end traffic, would average 7.5 Mbps, 9 Mbps, and 10.5 Mbps on the uncongested calibration network. These experiments are detailed in Appendix A.

Round-Trip Times

The routers in these simulations are *ns* nodes that I developed, called DelayBoxes, which, in addition to forwarding segments, also delay segments. DelayBox is an NS analog to dum-

Flow ID	RTT (ms)	Flow ID	RTT (ms)
1	55	11	77
2	97	12	31
3	86	13	18
4	46	14	17
5	88	15	118
6	21	16	53
7	98	17	691
8	95	18	411
9	17	19	16
10	31	20	27

Table 4.2: FTP Round-Trip Times

mynet [Riz97], which is used in Linux and FreeBSD to delay segments. With DelayBox, segments from a TCP connection can be delayed before being passed on to the next node. This allows each TCP connection to experience a different minimum delay (and hence a different minimum round-trip time), based on random sampling from a delay distribution. In these experiments, DelayBox uses an empirical RTT distribution from the PackMime HTTP traffic model (see Appendix A).

The range of delays assigned by DelayBox in any given experiment depends on the number of flows used, because delays are sampled per flow. For FTP traffic, I use 20 end-to-end flows in each direction and obtain RTTs ranging from 16 ms to almost 700 ms, with a median of 53 ms. This includes the 14 ms propagation delay from the network topology. These RTTs represent only propagation delay and do not include queuing delays. Table 4.2 shows the RTT of each of the end-to-end forward-path FTP flows. In each separate experiment, each FTP flow is assigned the base RTT. So, for example, flow 16 will always have a 53 ms base RTT no matter the TCP protocol or amount of cross-traffic.

For HTTP traffic, there are many more than 20 flows, resulting in a wider range of RTTs. The shortest RTT is 15 ms, the longest RTT is over 9 seconds, and the median RTT is 78 ms. Unlike FTP traffic, the HTTP traffic is dynamic, with the number of flows in the system at any one time being influenced by both the TCP protocol and the amount of cross-traffic.

HTTP traffic consists of pairs of requests and responses. An HTTP client first sends a request to an HTTP server. Then, the HTTP server processes the request and sends a response back to the HTTP client. In the implementation of PackMime in *ns*, HTTP request sizes are assigned in the same order for each experiment, but HTTP response sizes are assigned in the order that the requests are received by the HTTP servers. Since the binding of response size to request size is affected by drops (*e.g.*, drops of segments containing HTTP requests), there is no way to match a {request size, response size, RTT} triple from one experiment to

another. This makes it difficult to directly compare the performance of individual flows from the HTTP experiments, so I rely on aggregate statistics to evaluate HTTP performance.

Bandwidth-Delay Product

The bandwidth-delay product (BDP) represents the amount of data that the network can contain without queuing. The BDP is used to set the TCP maximum send window size for each flow and the maximum capacity in the queue at each router. It is calculated by multiplying the slowest link speed on the path by the RTT between the sender and receiver. On a network that carries only one flow, a TCP sender should be able to maintain a congestion window equal to the BDP without causing a queue to build-up at the router. Additionally, a rule of thumb is that a router's maximum queue capacity should be set to between 2-4 times the BDP. In my simulations, routers have 2xBDP queue buffers.

Computing the BDP for a group of flows with different RTTs is difficult. Because of the wide range of RTTs in the PackMime distribution, I chose to use the median RTT for the traffic type (53 ms for FTP and 78 ms for HTTP). The BDP is represented in bytes, but to set the window size and queue size in *ns*, segments are used. I divided the BDP by the maximum data size of a segment, 1420 bytes, to arrive at the BDP in units of segments. I chose to set the maximum send window for each flow to the BDP and the maximum queuing capacity at each router to twice the BDP. For FTP flows, the BDP is 46 1420-byte segments, which means that the queue capacity is 92 segments. For HTTP flows, the BDP is 68 1420-byte segments, resulting in a maximum queue length of 136 segments.

4.1.3 Simulation Running Time

I ran each FTP experiment for 300 simulated seconds. All 20 of the flows were started at time 0. The length of the simulation ensures that the flows reach steady-state before the end of the experiment, as demonstrated in Figure 4.1.3. This figure shows the running goodput of the 20 FTP flows (data bytes received by the FTP clients) as time increases. By time 150 seconds, goodput has stabilized to a little over 9 Mbps.

Given the bursty nature of the HTTP traffic sources, I used a 120-second warm-up interval before collecting any data. After the warm-up interval, the simulation runs until 250,000 HTTP request-response pairs have been completed[1]. I also require that at least one 10 MB response has begun its transmission by time 1120 (1000 seconds after the warm-up period ends). This ensures that there will be some very long-lived flows in the network along with short-lived flows that are typical of web traffic. For the multiple bottleneck experiments, generating cross-traffic caused additional simulation overhead in terms of memory and running time, which required some of the simulations to be shortened. For the 75-90% total load

[1]Depending on the load level, this takes 30-60 simulated minutes.

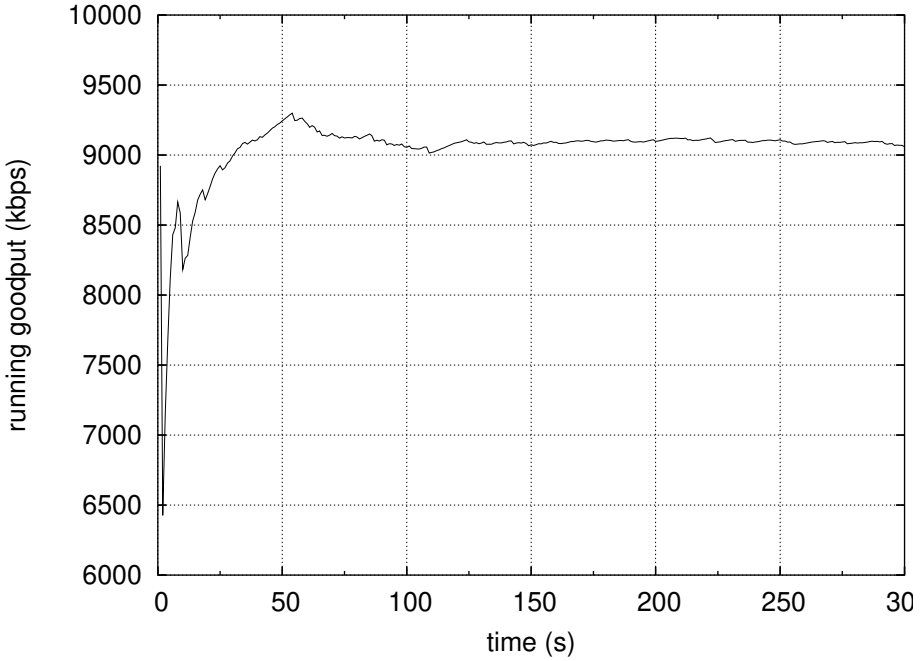


Figure 4.2: FTP Goodput as Running Time Increases

experiments, the simulations were terminated after 150,000 request-response pairs were completed. For the 105% total load experiments, the simulations were terminated after 100,000 request-response pairs were completed. More details on deciding the appropriate running time are in Appendix A.

4.1.4 Evaluation Metrics

The metrics used to evaluate Sync-TCP with FTP and HTTP traffic are different. In both scenarios, I will look at aggregate network-level metrics, such as average packet loss and average queue size at the bottleneck router. I will also look at application-specific metrics for both FTP and HTTP.

FTP Metrics

In evaluating Sync-TCP for FTP traffic, I will look at per-flow metrics, such as goodput (the rate of data received at the destination) and data packet drop percentage. I will also look at aggregate statistics, such as average network utilization, average packet drop percentage, and average queue size at each congested router.

Network utilization is calculated as the percentage of the link bandwidth delivered to the receivers. For example, 90% utilization means that on a 10 Mbps link, an average of 9 Mbps (including both data and ACKs) left the last router destined for the receivers. The

overall packet loss percentage is calculated as the percentage of total packets (including ACKs) dropped at a given queue at each congested router. The average queue size at each congested router is monitored at each packet arrival and departure. The average queue size is calculated and recorded for every observation over the entire experiment.

HTTP Metrics

In each experiment, I measured network-level metrics, including the average packet loss percentage at each congested router, the average queue size at each congested router, the average throughput, the average goodput for the HTTP clients, and the average link utilization at the link closest to the forward-path HTTP clients. I also measured application-level metrics, such as the HTTP response times and the average goodput per response. All of these metrics will be reported in summary form, with the results for every load level on a single graph. The application-level metrics are measured only for those flows that have completed responses. The network-level metrics are measured for all traffic, including flows that have not completed and which still have data in the network when the experiment ends.

The average packet loss percentage and the average queue size are calculated in the same way as in the FTP experiments. The average throughput is the rate of bytes (including ACKs) sent end-to-end on the forward path. The average goodput is calculated as the rate of data bytes (excluding ACKs) received by the forward-path HTTP clients. The average link utilization is the percentage of bandwidth used by the end-to-end flows (including ACKs) on the link closest to the forward-path HTTP clients.

The HTTP response time is the amount of time between a client sending a HTTP request and receiving the complete HTTP response from the server. The goodput per HTTP response is the goodput (as defined above) for each response (and does not include the HTTP request).

HTTP response time cumulative distribution functions (CDFs) are the main metric I use for evaluating the performance of HTTP traffic. The methodology for plotting the response time CDFs and using them to evaluate HTTP performance comes from Christiansen *et al.* [CJOS01]. Figure 4.3 is an example of a response time CDF. On these graphs, the x-axis is the HTTP response time in milliseconds. The probability that a response will finish in a certain amount of time or less is on the y-axis. In Figure 4.3, the arrows indicate a point on the curve where about 75% of the HTTP request-response pairs completed in 400 ms or less. The goal is for a large percentage of request-response pairs to complete in a short amount of time, so the closer the CDF curve is to the top-left corner of the graph, the better the performance.

For response time CDFs, I will show response times up to 1500 ms. To evaluate performance for request-response pairs that take longer than 1500 ms to complete, I will show the complementary cumulative probability function (CCDF) of HTTP response times. Figure 4.4 is an example of a response time complementary cumulative distribution function (CCDF).

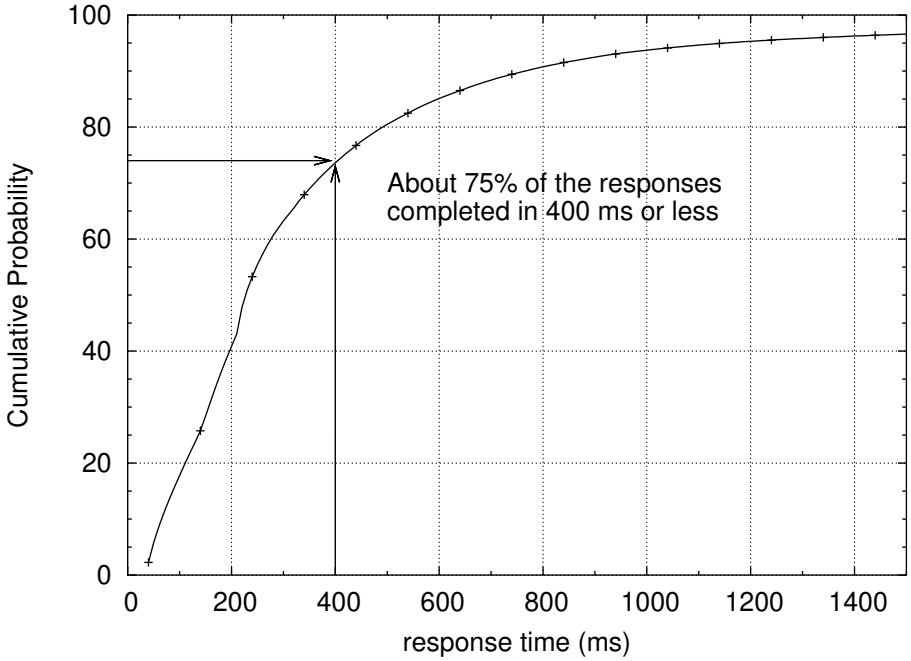


Figure 4.3: Response Time CDF Example

The response time CCDFs are plotted on a log-log scale, with the x-axis in seconds. The probability that a response will finish in a certain amount of time or longer is on the y-axis. The goal is to have the largest percentage of responses finish in the shortest amount of time, so better performance is indicated by the curve closest to the bottom-left corner of the graph. Since the x-axis is in seconds and is on a log scale, small differences towards the right side of the graph are actually large as compared with the CDFs.

Looking at only the summary statistics, response time CDFs, and response time CCDFs does not give a complete picture of the performance of HTTP traffic. I will present additional graphs that will be useful in answering why (and for what type of HTTP request-response pairs) one protocol performs better than other. These graphs include the CDFs of queue size at the bottleneck router, CCDFs of the size of completed responses, response time CDFs and CCDFs conditional on the size of the completed response, CCDFs of the RTTs of flows with completed responses, and RTT CCDFs conditional on the size of the completed response.

The CDFs of queue size (measured at every packet arrival and departure) are presented to show the percentage of queue events where the queue was of a certain size. This gives more information about the size of the queue over the course of the experiment than just the average queue size. It also gives an indication of how often the queue was empty. An empty queue means that no data is being transferred and the instantaneous link utilization is 0.

The HTTP experiments were run until a certain number of request-response pairs completed. Due to the traffic generator and congestion effects, there is no guarantee that each

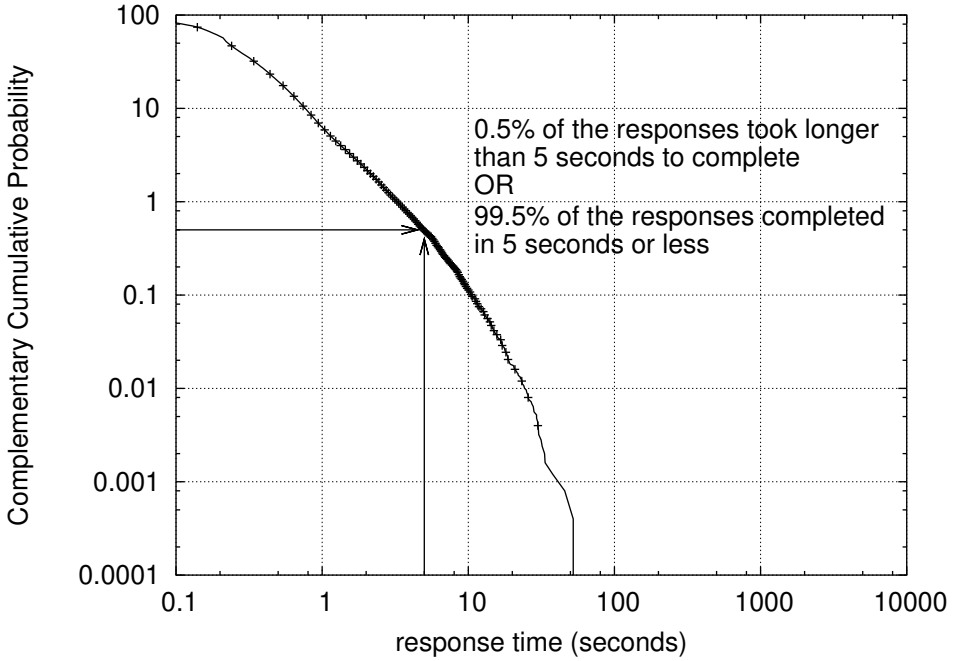


Figure 4.4: Response Time CCDF Example

experiment will see the same request-response pairs complete before the termination condition. In order to effectively evaluate the HTTP response time CCDFs, which show the longest response times, the distribution of the sizes of the completed responses and the distribution of the RTTs of the flows that had responses complete should be considered. For example, suppose that protocol A had a better HTTP response time CCDF than protocol B. But, if protocol B saw a flow with a larger maximum response size and/or a larger maximum base RTT than protocol A, it is hard to determine which protocol is better based on a comparison of only the HTTP response time CCDFs, since a flow's HTTP response time is largely affected by both the HTTP response size and the flow's base RTT. For this reason, I show the CCDFs of the size of the completed responses. Then, I show the CDFs and CCDFs of the HTTP response times conditional on the size of the completed responses. The response sizes were divided into two bins:

1. HTTP responses at most 25560 bytes (25 KB)
2. HTTP responses larger than 25560 bytes (25 KB)

These graphs should show any bias in the protocols towards short or long responses. For example, protocol A may result in better response times for short responses, but give worse response times for long responses, as compared with protocol B. This would be useful information for a thorough evaluation. I also look at the CCDFs of the RTTs for completed responses and the CCDFs of the RTTs conditional on the size of the response. Given this information, the impact of RTTs on which responses completed can be assessed. Since both

Protocol	Network Utilization %	Avg Packet Drop %	Avg Queue Size (Packets)
TCP Reno	90.59	1.56	41.51
Sync-TCP(Pcwnd)	91.97	0.06	26.49
Sync-TCP(MixReact)	85.79	1.03	30.97

Table 4.3: Aggregate FTP Statistics, Single Bottleneck

the RTT distribution and the response size distribution are heavy-tailed (see Appendix A), there will be many pairs where both the RTT and response size are small, a few pairs where the RTT is very large, a few pairs where the response size is very large, and a very small number of pairs where both the RTT and response size are very large. Because of the limited simulation run time, if there were any pairs that were assigned long RTTs and long response sizes, they probably would not have been able to complete before the required number of responses completed. Coupled with the conditional response time CDFs and CCDFs, these RTT CCDFs would bring out any bias that a protocol had towards a certain type of response.

4.2 FTP Evaluation

In this section, I will evaluate the performance of FTP traffic over both single and multiple bottlenecks using the following two Sync-TCP algorithms as compared to using TCP Reno:

- Sync-TCP(Pcwnd) - SyncPQlen congestion detection (section 3.5.1) and SyncPcwnd congestion reaction (section 3.6.1). SyncPQlen detects congestion when the computed queuing delay is over 50% of the maximum-observed queuing delay. SyncPcwnd reduces *cwnd* by 50% when congestion is detected.
- Sync-TCP(MixReact) - SyncMix congestion detection (section 3.5.5) and SyncMixReact congestion reaction (section 3.6.2). SyncMix computes the trend of the average queuing delay and returns the direction of the trend and how the average queuing delay relates to the maximum-observed queuing delay. SyncMixReact increases or decreases *cwnd* based on the congestion indication returned by SyncMix.

4.2.1 Single Bottleneck

Table 4.3 shows the aggregate statistics from the single bottleneck FTP experiments. Using Sync-TCP(Pcwnd) results in higher network utilization, lower average packet loss, and a lower average queue size than both TCP Reno and Sync-TCP(MixReact). TCP Reno has higher network utilization than Sync-TCP(MixReact), but also has higher packet loss and average queue size at the bottleneck link. Sync-TCP(MixReact) could have smaller packet loss because it is not fully using its share of the network or because it manages network resources more efficiently. To investigate this, I will present results of packet loss and goodput

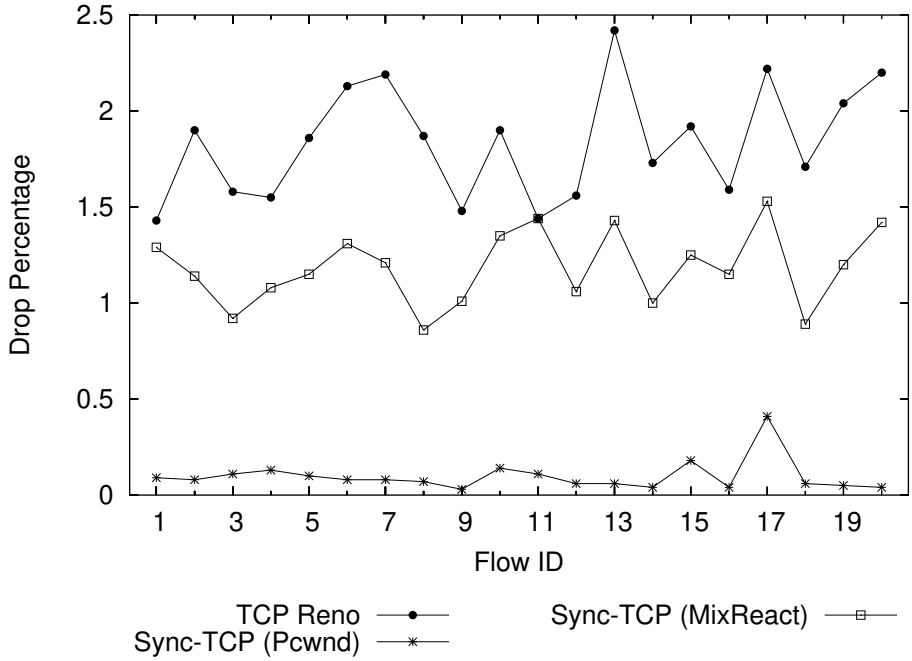


Figure 4.5: Packet Loss, Single Bottleneck

of each of the 20 FTP flows for the three protocols. I will also look into the operation of the Sync-TCP protocols as compared to TCP Reno by examining the queue size of the bottleneck over time and the adjustments made to *cwnd* over time for individual flows.

Packet Loss

Figure 4.5 shows average packet loss of each of the 20 FTP flows for TCP Reno, Sync-TCP(Pcwnd), and Sync-TCP(MixReact). (The lines between the points are included as a visualization aid only. There is no ordering implied by the x-axis.) For each of the 20 FTP flows, Sync-TCP(Pcwnd) results in less packet loss at the bottleneck router than either TCP Reno or Sync-TCP(MixReact). Both Sync-TCP protocols resulted in less packet loss than TCP Reno for every flow.

The idea behind reacting to congestion before loss occurs is that if flows slow down their sending rates, they will avoid overflowing queues and packet loss rates overall will be lowered. The main factor in lowering packet loss rates is reducing sending rates during times of congestion. Sending rates are lowered by reducing the size of senders' congestion windows. Sync-TCP(Pcwnd) avoids packet loss by reducing *cwnd* by 50% whenever the computed queuing delay is 50% of the maximum-observed queuing delay. Sync-TCP(MixReact) avoids packet loss by reducing *cwnd* by various amounts depending on the average queuing delay.

Both Sync-TCP algorithms use TCP Reno as their base protocols. Every flow begins in TCP Reno slow start. Sync-TCP(Pcwnd) only begins to operate once a flow has seen

packet loss (to make sure that it has a good estimate of the maximum amount of queuing in the network). Before this occurs, the flow is in TCP's standard slow start. In these FTP experiments, every flow sees packet loss during the experiment. Every flow but one (flow 18 with a 400 ms RTT) sees its first packet loss within 1 second of starting transfer. This means that the Sync-TCP(Pcwnd) congestion control algorithm is operating on all flows soon after the experiment begins. Sync-TCP(MixReact) begins operating once it receives nine ACKs. These FTP flows send more than 18 segments (since delayed ACKs are used), so Sync-TCP(MixReact) will operate on all flows in the experiment.

A Sync-TCP(Pcwnd) flow with the same number of *cwnd* reductions as a TCP Reno flow will by definition see less packet loss. TCP Reno only reduces *cwnd* after detecting packet loss, while Sync-TCP(Pcwnd) will reduce *cwnd* both after detecting packet loss and also when congestion is indicated. It is reasonable to assume that Sync-TCP(Pcwnd) flows will create less packet loss than TCP Reno flows because they are sensitive to increases in queuing delays and will reduce both *cwnd* before packet loss occurs and in response to packet loss.

The relationship between the congestion window reductions of Sync-TCP(Pcwnd) and Sync-TCP(MixReact) is more complex. Sync-TCP(Pcwnd) has larger *cwnd* decreases than Sync-TCP(MixReact) and smaller *cwnd* increases. Sync-TCP(Pcwnd) will decrease *cwnd* by 50% whenever congestion is detected, while Sync-TCP(MixReact) decreases *cwnd* by 10%, 25%, or 50% depending on the size of the average queuing delay. Additionally, in these experiments, Sync-TCP(Pcwnd) has fewer congestion window reductions than Sync-TCP(MixReact) for each flow. Sync-TCP(Pcwnd) reduces the congestion window at most once per RTT, while Sync-TCP(MixReact) has the potential to reduce the congestion window every 3 ACKs.

Queue Size

The goal of Sync-TCP(Pcwnd) is to avoid packet loss by keeping the network queuing delay smaller than 50% of its maximum. The goal of Sync-TCP(MixReact) is to quickly adjust to increasing queuing delays while make sure that the flow is using all of the available bandwidth without overflowing the queue. To look at how well the algorithms are achieving this goal, I will compare the queue size of the bottleneck router for TCP Reno, Sync-TCP(Pcwnd), and Sync-TCP(MixReact).

Table 4.3 shows that Sync-TCP(Pcwnd) has the smallest average queue size and that, on average, it is much smaller than 50% full (the maximum size is 92 packets). Figure 4.6 show the queue size in packets and queuing delay in milliseconds seen by packets arriving at the bottleneck router during a 10-second time period during the experiment for each protocol. The capacity of the queue at the bottleneck is 92 packets, and the maximum queuing delay seen by any packet at the bottleneck is 98 ms. Queue size and queuing delay are not always directly comparable, because queues buffer a certain number of packets rather than a certain

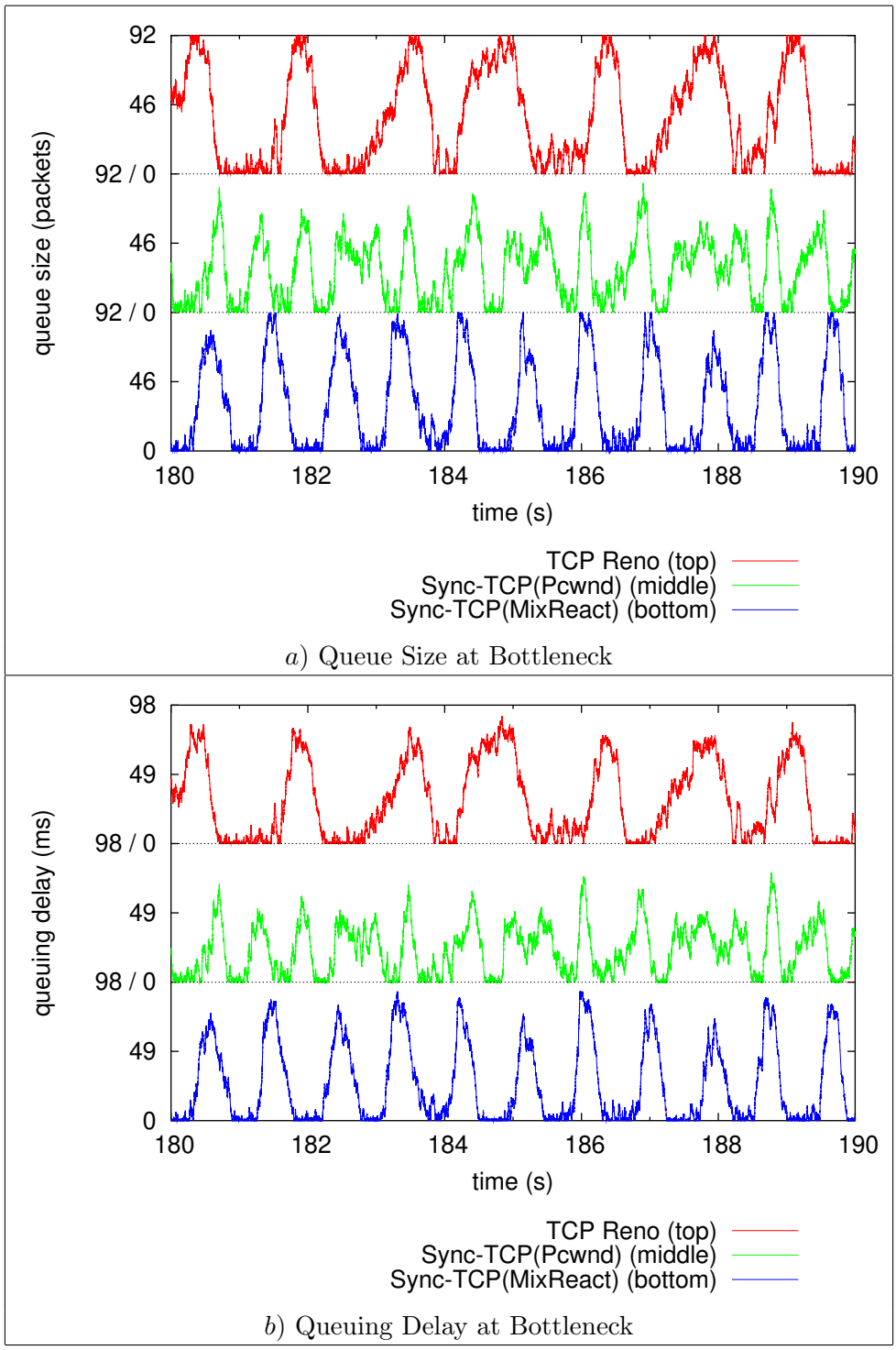


Figure 4.6: Queue Size and Queuing Delay, Single Bottleneck

number of bytes. For example, there will be a smaller queuing delay for a queue filled with 92 54-byte ACKs (4 ms) than with 92 1474-byte data packets (108 ms).

Sync-TCP(Pcwnd) seems to achieve its goal of keeping the queuing delay below 50% of the maximum-observed queuing delay most of the time (Figure 4.6b). The queue size with TCP Reno (Figure 4.6a) has the familiar peak and trough behavior. The TCP Reno flows increase their sending rates until the queue overflows and then they back off and slow down their sending rates. Sync-TCP(MixReact) has much the same queue size behavior as TCP Reno, but with a shorter period. Sync-TCP(Pcwnd) avoids the spikes reaching the maximum queue capacity by backing off when the queue is estimated to be half-full. TCP Reno flows continue to send data until the queue overflows. Sync-TCP(MixReact) flows increase the congestion window as long as the trend of the average queuing delay is decreasing. The averaging of the queuing delay smoothes and delays the signal, which may contribute to Sync-TCP(MixReact) flows increasing their sending rates based on information that is no longer valid. Additionally, Sync-TCP(MixReact) adjusts its AIMD congestion window adjustment parameters every three ACKs rather than at most once every RTT as with Sync-TCP(Pcwnd). This may contribute to the rate of fluctuations in the queue.

Goodput

Sync-TCP(Pcwnd) has less packet loss per flow than either TCP Reno or Sync-TCP(MixReact). Usually the tradeoff with improving loss rates is reducing throughput (or, goodput). If flows back off more often, they are not sending as much data and have the potential to receive lower goodput in return for lower packet loss. The goal is to have lower loss than TCP Reno, but maintain comparable goodput. This can be achieved by having the slower sending rates and lower loss of Sync-TCP balance the with faster sending rates and the transmission pauses while data is being recovered of TCP Reno. Table 4.3 shows that Sync-TCP(Pcwnd) has both the lowest packet loss and the highest network utilization of the protocols tested. This indicates that Sync-TCP(Pcwnd) does achieve this balance as compared with TCP Reno.

Figure 4.7 shows average goodput of each of the 20 FTP flows for TCP Reno, Sync-TCP(Pcwnd), and Sync-TCP(MixReact). In Figures 4.5 and 4.7, the flow ID is on the x-axis (and again, no ordering is implied between flows). This allows direct comparison of the loss percentage to goodput for each flow. For example, with Sync-TCP(MixReact), flow 5 has an average loss rate of just over 1% and an average goodput of just under 400 kbps. Flow 5 also has a base RTT of 88 ms (from Table 4.2).

Figure 4.7 shows that no protocol provided better goodput for every FTP flow. The goodput received by flows using TCP Reno and Sync-TCP(MixReact) is very similar. Sync-TCP(Pcwnd) results in some flows (especially, flows 2 and 3) receiving much less goodput than the corresponding flows with TCP Reno and Sync-TCP(MixReact). There are also flows

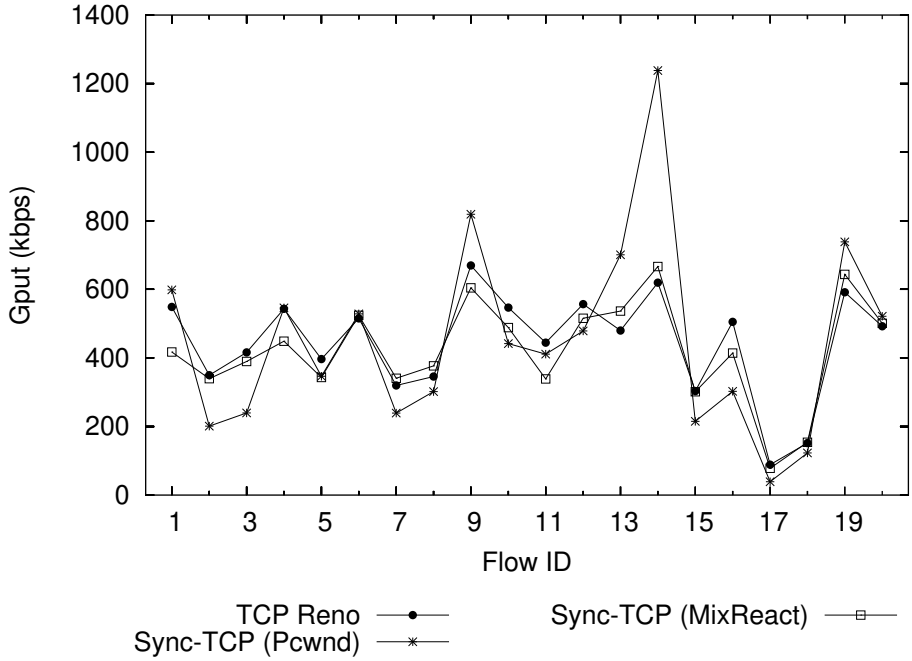


Figure 4.7: Goodput, Single Bottleneck

(such as, flows 9 and 14) that received much more goodput than the corresponding flows with the other protocols.

In TCP Reno, a flow's goodput is directly related to the flow's RTT and percentage of lost packets. Because of the *cwnd* adaptations in Sync-TCP(Pcwnd) and Sync-TCP(MixReact), this direct relationship does not exist. In both of the Sync-TCP protocols, a flow's goodput is also affected by the queuing delays seen by the flow's packets. In TCP Reno, queuing delays add to the flow's RTT, but in the Sync-TCP protocols, increased queuing delays result in congestion window decreases. Sync-TCP(Pcwnd) is essentially TCP Reno with additional *cwnd* reductions. Sync-TCP(MixReact) has a more complex relationship to TCP Reno since it adds *cwnd* reductions and also more aggressive increases of *cwnd* than TCP Reno.

Goodput can be lowered by both congestion window decreases and the amount of time spent retransmitting lost packets. TCP Reno and Sync-TCP(MixReact) see similar goodput because of Sync-TCP(MixReact)'s reduced packet loss (fewer retransmissions and 50% *cwnd* reductions) and Sync-TCP(MixReact)'s more aggressive *cwnd* increases when resources are available.

In the Sync-TCP protocols, the accuracy of a flow's estimate of the maximum amount of queuing available in the network is important to its performance. The 20 FTP flows in these experiments generally had the same estimate of the maximum amount of queuing in the network. For Sync-TCP(Pcwnd), there was a 26 ms difference between the flow with the smallest estimate (80 ms) and the flow with the largest estimate (106 ms). (Note that although

Flow ID	Goodput (kbps)	Base RTT (ms)	Maximum Queuing Delay Estimate (ms)
14	1238.39	16.54	105.746
9	819.11	17.41	99.06
19	738.63	16.36	96.282
13	700.64	17.51	89.948
1	598.62	54.77	100.828
4	546.14	46.04	101.996
6	528.46	20.90	88.194
20	521.45	26.56	87.416
12	478.62	31.11	86.958
10	441.72	31.30	90.152
11	410.94	77.30	101.088
5	346.33	88.02	97.492
16	302.66	52.97	82.844
8	301.80	94.66	95.676
3	239.74	85.70	88.322
7	239.04	98.48	93.268
15	215.06	118.42	93.658
2	201.00	97.21	91.272
18	123.01	410.78	96.924
17	39.02	691.18	79.946

Table 4.4: FTP, Single Bottleneck, Sync-TCP(Pcwnd), Per-Flow Summary

the maximum queuing delay for any packet at the bottleneck was 98 ms, there are additional queuing delays from the access links to the router that contribute to computed queuing delays higher than 98 ms.) For Sync-TCP(MixReact), there was only a 9 ms difference between the smallest estimate (93 ms) and the largest estimate (102 ms). Using Sync-TCP(Pcwnd), flows with larger estimates will reduce *cwnd* less frequently. Using Sync-TCP(MixReact), flows with larger estimates are more likely to see smaller *cwnd* reductions. Table 4.4 gives a summary of performance for all FTP flows with Sync-TCP(Pcwnd). The table is sorted by goodput received. Flows that received high goodput either had a small base RTT or large estimate of the maximum queuing delay. Flow 17 had the lowest goodput, the largest base RTT, and the smallest maximum queuing delay estimate. At the other end, flow 14 had the highest goodput, close to the smallest RTT, and the largest estimate of maximum queuing delay. The RTT actually has some effect on the estimate of maximum queuing delay, because flows with smaller RTTs will receive more ACKs (and thus, OTT and queuing delay estimates) in 300 seconds than flows with longer RTTs.

Table 4.5 shows the same information for Sync-TCP(MixReact). This table is also sorted by goodput, so the flows are not in the same order as in Table 4.4.

Flow ID	Goodput (kbps)	Base RTT (ms)	Maximum Queuing Delay Estimate (ms)
14	666.42	16.54	101.226
19	643.51	16.36	99.487
9	604.33	17.41	99.045
13	536.57	17.51	96.258
6	524.21	20.90	97.567
12	515.63	31.11	94.022
20	501.06	26.56	97.356
10	488.46	31.30	99.369
4	448.73	46.04	98.199
1	417.12	54.77	98.641
16	414.36	52.97	96.912
3	389.55	85.70	101.948
8	376.64	94.66	100.374
5	342.90	88.02	96.115
7	340.70	98.48	98.475
2	339.32	97.21	94.427
11	338.65	77.30	94.962
15	301.40	118.42	97.999
18	153.95	410.78	101.352
17	78.36	691.18	92.850

Table 4.5: FTP, Single Bottleneck, Sync-TCP(MixReact), Per-Flow Summary

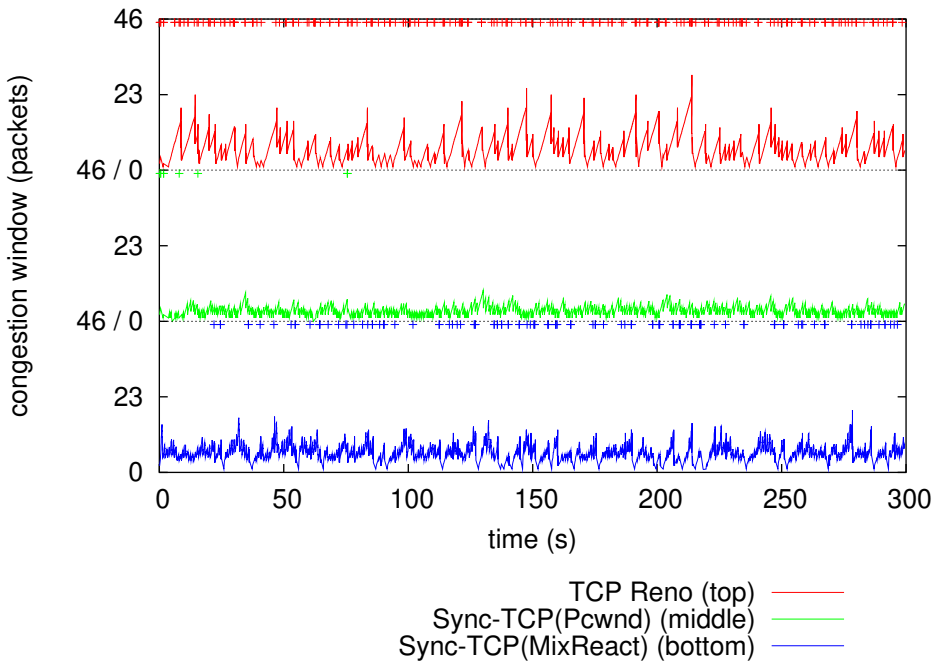


Figure 4.8: Congestion Window and Drops, Flow 2, 97 ms RTT, Single Bottleneck

To investigate how Sync-TCP(Pcwnd) and Sync-TCP(MixReact) adjust a flow's congestion window, I show the congestion windows from three flows using each of the three tested protocols in Figures 4.8-4.10. In these figures, the congestion windows of TCP Reno, Sync-TCP(MixReact), and Sync-TCP(Pcwnd) are plotted on top of each other for comparison. A '+' is plotted for each packet drop from the flow. The y-axis on each plot is divided into three 46-packet intervals, since the maximum send window in these experiments is 46 packets.

Figure 4.8 shows the congestion window and packet drops for flow 2, which has an RTT of 97 ms. Flow 2 received similar goodput with TCP Reno and Sync-TCP(MixReact), but less goodput with Sync-TCP(Pcwnd). The congestion window with TCP Reno shows the familiar sawtooth pattern where the congestion window builds slowly during congestion avoidance and then drops quickly because congestion window reductions in response to packet loss. Sync-TCP(MixReact), on the other hand, has some instances of both gradual increases and gradual decreases in the congestion window. The congestion window with TCP Reno grows larger than with Sync-TCP(MixReact), but the flow also sees more packet loss, resulting in similar goodput. Throughput the experiment, the Sync-TCP(Pcwnd) congestion window was very small. There was very little loss, so most of the congestion window reductions were due to the operation of the Sync-TCP(Pcwnd) algorithm, specifically reducing *cwnd* by half when congestion was indicated. For flow 2, with a maximum queuing delay estimate of 91 ms, any time the computed queuing delay reached 45.5 ms or more, its congestion window would be reduced by 50%. For flow 2, the congestion window remains small throughout the entire

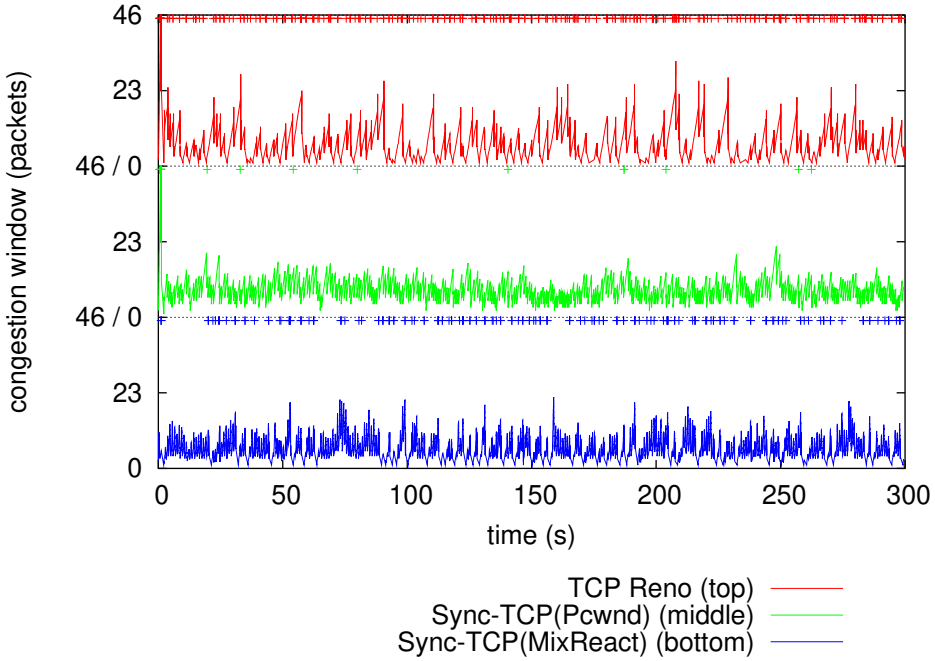


Figure 4.9: Congestion Window and Drops, Flow 14, 17 ms RTT, Single Bottleneck

experiment, especially as compared to TCP Reno, so it sends less data and receives lower goodput than TCP Reno.

Flow 14 with Sync-TCP(Pcwnd) receives higher goodput than with TCP Reno or Sync-TCP(MixReact). Its congestion window and drops are shown in Figure 4.9. With Sync-TCP(Pcwnd), the congestion window remains relatively high and seldom goes to 1 as TCP Reno and Sync-TCP(MixReact) do. This flow had the largest estimate of the maximum queuing in the network (105 ms), so computed queuing delays could reach 52.5 ms before the congestion window was reduced.

For comparison, the congestion window for flow 6 is included in Figure 4.10. Flow 6 received similar goodput with all three protocols. Sync-TCP(Pcwnd) saw the fewest packet drops (12), but has the smallest average congestion window (5 packets). TCP Reno saw the most packet drops (359), but had the largest average congestion window (9 packets). Sync-TCP(MixReact) was in the middle for both packet drops (199) and average congestion window size (7 packets). This demonstrates that a flow with a relatively small congestion window but few packet drops can see similar goodput as a flow that has larger congestion windows but more packet loss.

The performance of a Sync-TCP(Pcwnd) flow partly depends on how its estimate of the maximum queuing capacity in the network compares to that of the other flows. A flow with a smaller estimate will see lower goodput because it will have a larger percentage of congestion window reductions than a flow with a larger estimate. For example, flow 2 (with an estimate

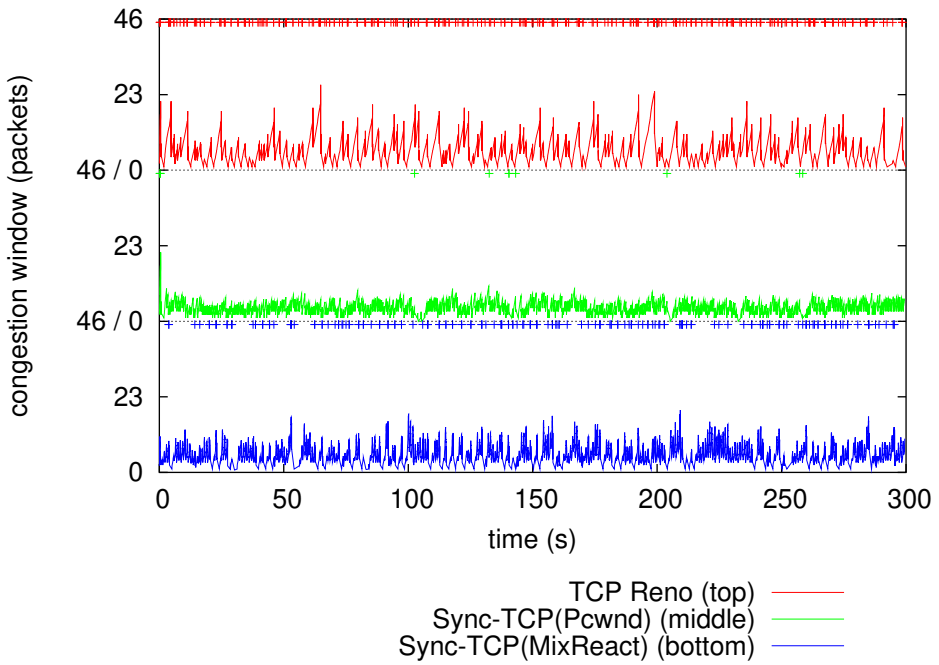


Figure 4.10: Congestion Window and Drops, Flow 6, 21 ms RTT, Single Bottleneck

of 91 ms) had an RTT of 97 ms, which means that there were a maximum of 3092 RTTs in the 300 second experiment. Of these, in 346 RTTs, or 11%, the congestion window was reduced by 50%. Flow 14 (with an estimate of 106 ms), on the other hand, had an RTT of 17 ms, giving it a maximum of 17,647 RTTs during the experiment. Flow 14 only had 331 congestion window reductions, 2% of the maximum number of possible reductions. Flow 6 (with an estimate of 88 ms), which saw the same goodput as with TCP Reno and Sync-TCP(MixReact), had congestion window reductions in 4% of its RTTs.

For all flows with Sync-TCP(MixReact), there are more congestion window increases than decreases. In the absence of packet loss, congestion window adjustments occur every three ACKs. For the 20 flows, about 60% of the adjustments are increases and 40% are decreases. Most of the decreases are of 10% (the trend is increasing and the average queuing delay is between 25-50% of the maximum-observed queuing delay). Most of the increases are of 50% (the trend is decreasing and the average queuing delay is between 0-25% of the maximum-observed queuing delay). There are almost no decreases of 50%, which would occur if the average queuing delay was between 75-100% of the maximum-observed queuing delay and the trend was increasing.

Overall, with a single bottleneck, the Sync-TCP protocols perform better than TCP Reno with regards to packet loss and queue size. In addition to lower loss, both Sync-TCP protocols provide comparable goodput to flows as TCP Reno.

Protocol	Utilization %	Average Queue Size (Packets) at R0	at R1	at R3
TCP Reno	44.86	2.03	37.75	38.58
Sync-TCP(Pcwnd)	50.88	2.76	29.83	36.37
Sync-TCP(MixReact)	43.88	1.57	31.89	32.64

Table 4.6: Aggregate FTP Statistics (Utilization and Queue Size)

Protocol	Average Packet Drop % at R0	at R1	at R3
TCP Reno	0	1.77	1.67
Sync-TCP(Pcwnd)	0.01	0.35	0.31
Sync-TCP(MixReact)	0	1.15	0.67

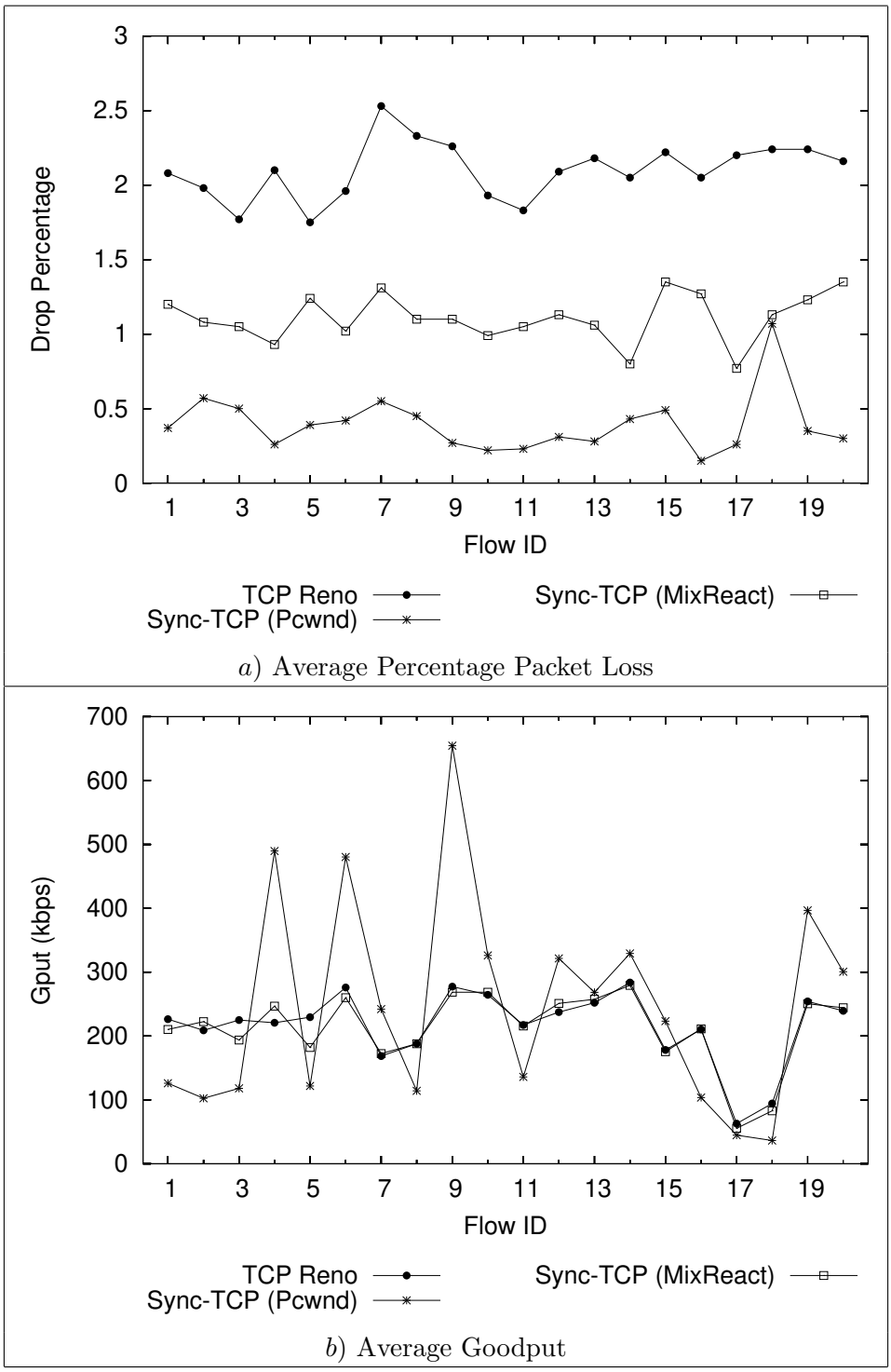
Table 4.7: Aggregate FTP Statistics (Packet Loss)

4.2.2 Multiple Bottlenecks

The cross-traffic in the multiple bottleneck FTP experiments consisted of 10 FTP flows added to the interior links of the network in each direction. This caused congestion on the two interior links (between router R1 and router R2 and between router R3 and router R4 in Figure 4.1).

Tables 4.6 and 4.7 show the aggregate statistics from the multiple bottleneck FTP experiments, including queue size and packet loss at each congested router. The average queue size and average packet loss include both end-to-end traffic and cross-traffic flowing through the congested routers. The packet loss and queue size at R0 is expected to be much smaller than that at R1 and R3 because R0 does not carry any cross-traffic. The only congestion at R0 is due to the aggregation of the FTP flows, and the sending rates of the flows will be affected more by the congestion present on the links with cross-traffic. As with the single bottleneck case, Sync-TCP(Pcwnd) has much lower overall packet loss than either TCP Reno or Sync-TCP(MixReact). In the multiple bottleneck case, network utilization is based on the amount of data that is received by the end-to-end receivers. Since the amount of data sent end-to-end due to the cross-traffic is less than in the single bottleneck case, the end-to-end network utilization will be lower. Sync-TCP(Pcwnd) has higher network utilization than either TCP Reno or Sync-TCP(MixReact), which have similar utilization.

Figure 4.11 shows the packet loss and goodput per flow for the multiple bottleneck experiments. As with the single bottleneck case, each of the 20 FTP flows has lower packet loss with Sync-TCP(Pcwnd) than with either TCP Reno or Sync-TCP(MixReact) (Figure 4.11a). Figure 4.11b shows that, like the single bottleneck experiments, no protocol has better goodput for every flow. Again, TCP Reno flows and Sync-TCP(MixReact) flows see very similar goodput. Sync-TCP(Pcwnd) has several flows (especially flows 4, 6, and 9) that receive much



a) Average Percentage Packet Loss

b) Average Goodput

Figure 4.11: Packet Loss and Goodput, Multiple Bottlenecks

Flow ID	Goodput (kbps)	Base RTT (ms)	Maximum Queuing Delay Estimate (ms)
9	654.45	3.41	255.898
4	489.60	32.04	247.408
6	480.23	6.90	239.586
19	396.60	2.36	220.734
14	329.12	2.54	216.384
10	326.13	17.30	222.072
12	321.48	17.11	222.458
20	300.65	12.56	208.948
13	268.29	3.51	196.002
7	242.07	84.48	242.026
15	223.21	104.42	240.318
11	136.00	63.30	187.636
1	126.04	40.77	177.004
5	121.67	74.02	183.476
3	118.00	71.70	181.294
8	114.07	80.66	180.232
16	103.67	38.97	160.238
2	102.45	83.21	180.192
17	44.77	677.18	173.962
18	36.42	396.78	174.512

Table 4.8: FTP, Multiple Bottlenecks, Sync-TCP(Pcwnd), Per-Flow Summary

better goodput than any of the other flows. As in the single bottleneck case, these three flows' maximum queuing delay estimates were among the highest of the 20 flows.

Tables 4.8 and 4.9 show the goodput, base RTT, and maximum queuing delay estimate for each FTP flow with Sync-TCP(Pcwnd) and Sync-TCP(MixReact), respectively. As with the single bottleneck tables, these tables are sorted by goodput. Those flow with smaller maximum queuing delay estimates and longer RTTs have lower goodput than flows with larger maximum queuing delay estimates and shorter RTTs.

The 20 flows with Sync-TCP(MixReact) generally had the same estimate of the maximum amount of queuing in the network, but there was a larger gap for Sync-TCP(Pcwnd). For Sync-TCP(Pcwnd), there was a 96 ms difference between the flow with the smallest estimate (160 ms) and the flow with the largest estimate (256 ms). For Sync-TCP(MixReact), there was only a 29 ms difference between the smallest estimate (159 ms) and the largest estimate (188 ms).

4.2.3 Summary

Over both single and multiple bottlenecks, Sync-TCP(Pcwnd) provides better performance for FTP flows with regards to overall packet loss, queue size, and link utilization than

Flow ID	Goodput (kbps)	Base RTT (ms)	Maximum Queuing Delay Estimate (ms)
14	279.19	2.54	180.486
9	268.48	3.41	179.612
10	268.45	17.30	185.069
6	260.02	6.90	177.175
13	257.38	3.51	162.350
12	251.00	17.11	184.665
19	250.45	2.36	170.573
4	246.83	32.04	184.789
20	244.15	12.56	187.997
2	222.54	83.21	186.517
11	215.65	63.30	181.358
16	211.12	38.97	171.214
1	210.21	40.77	175.413
3	193.76	71.70	174.589
8	187.85	80.66	170.924
5	181.98	74.02	178.324
15	175.41	104.42	180.716
7	172.53	84.48	175.195
18	82.88	396.78	159.218
17	55.48	677.18	162.614

Table 4.9: FTP, Multiple Bottlenecks, Sync-TCP(MixReact), Per-Flow Summary

either TCP Reno or Sync-TCP(MixReact). Sync-TCP(MixReact) provided lower loss and lower queue sizes than TCP Reno with comparable link utilization over both single and multiple bottlenecks. The early congestion notifications and reactions of Sync-TCP(Pcwnd) and Sync-TCP(MixReact) were successful in keeping queue sizes and packet loss rates smaller than TCP Reno.

These FTP experiments were designed to provide an evaluation of the congestion control mechanisms in a relatively simple environment. For all of these experiments, every flow had transitioned from slow start to using early congestion detection and reaction in Sync-TCP(Pcwnd) and Sync-TCP(MixReact). In the HTTP experiments, there will be many flows that complete without ever leaving slow start. Thus, there will be a mix of flows that are sensitive to increases in queuing delay and flows that are not.

4.3 HTTP Evaluation

In this section, I will compare the performance of traffic using Sync-TCP(Pcwnd), Sync-TCP(MixReact), and the "best" standards-track TCP protocol. The HTTP experiments are used for the actual evaluation of Sync-TCP with realistic traffic. The FTP experiments were used mainly to compare the operation of Sync-TCP in the steady-state with that of TCP Reno (which has been studied in-depth and is well-understood). Using an HTTP workload, I evaluated several standards-track TCP congestion control mechanisms including TCP Reno over drop-tail routers ("TCP Reno") and ECN-enabled SACK TCP over Adaptive RED routers ("SACK-ECN") with various RED parameters. This evaluation is presented in Appendix B. I found that there is a tradeoff in HTTP response time performance between TCP Reno and SACK-ECN. Also, the best RED parameters for SACK-ECN differed by HTTP traffic load level. For these reasons, I will compare the two Sync-TCP mechanisms to both TCP Reno and the best SACK-ECN at each load level.

The HTTP experiments were run for a range of load levels, from a lightly-loaded network (50% average utilization) to a heavily-overloaded network (105% average utilization). The bursty traffic used in these experiments contrasts with the FTP experiments where the traffic in the network was a constant amount for the duration of the experiment.

4.3.1 Single Bottleneck

The performance of HTTP over a single bottleneck with the Sync-TCP protocols was compared with HTTP performance with TCP Reno over drop-tail routers (referred to as "TCP Reno") and ECN-enabled SACK TCP over Adaptive RED routers (referred to as "SACK-ECN") with two different sets of RED parameters. The first set of parameters (referred to as "5 ms, qlen=340") uses the recommended settings for Adaptive RED [FGS01]: 5 ms target delay and an essentially infinite maximum queue size (340 packets). The large queue is to

Load	Average Queue Size Target (Packets)	Target Delay (ms)
80%	23	10
85%	36	15
90%	54	23
95%	73	31
100%	91	38
105%	103	44

Table 4.10: Parameters for "tuned" Adaptive RED

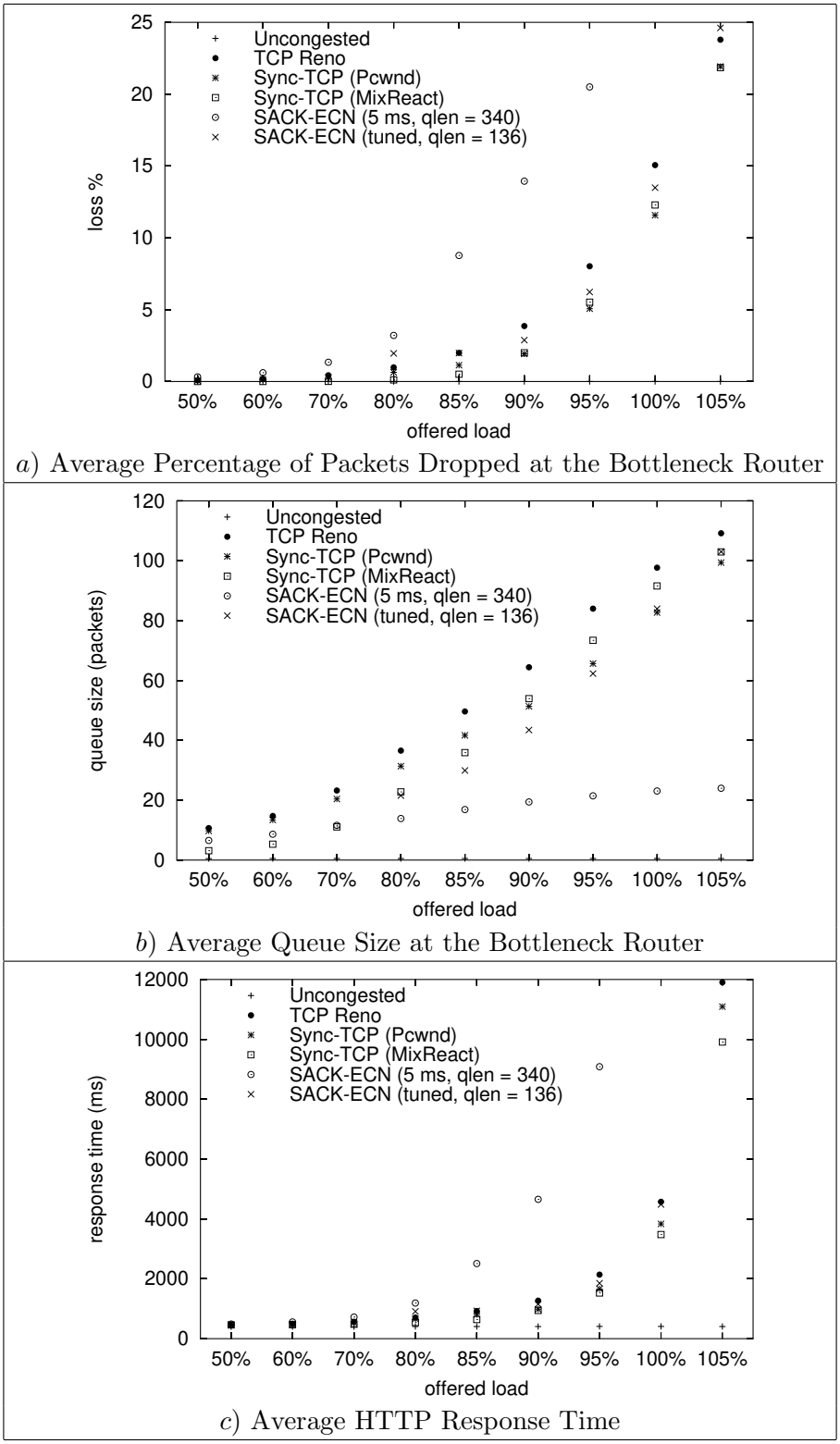
prevent tail-drops and isolate the behavior of the Adaptive RED mechanism. The second set of parameters ("tuned, qlen=136") is tuned to have the same average queue size as Sync-TCP(MixReact). The average queue size target and target delay, which is the parameter given to Adaptive RED, are listed in Table 4.10. The maximum queue size for the tuned SACK-ECN experiments is the same as with the drop-tail queue for Sync-TCP and TCP Reno (136 packets). Note that the tuned SACK-ECN experiments are only shown here for loads between 80-105% (see Appendix B.3).

Summary Statistics

Figures 4.12-4.14 show the performance of TCP Reno, Sync-TCP(Pcwnd), Sync-TCP(MixReact), and the two SACK-ECN cases for various metrics at all of the tested load levels. At every load level, the Sync-TCP protocols see less packet loss at the bottleneck router than any of the other protocols. Sync-TCP(MixReact) has slightly less loss (Figure 4.12a) than Sync-TCP(Pcwnd) between 50-85%, but at 90-95% loads, Sync-TCP(Pcwnd) has slightly less loss than Sync-TCP(MixReact). By adjusting their congestion windows based on queuing delays, the Sync-TCP protocols are able to avoid packet losses and keep the average queue size smaller than TCP Reno.

The average queue size at the bottleneck router for Sync-TCP(MixReact) is lower than TCP Reno at all load levels. Tuned SACK-ECN has lower average queue sizes than both Sync-TCP(MixReact) and Sync-TCP(Pcwnd), while SACK-ECN-5ms has the lowest average queue sizes. Sync-TCP(MixReact) results in lower queue sizes than Sync-TCP(Pcwnd) up to loads of 85%. Past loads of 90%, Sync-TCP(Pcwnd) results in lower queue sizes than Sync-TCP(MixReact) at the bottleneck link. This is similar to the behavior of loss as load increases.

Figure 4.12e shows the average goodput per response. At each load level, both Sync-TCP protocols have higher goodput than TCP Reno. At levels of 85-95%, tuned SACK-ECN has the highest goodput per response. The performance crossover with Sync-TCP(MixReact) and Sync-TCP(Pcwnd) that was present with packet loss and queue size is also evident with goodput per response.



a) Average Percentage of Packets Dropped at the Bottleneck Router

b) Average Queue Size at the Bottleneck Router

c) Average HTTP Response Time

Figure 4.12: HTTP Summary Stats, Single Bottleneck (drops, queue size, average response time)

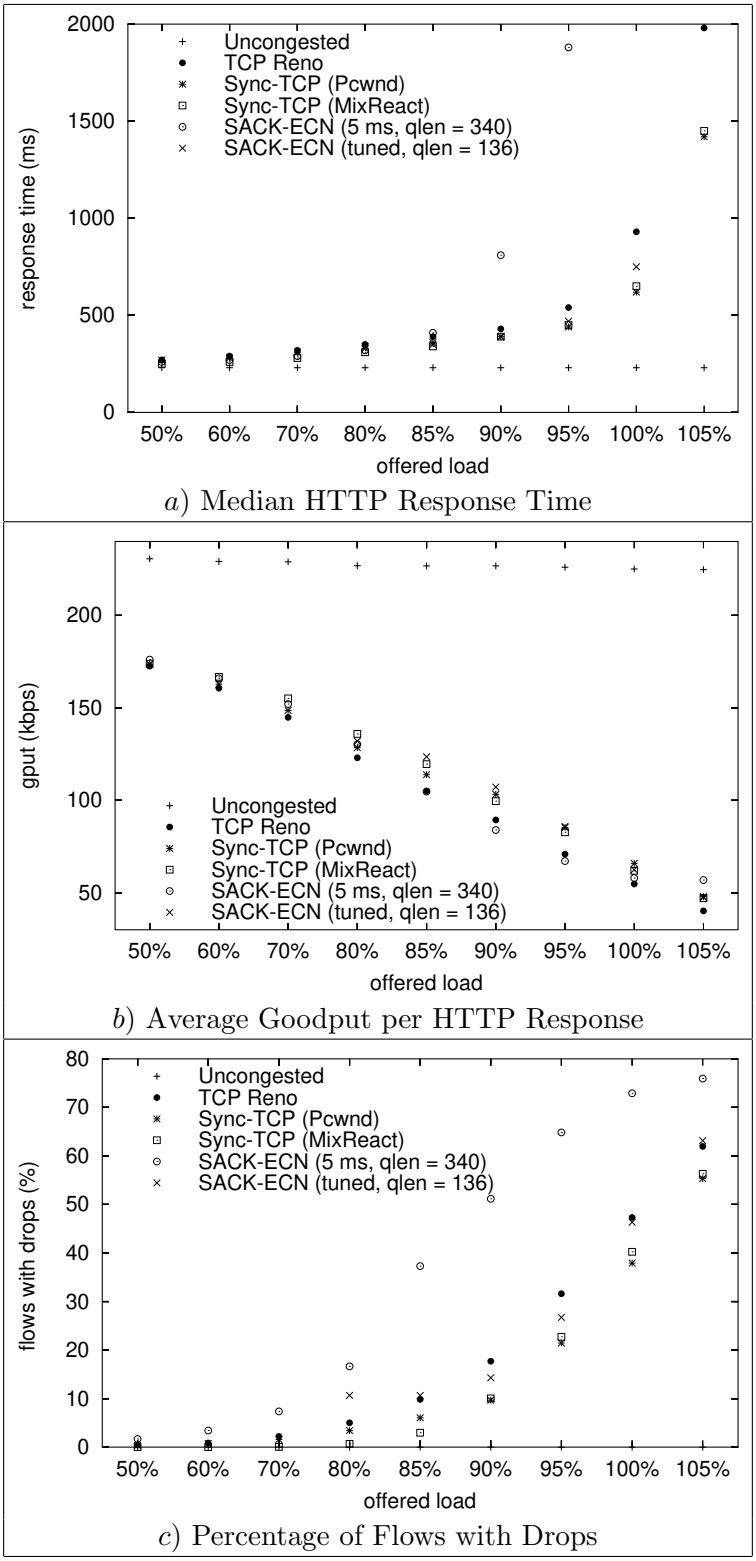


Figure 4.13: HTTP Summary Stats, Single Bottleneck (response time, goodput per response, flows with drops)

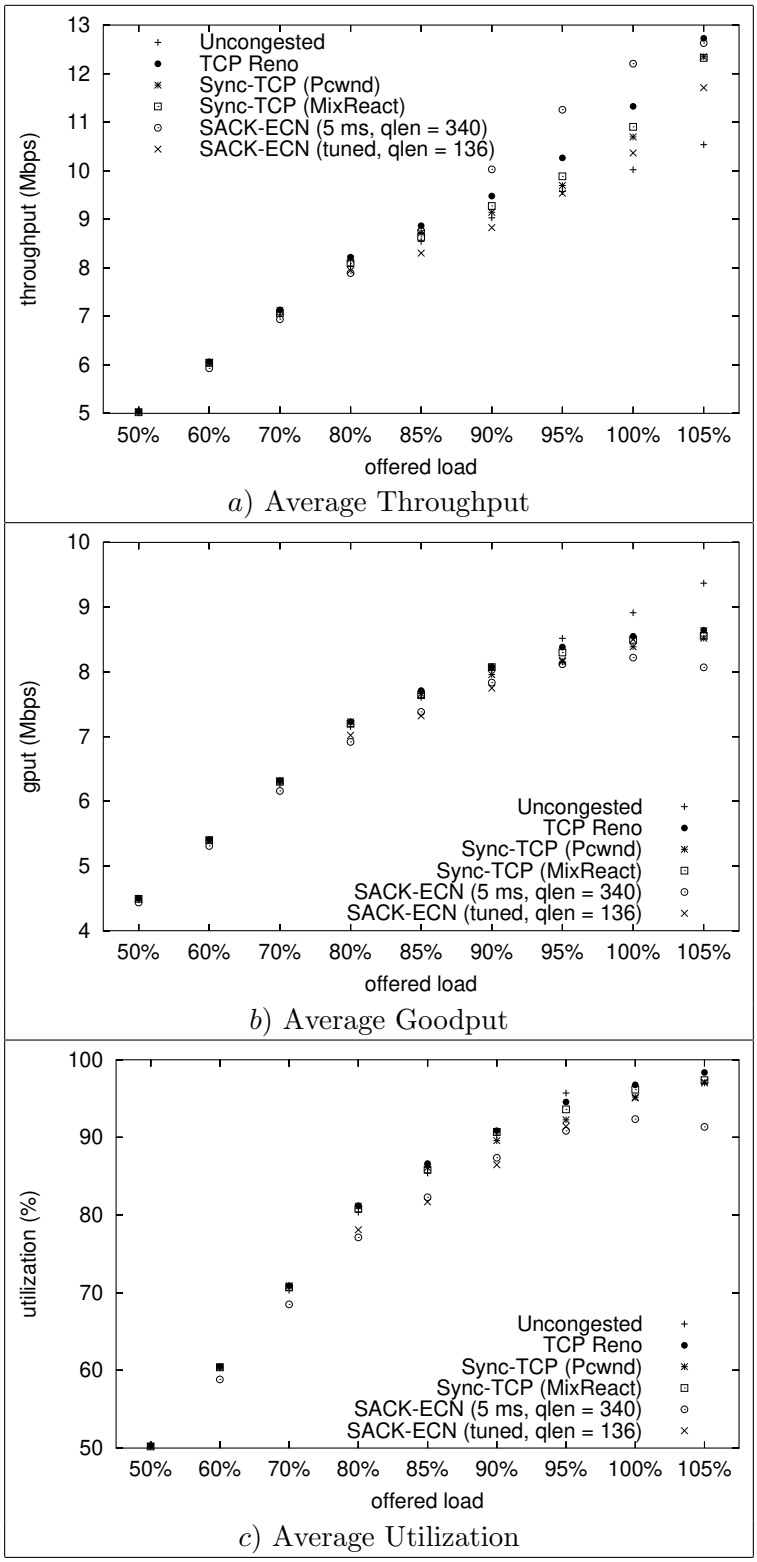


Figure 4.14: HTTP Summary Stats, Single Bottleneck (throughput, goodput, utilization)

Load	TCP Reno Packets Dropped	TCP Reno % Dropped	Sync-TCP (Pcwnd) Packets Dropped	Sync-TCP (Pcwnd) % Dropped	Sync-TCP (MixReact) Packets Dropped	Sync-TCP (MixReact) % Dropped
50%	3589	0.08	2598	0.06	0	0
60%	8151	0.18	6418	0.14	0	0
70%	21,522	0.43	14,717	0.29	312	0.01
80%	45,422	0.98	29,387	0.64	4969	0.11
85%	100,001	1.99	56,064	1.13	24,801	0.51
90%	181,997	3.86	87,998	1.92	91,673	2.00
95%	416,811	8.02	252,344	5.07	277,257	5.52
100%	795,797	15.06	585,585	11.57	627,287	12.29
105%	1,340,671	23.79	1,158,960	21.92	1,164,277	21.85

Table 4.11: Packet Loss

Table 4.11 shows the actual number of packets dropped and the corresponding packet drop percentage for each load level for TCP Reno, Sync-TCP(Pcwnd), and Sync-TCP(MixReact). The interesting thing to note is that at 50% and 60% loads, there are no packets dropped with Sync-TCP(MixReact). Additionally, the amount of packet loss increases dramatically for Sync-TCP(MixReact) at 90% load. Table 4.11 also shows that the 100% and 105% loads are severely overloaded with double-digit percentage packet loss for all of the protocols.

Summary Response Time CDFs

Figures 4.15 and 4.16 show the HTTP response time CDFs for each of the tested protocols at all load levels. These graphs are provided as a reference to show how increasing load affects the various protocols. The uncongested HTTP response time curve (from experiments run with 100 Mbps links between the routers), labeled as "UN" is the best case possible. There was no network congestion, and so, the HTTP response times were solely a factor of the base RTT and the HTTP response size.

Response Time CDFs

Figures 4.17-4.19 show the response time CDFs for the uncongested case (100 Mbps on the interior links), TCP Reno, Sync-TCP(Pcwnd), Sync-TCP(MixReact), and the best SACK-ECN for that load. For loads 50-70% (Figures 4.17a-c) the SACK-ECN protocol is the one using the recommended target delay setting of 5 ms. For loads 80% and greater (Figures 4.18 and 4.19), the tuned SACK-ECN version is used. At 50-60% loads, there is no large difference between any of the protocols. At the load levels from 70-85%, Sync-TCP(MixReact) performs better than the other protocols. At these load levels, a larger percentage of HTTP request-

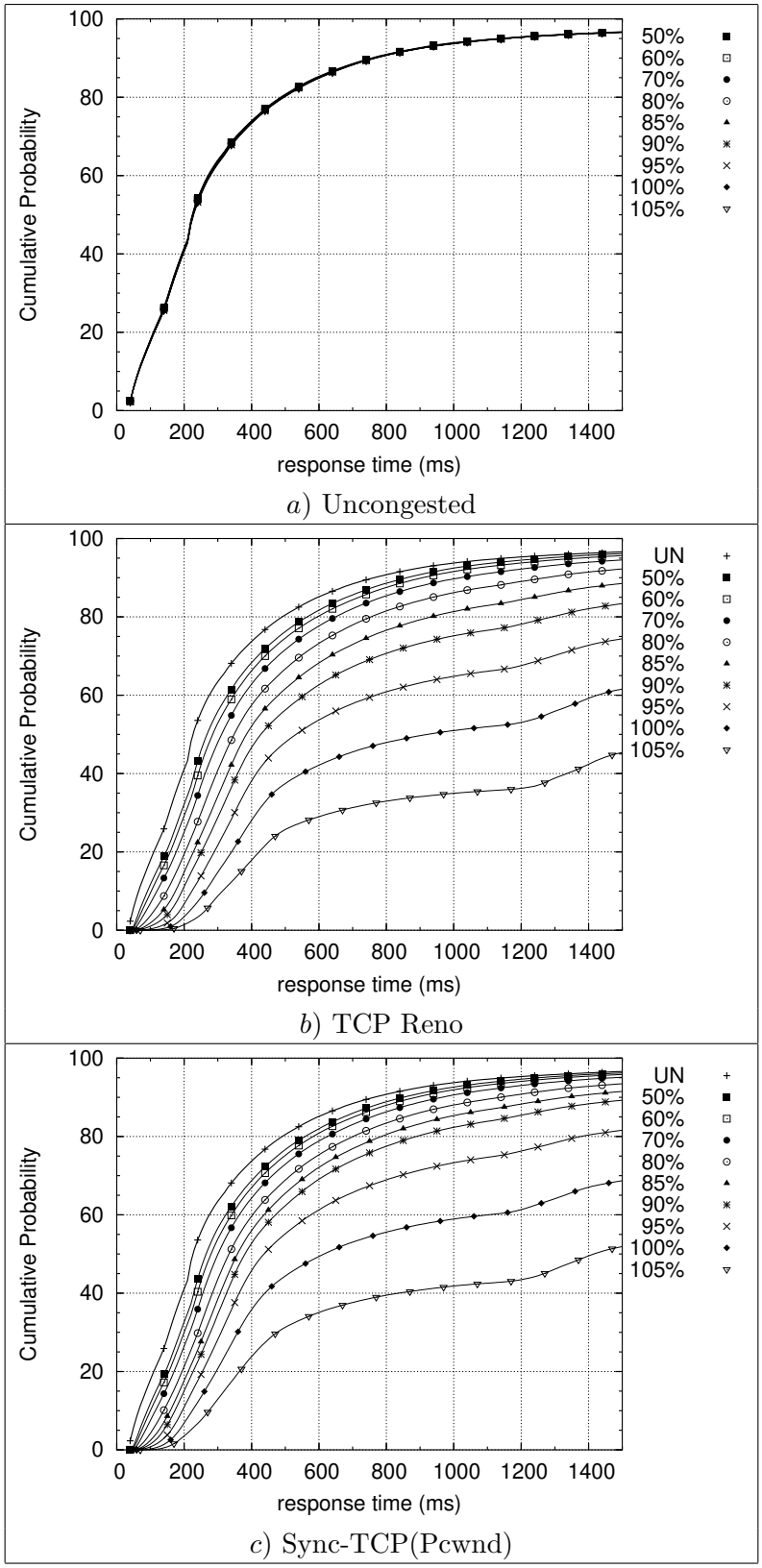


Figure 4.15: Summary Response Time CDFs (Uncongested, Reno, Sync-TCP(Pcwnd))

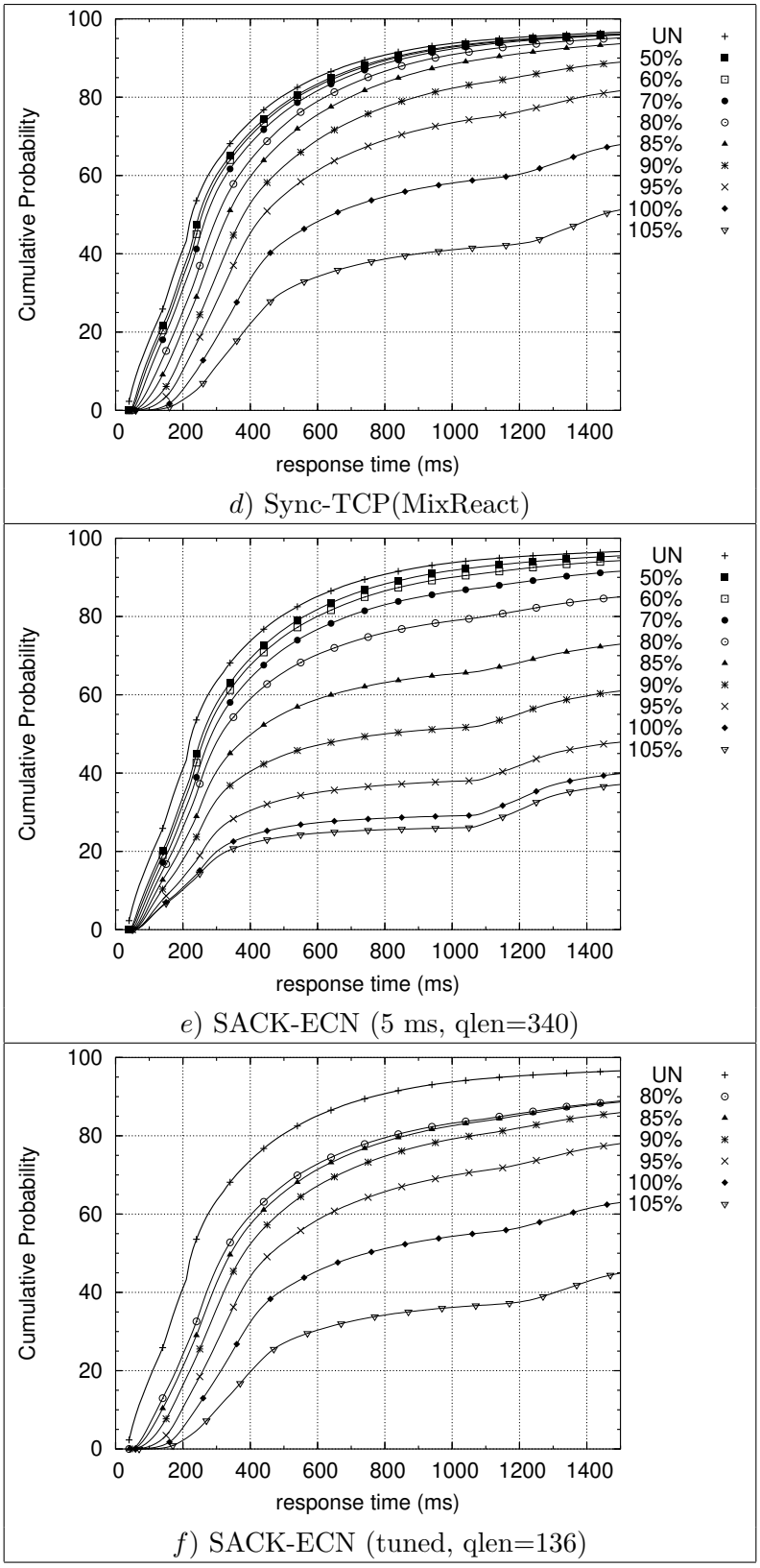


Figure 4.16: Summary Response Time CDFs (Sync-TCP(MixReact), SACK-ECN-5ms, SACK-ECN-tuned)

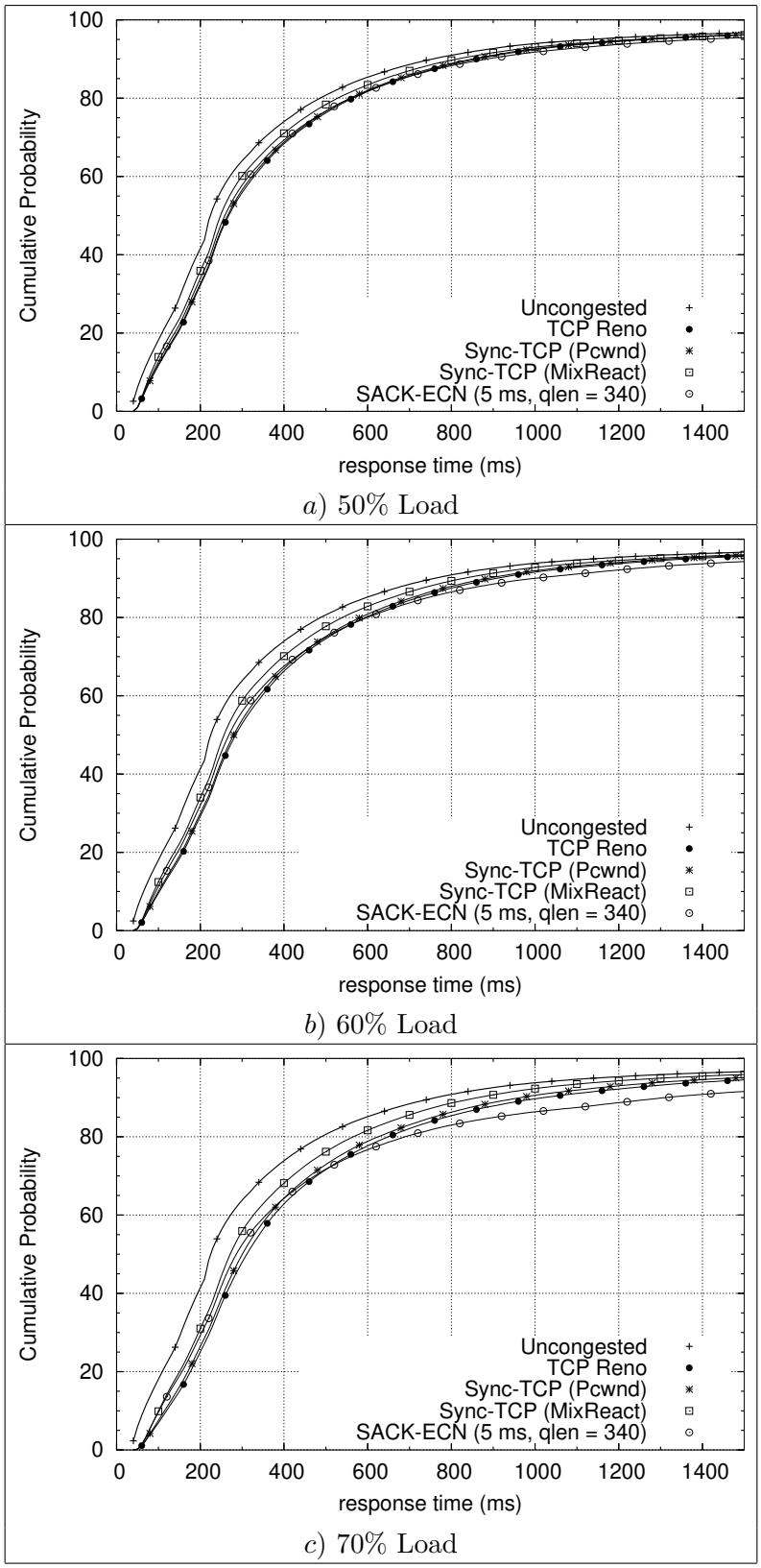


Figure 4.17: Response Time CDFs, Single Bottleneck, Light Congestion

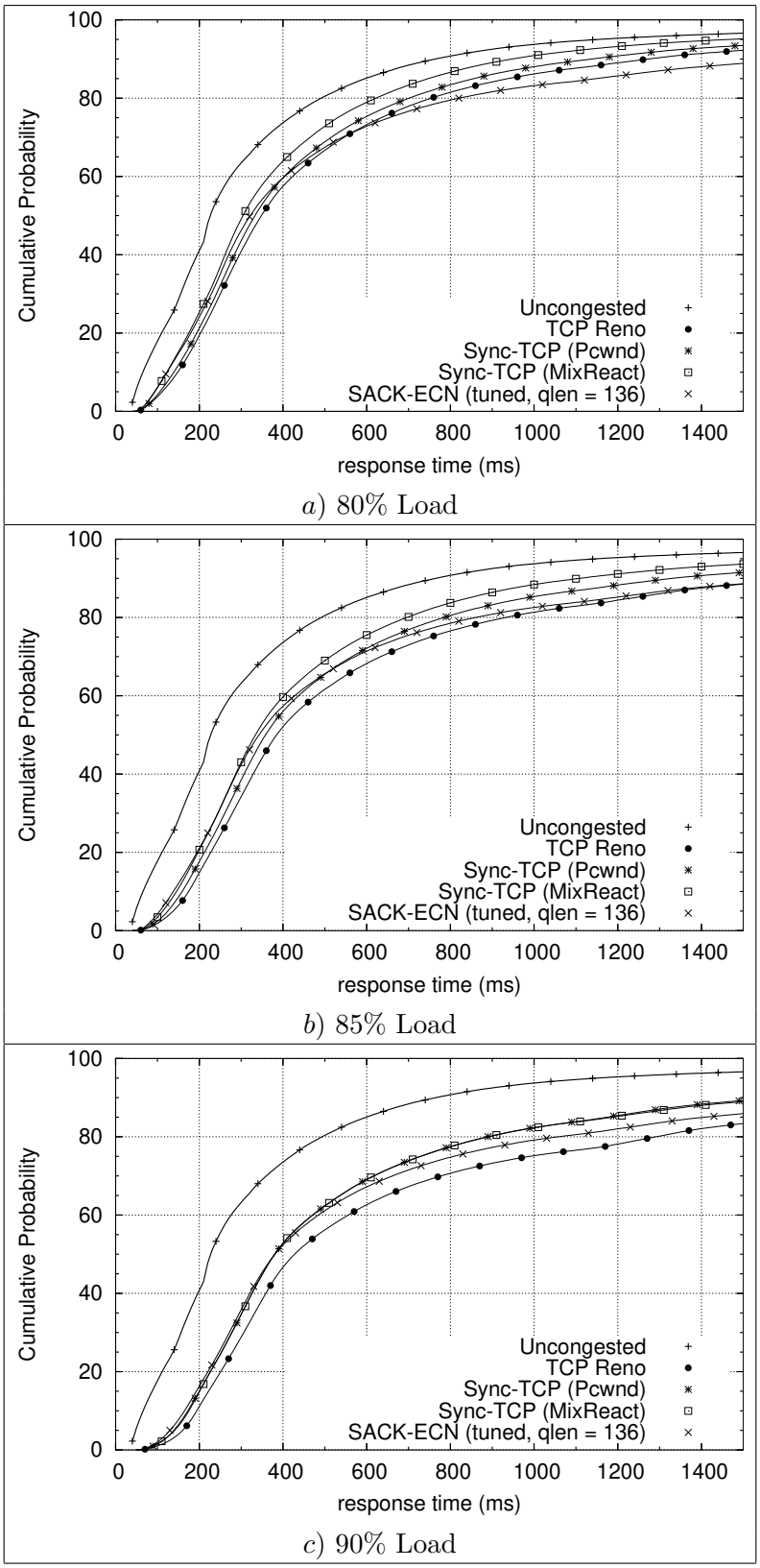


Figure 4.18: Response Time CDFs, Single Bottleneck, Medium Congestion

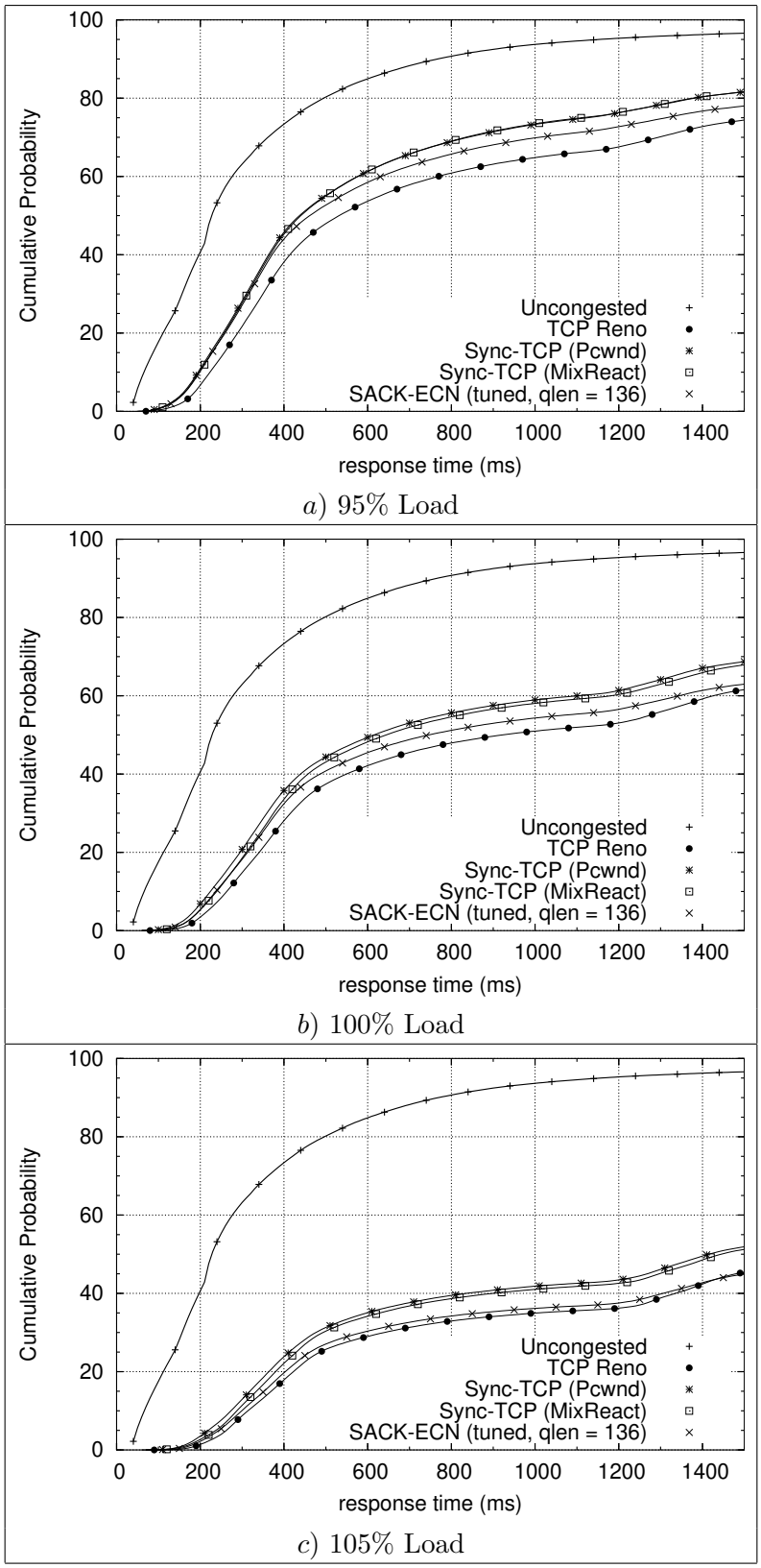


Figure 4.19: Response Time CDFs, Single Bottleneck, Heavy Congestion

response pairs completed in 1500 ms or less with Sync-TCP(MixReact) than with the other protocols. For example, at 85% load with Sync-TCP(MixReact), 94% of the request-response pairs completed in less than 1500 ms, while Sync-TCP(Pcwnd) had 92%, Reno had 89%, and SACK-ECN also had 89%. At load levels (Figure 4.18) higher than 85%, there is no large difference between Sync-TCP(Pcwnd) and Sync-TCP(MixReact). At every load level, for responses that completed in 1500 ms or less, Sync-TCP(MixReact) provides performance that is no worse, and most of the time is much better, than TCP Reno and SACK-ECN.

Again, the performance crossover between Sync-TCP(Pcwnd) and Sync-TCP(MixReact) is evident. For loads up to 85%, Sync-TCP(MixReact) performs better. At 90-95% loads, the performance of the two protocols is indistinguishable, and at loads of 100% and greater, Sync-TCP(Pcwnd) is slightly better. Figure 4.12f shows the percentage of flows that experienced packet loss. A Sync-TCP(Pcwnd) flow does not leave slow start and use early congestion detection until at least one packet has been dropped from the flow. Thus, the percentage of flows with packet drops is also the percentage of flows that are using the Sync-TCP(Pcwnd) congestion control algorithm. There is a large increase in the percentage of flows actually using Sync-TCP(Pcwnd) at 95% load. If a large number of flows use Sync-TCP(Pcwnd), they will backoff during times of congestion to avoid packet loss. This will reduce the size of the queue at the bottleneck. At loads up to 70%, Sync-TCP(Pcwnd) behaves much like TCP Reno. At these load levels, less than 2% of the flows experienced packet loss and used the Sync-TCP(Pcwnd) algorithm. With most of the flows in slow start, Sync-TCP(Pcwnd) sees very similar performance as TCP Reno. With FTP traffic, 100% of the flows actually used Sync-TCP(Pcwnd), and as a result, Sync-TCP(Pcwnd) saw very low packet loss. With HTTP traffic, there only needs to be a small percentage of the flows using Sync-TCP(Pcwnd) to perform better than TCP Reno, but a much larger percentage to perform better than Sync-TCP(MixReact). There is a concern that the flows using Sync-TCP(Pcwnd) will backoff and see worse performance that they would have if they had used TCP Reno. The extent of this occurring will be assessed when looking at response time CCDFs. With the current implementation of Sync-TCP(Pcwnd) and its packet loss requirement, to increase the number of flows using Sync-TCP(Pcwnd), the number of flows with packet loss must also increase.

A Sync-TCP(MixReact) flow leaves slow start and begins using Sync-TCP(MixReact) only after nine ACKs have been received. At all load levels, this occurs in only 5% of the completed request-response pairs (about 12,500 pairs). As with Sync-TCP(Pcwnd), response time CCDFs will be presented to show how these flows are affected.

Response Time CCDFs

Figures 4.20-4.22 show the HTTP response time CCDFs. At 50% (Figure 4.20a), Sync-TCP(MixReact) performs the worst of all of the protocols. For example, with Sync-TCP(MixReact), 30 request-response pairs take longer than 40 seconds to complete. The other protocols have no more than 9 pairs (SACK-ECN) that take longer than 40 seconds to complete. The performance of Sync-TCP(MixReact) in relation to the other protocols improves greatly at 70% load. The poor performance of Sync-TCP(MixReact) at 50-60% loads for responses that take longer than 4 seconds to complete can be attributed to the implementation of the SyncMix congestion detection mechanism when no packets have been dropped (see section 3.5.5). At 50-60% loads with Sync-TCP(MixReact), there were actually no packets dropped during the entire experiment. So, according to SyncMix, no flow had a "good" estimate of the maximum amount of queuing available in the network. One approach to improving performance in the absence of packet loss could be to look at the number of ACKs (and, thus, OTT estimates) that have returned since the maximum queuing delay was first observed. If this count crossed a reasonable threshold, and thus, the maximum-observed queuing delay was stable, then the current maximum-observed queuing delay could be said to be a good estimate of the maximum queuing available in the network. Another approach would be to use trend analysis of the average queuing delay as the main congestion detection factor and use a standard reaction based on if the trend was increasing or decreasing. This would be very similar to SyncTrend congestion detection with a new congestion reaction mechanism. Using such an optimization, I would expect that the HTTP response time performance of Sync-TCP(MixReact) at loads where there was a very low percentage of packet loss would improve. For other loads, though, the use of Sync-TCP(MixReact) in only 5% of the request-response pairs does not seem to detrimentally affect their performance. The performance of these flows, though, will be investigated in more detail in later sections.

Sync-TCP(Pcwnd) has rather poor performance compared to Sync-TCP(MixReact) at loads of 70-85%. For example, at 80% load with Sync-TCP(Pcwnd) there are 15 request-response pairs that take longer than 100 seconds to complete. With Sync-TCP(MixReact), there are only 6 such request-response pairs. At higher loads, the performance of Sync-TCP(Pcwnd) is no better than that of Sync-TCP(MixReact). With Sync-TCP(Pcwnd), the argument could be made that there are some number of flows (those that use early congestion detection and reaction) that see very poor performance, while other flows (those that remain in slow start) see very good performance and complete their request-response pairs quickly.

With SACK-ECN, there is a spike around 6 seconds. This corresponds to the initial timeout period that occurs if a SYN or SYN/ACK is dropped. At 70-85%, the same spike begins to emerge for TCP Reno and Sync-TCP(Pcwnd), while for Sync-TCP(MixReact) the spike is much lower.

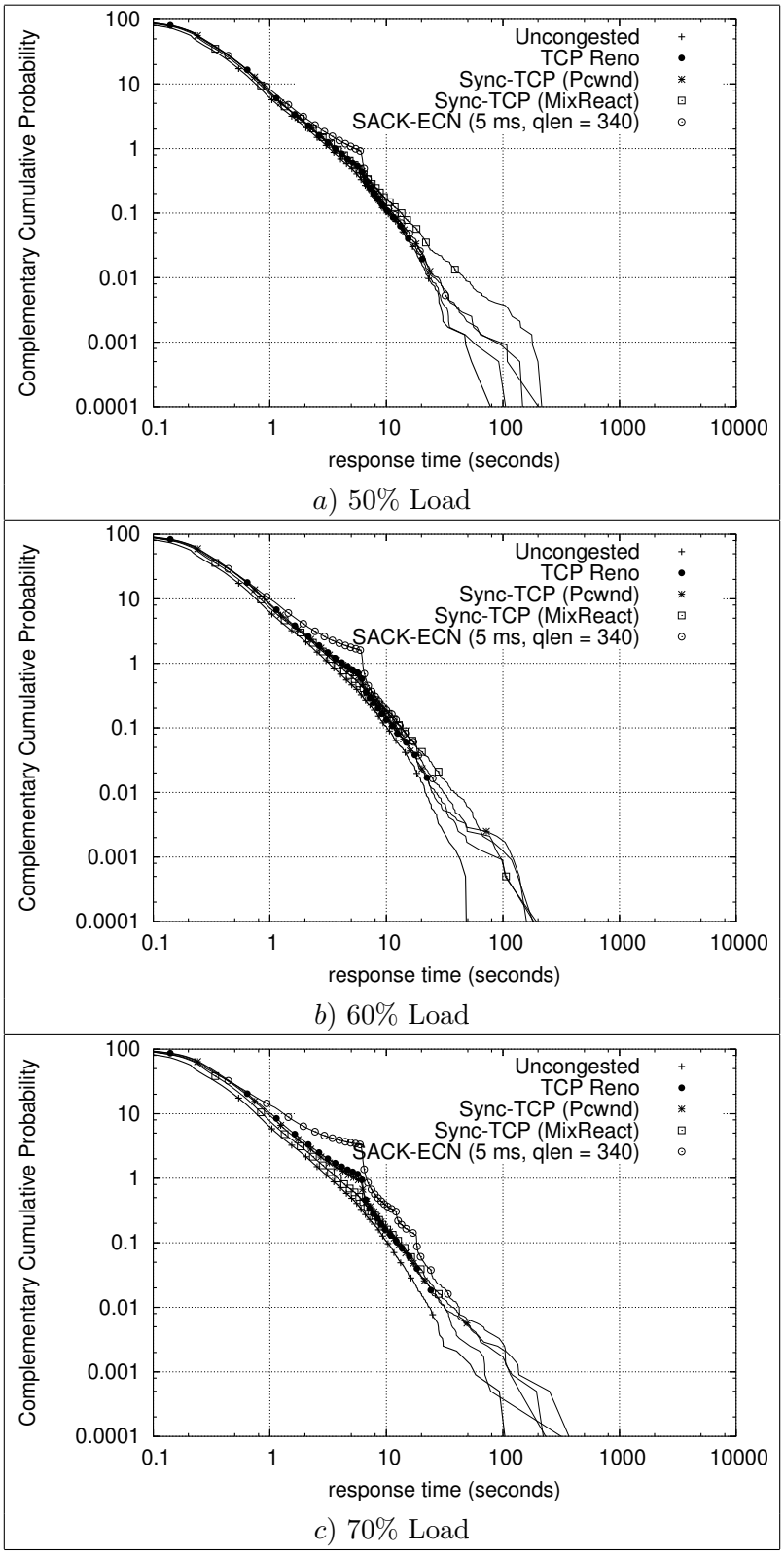


Figure 4.20: Response Time CCDFs, Single Bottleneck, Light Congestion

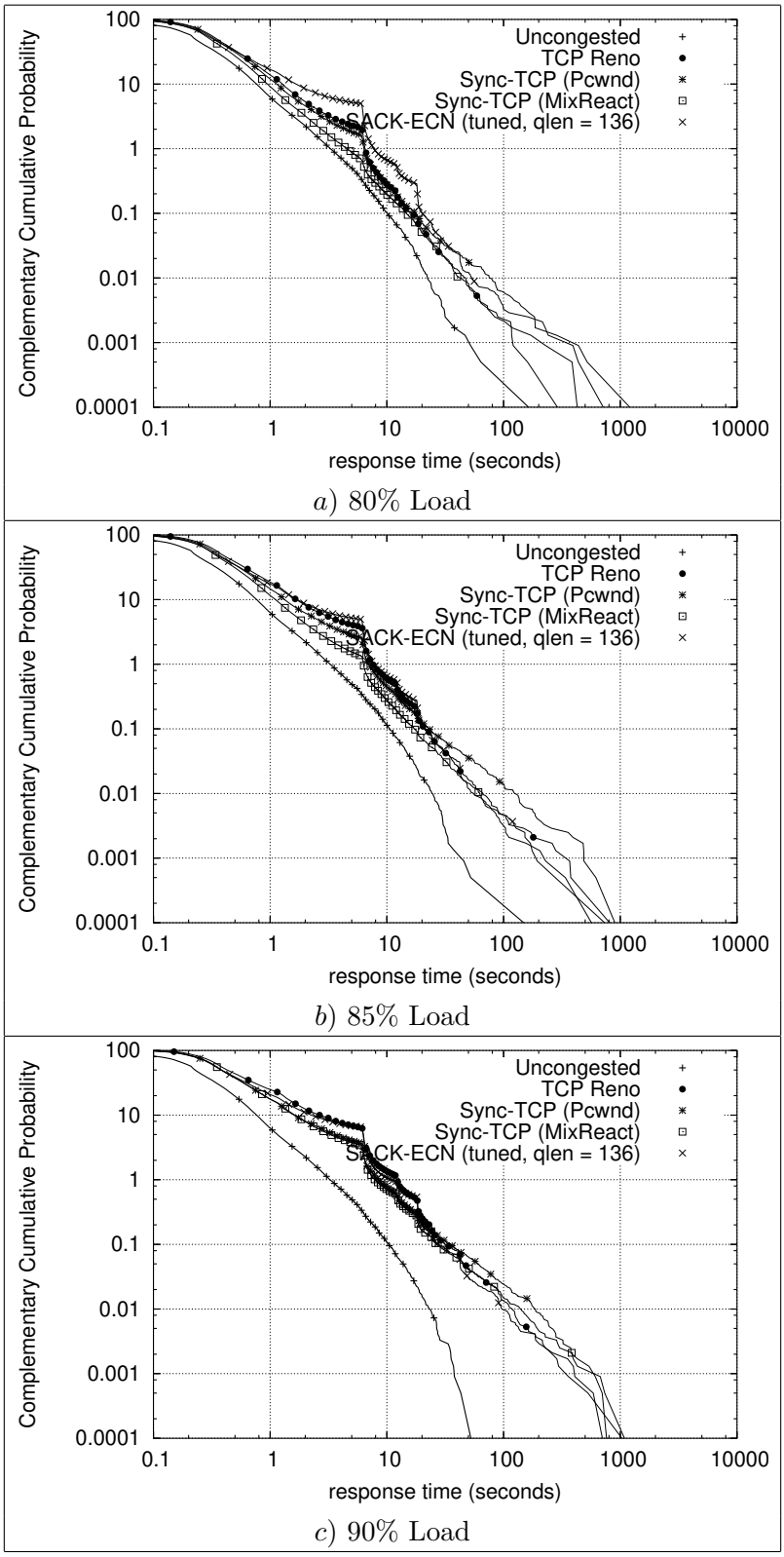


Figure 4.21: Response Time CCDFs, Single Bottleneck, Medium Congestion

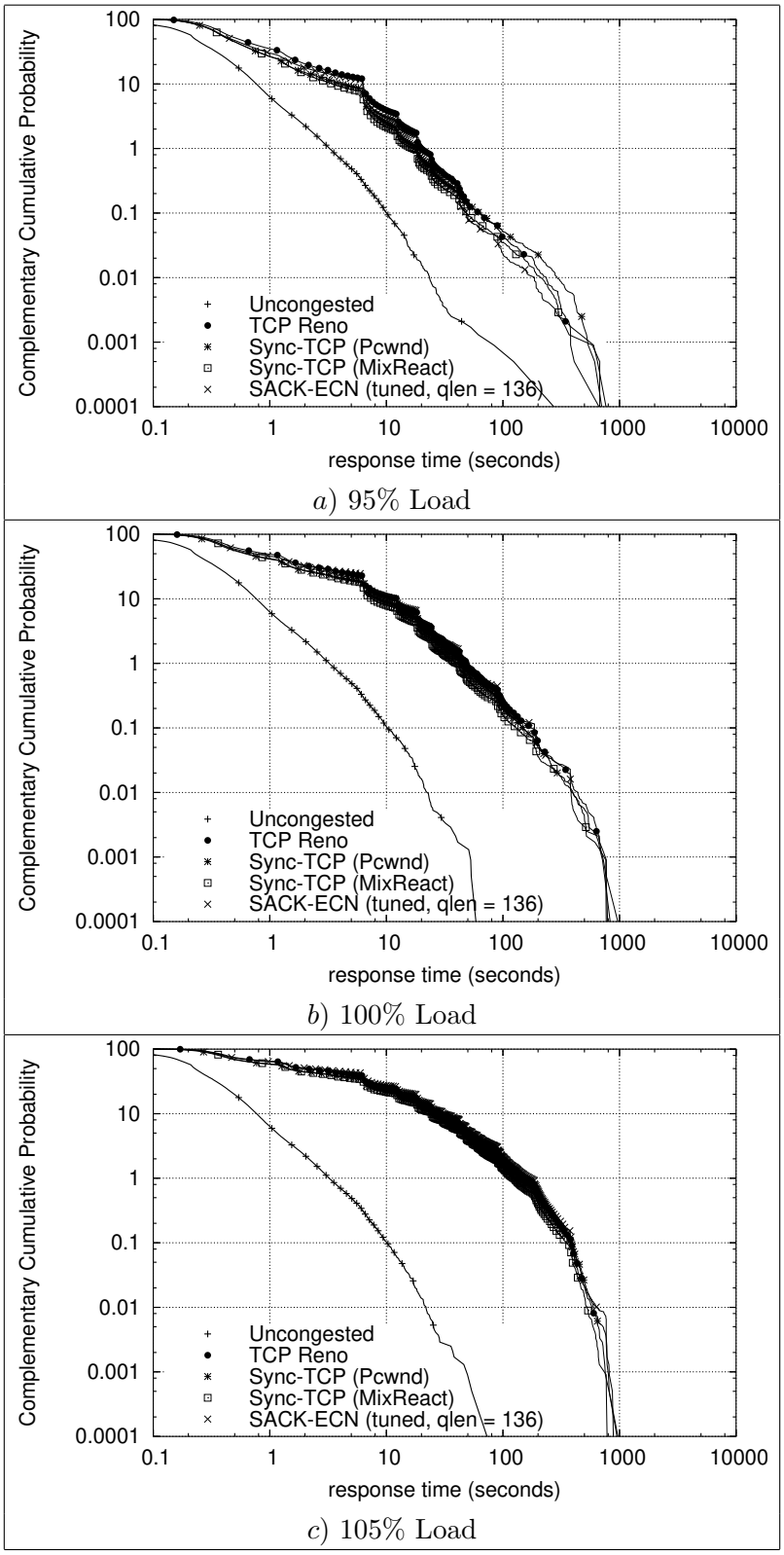


Figure 4.22: Response Time CCDFs, Single Bottleneck, Heavy Congestion

Queue Size CDFs

Figures 4.23-4.25 show the CDFs of queue size at the bottleneck router. These graphs give a more complete picture of how the queue size varies among the protocols. At load levels of 50-60%, Sync-TCP(MixReact) keeps the queue small, and more often, smaller than TCP Reno, Sync-TCP(Pcwnd), and SACK-ECN. For loads 50-70%, the queue size CDF of Sync-TCP(Pcwnd) is very similar to that of TCP Reno. This indicates that the small percentage of flows that are using Sync-TCP(Pcwnd) (under 2%) does not affect queue size greatly. For loads of 85-95%, SACK-ECN has the best queue size CDF, but these shorter queue sizes come at the expense of higher drop rates (Figure 4.12a). This is because Adaptive RED reduces queue sizes by signaling congestion early through marking or dropping packets.

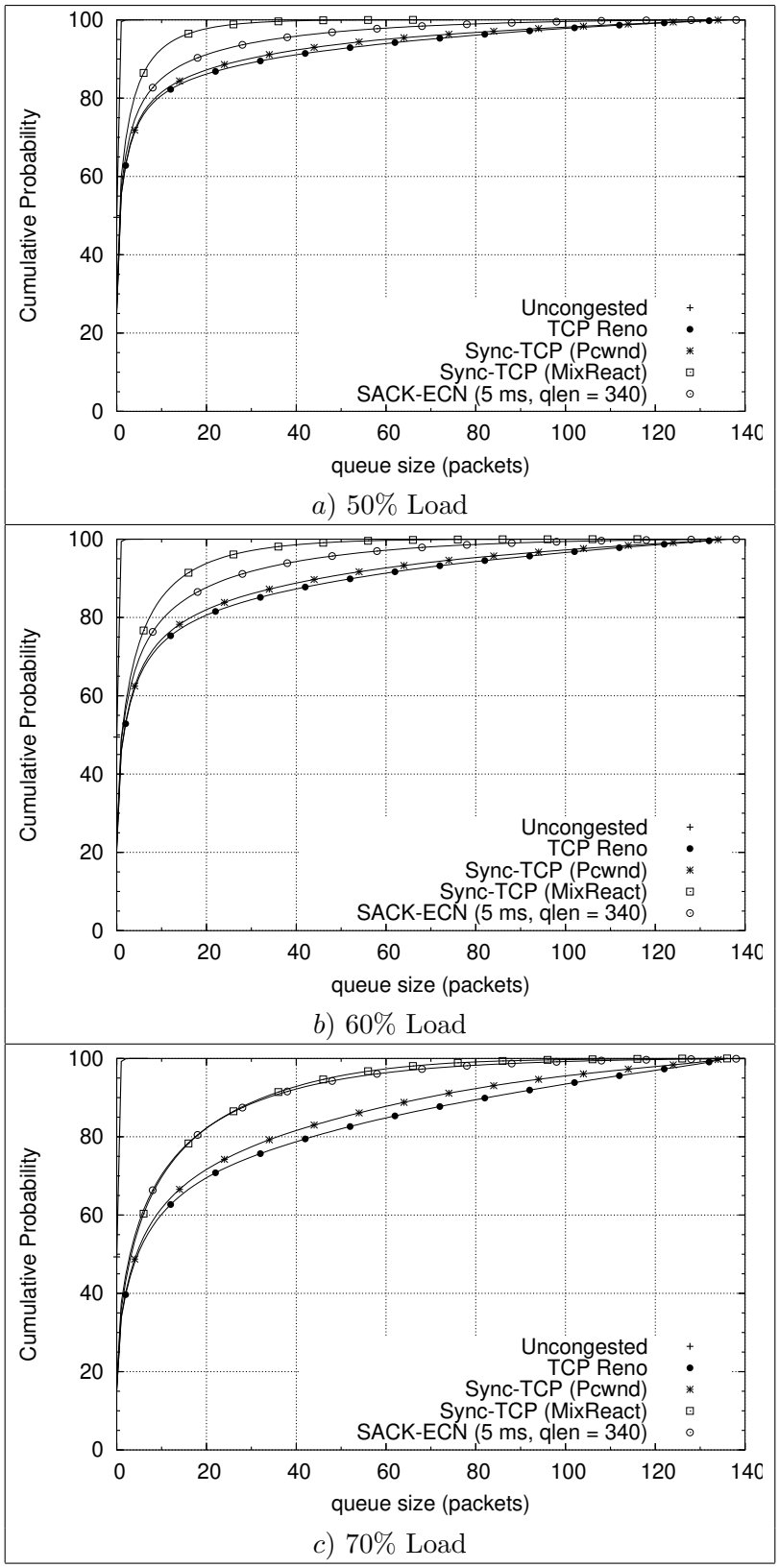


Figure 4.23: Queue Size CDFs, Single Bottleneck, Light Congestion

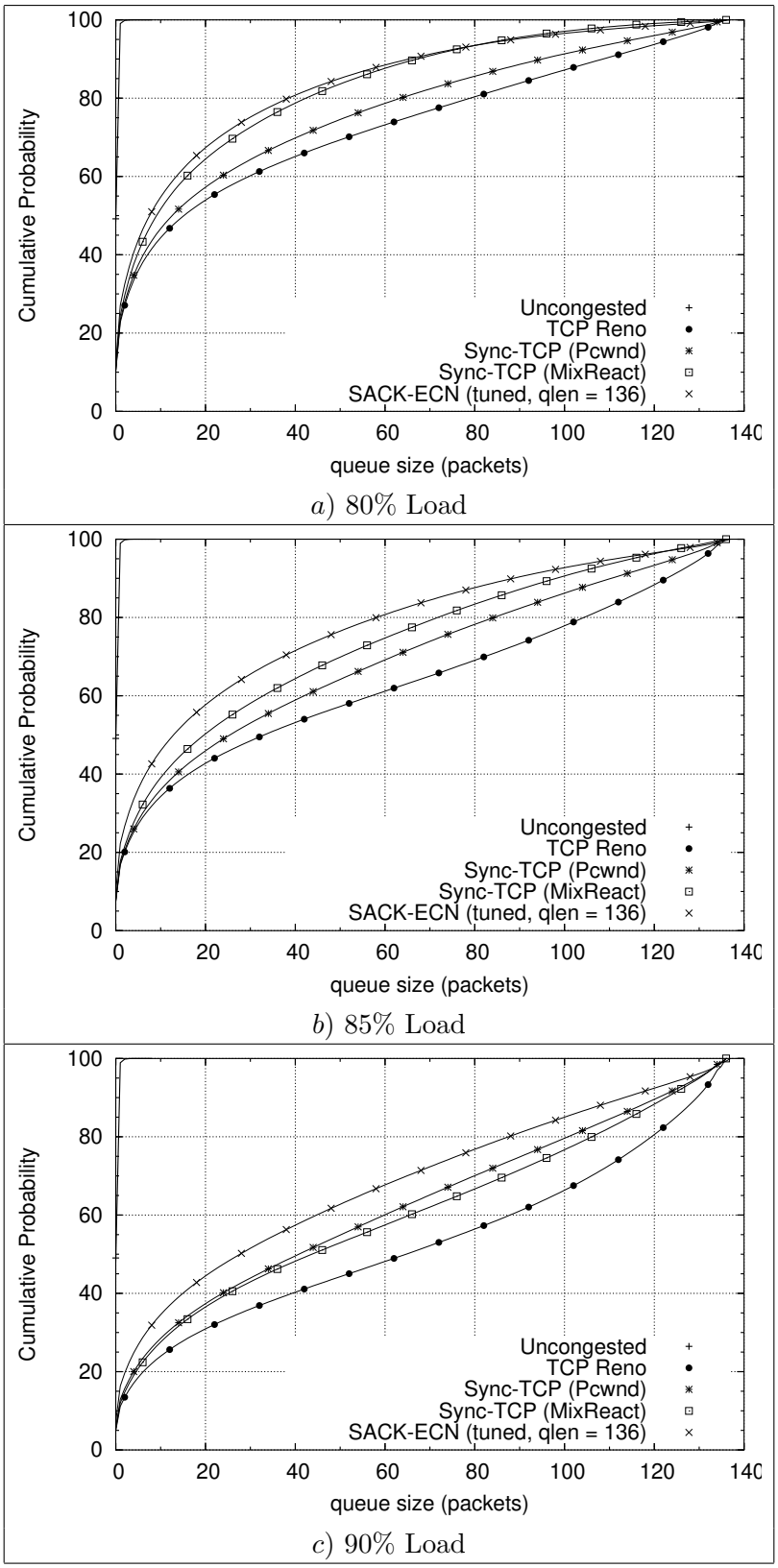


Figure 4.24: Queue Size CDFs, Single Bottleneck, Medium Congestion

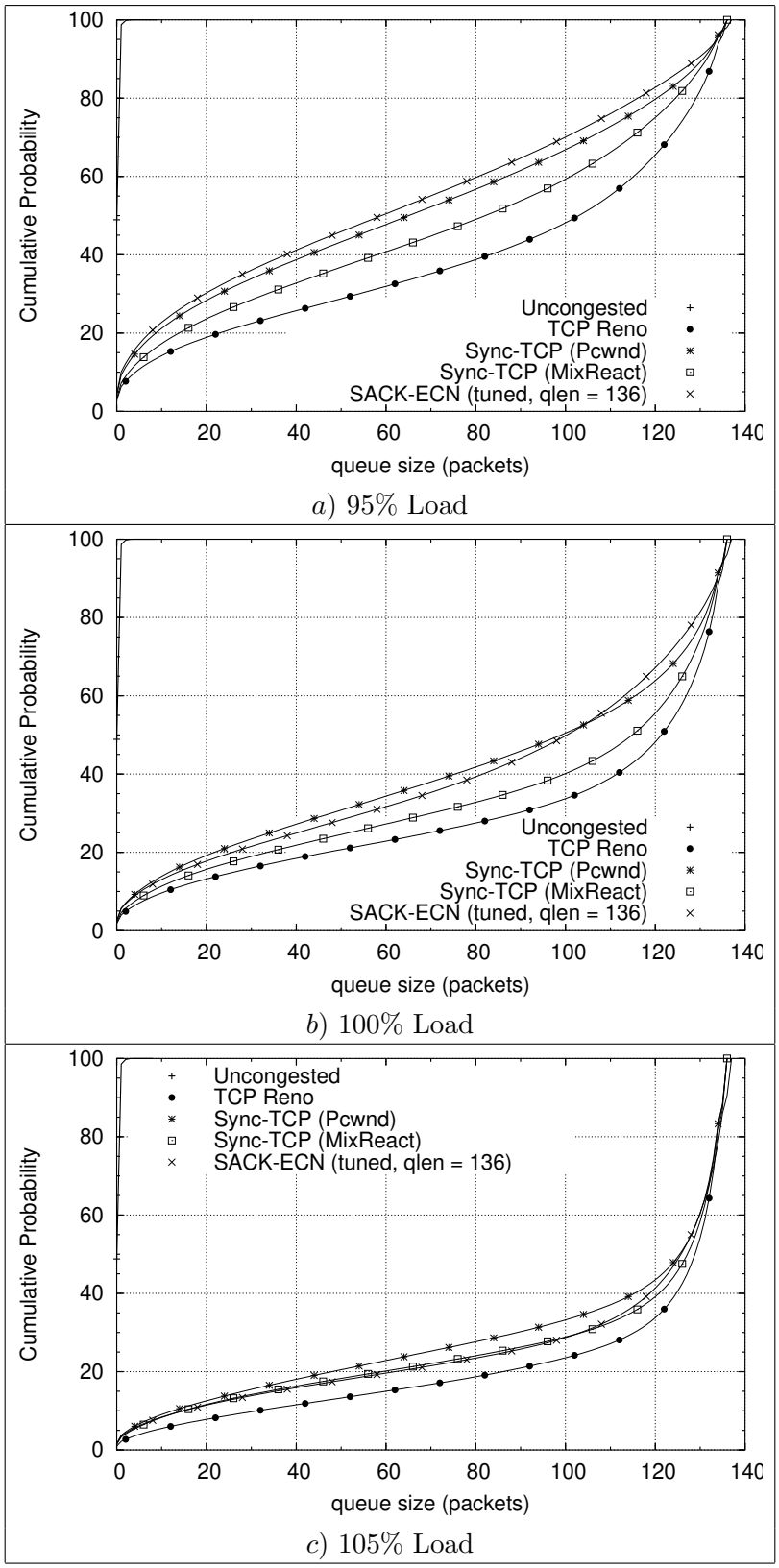


Figure 4.25: Queue Size CDFs, Single Bottleneck, Heavy Congestion

Completed Response Size CCDFs

Figures 4.26-4.28 show the CCDFs of the size of HTTP responses that completed. These CCDFs may differ by protocol or load level because the transfers of the largest responses (at the tail of the uncongested case) have started, but were not able to complete (due to congestion, protocol effects, or having a large RTT assigned) before 250,000 pairs completed. These graphs show that 90% of the responses were only slightly larger than 10 KB and less than 1% of the responses (about 1500 responses) were larger than 100 KB. The largest response to complete in any experiment was 49 MB. As the load level increases, the size of the largest completed response decreases. This effect can be attributed to the increased level of congestion and packet loss. These graphs also confirm that the size of the HTTP responses is heavy-tailed (on a log-log scale, a linear CCDF line implies that the distribution is heavy-tailed). There are a large number of small responses and a few very large responses.

From 50% to 85% load, all of the protocols except SACK-ECN have the same response sizes complete as in the uncongested case. As seen in Figure 4.27, at 90% load, using Sync-TCP(MixReact) allows a larger response to complete than with any of the other protocols. After 90% load (Figure 4.28, no protocol completes the largest responses that are completed when the network is uncongested.

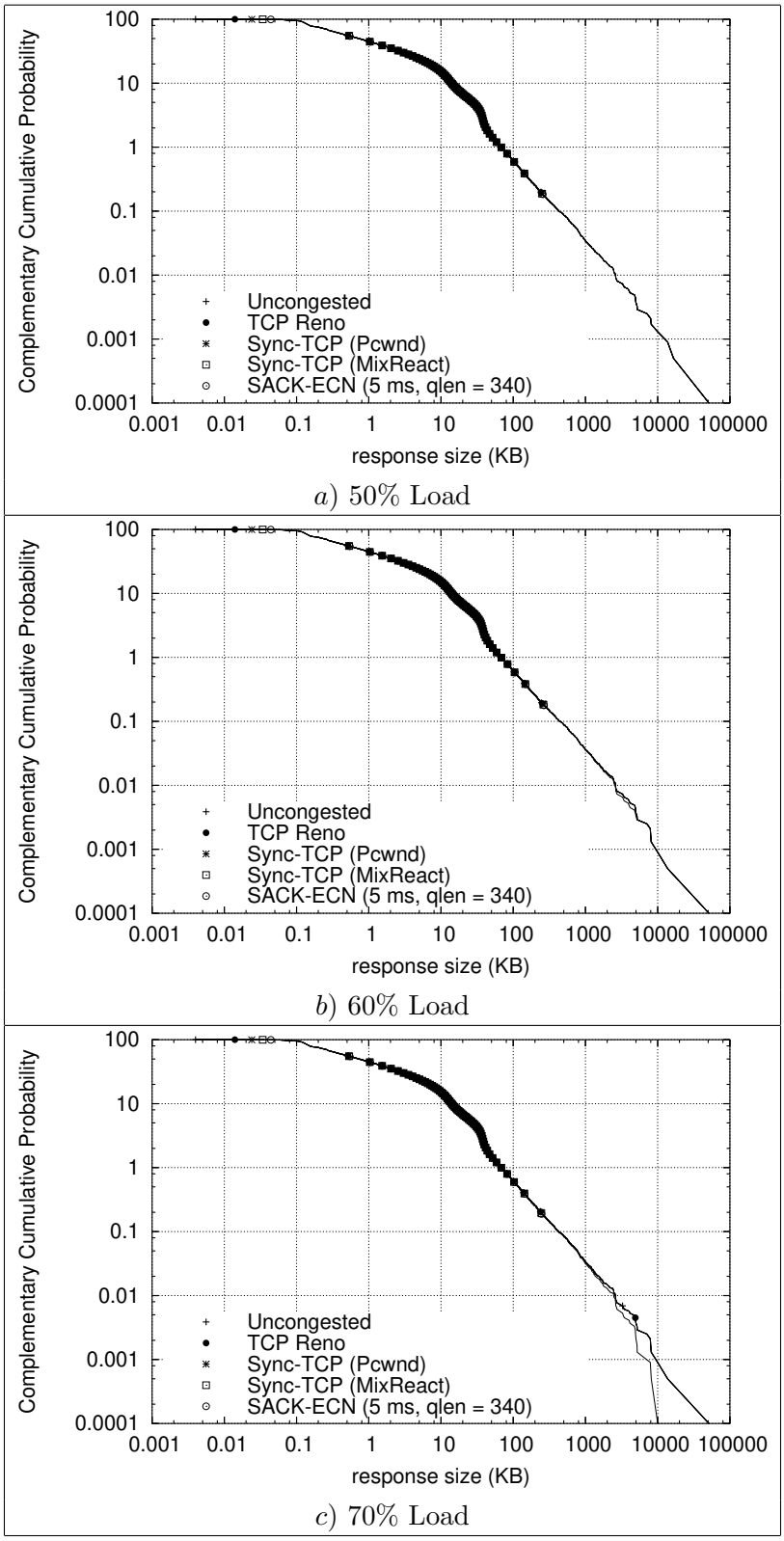


Figure 4.26: Response Size CCDFs, Single Bottleneck, Light Congestion

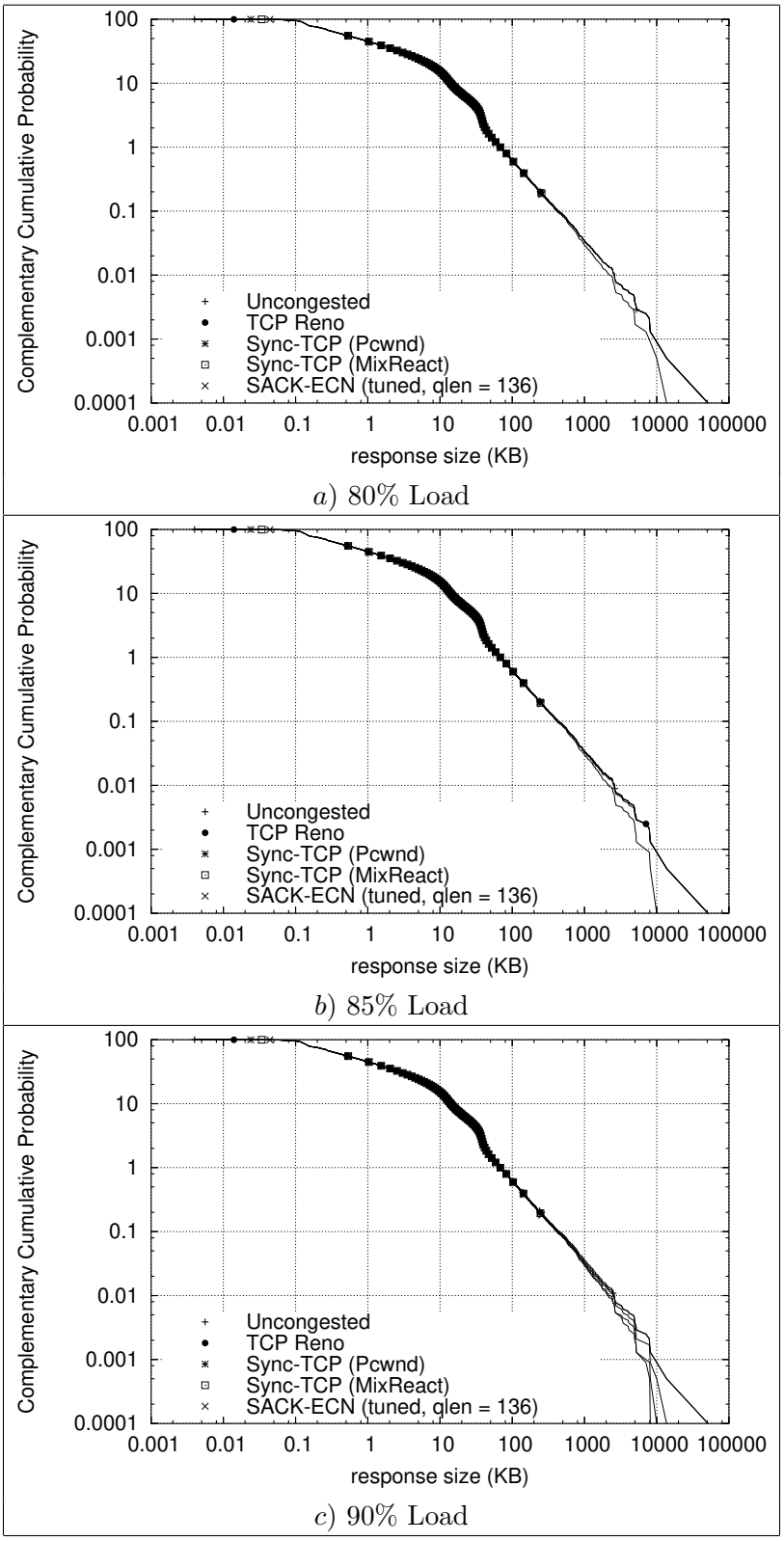


Figure 4.27: Response Size CCDFs, Single Bottleneck, Medium Congestion

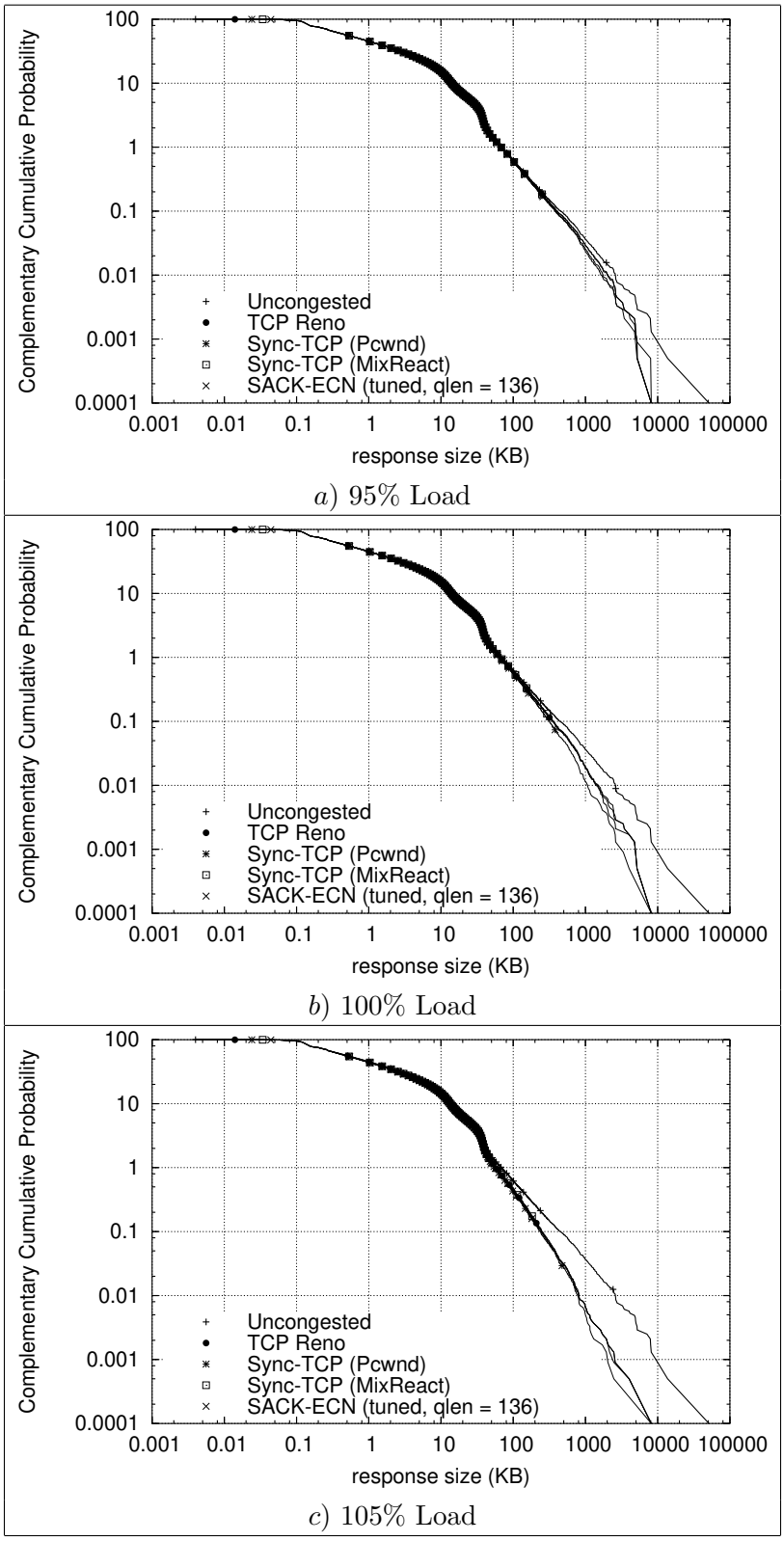


Figure 4.28: Response Size CCDFs, Single Bottleneck, Heavy Congestion

Conditional Response Time CDFs and CCDFs

The graphs presented next are HTTP response time CDFs and CCDFs conditional on the size of the completed response. Figures 4.29-4.31 show the HTTP response time CDFs for responses no larger than 25560 bytes (25 KB). These are responses that would have taken no more than 18 segments to complete. A connection sending this size response would likely not have used Sync-TCP(MixReact) early congestion detection and reaction. Not surprisingly, these graphs look much like the response time CDFs for all responses. These short responses are the most likely to complete in under 1500 ms and be displayed on the response time CDF graphs in Figures 4.17-4.19.

Figures 4.32-4.34 show the response time CCDFs for these short responses. Since the response time CCDFs show the longest response times, these graphs are much different than those for all responses (Figures 4.20-4.22). The longest response times for these shortest flows are less than 100 seconds for loads up to 85%. For all loads, the tails of these distributions are shorter than those that include all flows. Starting at 70% load, there is a visible spike around 6 seconds (corresponding to the SYN timeout) for Sync-TCP(Pcwnd) and TCP Reno. The spike also appears for Sync-TCP(MixReact) starting at 85% load. The presence of these spikes is not surprising since they are due to drops of SYN or SYN/ACK packets. At loads of 90-105%, where Sync-TCP(MixReact) loses its advantage over Sync-TCP(Pcwnd) in response times for all responses (Figures 4.21c and 4.22), Sync-TCP(MixReact) does slightly worse than Sync-TCP(Pcwnd).

At all load levels, the Sync-TCP protocols provide the best response time performance for these smallest responses. There is a performance cross-over with Sync-TCP(MixReact) and Sync-TCP(Pcwnd) around 90% load. These performance improvements for the Sync-TCP protocols are due to both shorter queue sizes and lower loss rates (especially for the comparison against tuned SACK-ECN where the queue sizes are smaller).

Figures 4.35-4.37 show the response time CDFs out to 10 seconds for responses that were larger than 25560 bytes (25 KB). Connections transferring these responses would have likely used Sync-TCP(MixReact) early congestion detection and reaction. The Sync-TCP protocols perform well for these larger responses compared to TCP Reno and SACK-ECN.

Figures 4.38-4.40 show the response time CCDFs for responses that were larger than 25 KB.

The Sync-TCP protocols perform well for these larger responses compared to TCP Reno and SACK-ECN. At 50% load (Figure 4.38a), Sync-TCP(MixReact) shows some bias against longer responses, but this is due to the fact that no packets were dropped, as has been mentioned previously. Otherwise, Sync-TCP(MixReact) provides no worse performance than TCP Reno for these large responses.

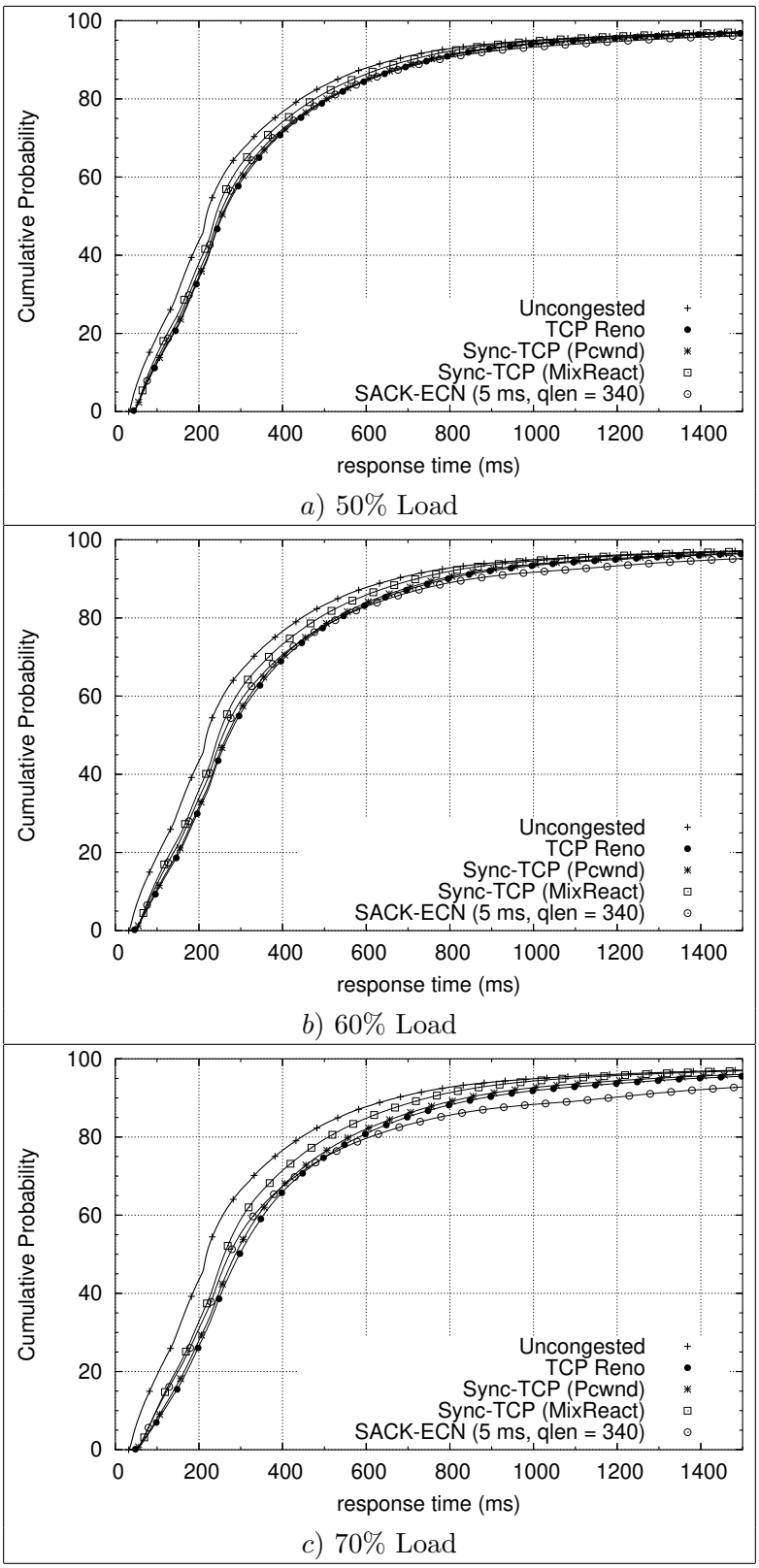


Figure 4.29: Response Time CDFs for Responses Under 25 KB, Light Congestion

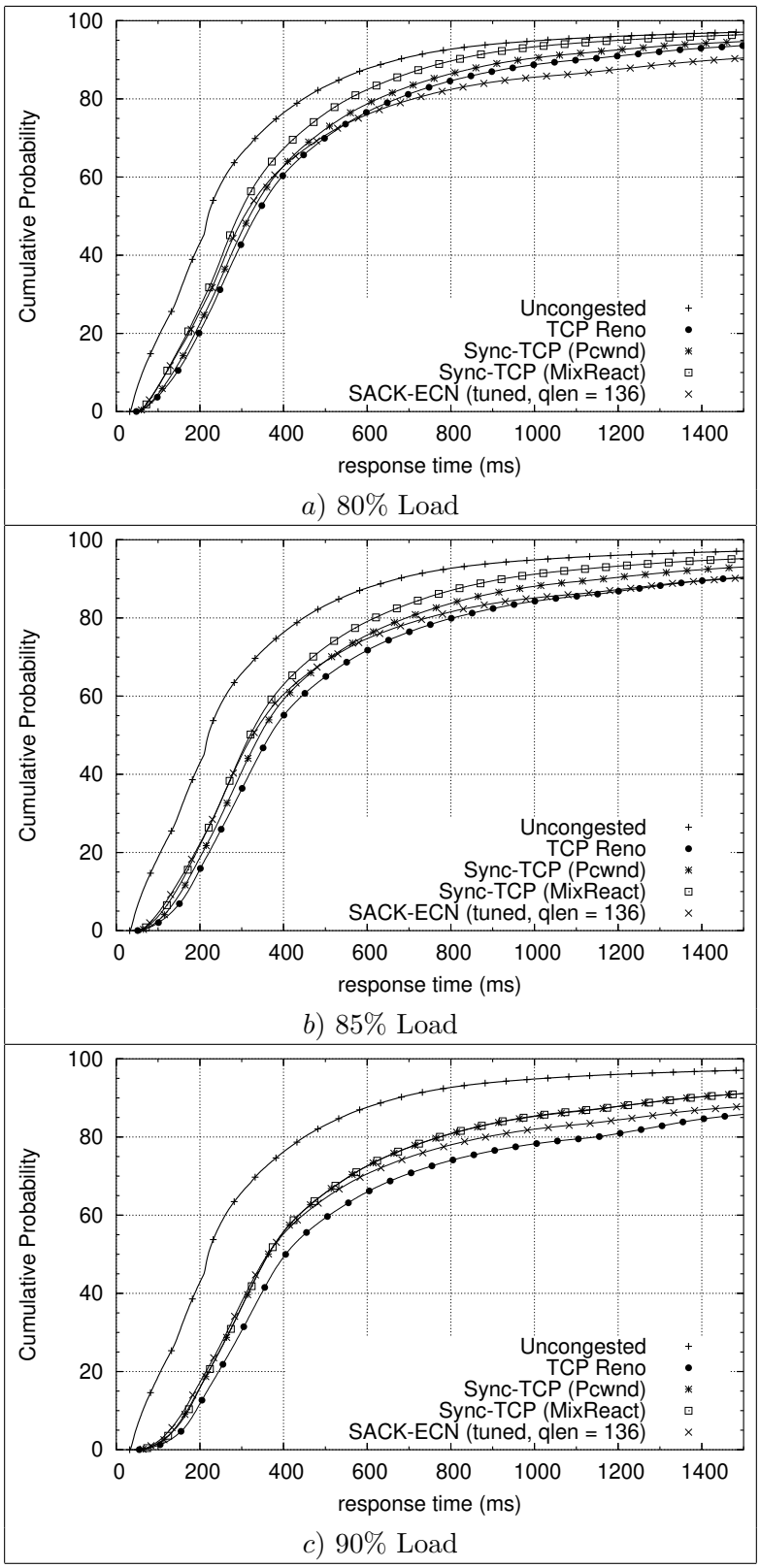


Figure 4.30: Response Time CDFs for Responses Under 25 KB, Medium Congestion

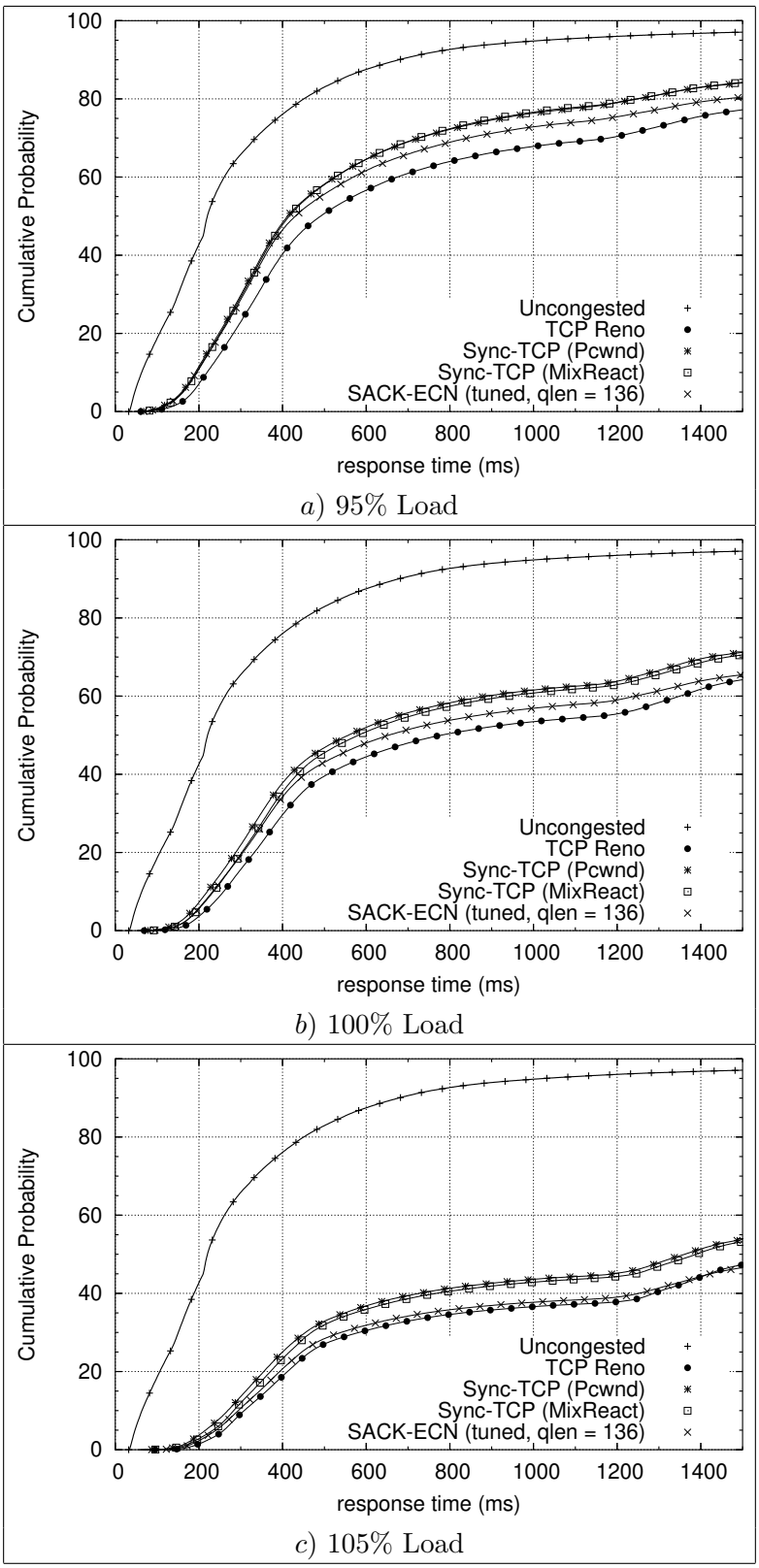


Figure 4.31: Response Time CDFs for Responses Under 25 KB, Heavy Congestion

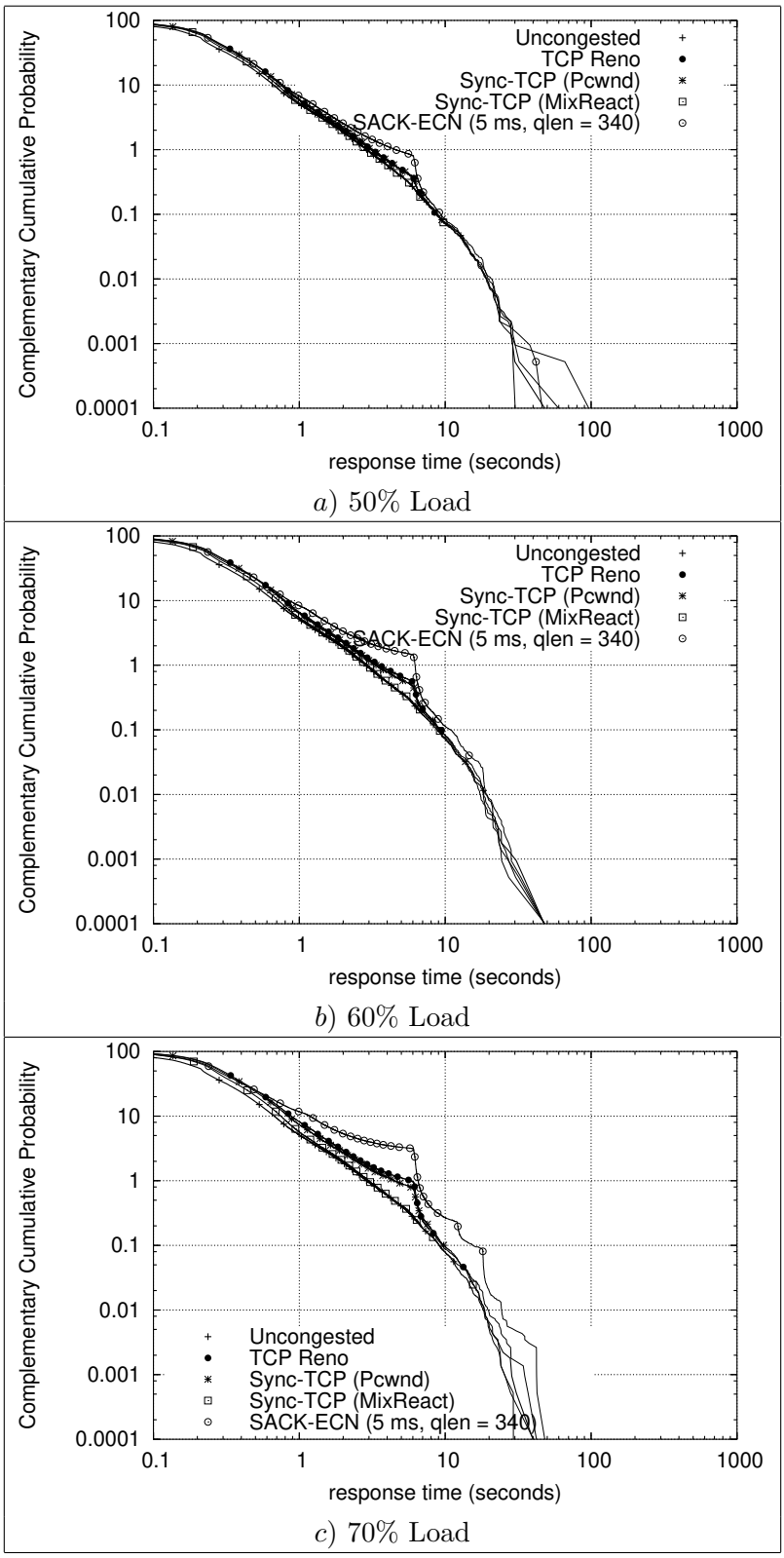


Figure 4.32: Response Time CCDFs for Responses Under 25 KB, Light Congestion

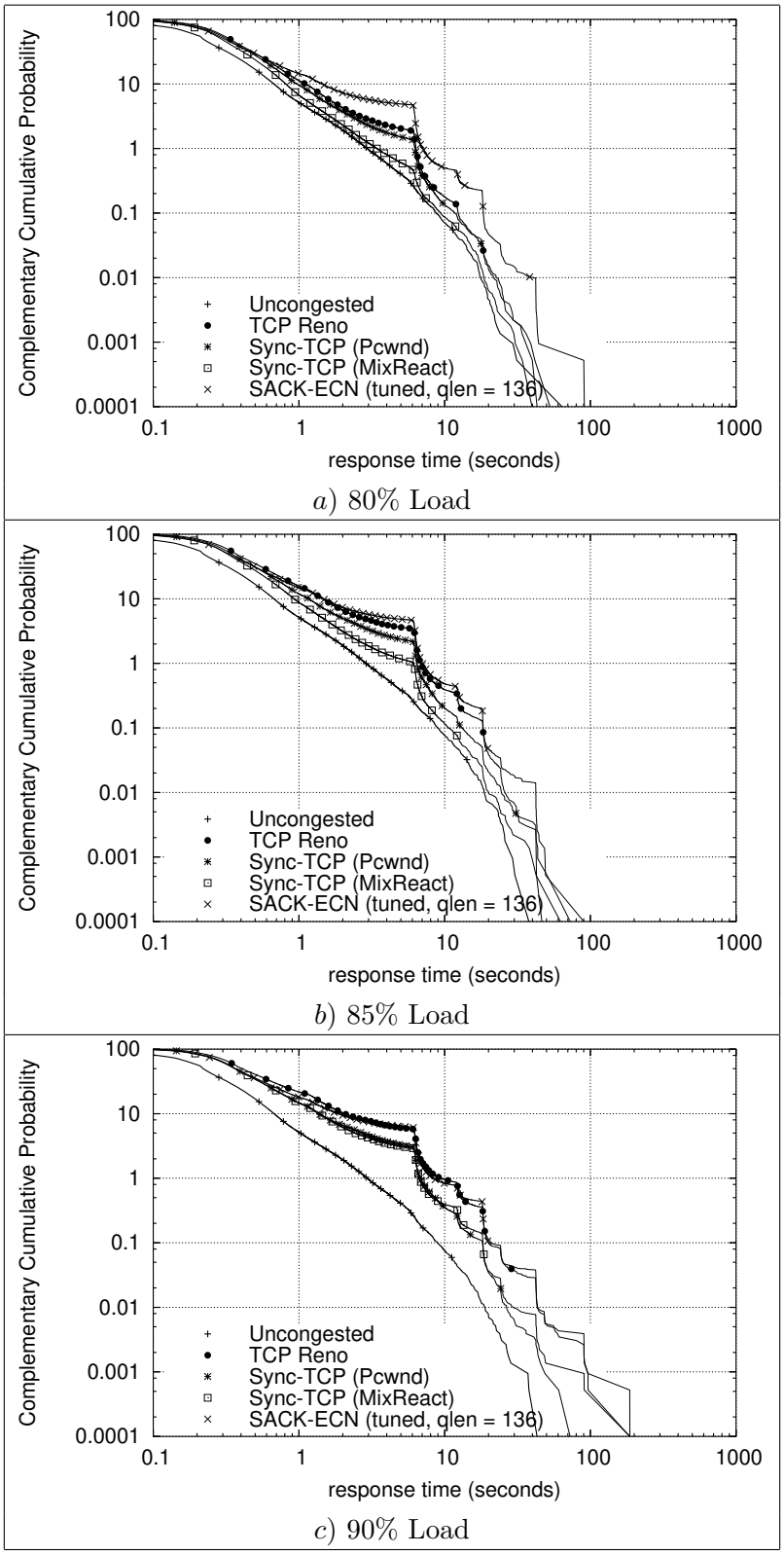


Figure 4.33: Response Time CCDFs for Responses Under 25 KB, Medium Congestion

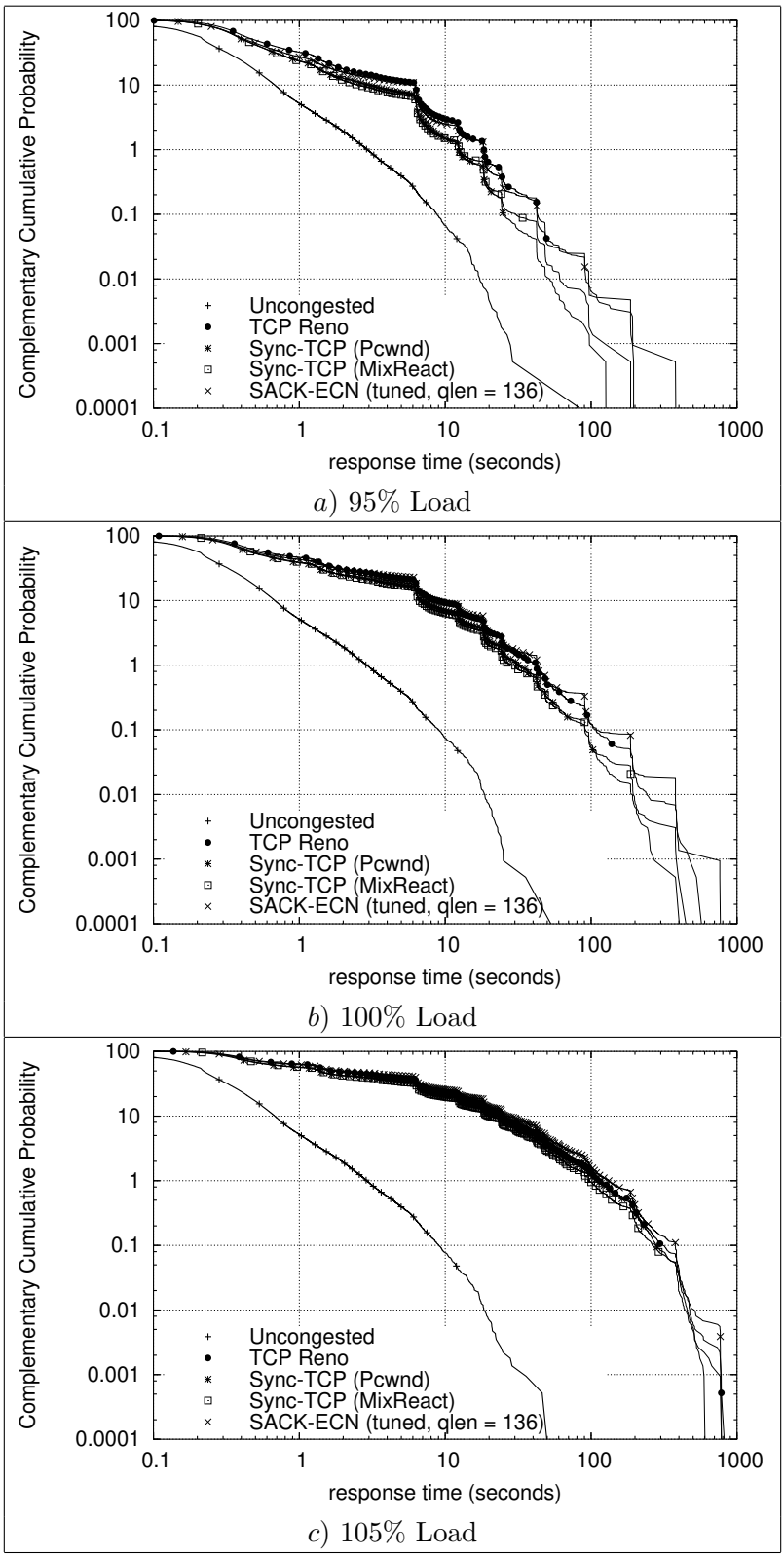


Figure 4.34: Response Time CCDFs for Responses Under 25 KB, Heavy Congestion

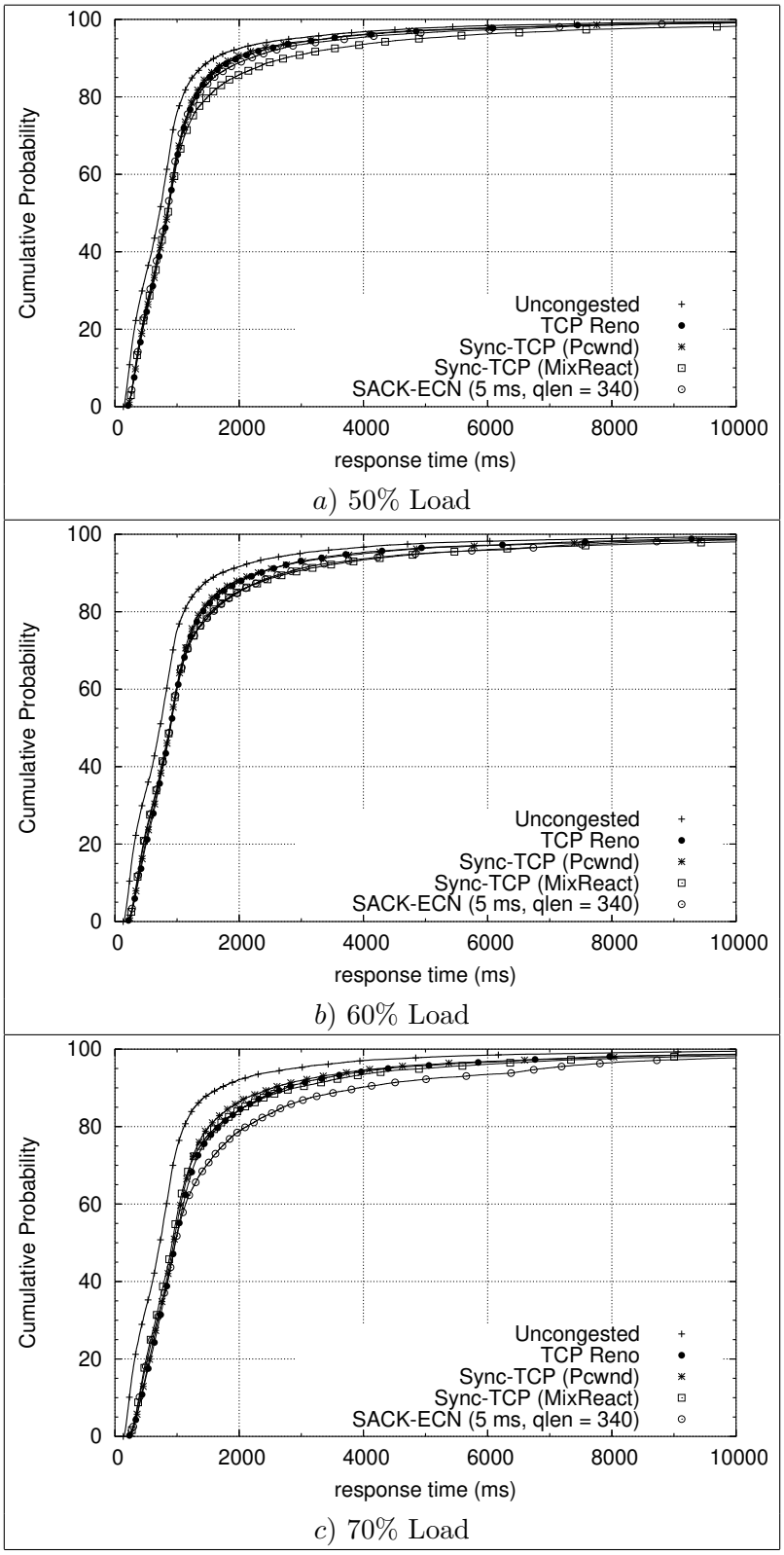


Figure 4.35: Response Time CDFs for Responses Over 25 KB, Light Congestion

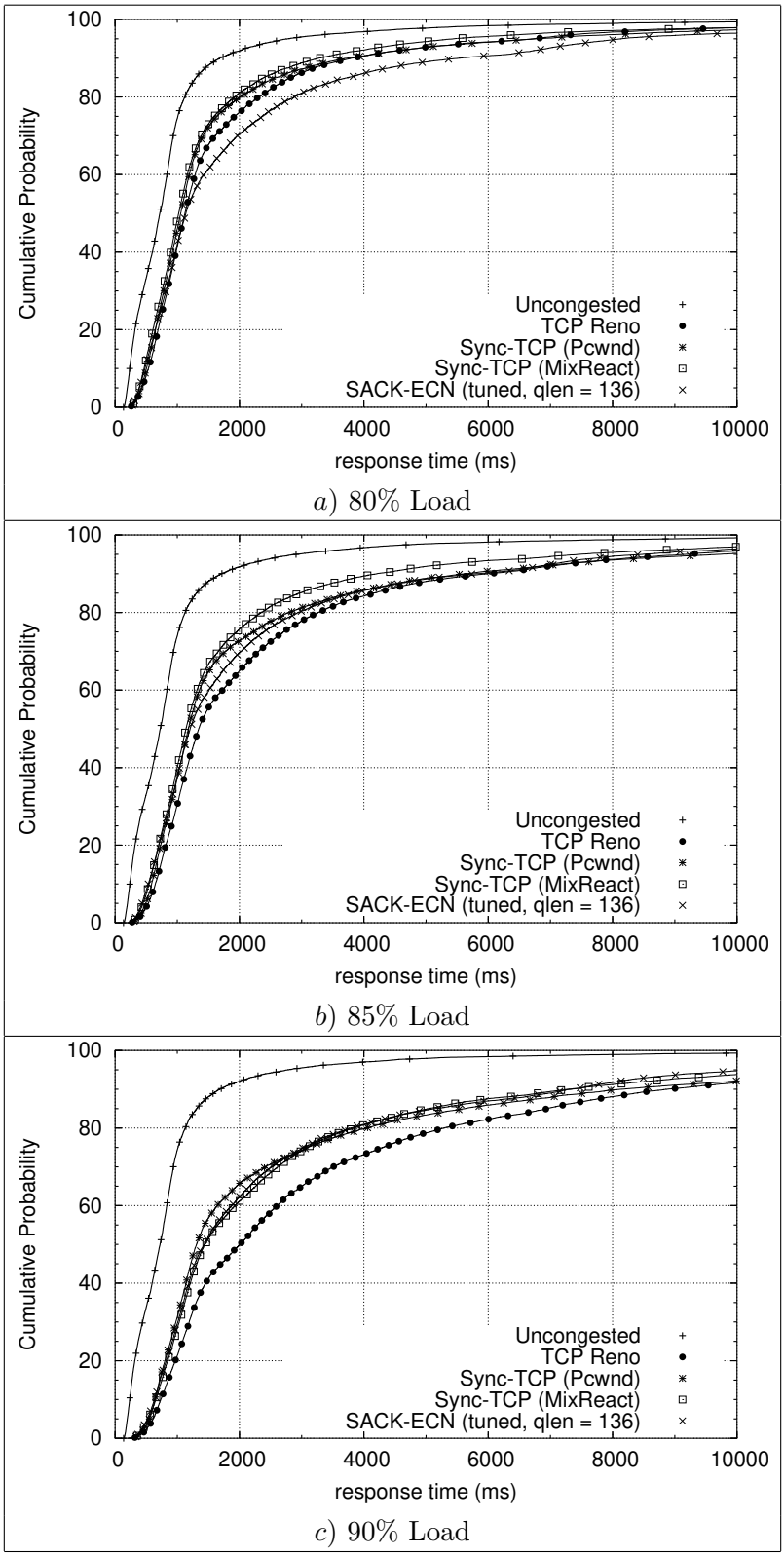


Figure 4.36: Response Time CDFs for Responses Over 25 KB, Medium Congestion

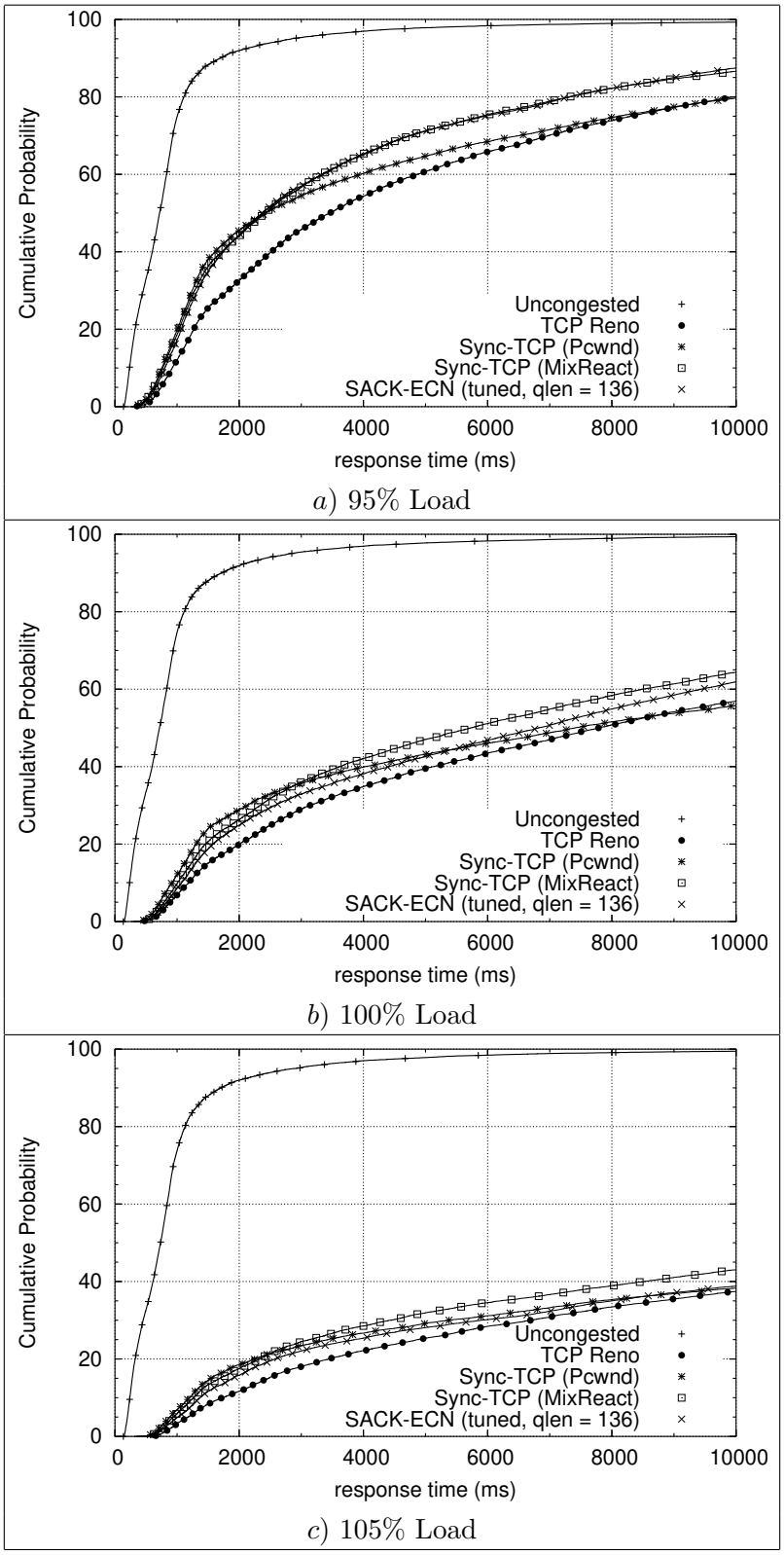


Figure 4.37: Response Time CDFs for Responses Over 25 KB, Heavy Congestion

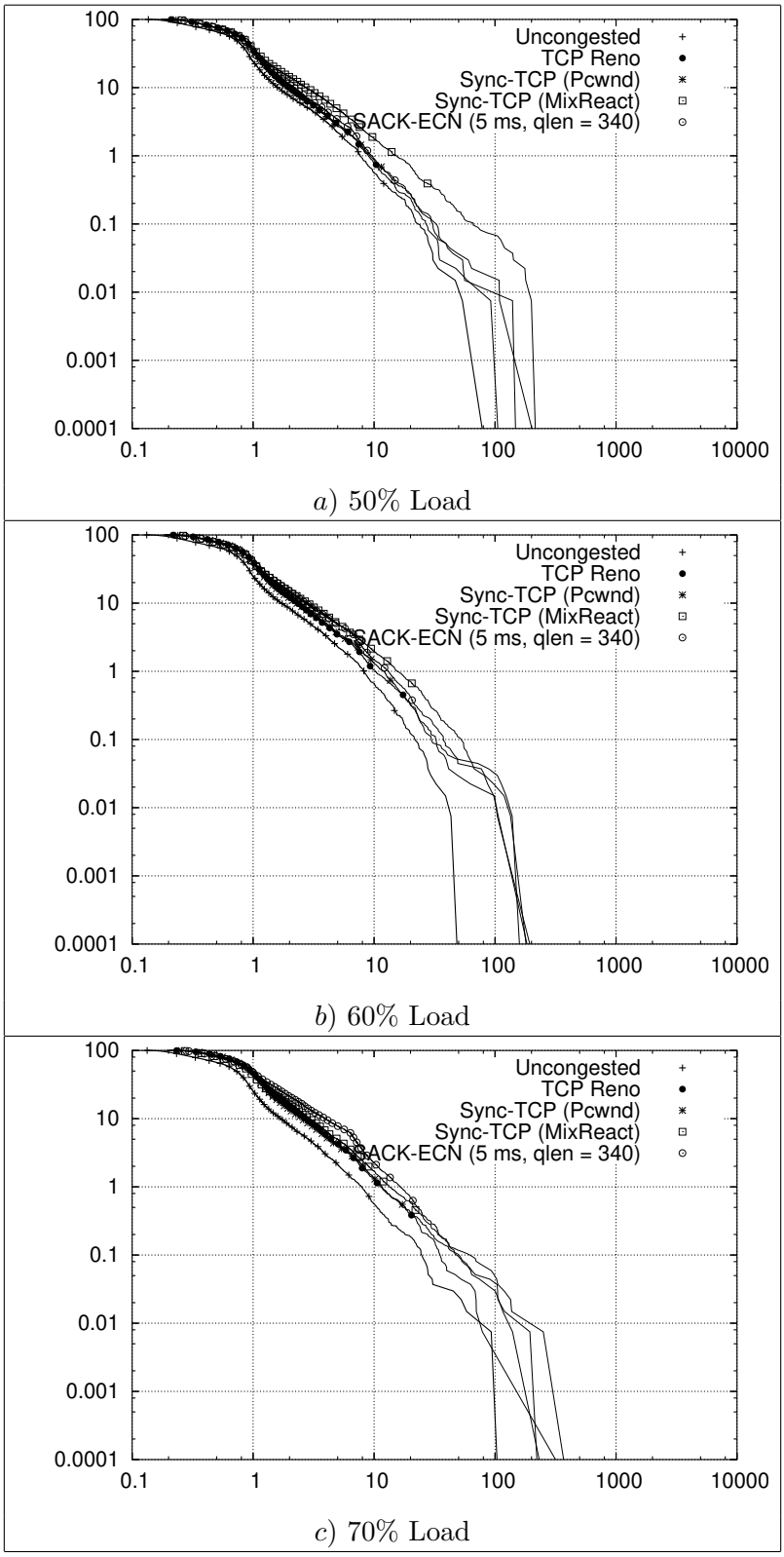


Figure 4.38: Response Time CCDFs for Responses Over 25 KB, Light Congestion

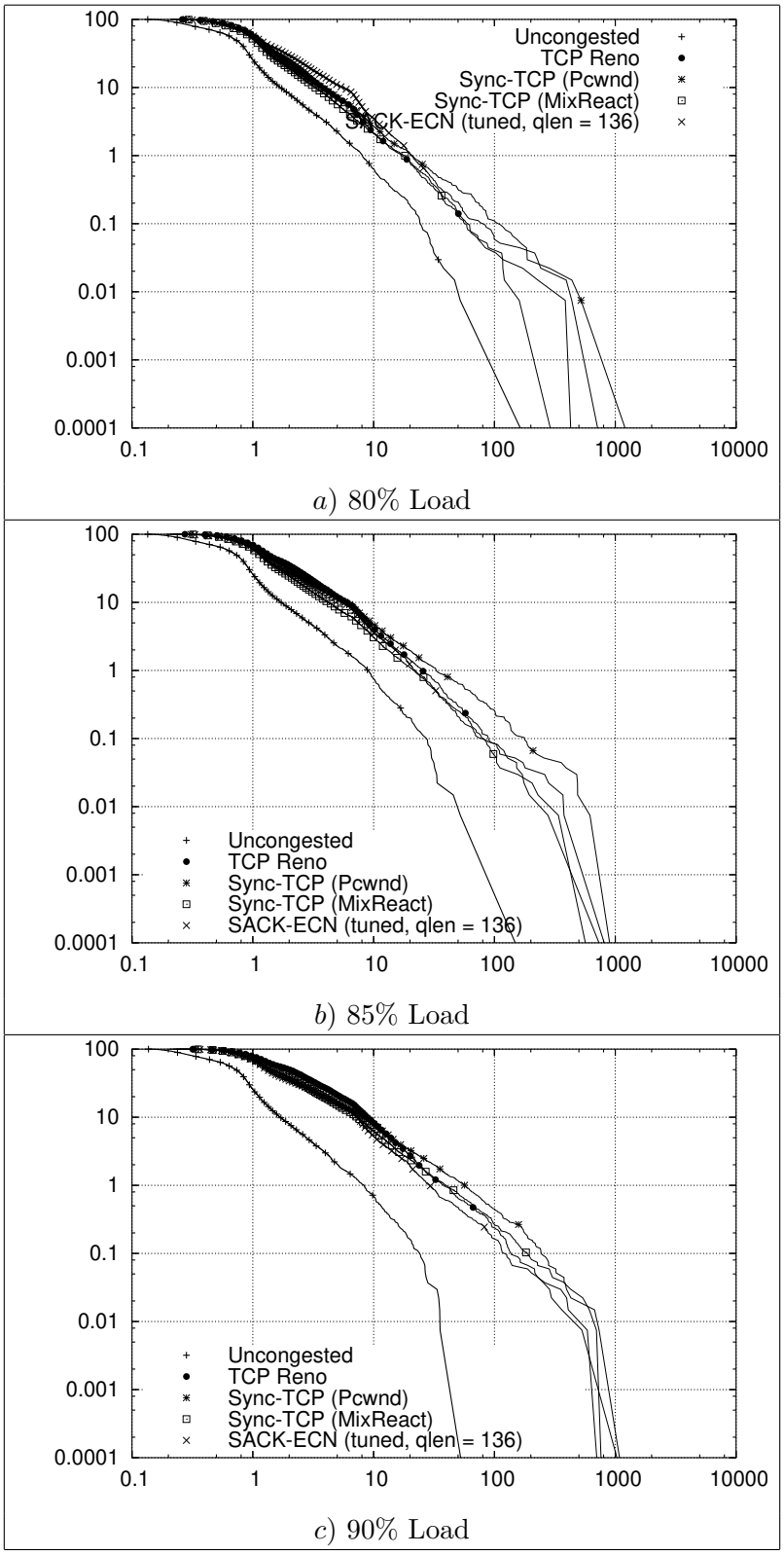


Figure 4.39: Response Time CCDFs for Responses Over 25 KB, Medium Congestion

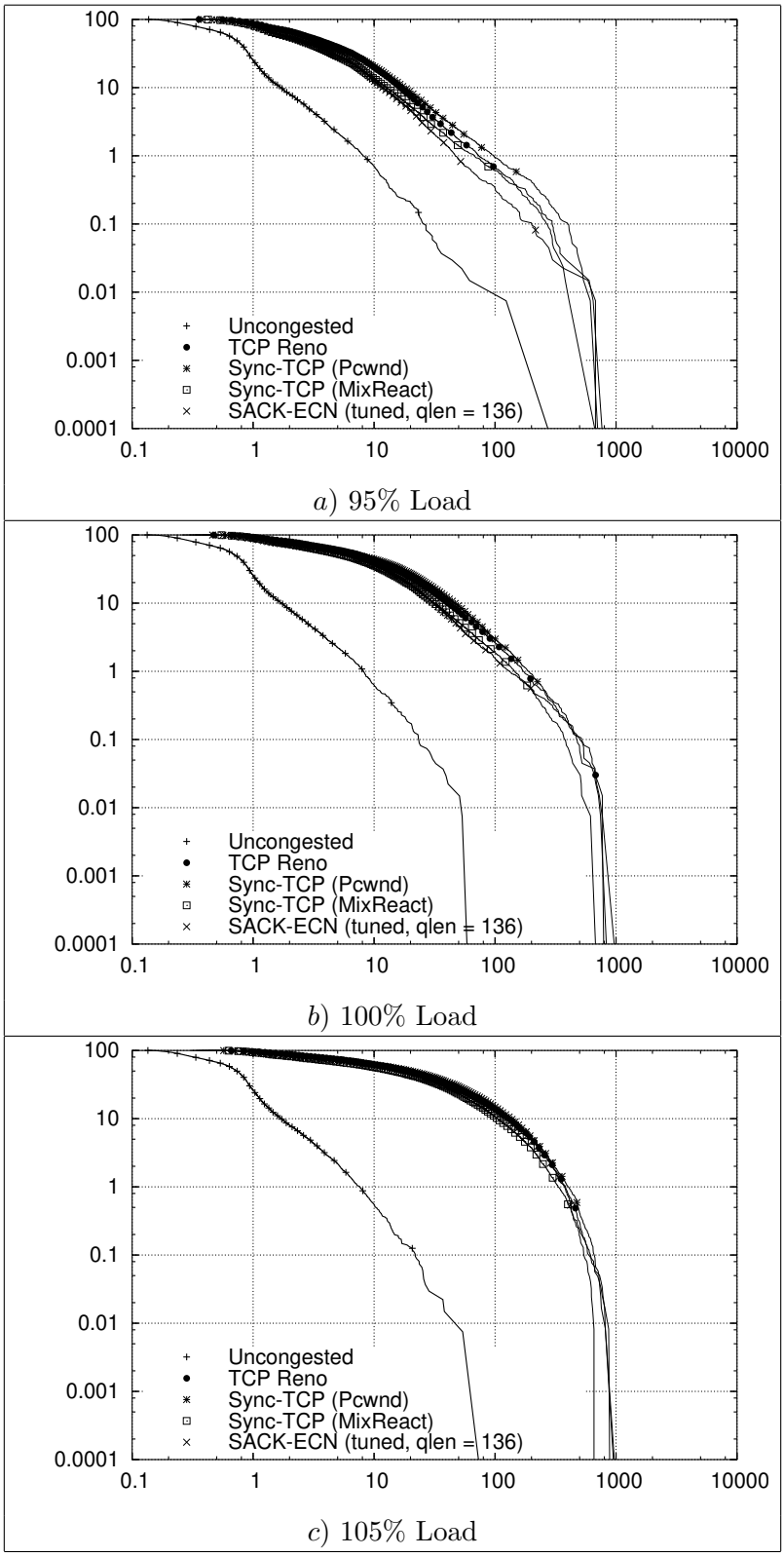


Figure 4.40: Response Time CCDFs for Responses Over 25 KB, Heavy Congestion

RTT CCDFs

Figures 4.41-4.43 show the CCDFs of the base RTTs assigned by DelayBox to the flows that had completed responses. There is no difference between the protocols until 105% load (Figure 4.43c). Flows with very long base RTTs might not be able to complete before 250,000 other request-response pairs in the presence of heavy congestion. These graphs show that 99% of the base RTTs are less than 1 second, but 1% of the 250,000 flows with completed request-response pairs (2500 flows) have base RTTs that are greater than 1 second. None of the flows with base RTTs greater than 1 second would be represented in the response time CDFs that only show response times less than 1500 ms.

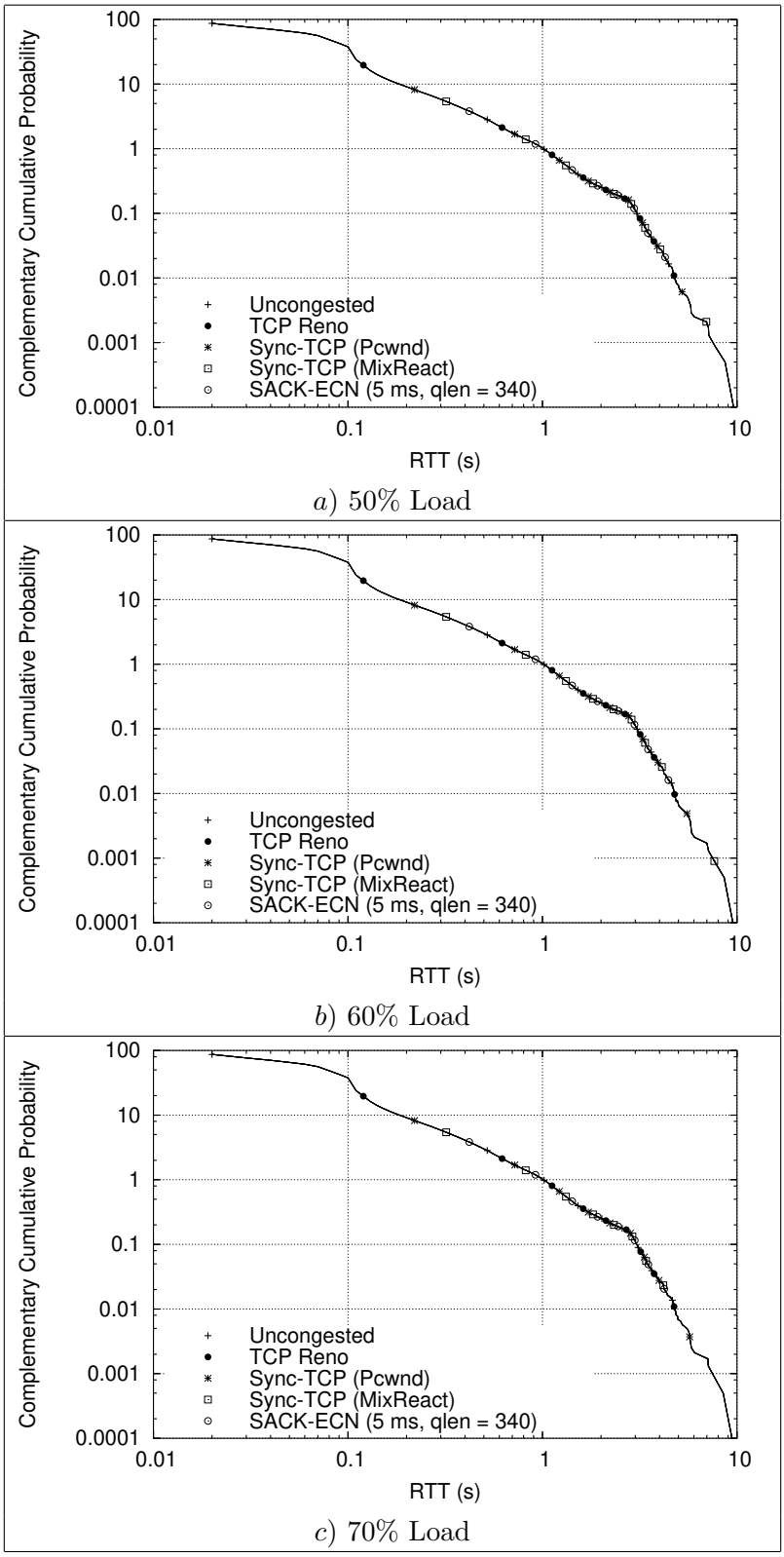


Figure 4.41: RTT CCDFs, Light Congestion

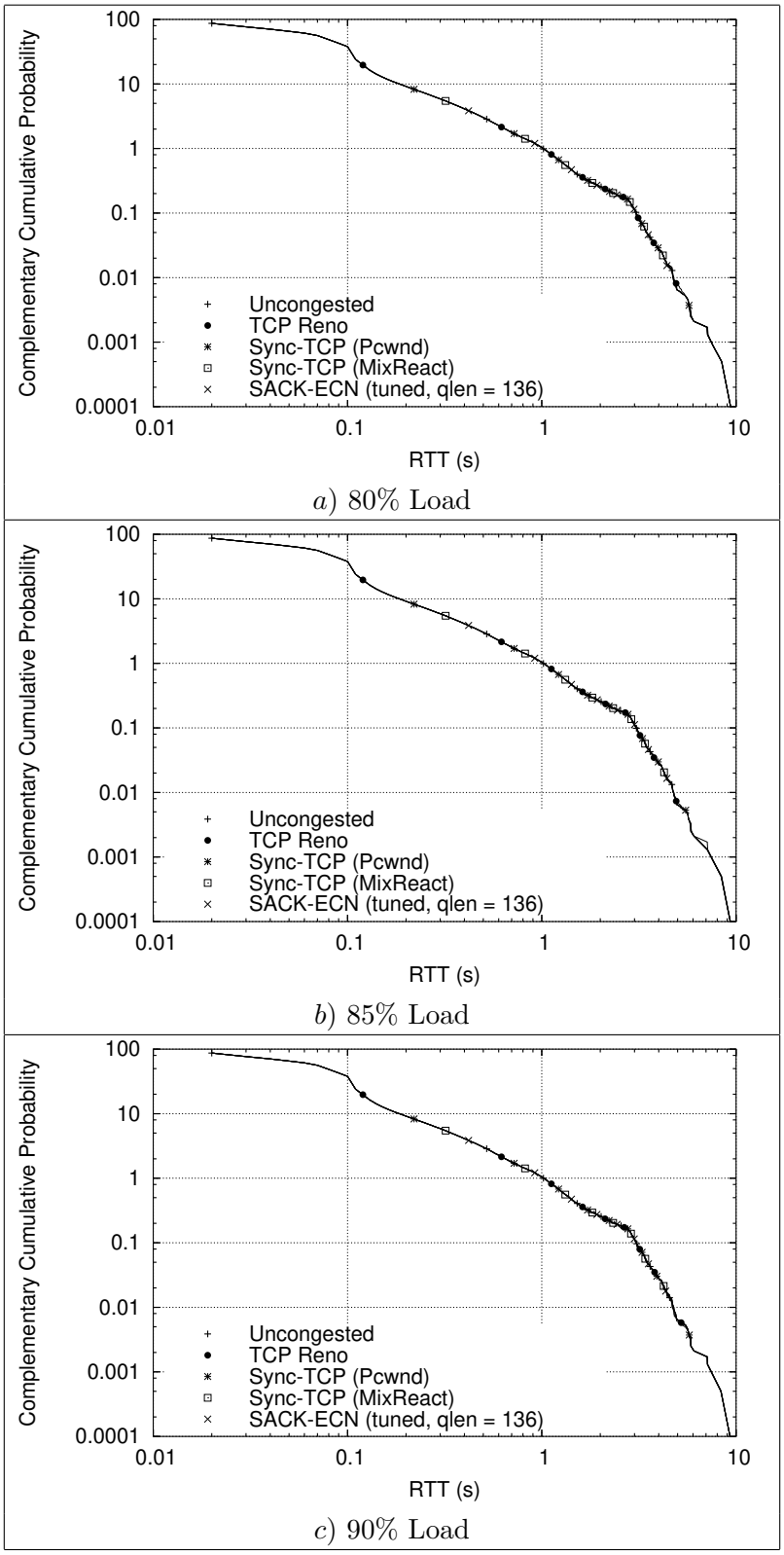


Figure 4.42: RTT CCDFs, Medium Congestion

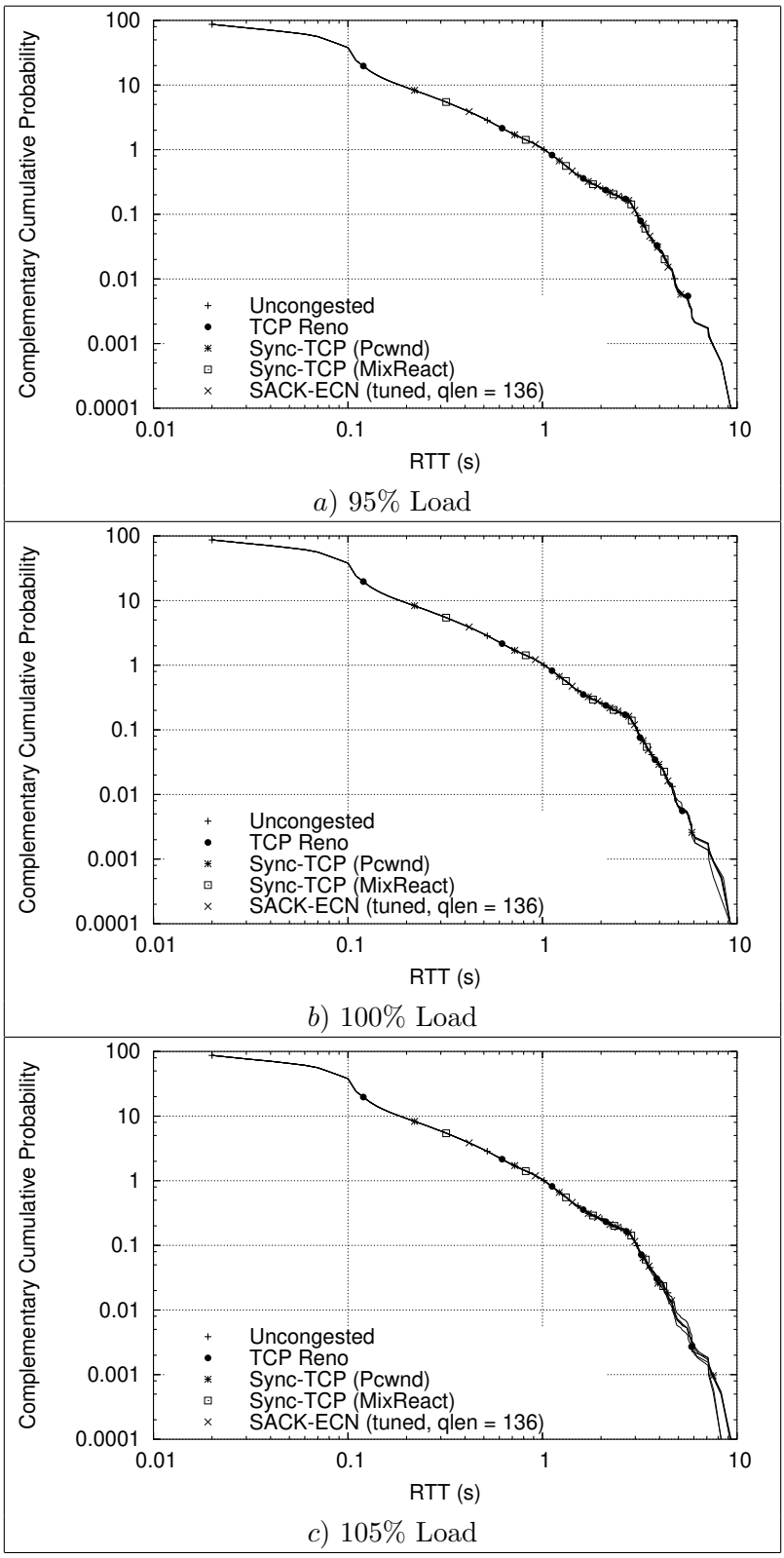


Figure 4.43: RTT CCDFs, Heavy Congestion

Conditional RTT CCDFs

Since response times are partly based on the size of the response and the RTT assigned, comparing the conditional response time CCDFs to the conditional RTT CCDFs is useful for evaluation. These graphs are presented to show that, of the responses that completed, the longest RTTs were generally assigned to the shortest responses. This is not by design, but due to the fact that most of the responses are short.

Figures 4.44-4.46 show the CCDFs of RTTs assigned to HTTP flows that had responses under 25 KB and completed within the first 250,000 flows. Since response time is a function of response size, RTT, level of congestion, and protocol, the shortest responses with large RTTs have a better chance of completing before the experiment ends than larger responses with large RTTs. Figures 4.47-4.49 show the RTT CCDFs for responses that were larger than 25 KB.

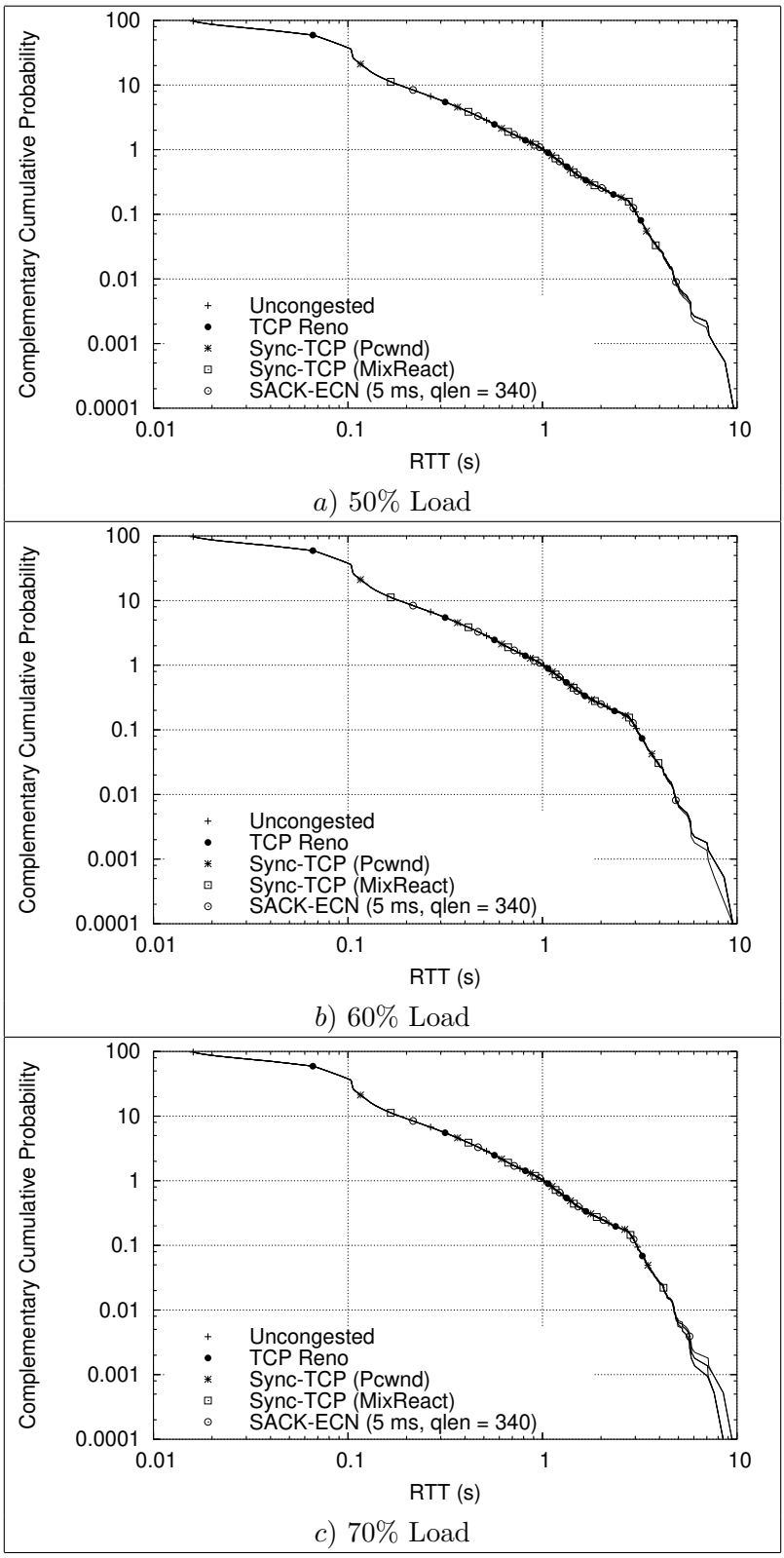


Figure 4.44: RTT CCDFs for Responses Under 25 KB, Light Congestion

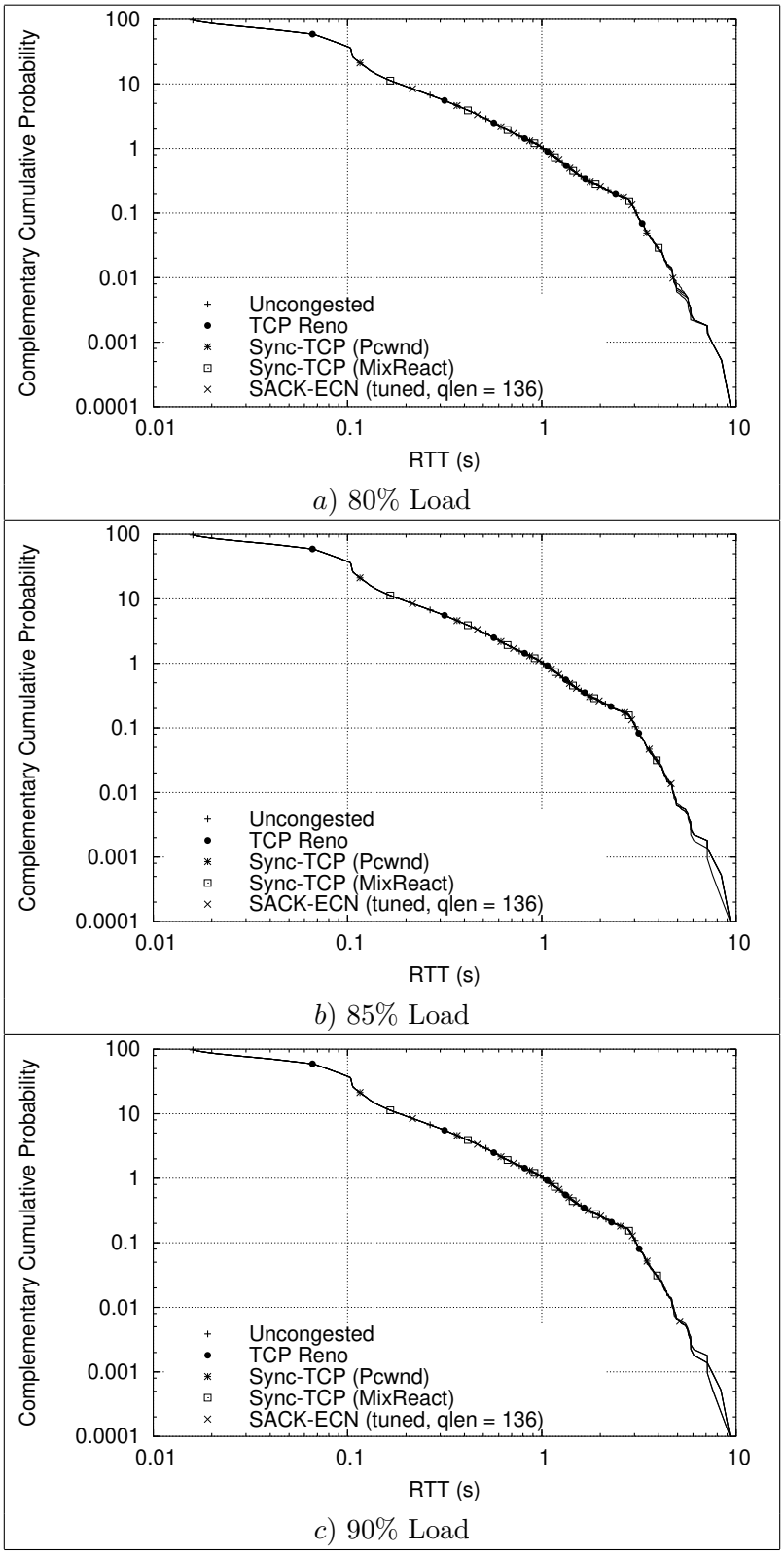


Figure 4.45: RTT CCDFs for Responses Under 25 KB, Medium Congestion

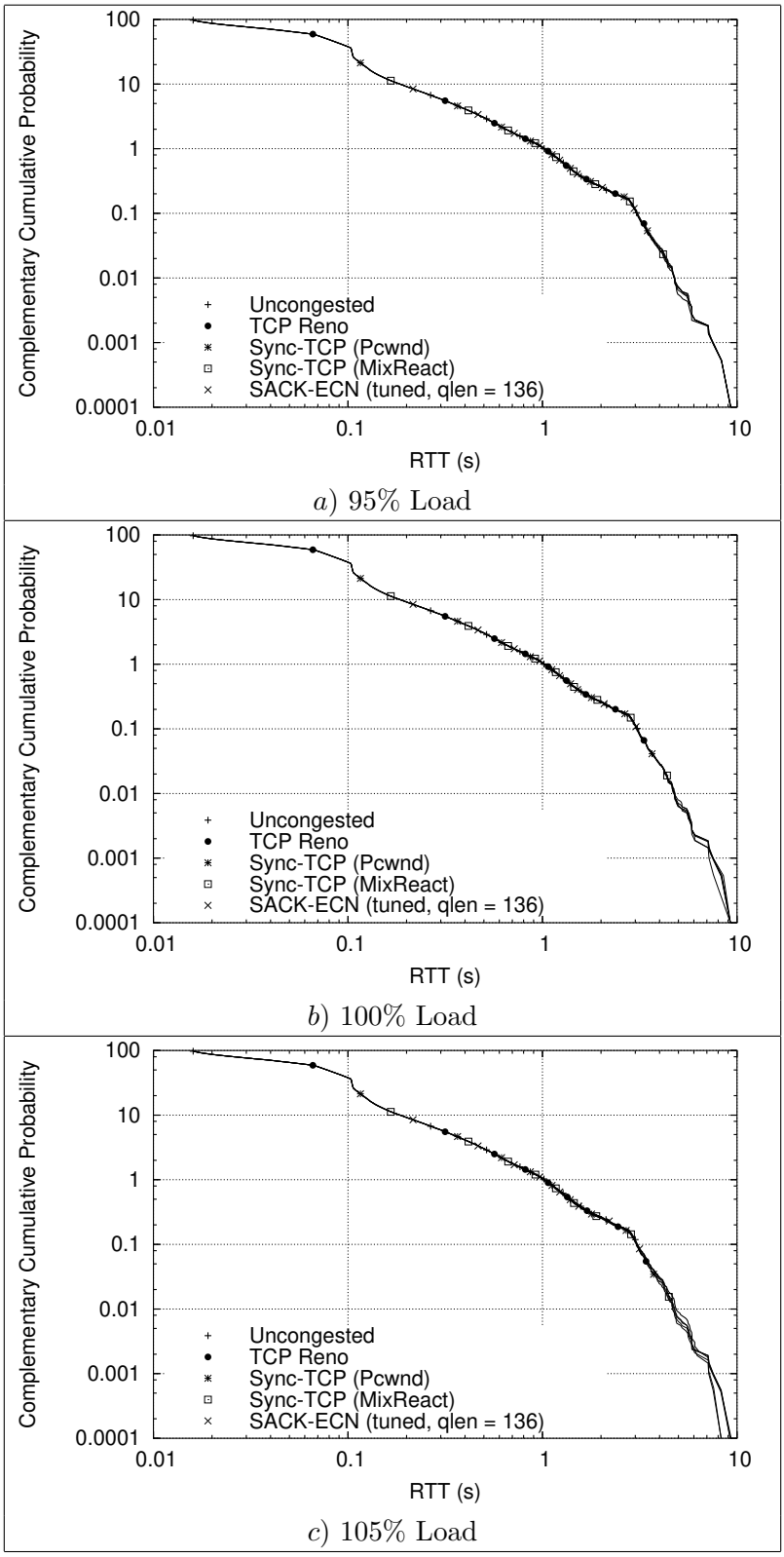


Figure 4.46: RTT CCDFs for Responses Under 25 KB, Heavy Congestion

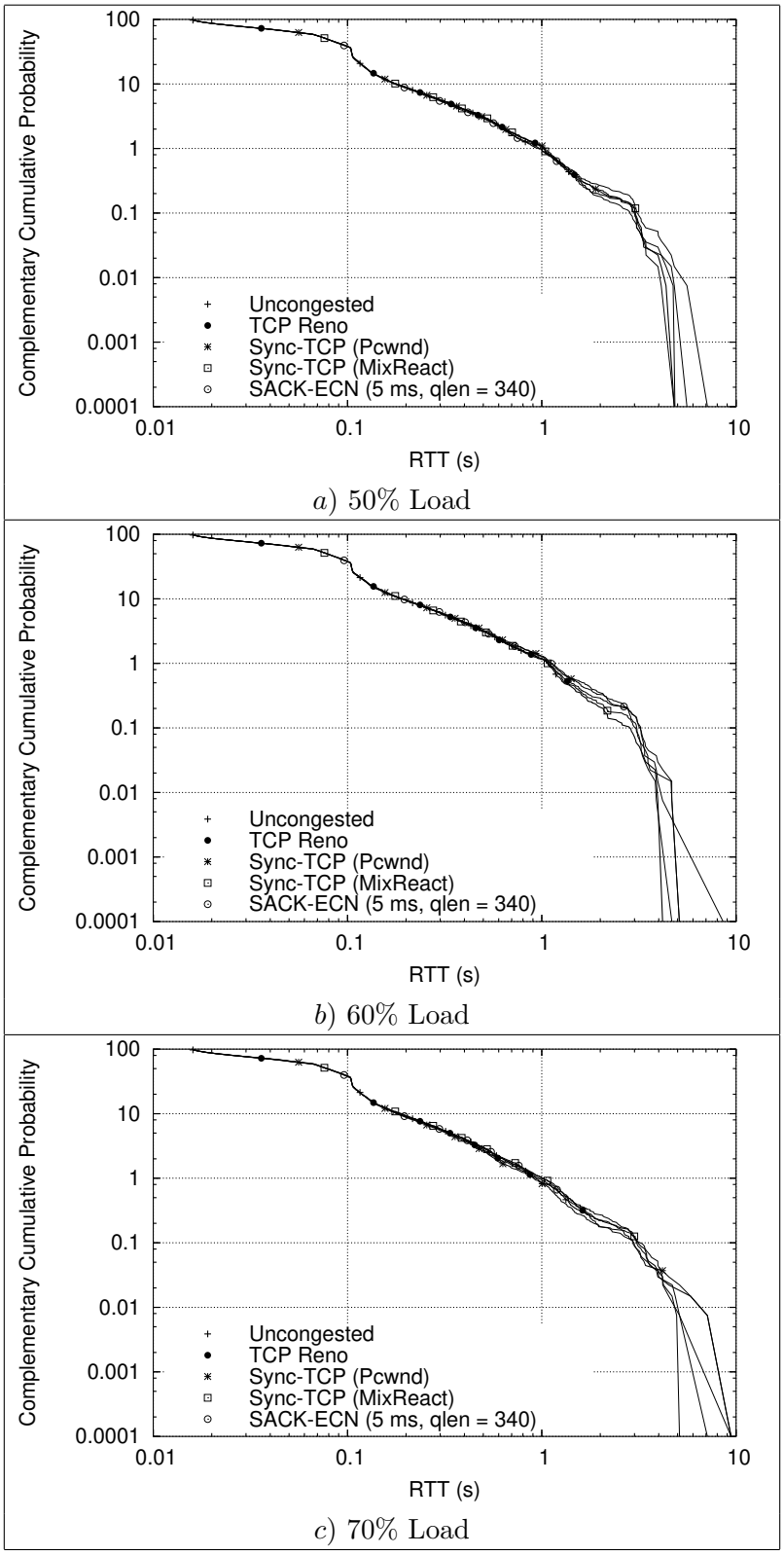


Figure 4.47: RTT CCDFs for Responses Over 25 KB, Light Congestion

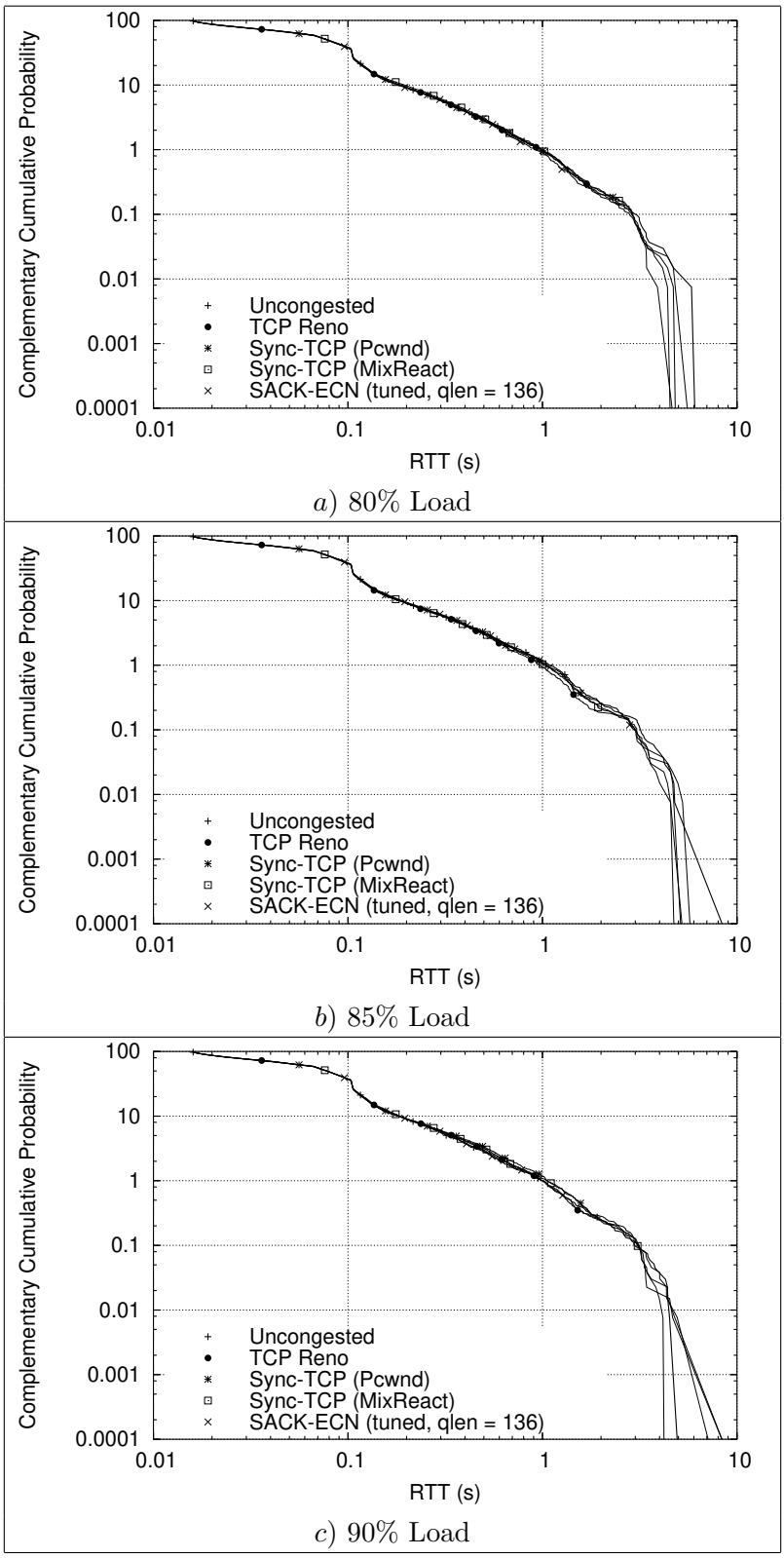


Figure 4.48: RTT CCDFs for Responses Over 25 KB, Medium Congestion

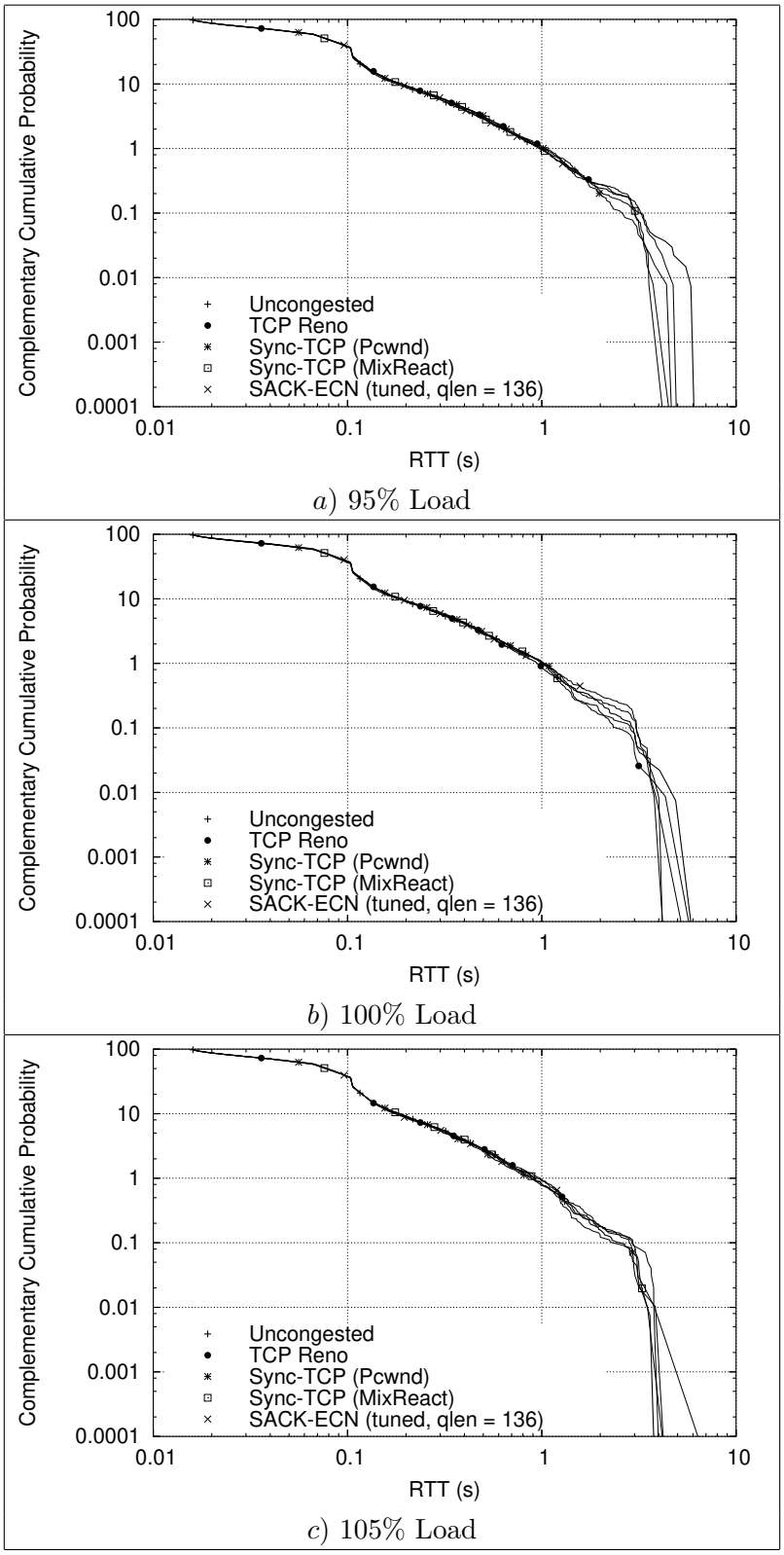


Figure 4.49: RTT CCDFs for Responses Over 25 KB, Heavy Congestion

Congestion Window Comparison

Figures 4.50 and 4.51 show the evolution of the congestion window (*cwnd*) over time, an estimate of the amount of unacknowledged data in the network, and the difference between *cwnd* and the amount of unacknowledged data for the flow transferring the largest completed response (49 MB) at 60% and 85% loads for TCP Reno, Sync-TCP(Pcwnd), and Sync-TCP(MixReact). The difference between *cwnd* and the amount of unacknowledged data in the network is the "headroom" of the connection. It represents the amount of data that could be sent by the connection (allowed by *cwnd*), but that something else is keeping the connection from sending (*e.g.*, fast recovery, timeout, end of data transfer). The TCP Reno values are plotted above the Sync-TCP(Pcwnd) values, which are plotted above the Sync-TCP(MixReact) values. The x-axis is simulation time (*i.e.*, the flows all started at approximately the same time in the simulation). Segment drops are indicated by a '+' near the top of the protocol's section of the *cwnd* graph. The flow's base RTT and number of drops are indicated in the key. These graphs show the entire duration of the three flows. In the 60% load case (Figure 4.50), the Sync-TCP(Pcwnd) flow completes first, and at 85% load, the Sync-TCP(MixReact) flow completes first. Note that these flows are not competing against each other, but are competing against similar levels of other traffic. All of the data presented in these graphs comes from different experiments.

Positive values of the difference between *cwnd* and the amount of unacknowledged data (Figures 4.50c and 4.51c) indicate that *cwnd* was large enough to allow there to be data to sent, but that no data was actually transmitted. This could be due to periods when ACKs were not returning and releasing new data even though there was room in the congestion window. Negative values of this difference indicate that *cwnd* was reduced and that no new data will be sent until the amount of unacknowledged data is less than the value of *cwnd*. Large positive spikes seem to be followed by large negative spikes and line up with segment drops, indicating data recovery, where *cwnd* is immediately reduced. With Sync-TCP(MixReact) at 60% load (Figure 4.50c), there is only one large positive spike, occurring at the end of the transmission. This spike is due to the fact that there is no more data to send for this flow even though *cwnd* is large enough to allow it. Sync-TCP(MixReact) at 60% load has no segment drops and no recovery instances, so no large spikes occur during the transmission.

The most interesting graphs for studying the behavior of the Sync-TCP protocols as compared to TCP Reno are the plots of *cwnd* over time (Figures 4.50a and 4.51a).

At 60% load (Figure 4.50a), both the TCP Reno and Sync-TCP(MixReact) flows have the same base RTT (105 ms). The Sync-TCP(Pcwnd) flow has a much shorter base RTT (38 ms), so it should (and does) complete sooner than the other flows. The TCP Reno flow has the familiar sawtooth pattern in its congestion window. The congestion window grows until a segment is dropped (as marked with a '+') and then is either cut in half or reduced to 1 segment. The reductions in *cwnd* due to segment loss are easy to see at this scale.

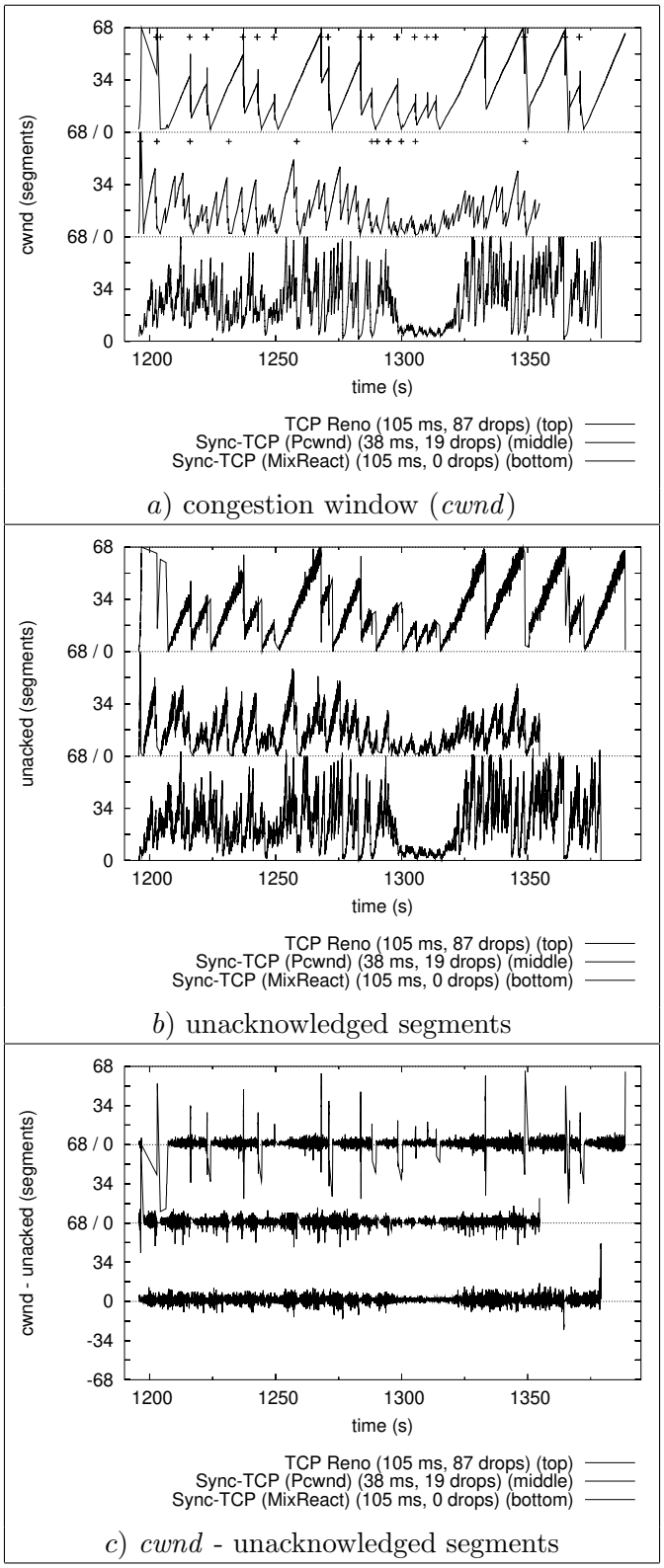


Figure 4.50: 60% Load, Congestion Window and Unacknowledged Segments

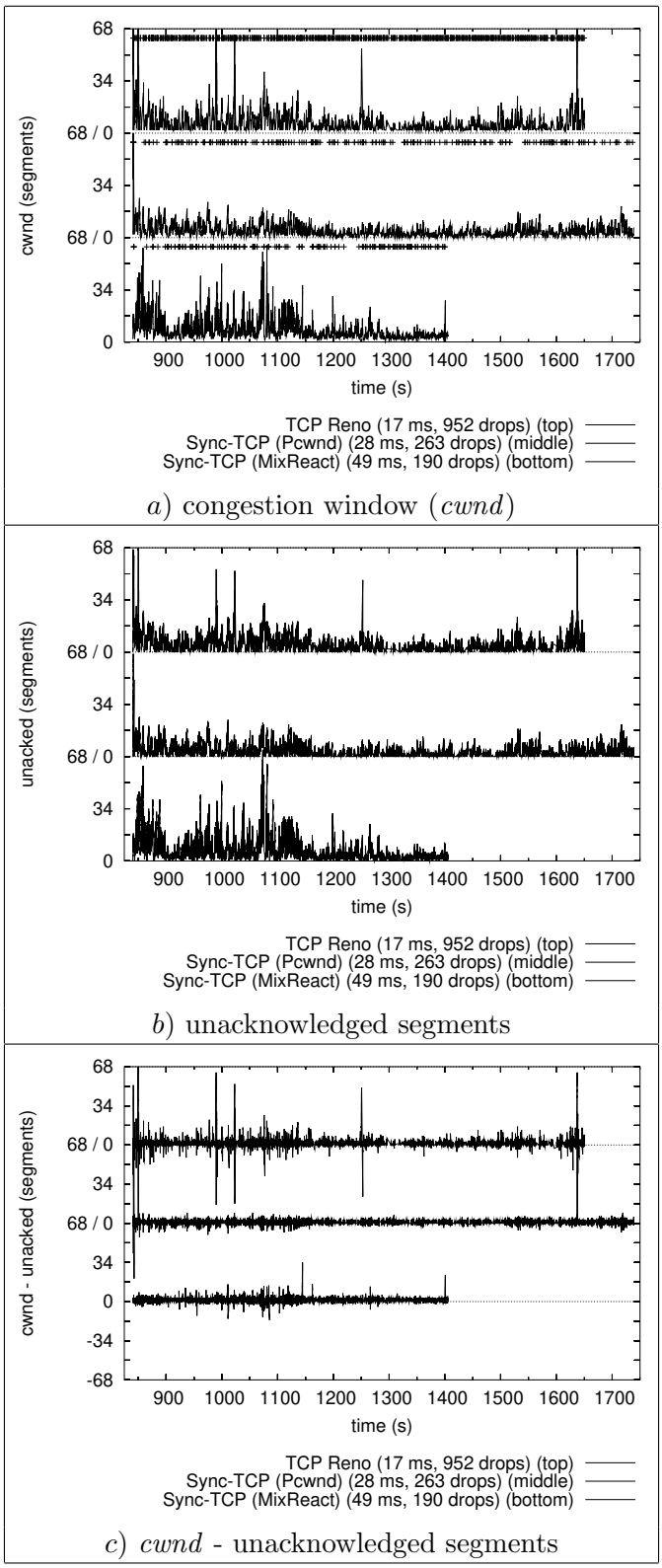


Figure 4.51: 85% Load, Congestion Window and Unacknowledged Segments

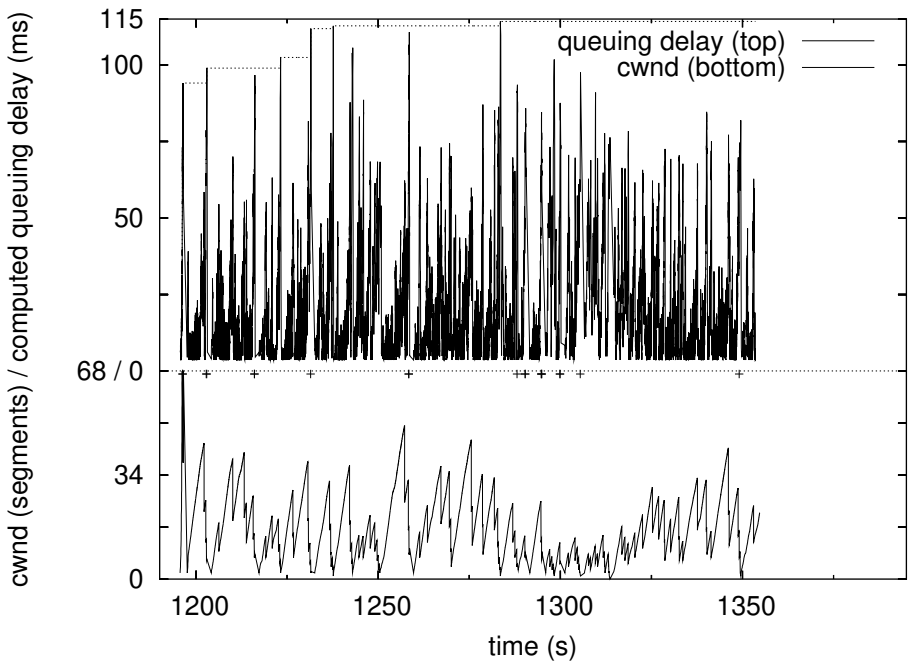


Figure 4.52: Sync-TCP(Pcwnd) - *cwnd* and Queuing Delay at 60% Load

The Sync-TCP(Pcwnd) flow at 60% load has a much shorter base RTT than the TCP Reno flow, so its linear increase of *cwnd* during congestion avoidance has a sharper slope in the graph than the TCP Reno flow. Sync-TCP(Pcwnd), like TCP Reno, either reduces *cwnd* by 50% or to 1 segment, but Sync-TCP(Pcwnd) will reduce *cwnd* by 50% more often than TCP Reno, since it reacts to high queuing delays as well as segment loss. Every reduction of *cwnd* that is not associated with a segment drop cuts *cwnd* in half. By reducing its congestion window in response to increases in queuing delay, the Sync-TCP(Pcwnd) flow avoids many of the drops that the TCP Reno flow experiences. The Sync-TCP(Pcwnd) flow has a much shorter RTT than the TCP Reno flow, but only finishes 34 seconds sooner. It reduces *cwnd* more often than TCP Reno, but does not compensate by being more aggressive with opening its congestion window. If the RTTs of the flows were closer, the Sync-TCP(Pcwnd) flow would likely finish after the TCP Reno flow even though many segment losses were avoided, because Sync-TCP(Pcwnd) reduces its congestion window more often, but is constrained to TCP Reno's linear increase during congestion avoidance. Figure 4.52 shows the Sync-TCP(Pcwnd) flow's (same flow as in Figure 4.50) computed queuing delay plotted above its congestion window. The black line above the computed queuing delay marks the maximum-observed queuing delay. This figure shows how increases in queuing delay affect the congestion window.

The Sync-TCP(MixReact) flow at 60% load has the same RTT as the TCP Reno flow, but finishes slightly sooner and avoids all segment loss (recall that the TCP Reno flow saw 87

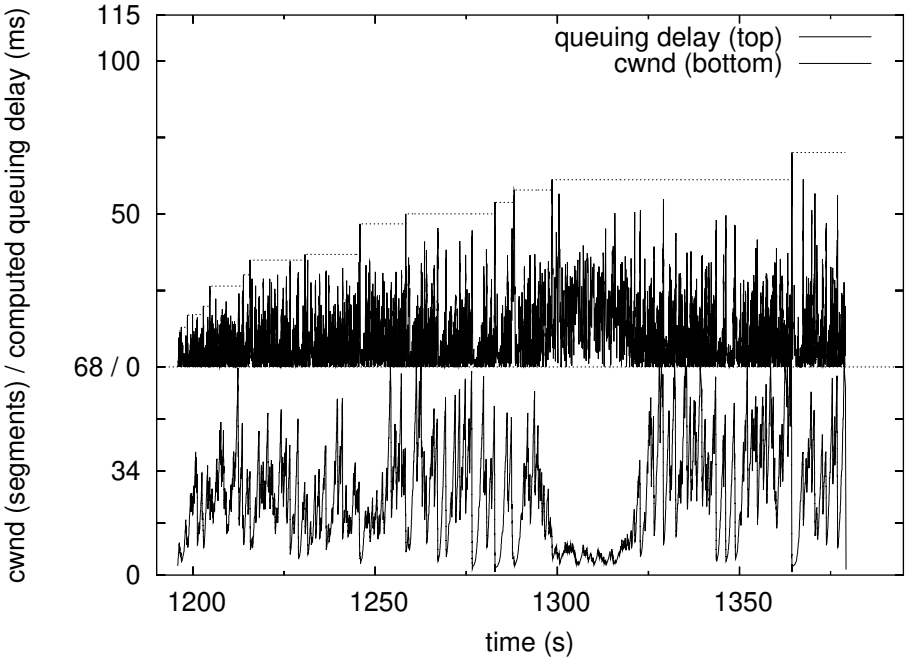


Figure 4.53: Sync-TCP(MixReact) - *cwnd* and Queuing Delay at 60% Load

drops). The congestion window adjustments with Sync-TCP(MixReact) are made every three ACKs when a new trend is computed. The frequent changes to the *cwnd* are apparent in the short period of the congestion window. Sync-TCP(MixReact) has decreases in *cwnd* of up to 50%, and Sync-TCP(MixReact) is never more aggressive in the magnitude of the decrease than TCP Reno or Sync-TCP(Pcwnd). The downward spikes in the Sync-TCP(MixReact) *cwnd* are not instantaneous, as would occur with a timeout, but are a series of gradual decreases (*e.g.*, there are three sharp downward spikes beginning near time 1275). Congestion window adjustments are made every three ACKs, which may be too frequently. Flows may be continuously reacting to the same period of congestion without waiting to see if the previous adjustment had an effect. In contrast to Sync-TCP(Pcwnd), Sync-TCP(MixReact) has *cwnd* increases potentially more aggressive than TCP Reno congestion avoidance. With Sync-TCP(MixReact), *cwnd* can grow by as much as 50% in one RTT (effectively doubling in two RTTs). Some of these sharp increases can been seen in Figure 4.50, especially in contrast to the gradual, linear increase of TCP Reno. Sync-TCP(MixReact) sees no segment loss, yet reduces its *cwnd* during periods of congestion. Since Sync-TCP(MixReact) avoids segment loss and can increase *cwnd* more aggressively than TCP Reno, why is the goodput (response size / duration) gain not much greater? The congestion window of a Sync-TCP(MixReact) flow is directly affected by the queuing delay the sender computes. Figure 4.53 shows this computed queuing delay plotted above the flow's *cwnd*. The black line above the queuing delay is the sender's maximum-observed queuing delay. This flow never sees loss, so the

sender will never see a queuing delay that represents the maximum amount of queuing in the network. As seen in Figure 4.52, after the Sync-TCP(Pcwnd) flow sees its first segment loss, its maximum queuing delay is almost 100 ms. During the entire flow, the computed queuing delay of the Sync-TCP(MixReact) flow never gets near 100 ms (its maximum is below 75 ms). Once the maximum queuing delay is greater than 10 ms, the congestion window of the Sync-TCP(MixReact) flow is being constrained by a maximum that does not fully represent the maximum amount of queuing in the network. The congestion detection and reaction mechanisms do adjust *cwnd* to the queuing delay, but since there is no good estimate of the maximum, the flow is reducing *cwnd* significantly when there is no actual danger of the queue overflowing. Especially during the period between time 1300-1325, *cwnd* is very low because the queuing delay during that period is clearly higher than during other periods of the flow's transfer. Even so, during this 25-second period, the queuing delay is low relative to the queuing delay during a similar region in the Sync-TCP(Pcwnd) experiment (Figure 4.52).

At 85% load (Figure 4.51), all flows have the same response size but different base RTTs. Sync-TCP(MixReact) experiences the fewest drops and finishes first, yet has the longest base RTT. The Sync-TCP(MixReact) flow received an average of 725.9 kbps, the flow using Sync-TCP(Pcwnd) received 456.6 kbps, and the flow received 506.4 kbps when using TCP Reno.

The Sync-TCP(Pcwnd) flow at 85% load, although it had only a slightly larger base RTT, finished much later (almost 90 seconds) than the TCP Reno flow, which experienced much more loss. The Sync-TCP(Pcwnd) flow was successful in reducing loss, but was not able to take advantage of the avoided drops and get improved goodput because it is not aggressive in increasing its congestion window during uncongested times.

The Sync-TCP(MixReact) flow at 85% load, on the other hand, performed quite well. It had the largest base RTT of the three flows, but finished first (a full 200 seconds before the TCP Reno flow) and had five times fewer drops than TCP Reno. This flow, unlike in the 60% load case, saw segment loss earlier in its transfer and had a good estimate of the maximum amount of queuing in the network. Although the flow takes longer to complete than its 60% load counterpart, there is much less bandwidth available in the network (because of the overall load increase). For comparison, Figure 4.54 shows the flow's *cwnd* and computed queuing delay in the same format as Figures 4.52 and 4.53. When the queuing delay is low (for example, near time 1050), the congestion window for the flow is allowed to grow. When the queuing delay is high (for example, after time 1150), the congestion window is kept relatively low. The more aggressive increase of Sync-TCP(MixReact) allows *cwnd* to grow much more rapidly than with TCP Reno. One advantage of detecting congestion every three ACKs is that the presence of a strong increasing trend in the average queuing delay can be detected and reacted to quickly. This allows for more aggressive increase without frequently overflowing the queue.

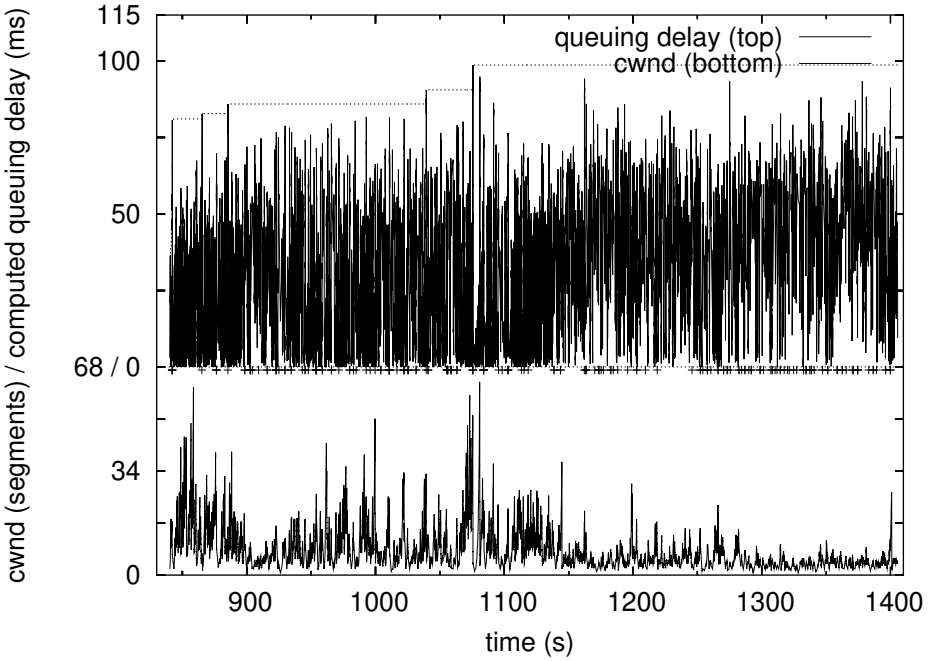


Figure 4.54: Sync-TCP(MixReact) - *cwnd* and Queuing Delay at 85% Load

Another thing to notice in the congestion window graphs is the behavior at the beginning of the flow. This is particularly evident at 60% load because the scale is smaller. Both TCP Reno and Sync-TCP(Pcwnd) begin in slow start and quickly increases their congestion windows to the maximum where packet loss then occurs. Sync-TCP(MixReact) avoids much of slow start and is able to increase *cwnd* without overflowing the network. Sync-TCP(MixReact) actually begins in slow start but transitions to early congestion detection after nine ACKs have been received and the trend of the average queuing delay can be computed. With delayed ACKs, during slow start *cwnd* grows to 10 segments with the receipt of the 8th ACK. When the 9th ACK is received, the trend can be computed and early congestion detection begins.

Response Size vs. RTT vs. Response Duration

I analyzed how Sync-TCP(MixReact) treats flows with particular response sizes and RTTs. To do this, I gathered the response size, RTT, and duration of the response (not including the request) for all flows in each experiment. The response durations were put in bins according to the response's size (in 1420-byte packets) and the RTT of its flow (in 10 ms). I calculated the average response duration for each bin that had 10 or more elements. Then, for each bin, I subtracted the average response duration of the TCP Reno flows from the average response duration of the Sync-TCP(MixReact) flows. This analysis was performed for the 60% and 85% loads (Figure 4.55). In these graphs, the x-axis is the RTT in 10 ms intervals, and the y-axis is the response size in 1420-byte packets. The color indicates the difference

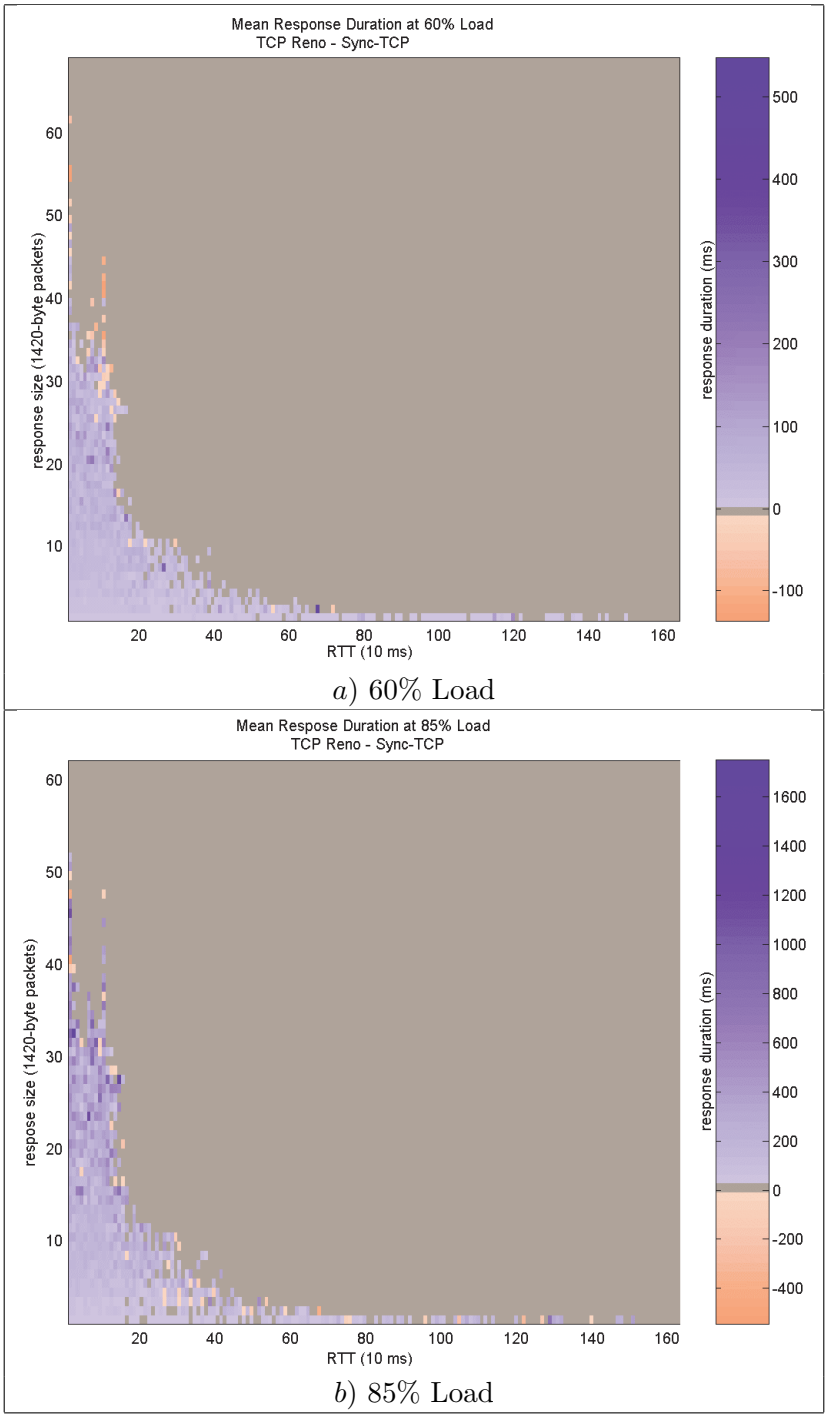


Figure 4.55: Difference in Mean Response Duration

between the average response duration for that bin obtained by TCP Reno and by Sync-TCP(MixReact). The scale of the colormap is in milliseconds. In these graphs, gray indicates bins where there were fewer than 10 flows for either TCP Reno or Sync-TCP(MixReact). Blue values indicate bins where the average response duration for TCP Reno was higher than Sync-TCP(MixReact), meaning that the flows finished faster on average with Sync-TCP(MixReact). Red values indicate that the average response duration for TCP Reno was lower than Sync-TCP(MixReact), meaning that the flows finished faster on average with TCP Reno.

At 60% load (Figure 4.55a), there are regions with large response sizes (30-45 packets) and moderate RTTs (around 170 ms) where TCP Reno performed better than Sync-TCP(MixReact). Again, this is to be expected because of the implementation of SyncMix when no packet loss occurs. At 85% load (Figure 4.55b), there are fewer bins where TCP Reno performs better than Sync-TCP(MixReact). There are also more bins where the blue color is darker, meaning that there was a greater difference in average response duration (*i.e.*, a bigger "win" for Sync-TCP(MixReact)).

4.3.2 Multiple Bottlenecks

The following experiments were run with various levels of cross-traffic to create multiple congested links. The levels of cross-traffic were based on the amount of load that result on the congested links from the combination of the end-to-end traffic and the cross-traffic. I tested against three levels of total load: 75%, 90%, and 105%. These resulted in cross-traffic loads as described in Table 4.1. Experiments with 50% and 60% end-to-end load have two main bottlenecks: at router R1 and at router R3 (Figure 4.1) where cross-traffic enters the network. Experiments with higher loads of end-to-end traffic have an additional bottleneck at R0, the end-to-end aggregation point.

Due to the additional overhead on memory and actual running time of creating the cross-traffic, these experiments ran until either 100,000 (for 105% total load) or 150,000 (for 75% and 90% total load) request-response pairs completed. These numbers of request-response pairs still meet the criteria for running time as described in Appendix A.3. Additionally, the multiple bottleneck experiments were only run for Sync-TCP(MixReact) and TCP Reno.

I will present summary data like that presented in the single bottleneck experiments, but in table form rather than in graphs. These tables include average loss and average queue size at each of the three congested routers (R0, R1, and R3), along with median and mean response times, goodput per response, throughput of end-to-end traffic, goodput of end-to-end traffic, and link utilization by end-to-end traffic. Throughput is measured on the link between router R0 and router R1. Goodput and link utilization are measured on the link between router R4 and router R5. Recall that throughput, goodput, and link utilization are

	TCP Reno	Sync-TCP(MixReact)
avg queue size at R0 (pckts)	4.3	2.7
avg queue size at R1 (pckts)	27.3	15.6
avg queue size at R3 (pckts)	19.3	10.2
% packet drops at R0	0.0	0.0
% packet drops at R1	0.8	0.1
% packet drops at R3	0.4	0.0
median rsptime (ms)	380.0	310.0
mean rsptime (ms)	833.9	527.0
goodput per response (kbps)	119.0	135.9
throughput (kbps)	5043.6	4987.8
goodput (kbps)	4479.9	4456.7
link utilization (kbps)	5012.5	4986.3

Table 4.12: 50% End-to-End Load, 75% Total Load, 2 Bottlenecks

	TCP Reno	Sync-TCP(MixReact)
avg queue size at R0 (pckts)	8.8	4.6
avg queue size at R1 (pckts)	23.2	10.8
avg queue size at R3 (pckts)	16.7	8.1
% packet drops at R0	0.0	0.0
% packet drops at R1	0.6	0.0
% packet drops at R3	0.4	0.0
median rsptime (ms)	380.0	300.0
mean rsptime (ms)	808.3	502.6
goodput per response (kbps)	123.6	142.0
throughput (kbps)	6029.3	5984.6
goodput (kbps)	5362.0	5346.2
link utilization (kbps)	6002.3	5984.3

Table 4.13: 60% End-to-End Load, 75% Total Load, 2 Bottlenecks

computed for all traffic on the link, not just for connections that had its response successfully complete and are included here for completeness.

75% Total Load

Tables 4.12-4.14 give summary statistics for the multiple bottleneck experiments where there was 75% total offered load on the congested links. The loss rates at each of the routers for both TCP Reno and Sync-TCP(MixReact) is very low. This matches expectations due to the performance for the single bottleneck case at 70% and 80% loads (Figure 4.12a). Since end-to-end loads of 50% and 60% would not normally cause much congestion at the aggregation point (R0), the average queue sizes at R0 are not very large. The average queue sizes at R1 and R3 are greater than at R0, but this is due to the additional traffic entering the network at those

	TCP Reno	Sync-TCP(MixReact)
avg queue size at R0 (pckts)	20.4	10.0
avg queue size at R1 (pckts)	21.3	8.8
avg queue size at R3 (pckts)	11.7	5.5
% packet drops at R0	0.3	0.0
% packet drops at R1	0.5	0.0
% packet drops at R3	0.2	0.0
median rsptime (ms)	380.0	300.0
mean rsptime (ms)	774.1	510.4
goodput per response (kbps)	124.0	143.0
throughput (kbps)	7086.2	7016.7
goodput (kbps)	6286.1	6257.0
link utilization (kbps)	7047.3	7015.0

Table 4.14: 70% End-to-End Load, 75% Total Load, 3 Bottlenecks

points. At 70% end-to-end load, the average queue size at R0 is comparable to that at R1, so I designate this as having three bottlenecks. For all of these metrics, Sync-TCP(MixReact) performs better (or comparable to – for throughput, goodput, and link utilization) than TCP Reno. In fact, with Sync-TCP(MixReact) there is almost no packet loss at all.

Figure 4.56 shows the HTTP response time CDFs for TCP Reno and Sync-TCP(MixReact) with multiple bottlenecks at 75% total load. The labels on the graph correspond to the amount of end-to-end traffic during the experiment. For example, Figure 4.56a shows the HTTP response time CDFs for TCP Reno with a total of 75% load at the congested links and 50%, 60%, and 70% load traveling end-to-end. These graphs are presented to show how additional end-to-end traffic affects HTTP response time performance, if the total amount of congestion is held relatively constant between experiments. Figure 4.56 shows little change in response times as the amount of end-to-end traffic increases.

Figures 4.57 and 4.58 present a direct comparison between TCP Reno and Sync-TCP(MixReact) for the HTTP response times for the multiple bottleneck case with 75% load at the congested links. Figure 4.57 shows the response time CDFs, while Figure 4.58 shows the response time CCDFs. Sync-TCP(MixReact) performs better than TCP Reno for each of the end-to-end loads. The TCP Reno flows also experienced more loss due to the drop of SYN or SYN/ACK packets during connection setup. This is shown by the spikes at 6 seconds in the response time CCDFs for TCP Reno.

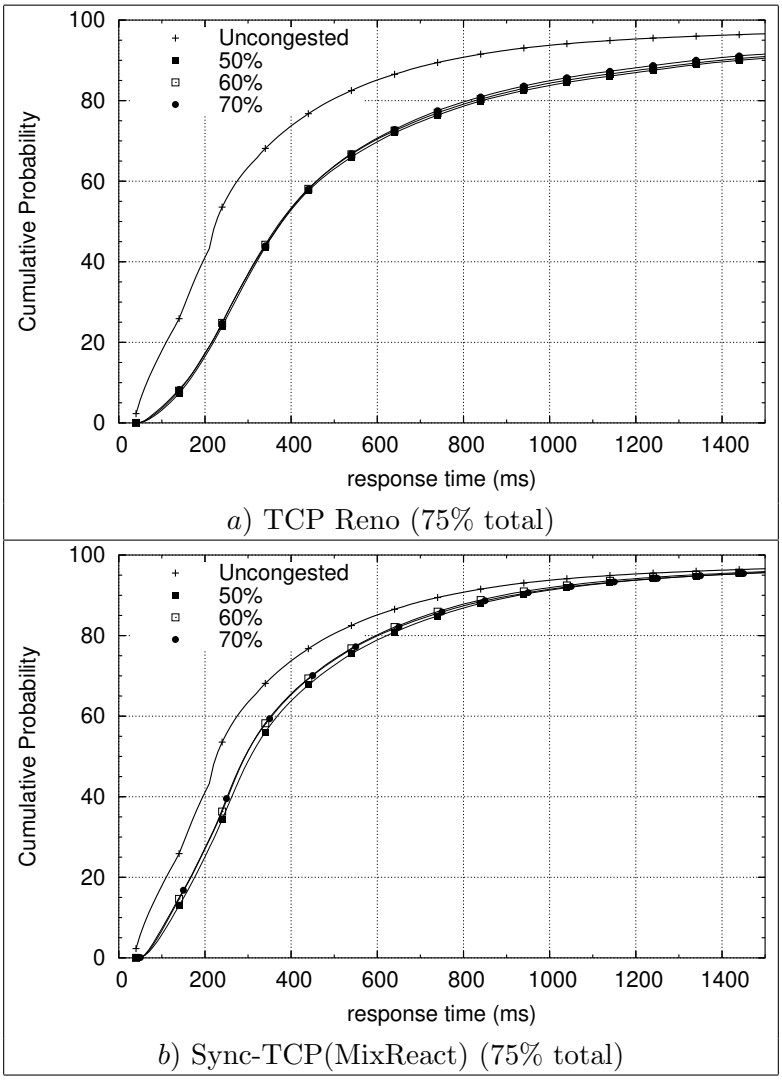


Figure 4.56: Response Time CDFs, Multiple Bottlenecks, 75% Total Load

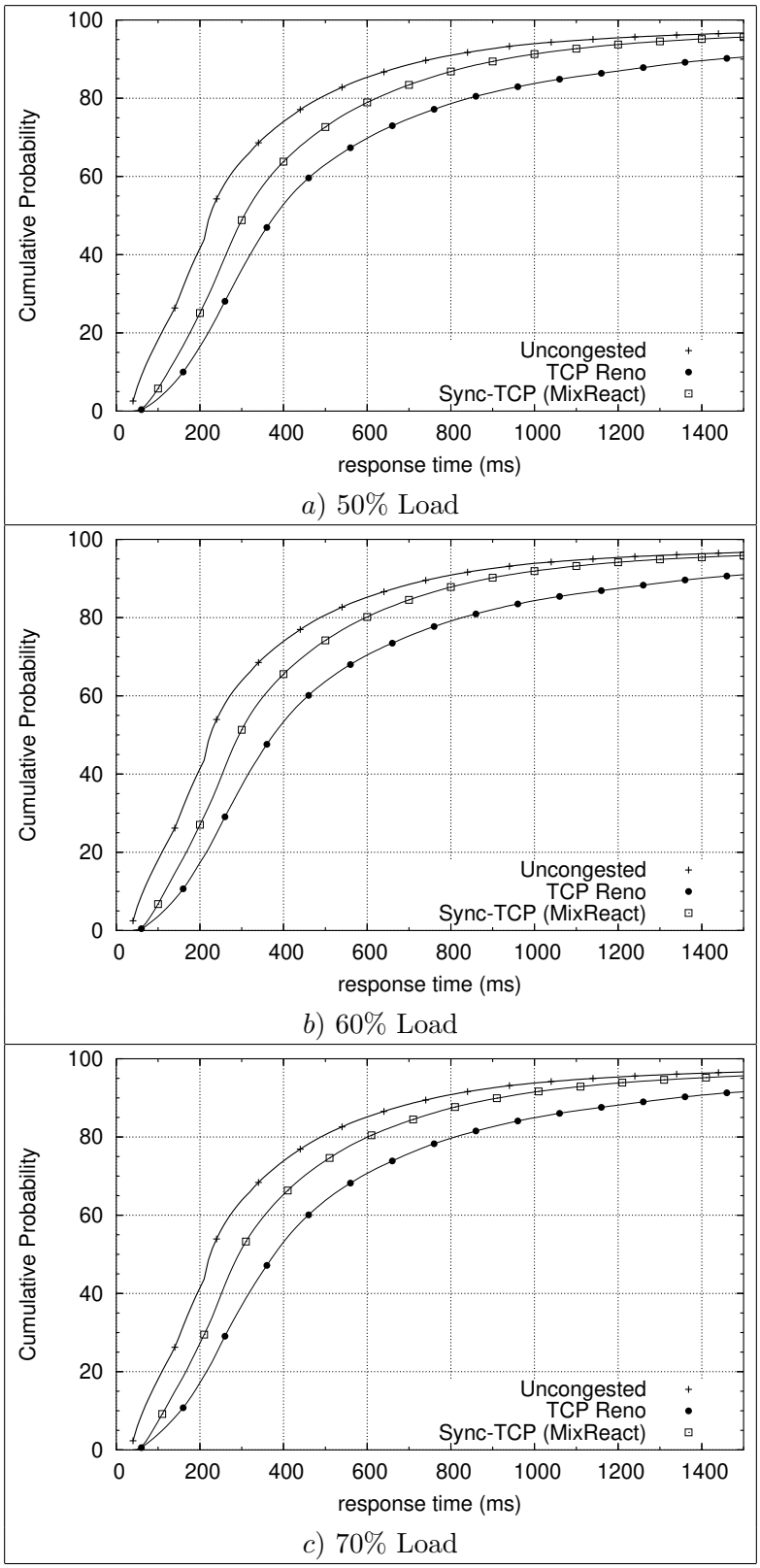


Figure 4.57: Response Time CDFs, Total 75% Load

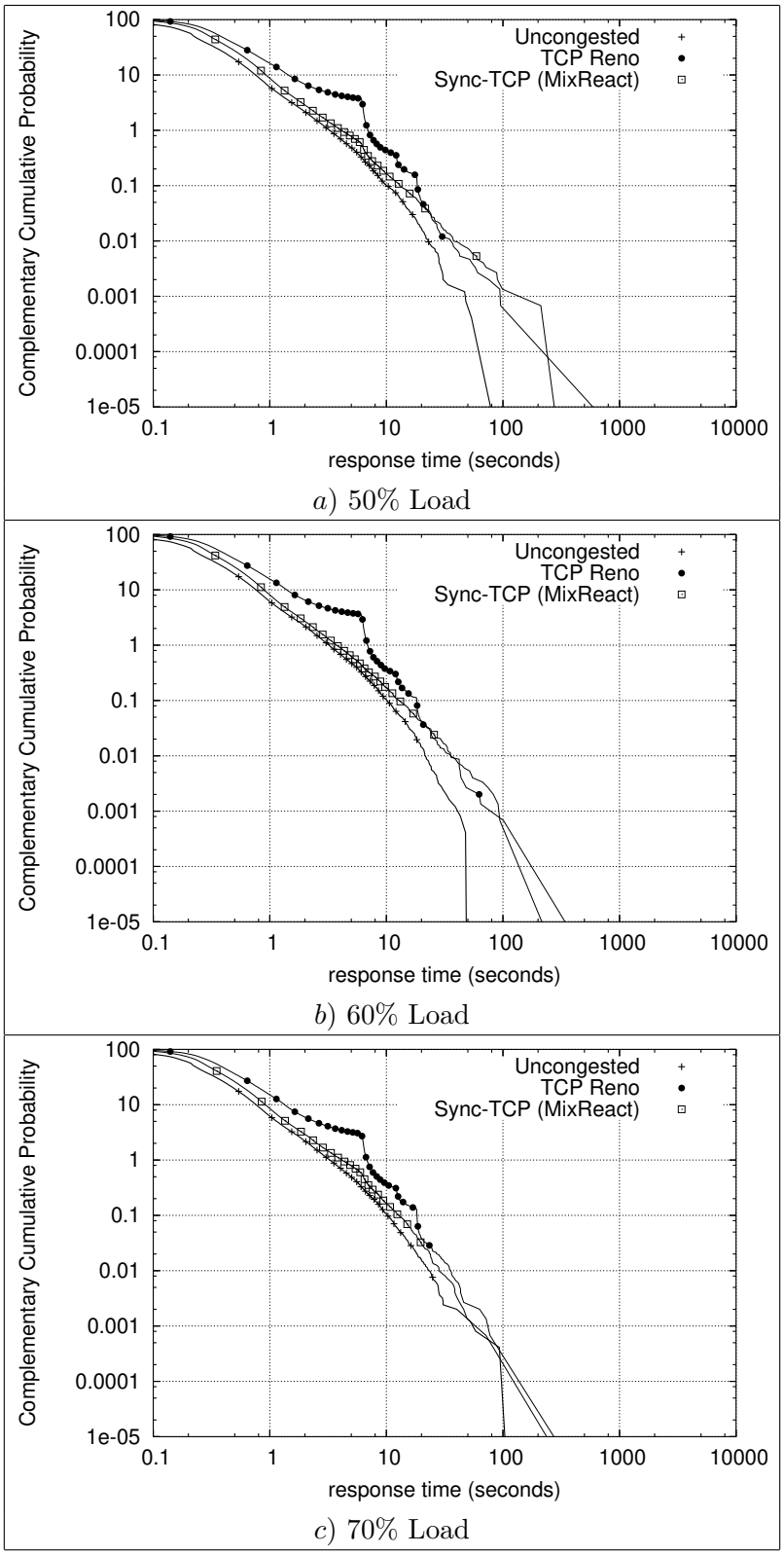


Figure 4.58: Response Time CCDFs, Total 75% Load

	TCP Reno	Sync-TCP(MixReact)
avg queue size at R0 (pckts)	3.0	2.4
avg queue size at R1 (pckts)	56.6	44.7
avg queue size at R3 (pckts)	39.8	28.1
% packet drops at R0	0.0	0.0
% packet drops at R1	4.0	1.9
% packet drops at R3	1.7	0.4
median rsptime (ms)	620.0	450.0
mean rsptime (ms)	2285.6	1181.6
goodput per response (kbps)	72.0	89.9
throughput (kbps)	5035.9	4897.3
goodput (kbps)	4329.9	4298.9
link utilization (kbps)	4870.0	4832.4

Table 4.15: 50% End-to-End Load, 90% Total Load, 2 Bottlenecks

	TCP Reno	Sync-TCP(MixReact)
avg queue size at R0 (pckts)	5.6	4.2
avg queue size at R1 (pckts)	52.4	42.6
avg queue size at R3 (pckts)	37.6	27.3
% packet drops at R0	0.0	0.0
% packet drops at R1	3.7	1.7
% packet drops at R3	1.6	0.4
median rsptime (ms)	610.0	440.0
mean rsptime (ms)	2138.8	1079.3
goodput per response (kbps)	76.1	92.9
throughput (kbps)	5983.0	5884.1
goodput (kbps)	5179.1	5186.4
link utilization (kbps)	5829.8	5825.8

Table 4.16: 60% End-to-End Load, 90% Total Load, 2 Bottlenecks

90% Total Load

Tables 4.15 – 4.19 show summary statistics for the multiple bottleneck experiments with the congested links at 90% load. With end-to-end loads greater than 70%, there is enough congestion just due to the aggregation of the end-to-end traffic to cause packet drops and queuing at R0. Since the cross-traffic is relatively light and does not try to consume the entire link (especially when the end-to-end traffic is at 80% or higher), there should be less packet drops and queue size at R3 than at R1 because R1 will be forwarding fewer packets that are headed for R3 (*i.e.*, the end-to-end traffic). For all of the metrics, Sync-TCP(MixReact) outperforms TCP Reno for all of the end-to-end load levels.

Figure 4.59 shows the response time CDF for TCP Reno and Sync-TCP(MixReact) at each load level for 90% total load with multiple bottlenecks. Figure 4.59a is interesting because

	TCP Reno	Sync-TCP(MixReact)
avg queue size at R0 (pckts)	14.3	7.5
avg queue size at R1 (pckts)	57.8	40.1
avg queue size at R3 (pckts)	44.0	31.0
% packet drops at R0	0.2	0.0
% packet drops at R1	6.2	1.8
% packet drops at R3	3.2	1.1
median rsptime (ms)	880.0	460.0
mean rsptime (ms)	3937.1	1303.7
goodput per response (kbps)	67.1	91.0
throughput (kbps)	7056.0	6828.7
goodput (kbps)	6006.0	5995.3
link utilization (kbps)	6777.8	6755.7

Table 4.17: 70% End-to-End Load, 90% Total Load, 3 Bottlenecks

	TCP Reno	Sync-TCP(MixReact)
avg queue size at R0 (pckts)	38.1	19.4
avg queue size at R1 (pckts)	57.0	40.5
avg queue size at R3 (pckts)	28.5	19.2
% packet drops at R0	2.6	0.3
% packet drops at R1	7.9	3.0
% packet drops at R3	1.1	0.4
median rsptime (ms)	1000.0	470.0
mean rsptime (ms)	4753.4	1529.2
goodput per response (kbps)	66.2	91.8
throughput (kbps)	8139.9	7830.3
goodput (kbps)	6813.8	6855.2
link utilization (kbps)	7693.5	7726.5

Table 4.18: 80% End-to-End Load, 90% Total Load, 3 Bottlenecks

	TCP Reno	Sync-TCP(MixReact)
avg queue size at R0 (pckts)	43.4	27.3
avg queue size at R1 (pckts)	46.6	31.0
avg queue size at R3 (pckts)	24.9	16.0
% packet drops at R0	2.2	0.7
% packet drops at R1	4.7	1.9
% packet drops at R3	0.8	0.2
median rsptime (ms)	660.0	440.0
mean rsptime (ms)	2892.9	1254.5
goodput per response (kbps)	75.8	98.2
throughput (kbps)	8553.6	8300.8
goodput (kbps)	7304.7	7270.1
link utilization (kbps)	8242.5	8198.4

Table 4.19: 85% End-to-End Load, 90% Total Load, 3 Bottlenecks

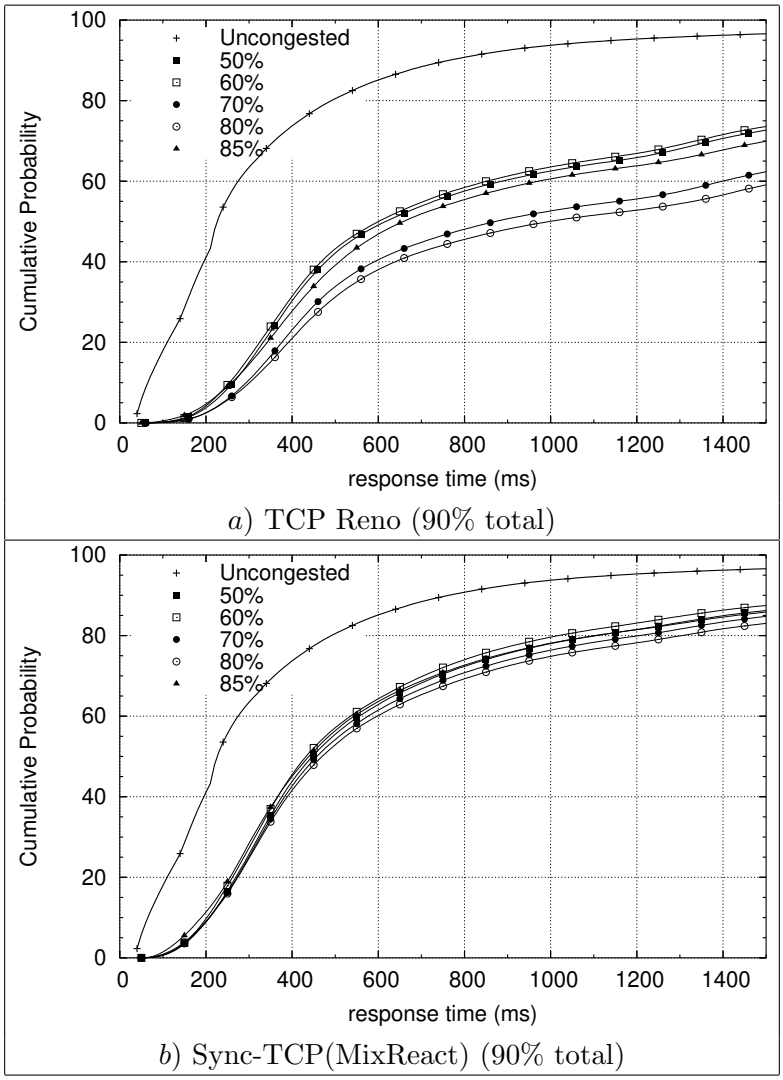


Figure 4.59: Response Time CDFs, Multiple Bottlenecks, 90% Total Load

end-to-end loads of 70% and 80% perform considerably worse than the other end-to-end loads. Tables 4.17 and 4.18 show that these loads saw higher loss than the other loads, resulting in poor response time performance.

Figures 4.60-4.63 compare TCP Reno and Sync-TCP(MixReact) for the multiple bottleneck experiments where the total load is 90% at the congested links. Figures 4.60 and 4.61 show the response time CDFs, while Figures 4.62 and 4.63 show the response time CCDFs. For all end-to-end loads, Sync-TCP(MixReact) performs better than TCP-Reno, even out to the longest response time.

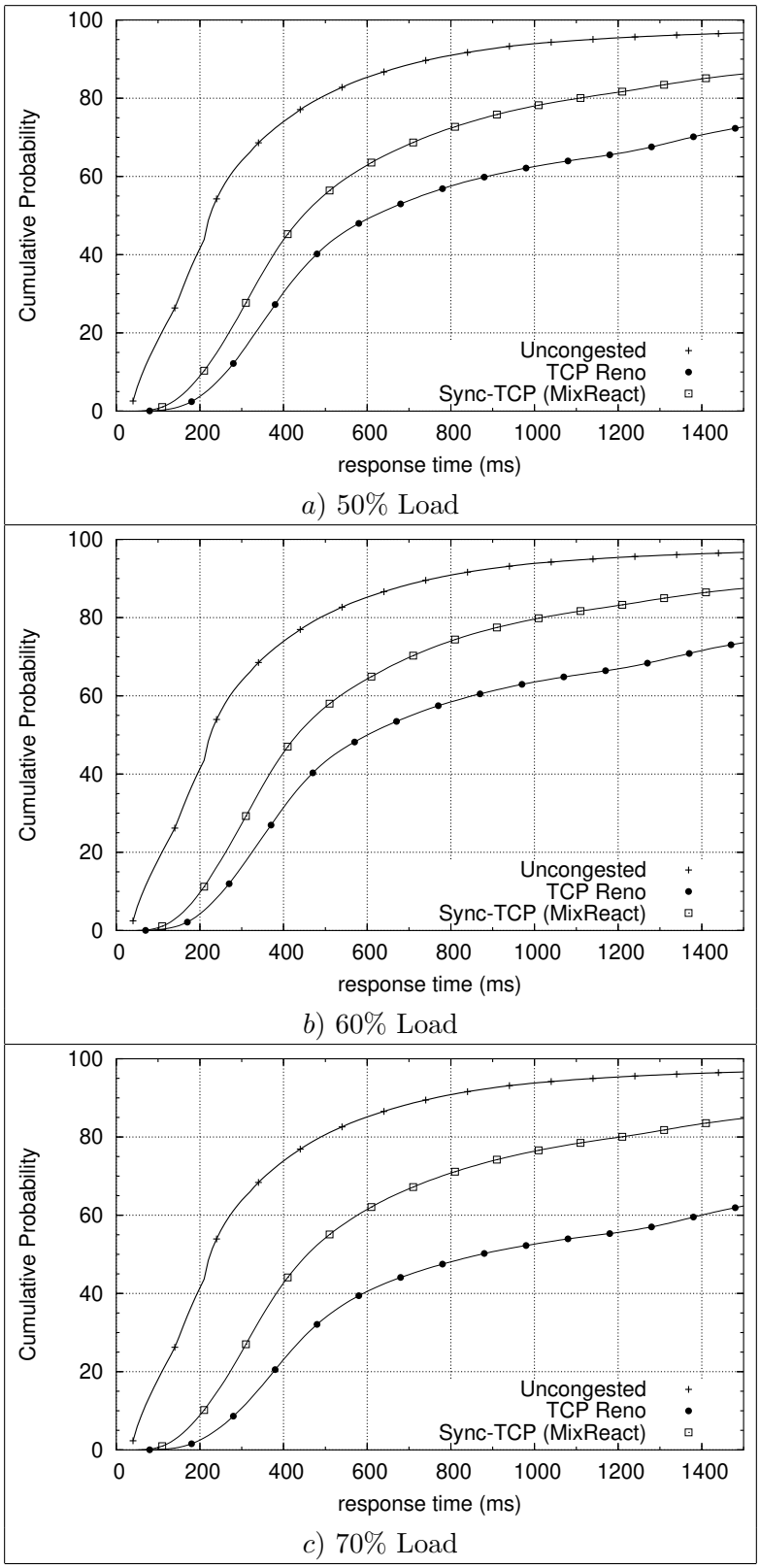


Figure 4.60: Response Time CDFs, Total 90% Load, Light End-to-End Congestion

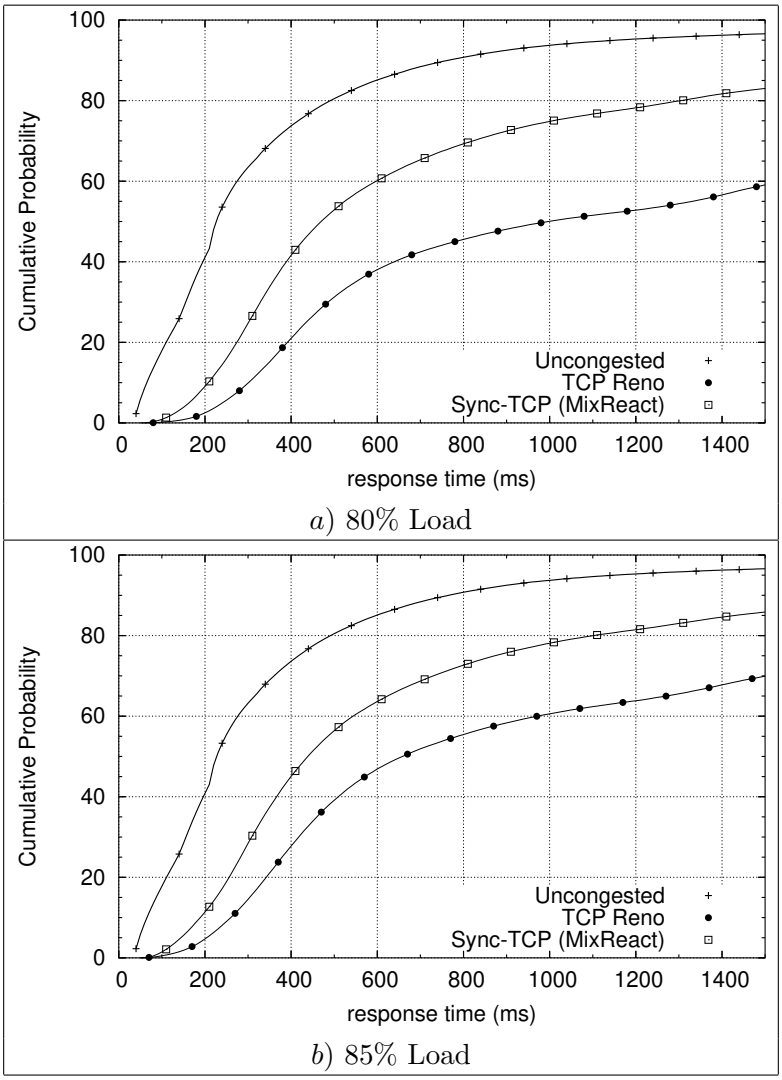


Figure 4.61: Response Time CDFs, Total 90% Load, Medium End-to-End Congestion

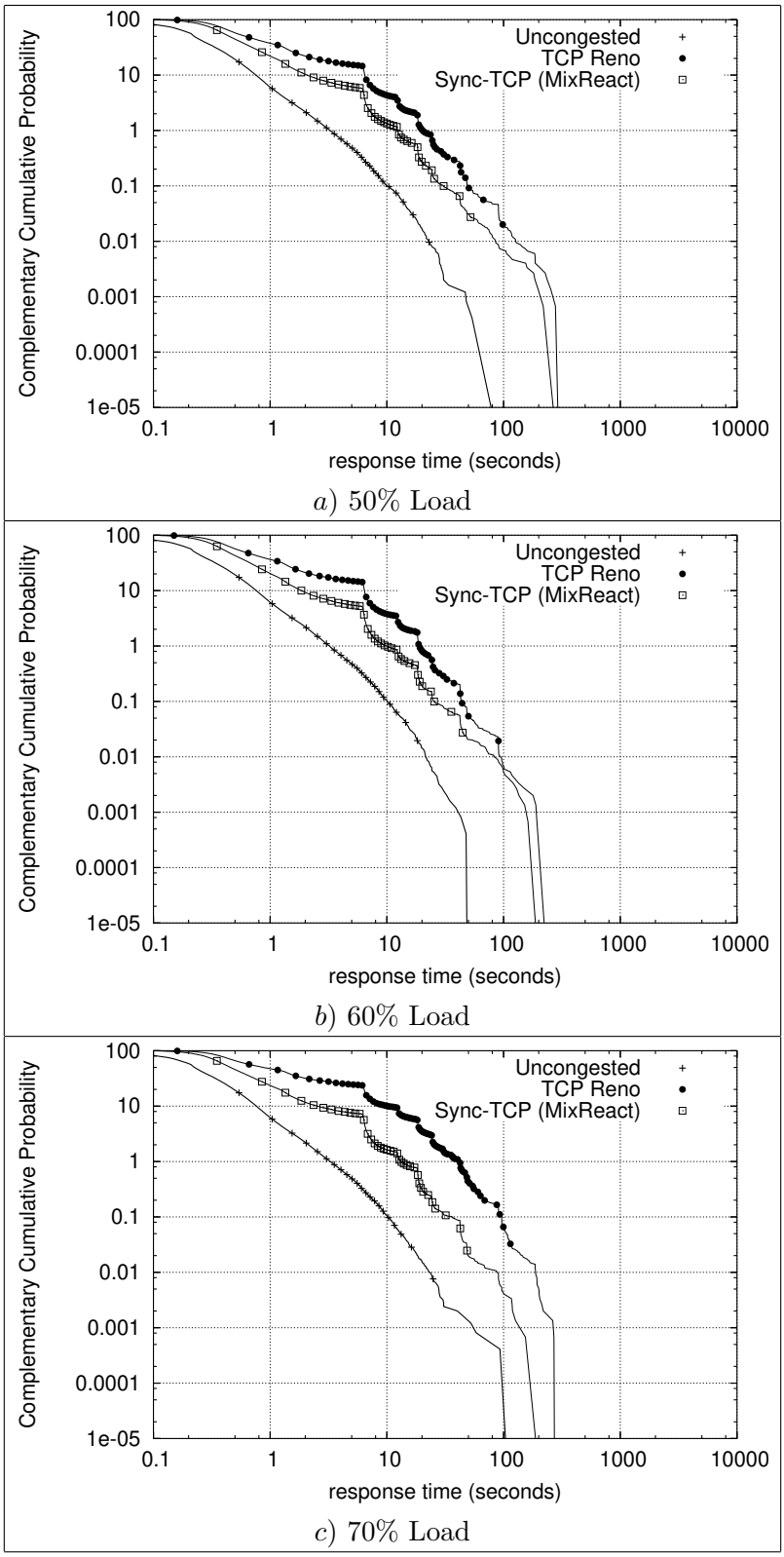


Figure 4.62: Response Time CCDFs, Total 90% Load, Light End-to-End Congestion

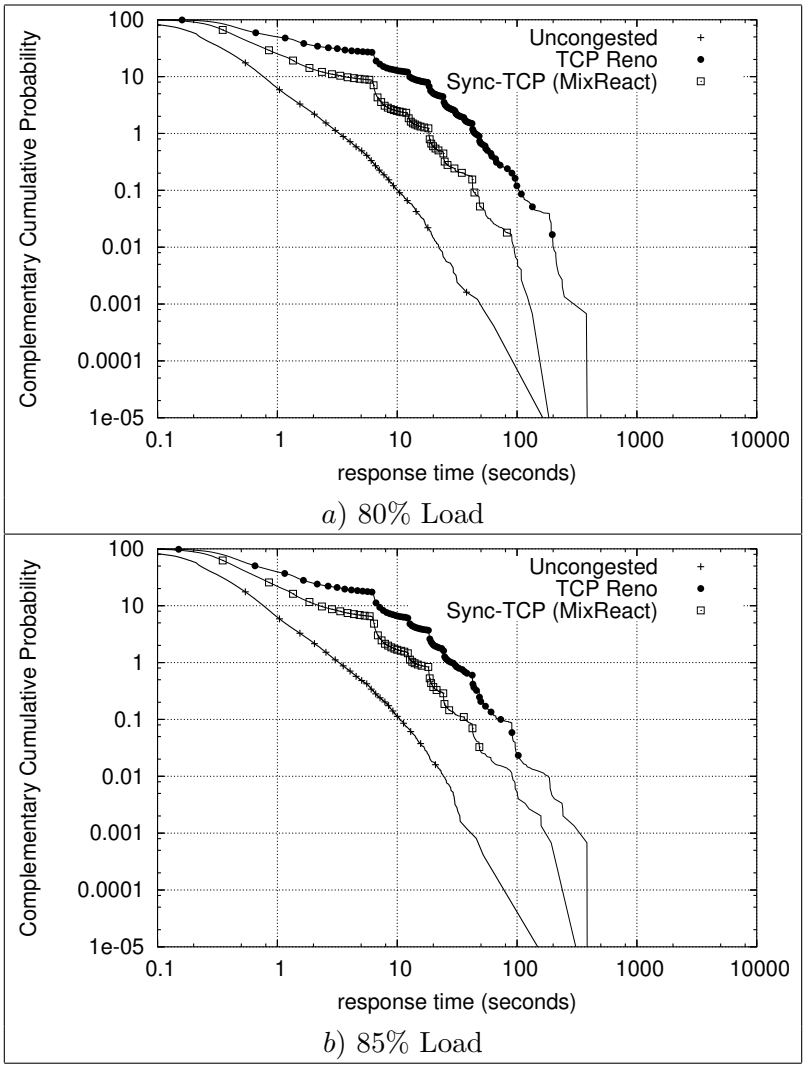


Figure 4.63: Response Time CCDFs, Total 90% Load, Medium End-to-End Congestion

	TCP Reno	Sync-TCP(MixReact)
avg queue size at R0 (pckts)	3.0	2.9
avg queue size at R1 (pckts)	114.9	108.2
avg queue size at R3 (pckts)	97.0	87.6
% packet drops at R0	0.0	0.0
% packet drops at R1	24.9	21.6
% packet drops at R3	11.2	9.1
median rsptime (ms)	12690.0	7590.0
mean rsptime (ms)	36388.8	26357.5
goodput per response (kbps)	19.1	24.3
throughput (kbps)	5745.0	5541.9
goodput (kbps)	3968.1	3973.7
link utilization (kbps)	4501.1	4507.8

Table 4.20: 50% End-to-End Load, 105% Total Load, 2 Bottlenecks

	TCP Reno	Sync-TCP(MixReact)
avg queue size at R0 (pckts)	16.1	13.5
avg queue size at R1 (pckts)	110.4	104.3
avg queue size at R3 (pckts)	85.0	79.4
% packet drops at R0	0.4	0.4
% packet drops at R1	27.0	22.0
% packet drops at R3	12.5	10.8
median rsptime (ms)	18590.0	12460.0
mean rsptime (ms)	45243.1	29986.0
goodput per response (kbps)	19.6	24.2
throughput (kbps)	6879.0	6528.2
goodput (kbps)	4652.1	4618.8
link utilization (kbps)	5304.8	5279.6

Table 4.21: 60% End-to-End Load, 105% Total Load, 2 Bottlenecks

105% Total Load

Tables 4.20-4.27 show the summary statistics for the multiple bottleneck experiments with 105% total load on the congested links. The loss rates and average queue sizes at all of the routers (especially at R1) are very large. Performance overall is poor, as expected, but Sync-TCP(MixReact) performed slightly better than TCP Reno at all end-to-end load levels. Figure 4.64 shows the HTTP response time CDFs for TCP Reno and Sync-TCP(MixReact) with multiple bottlenecks at 105% total load. As expected from looking at the summary tables, performance is poor at all end-to-end load levels. Figures 4.65-4.67 show the HTTP response time CCDFs for the 105% load multiple bottleneck experiments. Like the summary statistics, these graphs show that performance is poor overall. But, again, Sync-TCP(MixReact) has slightly better performance than TCP Reno.

	TCP Reno	Sync-TCP(MixReact)
avg queue size at R0 (pckts)	54.9	50.5
avg queue size at R1 (pckts)	101.9	99.4
avg queue size at R3 (pckts)	73.3	73.2
% packet drops at R0	7.0	5.9
% packet drops at R1	27.6	26.3
% packet drops at R3	11.6	11.9
median rsptime (ms)	30570.0	24800.0
mean rsptime (ms)	67932.9	61304.7
goodput per response (kbps)	22.4	23.7
throughput (kbps)	8271.9	8045.2
goodput (kbps)	5213.0	5159.1
link utilization (kbps)	6031.1	5982.9

Table 4.22: 70% End-to-End Load, 105% Total Load, 3 Bottlenecks

	TCP Reno	Sync-TCP(MixReact)
avg queue size at R0 (pckts)	60.8	63.2
avg queue size at R1 (pckts)	97.5	92.3
avg queue size at R3 (pckts)	73.8	69.0
% packet drops at R0	11.6	11.2
% packet drops at R1	24.6	24.2
% packet drops at R3	8.5	7.9
median rsptime (ms)	31610.0	25120.0
mean rsptime (ms)	64934.3	58761.1
goodput per response (kbps)	21.6	24.2
throughput (kbps)	9477.6	9267.4
goodput (kbps)	6034.9	5963.6
link utilization (kbps)	6996.5	6916.1

Table 4.23: 80% End-to-End Load, 105% Total Load, 3 Bottlenecks

	TCP Reno	Sync-TCP(MixReact)
avg queue size at R0 (pckts)	73.9	71.0
avg queue size at R1 (pckts)	97.5	92.3
avg queue size at R3 (pckts)	71.1	64.6
% packet drops at R0	12.1	12.1
% packet drops at R1	26.6	24.3
% packet drops at R3	6.1	5.3
median rsptime (ms)	30560.0	24690.0
mean rsptime (ms)	59178.1	53392.8
goodput per response (kbps)	21.4	23.4
throughput (kbps)	9996.3	9822.8
goodput (kbps)	6535.1	6485.3
link utilization (kbps)	7521.4	7493.1

Table 4.24: 85% End-to-End Load, 105% Total Load, 3 Bottlenecks

	TCP Reno	Sync-TCP(MixReact)
avg queue size at R0 (pckts)	74.9	71.6
avg queue size at R1 (pckts)	98.9	92.3
avg queue size at R3 (pckts)	52.2	54.5
% packet drops at R0	11.1	10.9
% packet drops at R1	25.1	22.3
% packet drops at R3	3.1	3.5
median rsptime (ms)	24620.0	18850.0
mean rsptime (ms)	49272.6	41036.0
goodput per response (kbps)	21.4	24.5
throughput (kbps)	10187.7	9990.5
goodput (kbps)	6987.5	6947.9
link utilization (kbps)	8048.4	7998.8

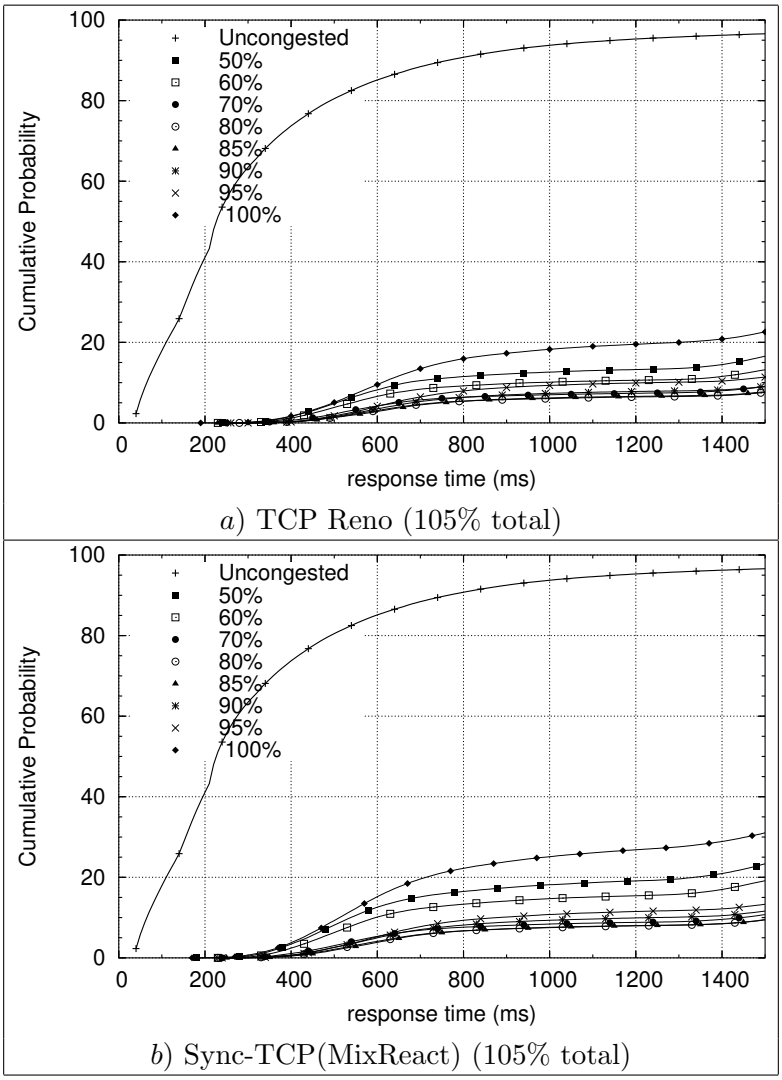
Table 4.25: 90% End-to-End Load, 105% Total Load, 3 Bottlenecks

	TCP Reno	Sync-TCP(MixReact)
avg queue size at R0 (pckts)	85.6	81.3
avg queue size at R1 (pckts)	102.0	96.8
avg queue size at R3 (pckts)	55.0	51.7
% packet drops at R0	12.6	12.4
% packet drops at R1	23.0	21.9
% packet drops at R3	2.8	2.7
median rsptime (ms)	18660.0	16650.0
mean rsptime (ms)	38570.0	34591.1
goodput per response (kbps)	20.8	23.1
throughput (kbps)	10695.3	10553.3
goodput (kbps)	7478.0	7415.5
link utilization (kbps)	8553.4	8485.0

Table 4.26: 95% End-to-End Load, 105% Total Load, 3 Bottlenecks

	TCP Reno	Sync-TCP(MixReact)
avg queue size at R0 (pckts)	82.2	72.9
avg queue size at R1 (pckts)	89.1	79.3
avg queue size at R3 (pckts)	51.3	47.7
% packet drops at R0	10.4	8.4
% packet drops at R1	12.6	10.3
% packet drops at R3	2.5	2.2
median rsptime (ms)	6850.0	6120.0
mean rsptime (ms)	17205.3	12219.1
goodput per response (kbps)	27.4	33.3
throughput (kbps)	10530.6	10240.2
goodput (kbps)	7931.6	7956.1
link utilization (kbps)	9068.7	9079.8

Table 4.27: 100% End-to-End Load, 105% Total Load, 3 Bottlenecks



a) TCP Reno (105% total)

b) Sync-TCP(MixReact) (105% total)

Figure 4.64: Response Time CDFs, Multiple Bottlenecks, 105% Total Load

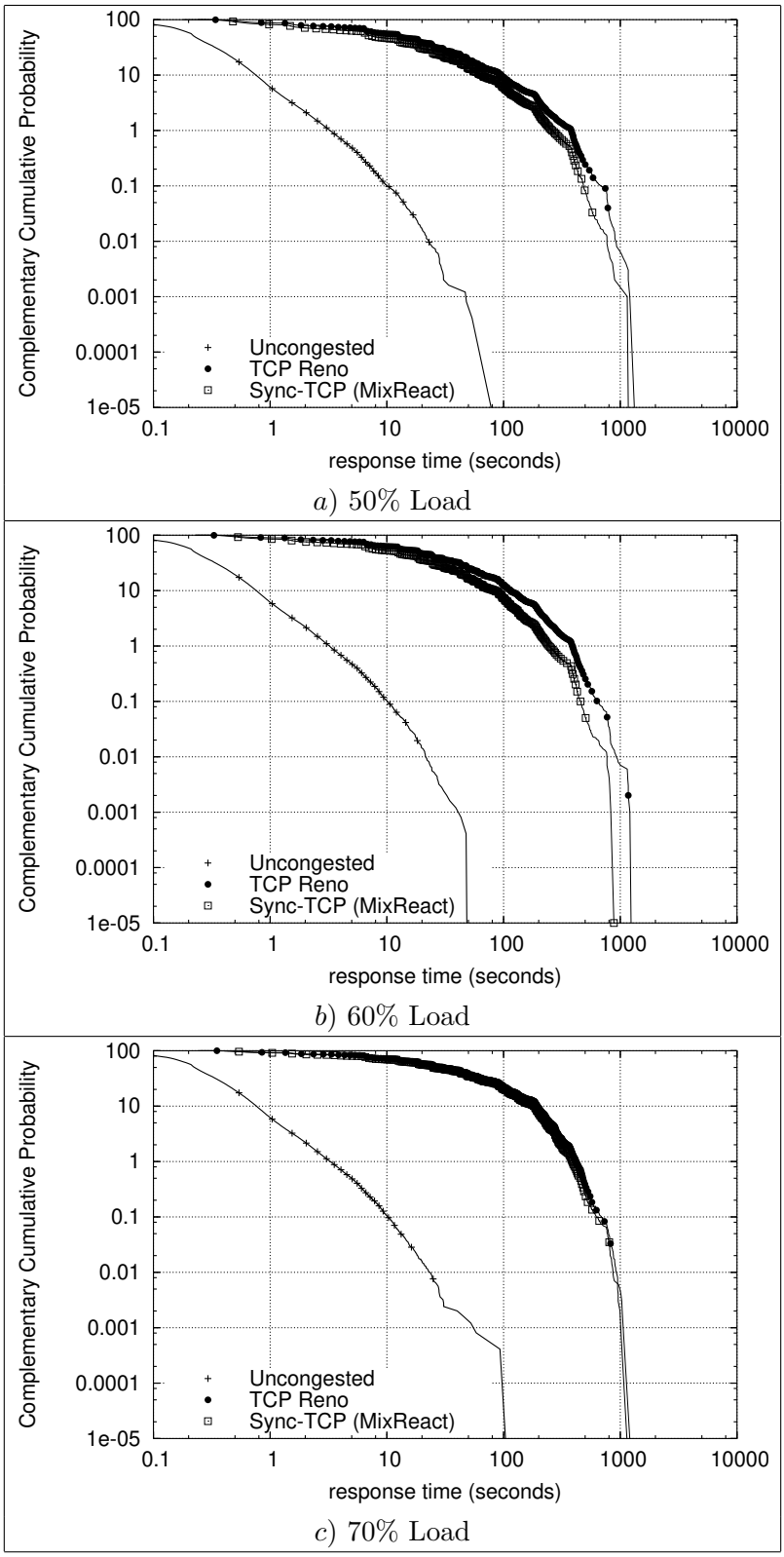


Figure 4.65: Response Time CCDFs, Total 105% Load, Light End-to-End Congestion

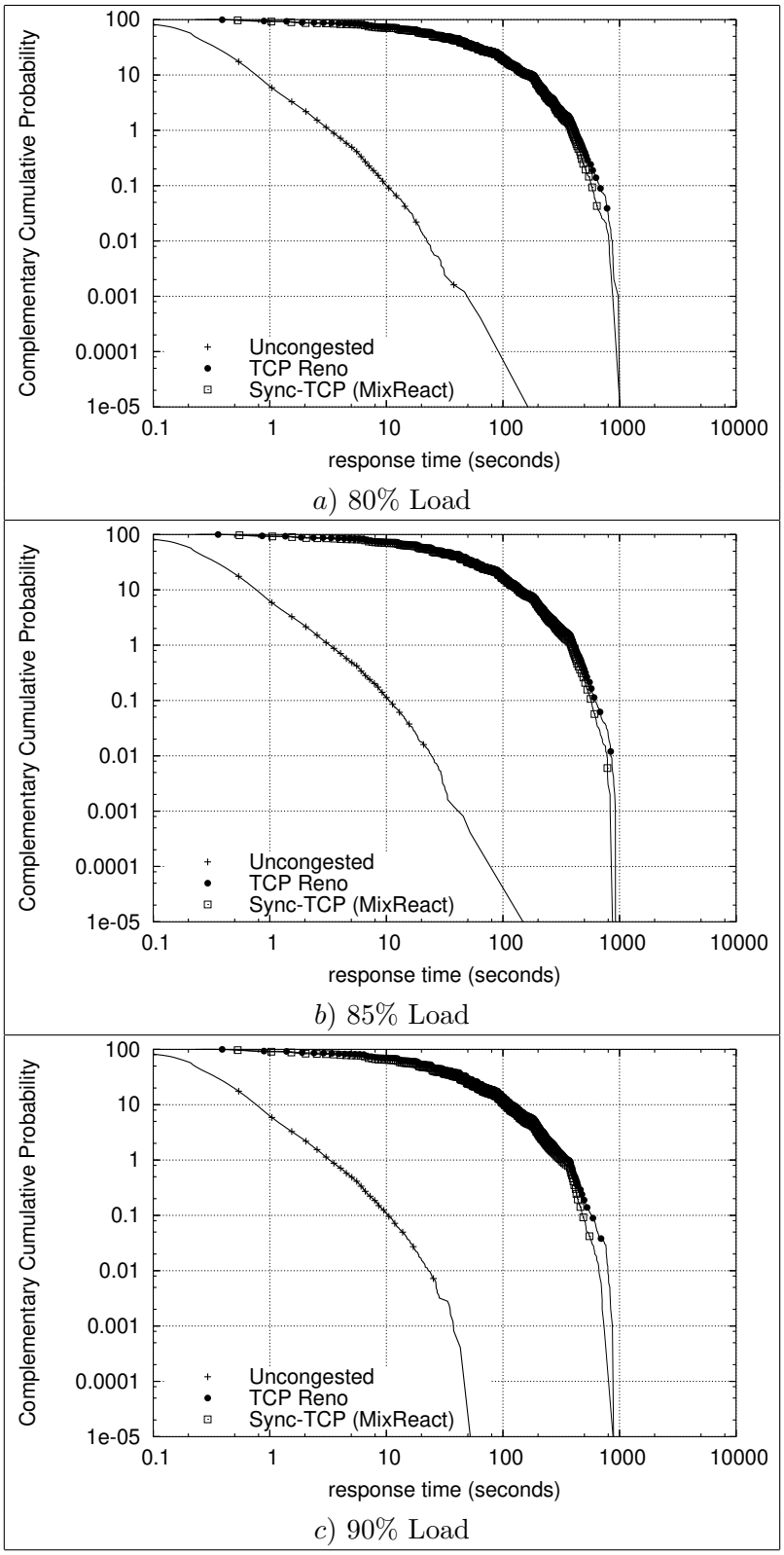


Figure 4.66: Response Time CCDFs, Total 105% Load, Medium End-to-End Congestion

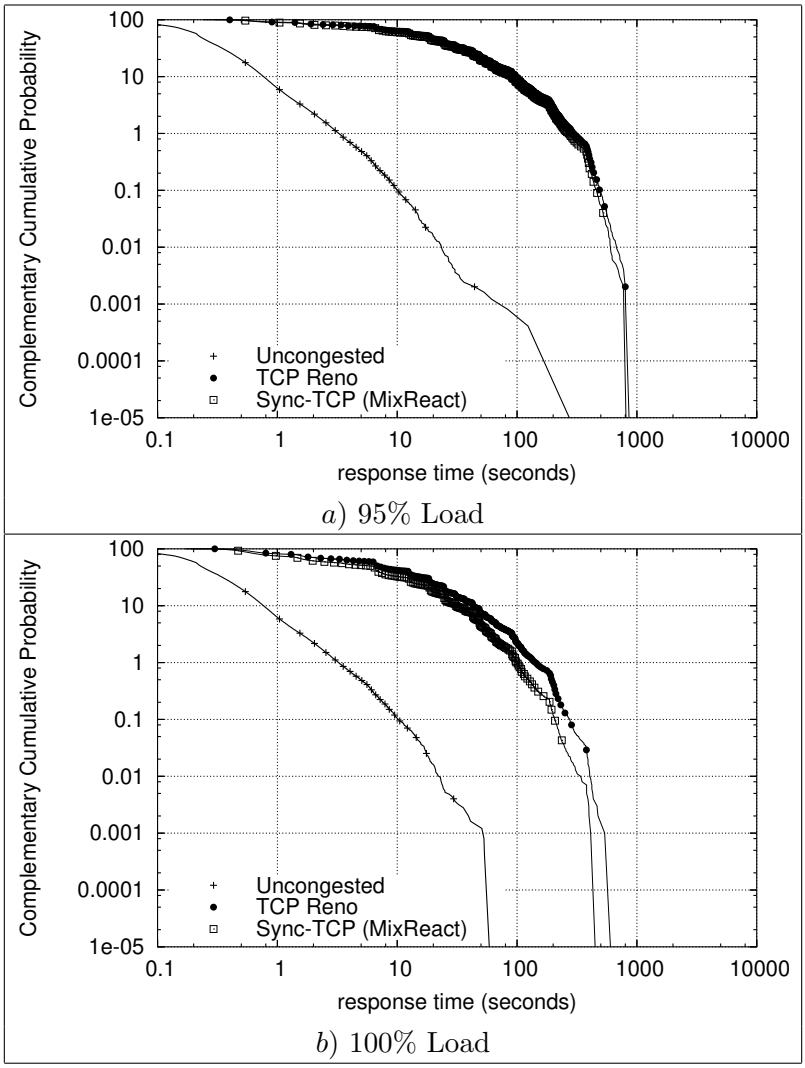


Figure 4.67: Response Time CCDFs, Total 105% Load, Heavy End-to-End Congestion

4.4 Summary

In Chapter 3, I introduced two Sync-TCP congestion control protocols: Sync-TCP(Pcwnd) and Sync-TCP(MixReact). In this chapter, I used experiments with FTP bulk transfers to examine the behavior of Sync-TCP(Pcwnd) and Sync-TCP(MixReact) as compared to TCP Reno. To evaluate the performance of the Sync-TCP protocols in a more complex environment, I tested them against TCP Reno over drop-tail routers and against TCP SACK over Adaptive RED routers with ECN marking in the context of HTTP traffic. I found that both of the Sync-TCP protocols resulted in less packet loss and smaller queue sizes at the bottleneck link than TCP Reno for both FTP and HTTP. Both Sync-TCP protocols had a larger percentage of HTTP responses complete in 1500 ms or less than either TCP Reno or SACK-ECN at load levels of 60-105% of link capacity. At 50% of link capacity, there is no difference among the protocols.

Sync-TCP(Pcwnd) performed well with HTTP traffic, lowering packet loss and queue sizes as compared to TCP Reno. Sync-TCP(Pcwnd) saw its best performance at higher loads when there were more flows that had experienced packet loss. Due to the design of the SyncPQlen congestion detection algorithm that Sync-TCP(Pcwnd) uses, no early congestion detection is performed until after a flow sees a packet drop. As the number of flows that use Sync-TCP(Pcwnd) congestion control increases, so does the performance of Sync-TCP(Pcwnd) relative to TCP Reno and Sync-TCP(MixReact). Thus, the performance of Sync-TCP(Pcwnd) is tied to the number of flows that experience packet loss.

Sync-TCP(Pcwnd) reduces *cwnd* by 50% at most once per RTT when the computed queuing delay is greater than 50% of the maximum-observed queuing delay. Flows that use this early congestion detection and reaction mechanism strive to keep the bottleneck queue under half-full. These flows will see fewer drops than if only TCP Reno was used, because the Sync-TCP(Pcwnd) flows will try to avoid overflowing the queue.

Sync-TCP(MixReact) achieves its performance improvement with only 5% of the completed flows (*i.e.*, flows that had a completed request-response pair) using its early congestion detection and reaction mechanisms. The other 95% of the flows are short enough to complete before nine ACKs arrive at the sender. Using Sync-TCP(MixReact) does not adversely affect the flows which use it except in the case where the flow avoided all packet loss. The 5% of flows using the Sync-TCP(MixReact) congestion control algorithms avoid packet loss by reacting to increases in queuing delay in a continuous manner and all flows benefit from the resulting smaller queues.

Sync-TCP(MixReact) uses a more complex congestion signal than either TCP Reno or Sync-TCP(Pcwnd) in order to have a fine-grained control of the congestion window. Depending upon the trend and magnitude of the average computed queuing delay, a flow will increase or decrease *cwnd* by 10%, 25%, or 50%. This allows flows to quickly increase their sending rates when the average queuing delay is relatively low and decreasing while also

quickly decreasing *cwnd* when the average queuing delay is relatively high and increasing. This fine-grained control allows flows to back off according to the severity of congestion. It also allows the flows to be more aggressive than TCP Reno in increasing *cwnd* in order to compensate for the additional *cwnd* decreases not associated with packet loss occurs. Sync-TCP(MixReact) allows flows to avoid packet loss by reacting to increasing congestion while making use of network resources when they are available.

Overall, the two Sync-TCP protocols perform better than the best standards-track TCP protocols. These protocols take advantage of synchronized clocks and the exact timing information of one-way transit times for early congestion detection. Both of these protocols can yet be improved, but still outperform the best standards-track AQM mechanism.

Chapter 5

Summary and Conclusions

Network congestion occurs when there is a long-term build-up in the size of queues in routers. Queues increase in size when the rate of incoming traffic is greater than the speed of the out-bound link. Congestion can result from connections sending data too quickly or too many new connections entering the network at the same time. Congestion often, but not always, leads to packet loss. If the finite queues fill to capacity, packets are no longer admitted to the queue and are dropped. Long-term congestion, where there are persistent large queues, may not lead to packet loss, but is still not a desirable situation. The longer a queue, the longer packets at the end of the queue must wait before being forwarded. This leads to large queuing delays, which degrades the performance of interactive applications.

Congestion control algorithms were developed to alleviate periods of congestion and avoid future occurrences of congestion collapse. The congestion control algorithms that have been proposed forced flows to carefully increase their sending rates in uncongested periods and to reduce their sending rates after incidents of packet loss. The goal of any congestion control algorithm is to maximize network utilization while minimizing data loss and queuing delay. To maximize network utilization, there should always be a backlog of data for a router to forward, so that the out-bound link is never idle. To minimize data loss, the router's queue should never overflow. To minimize queuing delay, the queue should be small. The ideal is for there always to be exactly one packet in the queue. Short-lived bursts of traffic are common occurrences in the Internet. These bursts do result in an increase in the size of network queues, but they are short-lived, and so, should not trigger a congestion indication (whether a congestion notification signal or packet loss). Buffers in routers should be large enough to allow for short bursts of incoming traffic, but should also limit the maximum amount of queuing delay when large queues persist.

Researchers have looked at the congestion control problem either by making changes to routers and leaving end systems untouched or by making changes only to end systems and treating the network as a "black box." Router-based mechanisms, such as active queue management (AQM), come from the idea that drop-tail routers are the problem. These

mechanisms make changes to the routers so that they notify senders when congestion is occurring. End-to-end approaches are focused on making changes to TCP Reno, rather than the drop-tail queuing mechanism. Most of these approaches try to detect and react to congestion earlier than TCP Reno (*i.e.*, before segment loss occurs) by monitoring the network using end-to-end measurements.

Some routers use AQM techniques to control the size of their queues over a long-term. AQM schemes like Random Early Detection (RED) monitor the queue size and decide to mark packets when the average queue size reaches certain thresholds. If a packet is chosen by RED to be marked, the packet is either dropped or a bit is set in its header. If the sender and receiver understand header-based congestion notifications, the packet is marked and forwarded. If not, the packet is dropped. In this way, the flow's sender gets an early notification that the queue is building up and congestion is occurring. The sender, then, can reduce its sending rate and help to reduce the queue size before the queue overflows. RED specifically is designed in such a way that short-lived traffic bursts do not result in congestion notifications.

Strictly end-to-end congestion control mechanisms rely only on the information that can be obtained by end systems alone. When a flow is competing with other flows for network resources, the congestion control algorithm must infer information about the network. For example, when a TCP Reno flow detects that a segment has been lost, it infers that a queue has overflowed. There is no explicit signal from the network letting the sender know that a segment has been dropped. Conversely, when a TCP Reno flow receives an ACK, it infers that the queues in intermediate routers did not overflow. There is no indication from the routers how full the queues are or if they overflowed and dropped segments from a different flow. TCP Reno relies on a binary signal of congestion – either segments are dropped, hence there is congestion, or there is no congestion. When there is no congestion, TCP Reno probes for additional bandwidth. This works toward the goal of increasing network utilization. When there is congestion, TCP Reno reduces its sending rate in order to minimize further data loss. Unfortunately, the only way for TCP Reno to detect congestion is through segment loss. So, TCP Reno strives to fill network buffers and then reacts only when they overflow.

Delay-based congestion control algorithms (DCA) monitor a flow's round-trip time (RTT) and react accordingly. An increase in RTT is interpreted as an increase in queuing delay. An increase in queuing delay means that queues are building up and congestion is likely to be occurring. Changes in delay are another data point, along with segment drops, that a congestion control algorithm can use to determine when congestion is occurring. DCA algorithms allow a sender to react to network congestion before segment loss occurs. The goal of these algorithms is to more efficiently use network resources by keeping smaller queues and avoiding segment loss.

Round-trip times are often not reliable sources of forward, or data, path queuing delays. Many Internet paths are asymmetric, where data segments face different congestion conditions (and may even traverse different links) than ACKs. The goal of congestion control algorithms is to help alleviate congestion on the forward path, where data segments travel. DCA algorithms could use the forward-path one-way transit times (OTTs) instead of RTTs in detecting the presence of congestion. But, in order to accurately measure OTTs, synchronized clocks are required on the sender and receiver.

In this dissertation, I studied methods for using synchronized clocks and OTTs in TCP congestion control. Synchronized clocks allow the sender and receiver to exchange accurate OTT information. These OTTs can then be used to estimate the queuing delay on the data path by subtracting the current OTT from the minimum-observed OTT. Once the queuing delay is determined, the congestion control algorithm can detect and react to periods of congestion.

In this dissertation, I have made the following contributions:

- I developed a method for measuring a flow's OTT and returning this exact timing information to the sender.
- I performed an empirical comparison of several methods for using OTTs to detect congestion.
- I developed Sync-TCP, an end-to-end congestion control mechanism based on using OTTs for congestion detection.
- I performed a study of standards-track TCP congestion control and error recovery mechanisms in the context of HTTP traffic.

5.1 Acquiring and Communicating One-Way Transit Times

In developing Sync-TCP, I assumed the presence of synchronized clocks. Since the evaluation was performed in simulation, this was a reasonable assumption. One way of actually acquiring synchronized clocks is to use the Global Positioning System (GPS) to receive time and position updates from the orbiting satellites.

I inserted a new timestamp option into the TCP header to carry the exact timing information obtained by the use of synchronized clocks. This option is carried in every segment (both data and ACK) and contains the time the segment was sent, the time the most recently received segment was sent, and the OTT of the most recently received segment. The receiver of a segment performs the calculation of the OTT by subtracting the time the segment was sent from the time the segment was received. In the next segment to be transmitted to the sender (usually an ACK), the receiver places this OTT, the timestamp found in the received segment's header, and the time this segment is to be sent. When this ACK is received, the sender can obtain the OTT experienced by the data segment, the time the data segment was received, and the OTT of the ACK itself.

I assume that the OTT accurately tracks the queuing delay and that there are no routing changes which would cause a change in the propagation delay. In a single bottleneck environment, the OTT does accurately track the queuing delay at the congested router. In an environment with multiple congested links, the OTT corresponds to the sum of the queuing delays at each router.

5.2 Congestion Detection

I looked at several methods of detecting congestion using queuing delays, which are based on OTTs:

- SyncPQlen (percentage of the maximum queuing delay) – reports congestion whenever the current queuing delay is greater than 50% of the maximum-observed queuing delay.
- SyncPMinOTT (percentage of the minimum OTT) – reports congestion whenever the current queuing delay is greater than 50% of the minimum-observed OTT.
- SyncAvg (average queuing delay) – reports congestion whenever the average queuing delay crosses a threshold.
- SyncTrend (trend analysis of queuing delays) – reports congestion when the trend of the queuing delays is increasing.
- SyncMix (trend analysis of average queuing delay) – reports both the direction of the trend of the average queuing delay and how close the average queuing delay is to the maximum-observed queuing delay.

I compared the above congestion detection mechanisms in the context of 20 FTP flows with no change in the reaction mechanism (*i.e.*, the TCP Reno algorithm was followed for changes to the congestion window). SyncMix is the combination of the best parts of the other algorithms and was found to provide the best congestion detection.

5.3 Congestion Reaction

I looked at two different congestion reaction mechanisms. The first mechanism, SyncPcwnd, performs the same adjustment to *cwnd* as TCP Reno upon the receipt of three duplicate ACKs and as an ECN-enabled sender upon the receipt of a congestion notification. Whenever congestion is detected, a sender using SyncPcwnd reduces *cwnd* by 50%. SyncPcwnd is designed to be used with a congestion detection mechanism that provides a binary signal of congestion. The second congestion reaction mechanism was specifically designed to be used with the SyncMix congestion detection method and is called SyncMixReact. SyncMixReact adjusts *cwnd* by setting α and β in the AIMD algorithm based on state of the network as reported by SyncMix. Since a sender using SyncMix and SyncMixReact is sensitive to increases in queuing delay, the sender should be allowed to increase its congestion window more aggressively than

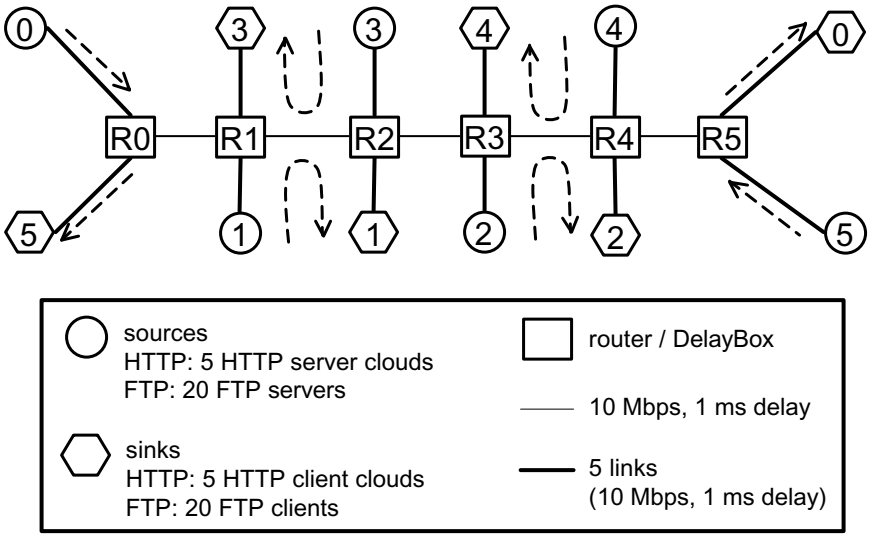


Figure 5.1: Simplified 6Q Parking Lot Topology

TCP Reno's congestion avoidance. In SyncMixReact, the maximum increase to *cwnd* occurs when the average queuing delay is between 0-25% of the maximum queuing delay and its trend is decreasing. When this occurs, α is set to $\frac{x_i(t)}{2}$, where $x_i(t)$ is the current congestion window. This increases the congestion window by 50% in one RTT. The maximum decrease occurs when the average queuing delay is between 75-100% of the maximum-observed queuing delay and the trend of the average queuing delay is increasing. SyncMixReact then sets β to 0.5, reducing the congestion window by 50%, which is the same adjustment that TCP Reno makes when a segment loss is detected by three duplicate ACKs.

5.4 Sync-TCP

Sync-TCP is a family of end-to-end congestion control mechanisms based on using OTTs for congestion detection. The combination of SyncPQlen congestion detection and SyncPcwnd congestion reaction is referred to as "Sync-TCP (Pcwnd)." Likewise, the combination of SyncMix congestion detection and SyncMixReact congestion reaction is referred to as "Sync-TCP (MixReact)."

5.4.1 Evaluation

The evaluation of Sync-TCP was performed in NS-2 simulations using a parking lot topology with six routers as illustrated in Figure 5.1. Each link was 10 Mbps, and congestion was created by adding cross-traffic. I looked at an uncongested network in order to calibrate the generated traffic. I also looked at a one bottleneck network where congestion was created by many HTTP flows entering the same router.

I evaluated Sync-TCP using both FTP and HTTP traffic. In all experiments, I used two-way traffic and delayed ACKs. For experiments with HTTP traffic, I used Bell Labs' PackMime model to generate HTTP traffic. In the case of a single congested link, each simulation was run until 250,000 HTTP request-response pairs were completed with an average intensity between 50-105% of link capacity. In the two congested link setup, I ran various levels of HTTP cross-traffic so that the total average loads on the congested links were 75%, 90%, and 105%. Because of the increase in simulation overhead and run time, the multiple congested link simulations with 75% and 90% load were run until 150,000 request-response pairs were completed. With 105% total load, 100,000 pairs were collected. For experiments with FTP traffic, 20 infinite bulk transfer streams that ran for 10 minutes were used. For the two congested link setup, 10 bulk transfer streams were used as cross-traffic.

For FTP traffic, I looked at the average throughput and loss percentage for each flow. For HTTP traffic, I looked at HTTP response times as the main performance metric. I also looked at throughput per response, loss rates, and queue size at the congested routers.

In the evaluation of Sync-TCP, I tested Sync-TCP (Pcwnd) and Sync-TCP (MixReact) against the best standards-track TCP protocols, TCP Reno over drop-tail routers and ECN-capable TCP SACK over Adaptive RED routers, referred to here as "SACK-ECN."

5.4.2 Results

For FTP, I found that both Sync-TCP(Pcwnd) and Sync-TCP(MixReact) resulted in less packet loss and lower queue sizes at the bottleneck link than TCP Reno. Additionally, for both Sync-TCP protocols, goodput per flow was similar to that of TCP Reno. With FTP traffic, 100% of the FTP flows encountered the conditions for both Sync-TCP protocols that caused the transition from slow start to the Sync-TCP early congestion detection phase. The exact timing information provided by synchronized clocks and OTTs allowed the flows to back off early to keep queues small and avoid loss, but still maintain comparable goodput to TCP Reno.

For HTTP traffic, there was no difference among the protocols at 50% load. At loads of 60-105% of link capacity, both Sync-TCP(Pcwnd) and Sync-TCP(MixReact) had better HTTP response time performance than both TCP Reno and SACK-ECN for response that completed in under 1500 ms.

With HTTP traffic, not all of the flows saw conditions that triggered the transition from slow start to the Sync-TCP early congestion phase. For Sync-TCP(Pcwnd) the number of flows that used early congestion detection varied with the load level. The trigger for Sync-TCP(Pcwnd) is segment loss, so as the load increased, the percentage of flows that experienced segment loss also increased. Once over 10% of the Sync-TCP(Pcwnd) experienced segment loss and used the early congestion detection mechanisms, response time performance improved relative to the other protocols. For Sync-TCP(MixReact), the trigger to use early congestion

detection is the receipt of nine ACKs. Since the response size distribution did not change with load level, the percentage of completed responses greater than 25 KB (*i.e.*, belonging to flows that received nine ACKs or more) also remained at 5% regardless of load level. The flows that used the early congestion detection mechanism were sensitive to increases in queuing delay and backed off to keep queues short. The other 95% of the flows were not long enough (since they generated under nine ACKs) to individually cause long-term congestion, so the longer flows did not suffer from the shorter flows consuming all of the newly available bandwidth when the longer flows backed off. All of the flows with Sync-TCP(MixReact) benefited from the lower queue sizes resulting from some flows backing off.

Overall, the two Sync-TCP protocols perform better than the best standards-track TCP protocols. The Sync-TCP protocols take advantage of synchronized clocks and the exact timing information of OTTs for early congestion detection. Even though both Sync-TCP(Pcwnd) and Sync-TCP(MixReact) have some problems that should be addressed, they still outperformed the best standards-track AQM mechanism.

5.5 Standards-Track TCP Evaluation

In Appendix B, I report the results of an extensive study of the current standards-track TCP protocols coupled with the most advanced standards-track AQM techniques. This study was performed to determine the best standards-track TCP protocol and queue management combination for HTTP traffic. This combination was then used as a point of comparison in evaluating Sync-TCP.

This study was carried out in much the same manner as the evaluation of Sync-TCP, using the same evaluation metrics. I studied the following combinations of end system congestion control and queue management:

- TCP Reno over drop-tail routers with a queue length of twice the bandwidth-delay product – "TCP Reno".
- ECN-enabled TCP SACK over Adaptive RED routers with a target delay of 5 ms and a queue length of five times the bandwidth-delay product (essentially an infinite queue) – "SACK-ECN (5 ms, qlen = 340)".
- ECN-enabled TCP SACK over Adaptive RED routers with a target queue size tuned to the average queue size of Sync-TCP (MixReact) for the specific traffic load level and a queue length of five times the bandwidth-delay product – "SACK-ECN (tuned, qlen = 340)".
- ECN-enabled TCP SACK over Adaptive RED routers with a target queue size tuned to the average queue size of Sync-TCP (MixReact) for the specific traffic load level and a queue length of twice the bandwidth-delay product - "SACK-ECN (tuned, qlen = 136)".

For HTTP traffic loads of 50-70% of the capacity of the bottleneck 10 Mbps link, SACK-ECN (5 ms, qlen = 340) provided the best performance in terms of average goodput per response and average queue size. For loads 85% and higher, SACK-ECN (tuned, qlen = 136) provided the best performance in terms of average packet loss at the bottleneck link, average goodput per response, and HTTP response times. At loads of 60-70%, there is a tradeoff between TCP Reno and SACK-ECN (5 ms, qlen = 340) in HTTP response time performance. With this tradeoff, a higher percentage of flows finish in a short amount of time with SACK-ECN than TCP Reno, but the performance of flows using TCP Reno improves as the response time increases. At 80% load, this tradeoff is present between TCP Reno and SACK-ECN (tuned, qlen = 136).

In total, there was no clear "best" protocol at all load levels. For this reason, the comparison to Sync-TCP was made against TCP Reno and SACK-ECN (5 ms, qlen = 340) for loads 50-70% and against TCP Reno and SACK-ECN (tuned, qlen = 136) for loads 80% and higher.

5.6 Future Work

From this dissertation, there are some clear directions for future work. These directions can be classified as further analysis of Sync-TCP, extensions to Sync-TCP, or additional uses for synchronized clocks.

5.6.1 Further Analysis

As in all evaluations, there is more analysis that can be performed. I would like to further investigate the accuracy of clock synchronization needed for Sync-TCP, the performance of Sync-TCP over many congested links, the performance of Sync-TCP over links with different levels of congestion, and the performance of Sync-TCP that competes with ECN-capable TCP Reno flows over Adaptive RED routers.

Accuracy of Clock Synchronization

In this work, I have assumed that end systems' clocks were perfectly synchronized. Future work could consider the threshold of clock synchronization before Sync-TCP congestion detection performs poorly. For instance, GPS clock synchronization promises sub-millisecond accuracy. Using GPS indoors is problematic, but one solution could be to use NTP to propagate the clock signal. It is reasonable to expect that clocks of computers on a WAN could be synchronized to within 1-2 ms if there were GPS time sources that could be used as NTP servers located on managed networks with low delay variability. If the threshold for Sync-TCP operation is low (meaning that the only a low degree of accuracy is needed), NTP could be a viable alternative to GPS synchronization, or relative delays may be able to used instead

of requiring synchronization and OTTs. Or, on paths that have a high probability of being symmetric, RTTs could be used.

Multiple Congested Links

The congestion signal that Sync-TCP monitors is queuing delay. The queuing delay that Sync-TCP uses on multiple congested links is the sum of all queuing delays on the path. As more congested links are introduced in the path, this signal gets more diluted. For example, if Q_i is the queuing delay experienced by packets of flow i, q_j is the queuing delay at router j, and N is the number of routers in the path, then $Q_i = \sum_{j=1}^{N} q_j$. As N grows, the contribution of each q_j diminishes. I would like to look at the maximum number of congested links and amount of queuing delay that could be tolerated before Sync-TCP congestion detection breaks.

Incremental Deployment of Sync-TCP

Note that in all of the evaluations of Sync-TCP, I assumed that all flows were operating under the same protocol. For example, I did not evaluate the behavior of any Sync-TCP flows competing against TCP Reno flows over drop-tail routers. The goal of TCP Reno congestion control is to overflow the network queues by continually probing for available bandwidth. The goal of Sync-TCP congestion control is to keep the queue relatively small. In a mixed environment, TCP Reno flows would continue to increase their sending rates as Sync-TCP flows backed off due to increased queuing delay. Therefore, the TCP Reno flows would see higher throughput than the Sync-TCP flows. Additionally, Sync-TCP flows competing with TCP Reno flows would see lower throughput than when competing solely with other Sync-TCP flows.

The inability of traditional delay-based congestion avoidance algorithms to coexist with TCP Reno and, therefore, be incrementally deployed has been explored by Martin [MNR00]. This result carries over to OTT-based algorithms such as Sync-TCP based on their sensitivity to queuing delays. Although the inability to be incrementally deployed is a shortcoming of delay-based congestion control algorithms, the study of these algorithms can offer insight into the nature and behavior of congestion and could point to new directions for future congestion control techniques. Additionally, Martin's work indicated that if a certain percentage of flows were sensitive to delays, overall network performance could be improved (*i.e.*, lower packet loss and queue sizes). I would like to investigate what percentage of Sync-TCP flows would be needed in an environment of TCP Reno over drop-tail routers to affect such an improvement.

Additionally, I would like to investigate if Sync-TCP flows could coexist with TCP Reno flows if all of the routers were running Adaptive RED. The Adaptive RED routers would try to keep their average queue sizes low by dropping packets, so the Sync-TCP flows may not always see the queuing delay increasing until the queues overflowed.

5.6.2 Extensions to Sync-TCP

Using the measurements provided by the Sync-TCP timestamp option, several extensions could be made to Sync-TCP to improve performance.

Estimate of Queuing Available in the Network

Sync-TCP(MixReact) has poor performance for long flows that do not experience segment loss (for example, as in the 50-60% load HTTP experiments). These flows do not have a good estimate of the maximum amount of queuing in the network and, if the maximum-observed queuing delay is less than 10 ms, they use a modified algorithm to determine the magnitude of the change in queuing delay returned by the SyncMix congestion detection mechanism. Both this algorithm and the 10 ms threshold do not allow the congestion window to grow adequately to take advantage of the fact that flows using Sync-TCP(MixReact) are backing off and avoiding packet loss. One approach could be to look at the number of ACKs (and, thus, OTT estimates) that have returned since the maximum-observed queuing delay was computed. If this count crossed a reasonable threshold, and thus, the maximum-observed queuing delay was stable, then the current maximum-observed queuing delay could be said to be a good estimate of the maximum queuing available in the network. Another approach would be to use trend analysis of the average queuing delay as the main congestion detection factor and use a standard reaction based on if the trend was increasing or decreasing. This would be very similar to SyncTrend congestion detection with a new congestion reaction mechanism.

ACK Compression

Senders could use the timestamps to detect and compensate for ACK compression (See section 4.1.2). This would prevent bursts of data segments resulting from ACK compression. Using Sync-TCP timestamps, a sender can determine the original spacing of the ACKs and increase its congestion window at that rate.

Delayed Acknowledgments

I have seen that not using delayed acknowledgments can cause increased loss rates from additional ACKs competing for buffer space with data packets. Using Sync-TCP, receivers could measure the OTTs and congestion on the ACK path. They could then perform ACK congestion control, which could help reduce the chance that ACKs were dropped and help alleviate congestion on the ACK path. The reduction in the frequency of delayed ACKs could improve performance, but the impact on the operation of TCP congestion control would need to be investigated. Perhaps, especially on short-delay links, an “ACK every other packet” policy is more aggressive than is needed.

Fairness

The Sync-TCP congestion reaction mechanism could be adjusted to use the exact RTTs as a basis for changes to the congestion window. This could improve fairness to flows that have long RTTs.

5.6.3 Additional Uses for Synchronized Clocks in TCP

Synchronized clocks could be useful in other parts of TCP. More accurate round-trip timers are one area of interest. Additionally, synchronized clocks on end systems could be used to report reliable delay statistics to time-critical interactive applications. These applications could then adapt to give the user the best performance based on current network conditions.

The application of Sync-TCP depends on the availability of synchronized clocks on end systems. With the increasing popularity of mobile devices, many of which are already equipped with GPS receivers, comes the opportunity to look for ways to use Sync-TCP to improve network performance for these devices. For example, in wireless environments, data transmission is susceptible to failures due to interference. Many times segments are lost because of these transmission errors rather than because of congestion. Basing congestion detection on queuing delay rather than segment loss could greatly improve performance in such environments. For example, segment losses that were not preceded by large delays could be assumed to be link errors rather than congestion errors. Sync-TCP currently uses segment loss as a signal that the maximum-observed queuing delay is a good estimate of the maximum amount of queuing in the network. If link errors in a wireless network occurred, the subsequent computed queuing delays probably would not be a good estimate of the maximum queuing in the network. So, instead of waiting for a segment drop, Sync-TCP could count the number of ACKs that have arrived since the maximum-observed queuing delay was computed and declare that the algorithm has a good estimate when the number of ACKs is significant. Wireless environments also bring up other issues, such as retransmissions triggered at the link level, which would increase delays, but not necessarily be due to congestion. There may be little that an end-to-end protocol can do to detect such occurrences, though.

5.7 Summary

This dissertation demonstrates that one-way transit times (OTTs), provided by the use of synchronized clocks, can be used to improve TCP congestion control. Two versions of Sync-TCP, a family of end-to-end congestion control protocols that take advantage of the knowledge of OTTs, have been shown to reduce packet loss rates and queue sizes over TCP Reno. In the context of HTTP traffic, flows see improved response time performance when all flows use one of the Sync-TCP protocols as opposed to TCP Reno.

Appendix A

Methodology

In this appendix, I present additional details of the evaluation of Sync-TCP described in Chapter 4. In Section A.1, I discuss decisions made in the design of the network configuration. In Section A.2, I discuss the PackMime web traffic model and characteristics of the traffic generated using the PackMime model. Section A.3 describes the challenges in running simulations involving HTTP traffic, which has components modeled with heavy-tailed distributions. Finally, in Section A.4, I describe the additions I made to *ns* to implement per-flow base RTTs and web traffic generation based on the PackMime traffic model.

A.1 Network Configuration

Initial evaluations of network protocols often use a simple traffic model: a small number of long-lived flows with equal RTTs, data flowing in one direction, ACKs flowing in the opposite direction, and every segment immediately acknowledged. These scenarios may be useful in understanding and illustrating the basic operation of a protocol, but they are not sufficiently complex to be used in making evaluations of the potential performance of one protocol versus another on the Internet, which is an extremely complex system. Networking researchers cannot directly model the Internet, but should use reasonable approximations when designing evaluations of network protocols [PF97].

A.1.1 Evaluating TCP

Allman and Falk offer guidelines for evaluating the performance of a set of changes to TCP, such as Sync-TCP [AF99]. These guidelines include the following :

- *Use delayed ACKs.* A receiver using delayed acknowledgments will wait to see if it has data to transmit to the sender before sending an ACK for received data. If the receiver has data to transmit before the timeout period (usually 100 ms) or before there are two unacknowledged data segments, the ACK is piggy-backed with the outgoing data segment bound for the sender. Delayed ACKs are commonly used in the Internet and should be used in TCP evaluations. **Delayed ACKs are used in the Sync-TCP evaluations.**
- *Set the sender's maximum send window to be large.* A sender's congestion window cannot grow larger than its maximum send window, so to assess the effect of a congestion

control protocol on the sender's congestion window, the congestion window should not be constrained by the maximum send window. The bandwidth-delay product is the maximum amount of data that can be handled by the network in one RTT. **The sender's maximum send window is set to the bandwidth-delay product in the Sync-TCP evaluations.**

- *Test against newer TCP implementations.* TCP Reno is a popular implementation of TCP, but performance of new TCP protocols should be compared against the performance of the latest standards-track TCP implementations, such as TCP with selective acknowledgments (SACK) and Explicit Congestion Notification (ECN). **Sync-TCP is evaluated against TCP Reno and ECN-enabled TCP SACK.**
- *Set router queue sizes to at least the bandwidth-delay product.* The maximum number of packets that a router can buffer affects maximum queuing delay that packets entering the router experience. **The router queue sizes are set to two times the bandwidth-delay product in the Sync-TCP evaluations.**
- *Test with both drop-tail and Random Early Detection (RED) queuing mechanisms.* Although most routers in the Internet use drop-tail queue management, RED (or some variant of RED) active queue management is becoming increasing popular. **Sync-TCP over drop-tail routers is compared to TCP Reno over drop-tail routers and ECN-enabled TCP SACK over Adaptive RED routers.**
- *Use competing traffic.* Most traffic in the Internet is bi-directional (*e.g.*, TCP is inherently two-way). In the case of TCP, data segments in one direction flow alongside ACKs for data flowing in the opposite direction. Therefore, ACKs flow on the same paths and share buffer space in the same queues as data segments. This has several implications [Flo91b, ZSC91]. First, since buffers in routers are typically allocated by IP packets, rather than bytes, an ACK occupies the same amount of buffer space as a data segment. Second, since ACKs can encounter queues, several ACKs could arrive at a router separated in time, be buffered behind other packets, and then be sent out in a burst. The ACKs would then arrive at their destination with a smaller inter-arrival time than they were sent. This phenomena is called *ACK compression*. ACK compression could lead to a sudden increase in a sender's congestion window and a bursty sending rate. Third, since there could be congestion on a flow's ACK path, ACKs could be dropped. Depending on the size of the sender's congestion window, these ACK drops could be misinterpreted as data path drops and retransmissions would be generated. This could be detrimental to a sender, which would lower its sending rate in response to the perceived segment drop. These types of complexities occur in the Internet and, therefore, are useful in simulations. **Two-way competing traffic is used in the Sync-TCP evaluations.**
- *Use realistic traffic.* Much of the traffic in the Internet is HTTP traffic, which typically contains a large number of short-lived flows and a non-negligible number of very long-

	One-Way		Two-Way	
segments	260,218		446,129	
data	260,156	(100%)	229,021	(51%)
ACK	62	(0%)	217,108	(49%)
drops	6181		11,169	
data	6178	(100%)	7106	(64%)
ACK	3	(0%)	4063	(36%)
utilization	99.83%		90.30%	
goodput	9983.11	kbps	8722.81	kbps

Table A.1: Performance of One-Way FTP Traffic and Two-Way FTP Traffic Without Delayed ACKs

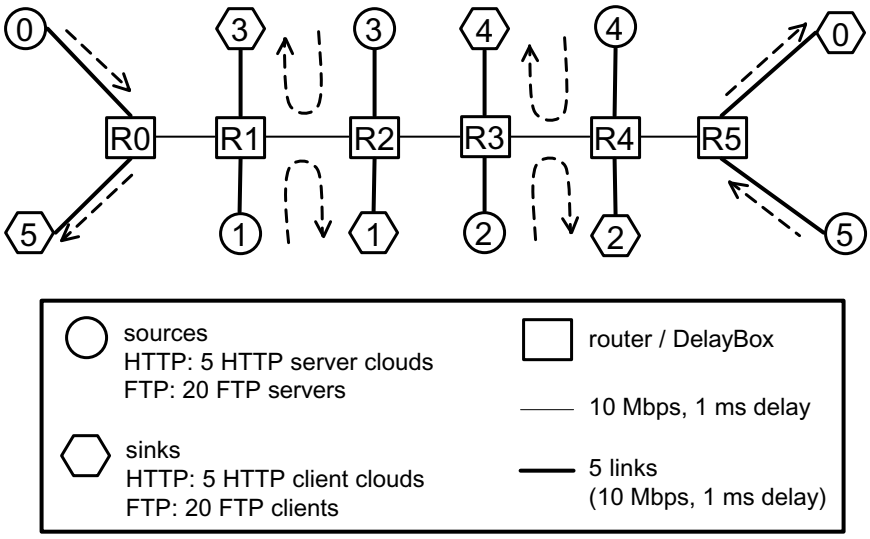


Figure A.1: Simplified 6Q Parking Lot Topology

lived flows. **A recent HTTP traffic model is used to generate web traffic in the Sync-TCP evaluations.**

A.1.2 Impact of Two-Way Traffic and Delayed ACKs

With two-way traffic and no delayed acknowledgments, ACKs make up almost half of all segments sent over the forward path. The addition of ACKs on the forward path increases total segment drops over that of one-way traffic. Table A.1 shows the results of an experiment with 20 FTP bulk-transfers without delayed ACKs flowing in one direction ("one-way") and 20 FTP bulk-transfers flowing in each direction ("two-way") over the parking lot topology (Figure A.1). With two-way traffic, ACKs make up half of the segments sent over the bottleneck link and a significant portion of the total segment drops. Link utilization is lower with two-way traffic than with one-way traffic because ACKs make up half of the packets

	No Delayed ACKs		Delayed ACKs	
segments	446,129		363,553	
data	229,021	(51%)	229,778	(63%)
ACK	217,108	(49%)	133,775	(37%)
drops	11,169		5685	
data	7106	(64%)	4149	(73%)
ACK	4063	(36%)	1537	(27%)
utilization	90.30%		90.59%	
goodput	8722.81	kbps	8868.83	kbps

Table A.2: Performance of Two-Way FTP Traffic Without and Without Delayed ACKs

that the router on the bottleneck link forwards. Thus, two-way traffic creates quantitatively and qualitatively different congestion dynamics and hence is important to understand.

When using two-way traffic, ACKs for reverse-path data segments compete with forward-path data segments for space in the bottleneck router's queue. Using delayed acknowledgments reduces the frequency of ACKs, limiting them to an average of one ACK for every two segments received. Table A.2 compares the results from 20 two-way FTP bulk-transfers without delayed ACKs (as described above) with 20 two-way FTP bulk-transfers with delayed ACKs (the "TCP Reno" experiments described in Chapter 4). On a congested link, using delayed ACKs lowers the number of packets drops without reducing goodput as compared to an ACK-per-packet policy.

A.2 HTTP Traffic Generation

The web traffic generated in the evaluation of Sync-TCP is based on a model of HTTP traffic, called PackMime, developed at Bell Labs [CCLS01b]. In this section, I discuss PackMime, characteristics of the traffic generated by PackMime, and how PackMime is used to generate programmable levels of congestion. Later, in section A.4.1, I will describe the implementation of PackMime in *ns*.

A.2.1 PackMime

PackMime is based on the analysis of HTTP 1.0 connections from a trace of a 100 Mbps Ethernet link connecting an enterprise network of approximately 3,000 hosts to the Internet [CLS00, CCLS01a]. HTTP 1.0 requires a separate TCP connection for each HTTP request-response pair. In other words, only one HTTP request and its response are carried in a TCP connection. The fundamental parameter of PackMime is the HTTP connection initiation rate (which, for HTTP 1.0, is also the TCP connection initiation rate). The PackMime model also includes distributions of base RTTs, the size of HTTP requests, and the size of HTTP responses.

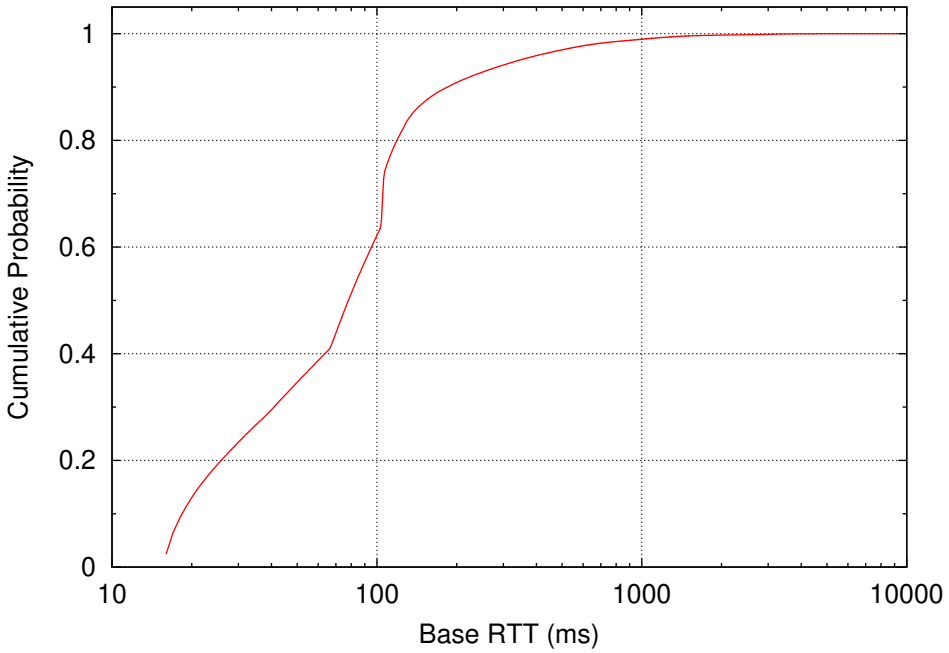


Figure A.2: HTTP Round-Trip Times

PackMime specifies when new HTTP requests should be made. When the time comes for a new request, the request size and server for the client to contact are chosen according to the distributions in PackMime. When a server receives a request, the response size and the server delay time are chosen. Once the server delay time has passed (simulating the server processing the request), the server will transmit the response back to the client.

The RTTs generated by the PackMime model range from 1 ms to 9 seconds. The topology used in the Sync-TCP evaluation (Figure A.1) adds additional propagation delay of 14 ms for end-to-end flows (1 ms on each link in each direction). With the added propagation delay, the base RTTs range from 15 ms to 9 seconds. The mean RTT is 115 ms, and the median RTT is 78 ms. The CDF of the RTTs is presented in Figure A.2.

Both the request size distribution and the response size distribution are heavy-tailed. Figure A.3 shows the request size CDF for 85% load. There are a large number of small request sizes and a few very large request sizes. Over 90% of the requests are under 1 KB and fit in a single segment. The largest request is over 1 MB.

Figures A.3 and A.4 show the CDF and CCDF of the response sizes for 85% load. In the PackMime model, the body of the response size distribution is described empirically, and the tail is a Pareto ($\alpha = 1.23$) distribution. (The Pareto distribution is an example of a heavy-tailed distribution.) About 60% of the responses fit into one 1420-byte segment, and 90% of the responses fit into 10 segments, yet the largest response size is almost 50 MB. Using these

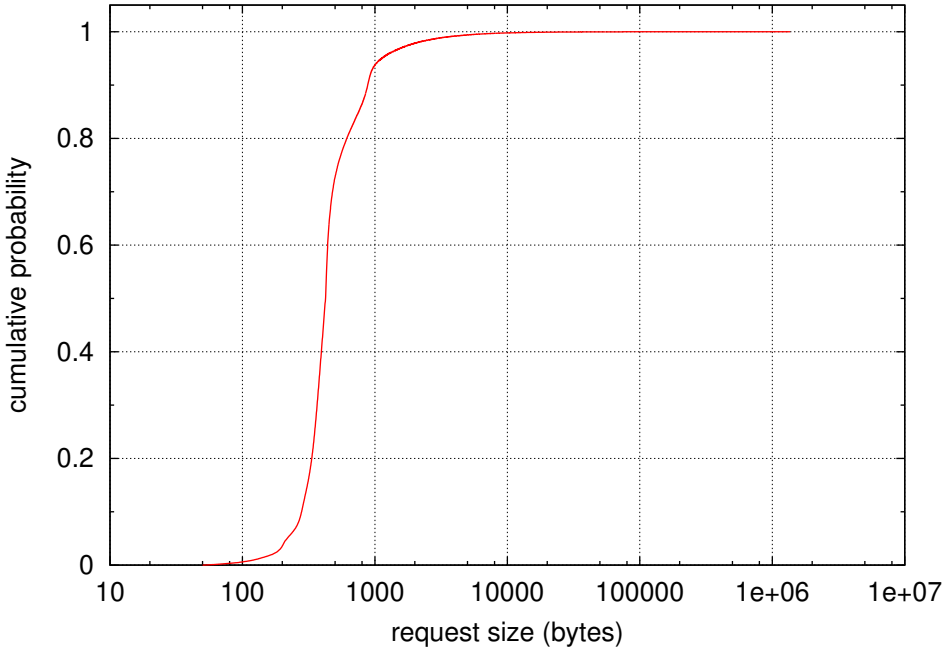


Figure A.3: CDF of HTTP Request Sizes

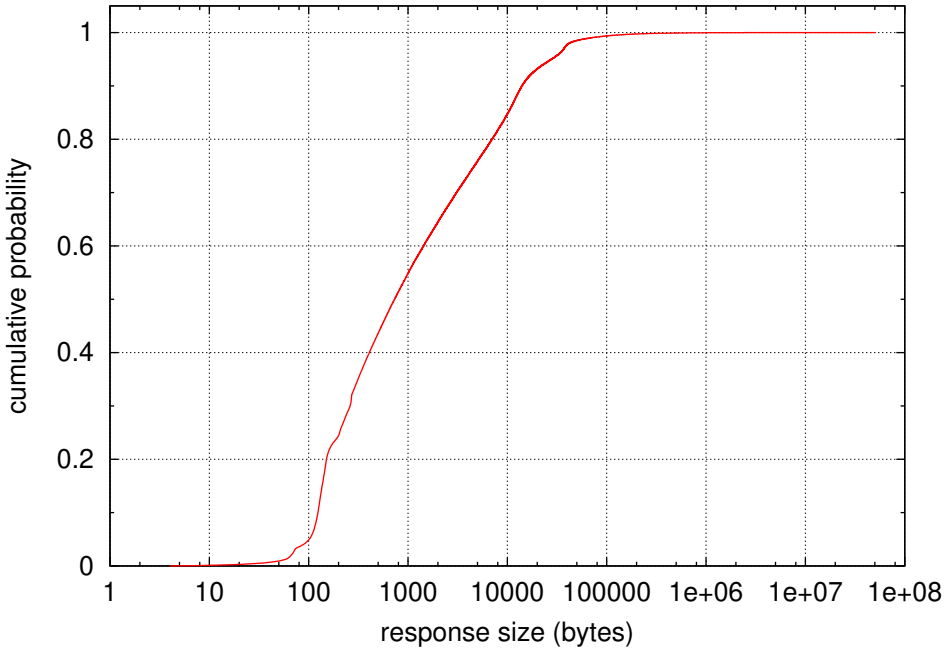


Figure A.4: CDF of HTTP Response Sizes

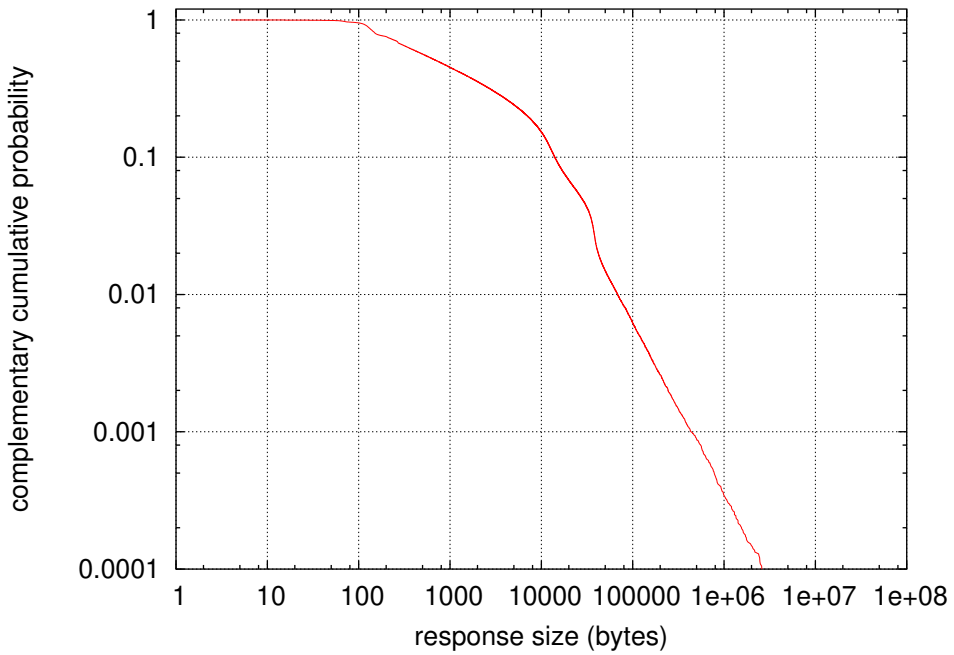


Figure A.5: CCDF of HTTP Response Sizes

distributions to generate web traffic will result in many short-lived transfers, but also some very long-lived flows.

I implemented PackMime traffic generation in *ns* using Full-TCP, which includes bi-directional TCP connections, TCP connection setup, TCP connection teardown, and variable segment sizes. The experimental network consists of a number of PackMime "clouds." Each PackMime cloud represents a group of HTTP clients or servers. The traffic load produced by each PackMime cloud is driven by the user-supplied connection rate parameter, which is the average number of new connections starting per second, or the average HTTP request rate. The connection rate corresponding to each desired link loading was determined by a calibration procedure described below. New connections begin at their appointed time, whether or not any previous connection has completed. Thus, the number of new connections generated per second does not vary with protocol or level of congestion, but only with the specified rate of new connections. For illustration, in Figure A.6, I show the number of new connections per second started by five PackMime clouds for a resulting average load of 85% of the 10 Mbps bottleneck. This was generated by specifying a connection rate of 24.1 new connections per second on each of the five PackMime clouds.

A.2.2 Levels of Offered Load

The levels of offered load used in the evaluation of Sync-TCP are expressed as a percentage of the capacity of a 10 Mbps link. I initially ran the network with all links configured at 100

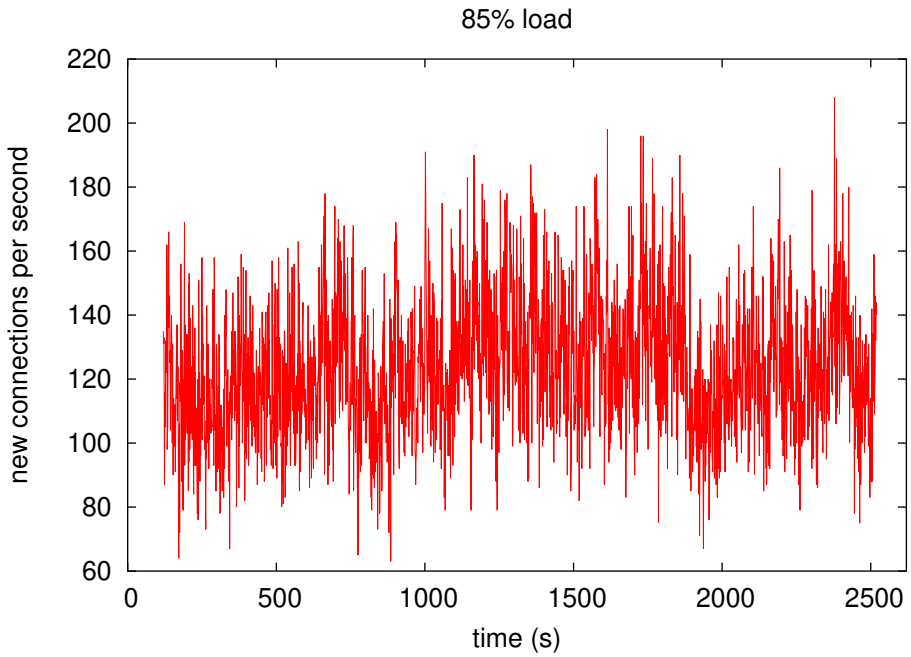


Figure A.6: New Connections Started Per Second

Mbps (for an uncongested network) to determine the PackMime connection rates that would result in average link utilizations (in both forward and reverse directions) of 5, 6, 7, 8, 8.5, 9, 9.5, 10, and 10.5 Mbps. The connection rate that results in an average utilization of 8.5% of the uncongested 100 Mbps link will be used to generate an offered load on the 10 Mbps link of 8.5 Mbps, or 85% of the 10 Mbps link. Note that this "85% load" (*i.e.*, the connection rate that results in 8.5 Mbps of traffic on the 100 Mbps link) will not actually result in 8.5 Mbps of traffic on the 10 Mbps link. The bursty HTTP sources will cause congestion on the 10 Mbps link, and the actual utilization of the link will be a function of the protocol and router queue management scheme used. Note, also, that this 8.5 Mbps of traffic on the uncongested link is an average over a long period of time. For example, Figure A.7 shows the offered load per second at the 85% level on an unconstrained network. Notice that just before time 1000, there is a large burst in the offered load. This burst is typical of HTTP traffic and is included in the average load that results in 8.5 Mbps. This is just an illustration that although I will refer to 85% load, the load level as time passes is not uniform. The burst in the offered load corresponds to a transfer of a 4.5 MB file followed about 30 seconds later by a 49 MB file. Figure A.8 shows the same 85% load on a constrained 10 Mbps network. After the large transfer begins, the number of bytes transmitted per second increases greatly and does not drop below 6 Mbps until the transfer completes around time 1600.

Table A.3 shows the HTTP connection rates per PackMime cloud required to generate the desired load on the forward and reverse paths. Note that these connection rates are dependent

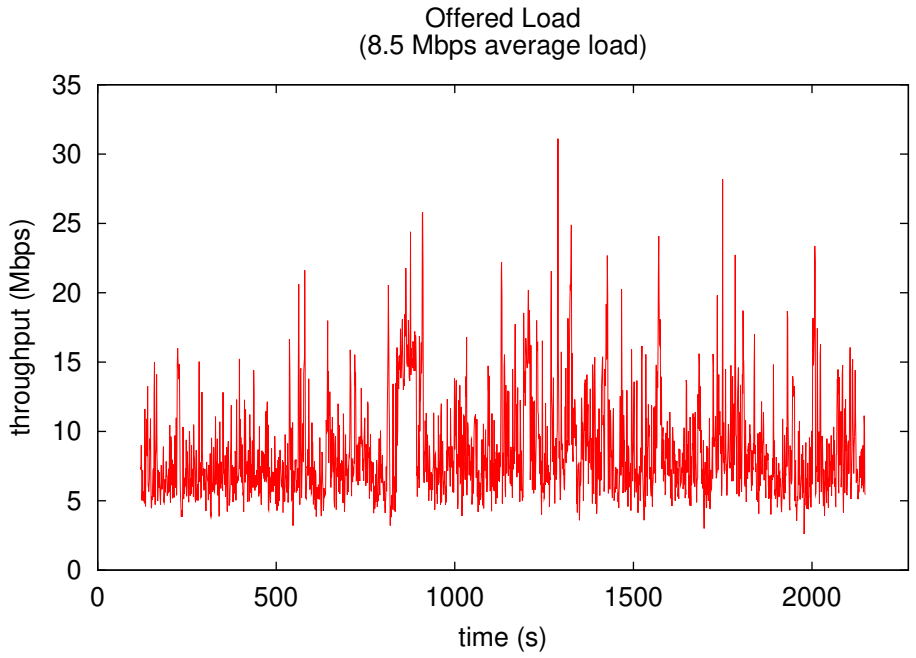


Figure A.7: Offered Load Per Second

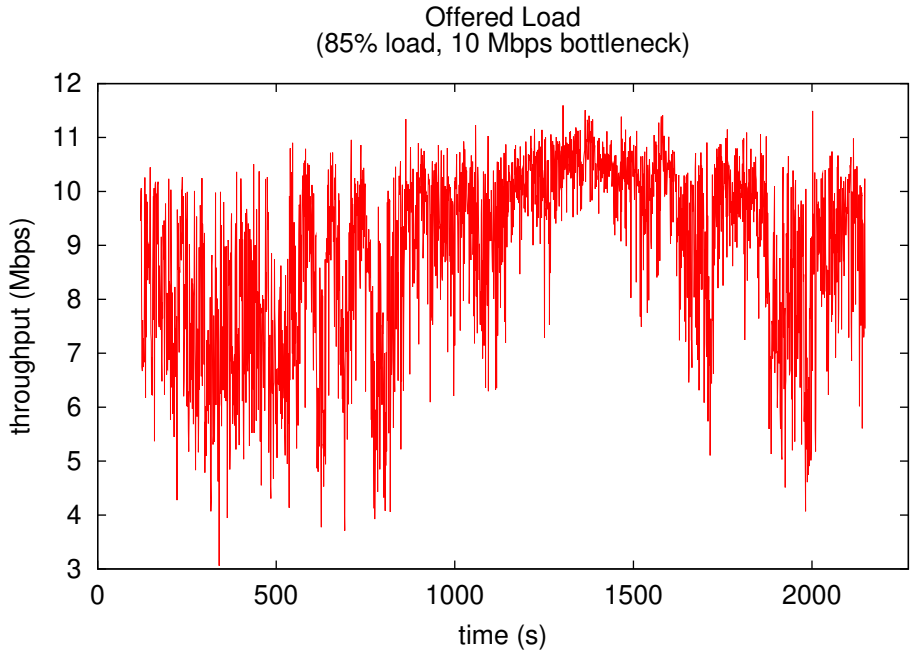


Figure A.8: Offered Load Per Second on a Congested Network

% Load	Forward Path c/s	Reverse Path c/s	Duration (s)
50	14.2	13	3600
60	16.9	15.7	3300
70	19.9	18.7	3000
80	22.7	21.5	2400
85	24.1	22.9	2400
90	25.4	24.2	2100
95	26.9	25.7	2100
100	28.2	27	1920
105	29.6	28.4	1800

Table A.3: HTTP End-to-end Calibration

	75% Total Load		90% Total Load		105% Total Load	
End-to-End % Load	Link 12 c/s	Link 34 c/s	Link 12 c/s	Link 34 c/s	Link 12 c/s	Link 34 c/s
50	7.7	7.2	11.5	11.4	15.3	15.5
60	4.7	4.1	8.7	8.6	12.7	12.5
70	1.5	1.3	6.3	5.7	10.2	9.8
80	-	-	3.3	2.7	7.7	7.2
85	-	-	1.5	1.3	6.3	5.7
90	-	-	-	-	4.7	4.1
95	-	-	-	-	3.3	2.7
100	-	-	-	-	1.5	1.3
105	-	-	-	-	-	-

Table A.4: HTTP Cross-Traffic Calibration

upon the seed given to the pseudo-random number generator. Using a different seed would cause different requests and response sizes to be generated at different times which would affect the offered load. The table also gives the simulation length needed for 250,000 HTTP request-response pairs to complete. The simulation run length will be discussed further in section A.3.

For the multiple bottleneck HTTP experiments, the amount of cross-traffic to generate also had to be calibrated. Table A.4 shows the forward-path connection rates that correspond to the desired load of cross-traffic.

A.2.3 Characteristics of Generated HTTP Traffic

Previous studies have shown that the size of files transferred over HTTP is heavy-tailed [BC98, Mah97, PKC96, WP98]. With heavy-tailed file sizes, there are a large number of small files and a non-negligible number of extremely large files. These studies have also shown that the arrival pattern of web traffic is self-similar, which exhibits long-range dependence, or

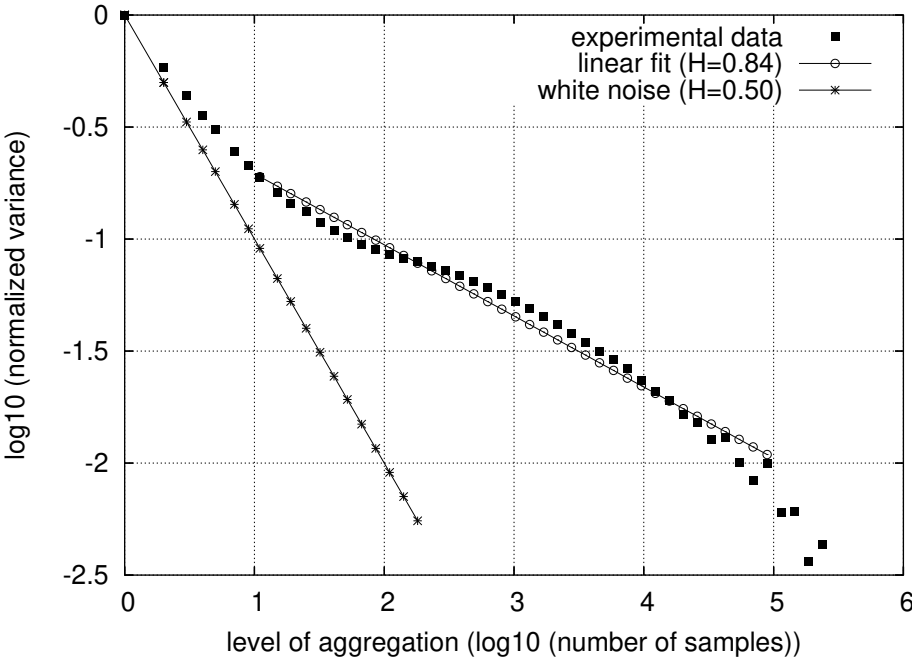


Figure A.9: Packet Arrivals

burstiness over several time scales. In other words, as the aggregation interval increases, the rate of packet arrivals are not smoothed out (as it would be, for example, if the arrivals were Poisson). Figure A.9 demonstrates the long-range dependence of the packet arrivals in the PackMime model[1]. The Hurst parameter characterizes the amount of long-range dependence in a time series. A Hurst parameter of 0.5 is white noise, meaning that there is no long-range dependence, while a Hurst parameter greater than 0.8 means that there is strong long-range dependence. The PackMime model exhibits long-range dependence with a Hurst parameter of 0.84.

A.3 Simulation Run Time

Typically, simulations are run for as long as it takes the sample metric of interest to reach within a certain range of the long-run metric. I am interested in looking at throughput, goodput, and HTTP response times over many HTTP connections. *Throughput* is defined as the amount of data per second sent from a sender to a receiver. I measure throughput by counting the number of bytes (both data and ACKs) transferred every second on the link before the bottleneck router. Because throughput includes dropped segments and retransmissions, it may exceed the bottleneck link bandwidth. *Goodput* is defined as the amount of data per second received by a receiver from a sender. I measure goodput by counting the number of

[1]The data in the figure was computed using methods developed by Taqqu [Taq].

data bytes that are received by each receiver every second. Goodput measures only the segments that are actually received and so does not include segment losses. On an uncongested network, the goodput is roughly equal to the throughput minus any ACKs. *Response times* are application-level measurements of the time between the sending of a HTTP request and the receipt of the full HTTP response.

To assess appropriate simulation length, I will run my experiments on an uncongested network and so will look at only throughput and response times (since goodput is approximately the same as throughput in the uncongested case). A major component of both throughput and response time is the size of the HTTP response. The larger a response, the more data a server tries to put on the network in a small amount of time, which will increase throughput. As response sizes grow, so do the number of segments needed to carry the data, which increases response times.

The PackMime model, used for HTTP traffic generation, has a HTTP response size distribution where the body is described empirically and the tail is a Pareto ($\alpha = 1.23$) distribution. The Pareto distribution is an example of a heavy-tailed distribution, where there are many instances of small values and a few, though non-negligible, instances of very large values. Pareto distributions with $1 \leq \alpha \leq 2$, such as the PackMime response size tail, have infinite variance, meaning that the variance never converges even with very large sample sizes. Crovella and Lipsky show that when sampling from heavy-tailed distributions it can take a very long time for statistics such as the mean to reach steady state [CL97]. For example, to achieve two-digit accuracy for the mean of a Pareto distribution with $\alpha = 1.23$, over 10^{10} samples are required[2]. Even then, the mean is still rather unstable, because of "swamping" observations that can double the mean with just one sample. The probability that a swamping observation could occur in a simulation using a Pareto ($\alpha = 1.23$) distribution is greater than 1 in 100. For these reasons, Crovella and Lipsky state that when $\alpha < 1.5$, simulation convergence becomes impractical. In determining the run length in my experiments, I look at the HTTP response sizes, so in this case, each sample is a HTTP response size. Assuming that the desired load is 85%, there are about 120 requests per second on the forward path. Assuming that for 120 requests per second there are 120 responses per second, generating 10^{10} HTTP responses would take over 80 million seconds, or more than 2 1/2 years, of simulation time.

Figure A.10 shows the running mean response size for 250,000 samples at 85% load, with the theoretical mean[3] of 7560 bytes marked. (This figure corresponds to the response size CDF and CCDF in Figures A.4 and A.5.) Gathering 250,000 response size samples takes about 40 simulated minutes and six hours in real-time. The sample mean is far from converging to the

[2]The sample mean, A_n, of a Pareto distribution converges as $|A_n - \mu| \sim n^{1/\alpha - 1}$, where n is the number of samples. For k-digit accuracy, $n \geq c\, 10^{\frac{k}{1-1/\alpha}}$. If we let $c = 1$, $k = 2$, and $\alpha = 1.23$, $n \geq 10^{10}$.

[3]The theoretical mean for the PackMime HTTP response size distribution was provided by Jin Cao of Bell Labs.

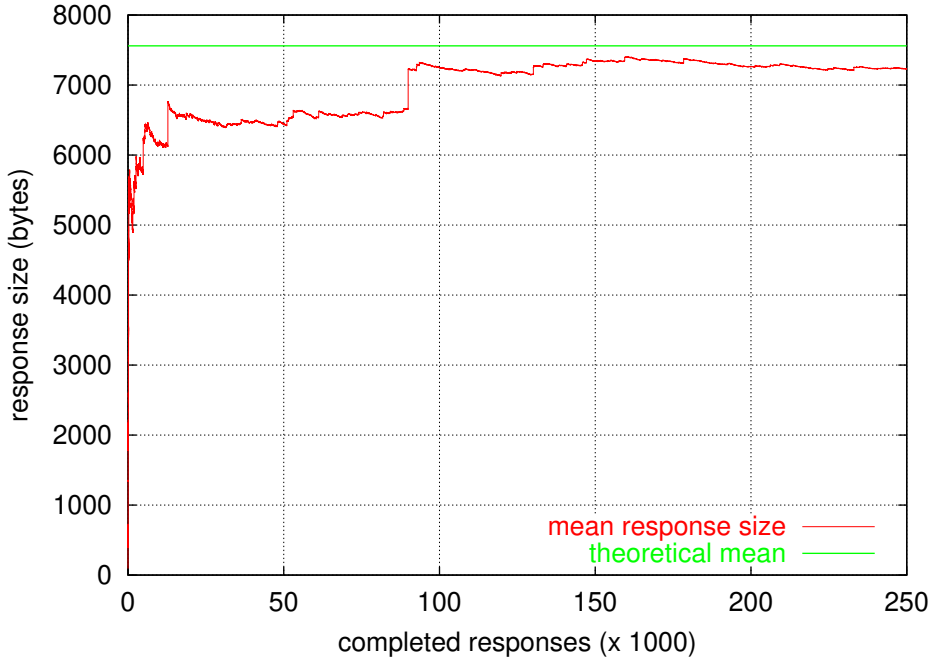


Figure A.10: Mean Response Size

theoretical mean. Therefore, I do not expect to see the mean HTTP response size converge in my simulations.

Since traditional measures like confidence intervals on means will not work, other criteria must be used. Huang offered a suggestion for determining simulation run length: to obtain the 90th quantile of true throughput, run the simulation for at least as long as it takes the 90th quantile of the response size to converge, though no metric for convergence was offered [Hua01]. (By definition, 90% of the values are less than the 90th quantile.) Figure A.11 shows the running 90th quantiles of HTTP response sizes for 250,000 completed responses. After 100,000 samples, the range of the 90th quantiles is less than 100 bytes. To determine how much the 90th quantile is converging, I computed the coefficient of variation (COV). The COV describes how much the standard deviation of the metric, in this case, the 90th quantile of response size, is changing in relation to the mean. Figure A.12 shows the COV of the 90th quantiles of response size[4]. A COV of 5% or less is considered to be stabilizing. The COV of the 90th quantile of response size remains under 3% after 100,000 samples, so at this point, the 90th quantile of response size is converging.

I could also look at the mean and quantiles of throughput and response time, since these are the metrics I am actually interested in measuring. It is a "rule of thumb" in networking research that workloads based on heavy-tailed distributions produce long-range dependent traffic [Par00]. This long-range dependence will appear in the throughput and response

[4] $COV = \frac{\sigma}{\mu}$. The COV is computed every 100 responses.

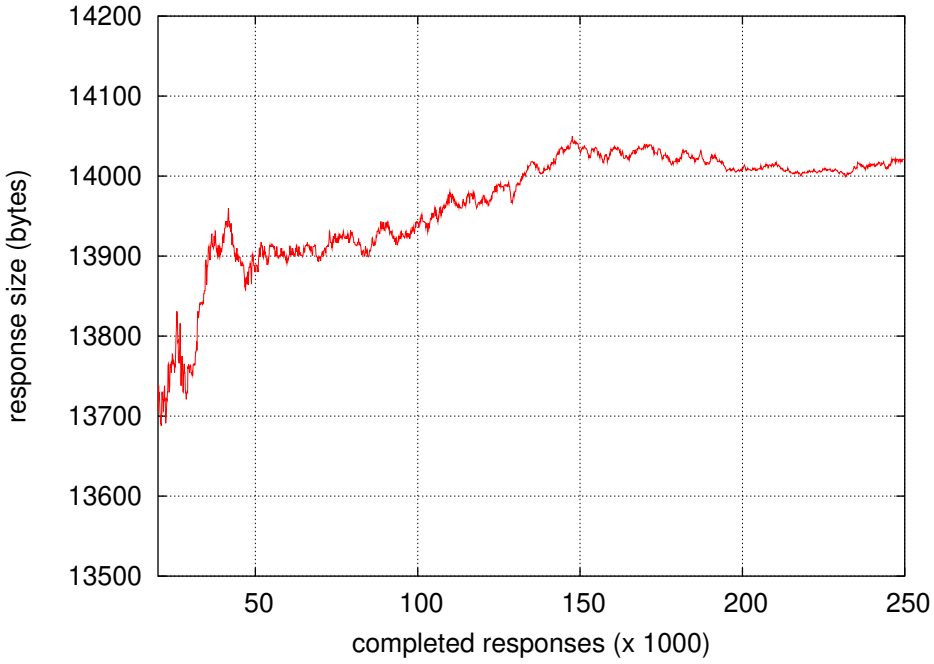


Figure A.11: 90th Quantiles of Response Size

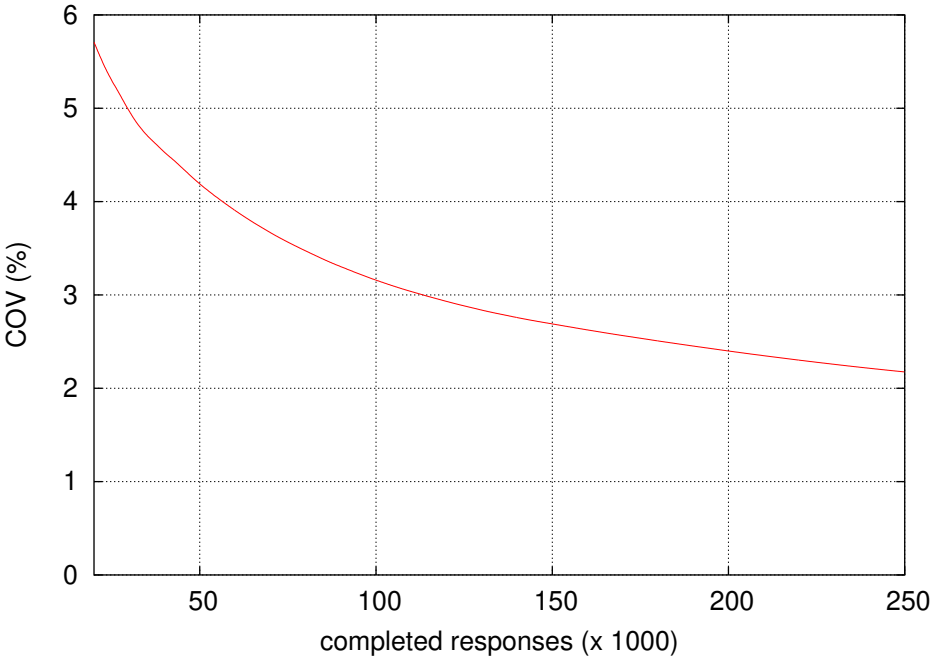


Figure A.12: Coefficient of Variation of 90th Quantiles of Response Size

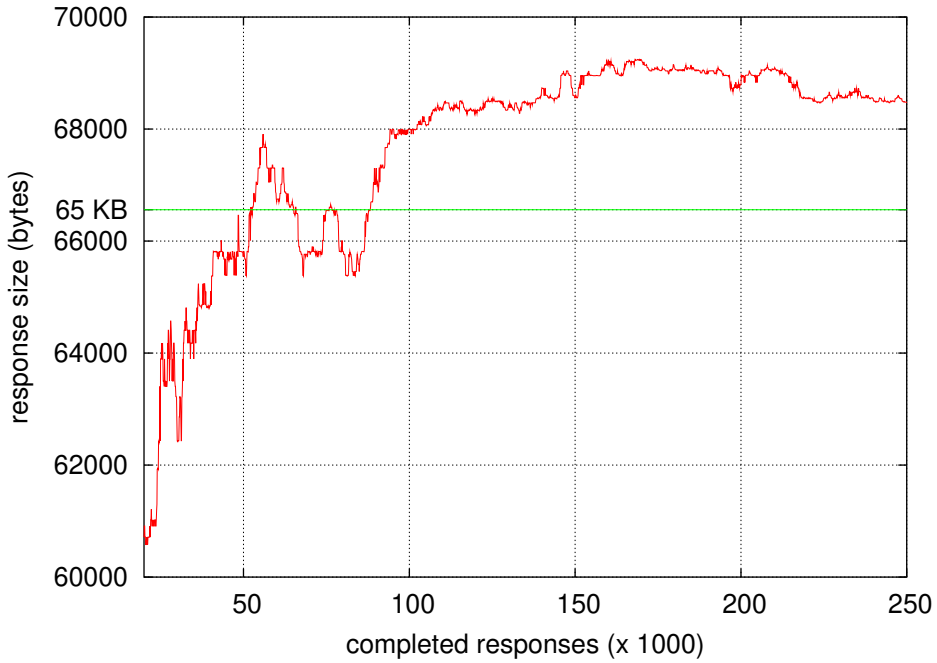


Figure A.13: 99th Quantiles of Response Size

time data. Whereas the response size values are independently sampled from a distribution, throughput and response times are measured values which exhibit dependence. Further, these metrics are protocol- and network-dependent. If I choose the simulation run length based upon throughput or response time, whenever the network setup changed, the simulation run length would have to be re-evaluated.

Operationally, I want to run the simulations long enough to see a mix of "typical" HTTP traffic. This includes small transfers as well as very large transfers. The longer the simulation runs, the more large transfers complete. The sizes of the largest responses can be determined by looking at the 99th quantile of response size. Figure A.13 shows the 99th quantiles of response size for 85% load. After 100,000 completed responses, 1% of all responses are greater than 66,560 bytes (65 KB). This may not seem like a very large response, but it is almost two orders of magnitude larger than the median response size, which is about 730 bytes.

From the 90th quantile of response size heuristic and the large response size heuristic, it looks to be sufficient to run the simulation for 100,000 responses. I will be more conservative and require that 250,000 responses be completed (when CPU and memory resources are abundant for simulation). To ensure that each run has a wide range of response sizes, I also require that the 99th quantile of response size be greater than 65 KB. Also, the maximum-sized response that has begun transfer should be greater than 10 MB. This maximum-sized response is not required to complete, because with heavy loads of traffic there will be much congestion and many packet drops. With the congestion, this maximum-sized response may

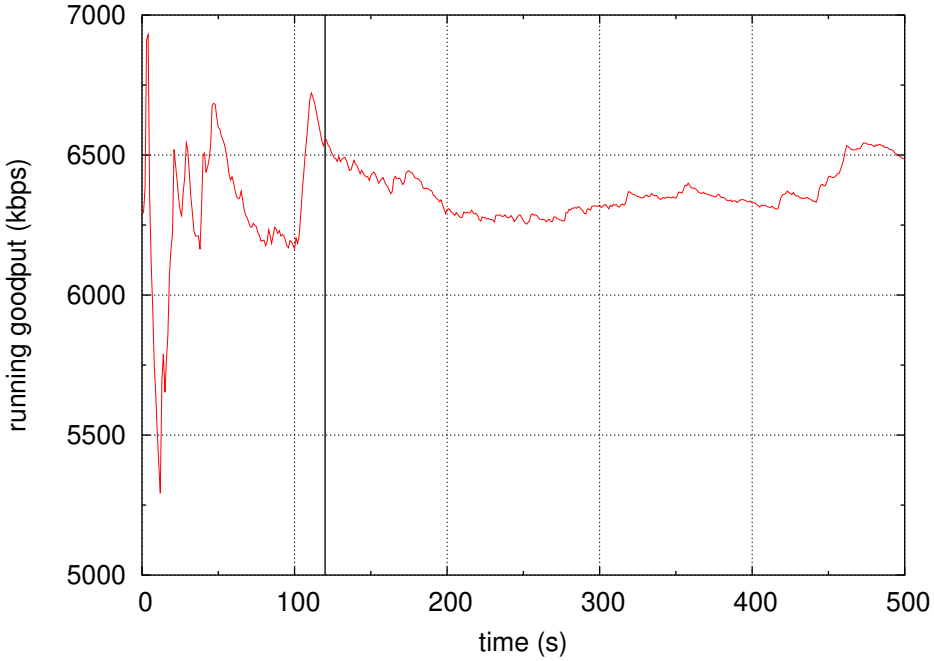


Figure A.14: HTTP Warmup Interval

not complete before 250,000 other responses complete. Even though the maximum-sized response does not complete, its transfer still contributes to the congestion in the network.

In some situations, especially with heavy levels of congestion and multiple bottlenecks, completing 250,000 request-response pairs takes a very long running time (over 6 hours in real-time) and a very large amount of memory (over 500 MB). In these situations, only 100,000-150,000 completed request-response pairs will be required. These numbers of completed request-response pairs still meet the requirements of the heuristics described above.

Before collecting any data in the simulation, including counting completed request-response pairs, a warmup interval of 120 seconds was used. Figure A.14 shows the running goodput of HTTP traffic at 85% load on an uncongested network. The vertical line at time 120 marks the beginning of data collection for the experiments. The running goodput after this point is much less bursty than before time 120.

Ending the simulation before the statistics reach steady-state leaves the HTTP traffic in a transient state. When evaluating the performance of a protocol, I will always compare it against a baseline, which will have been run in the same transient state. When comparing two protocols head-to-head, I will use the same pseudo-random number seeds. This will ensure that the request sizes, response sizes, request generation rate, and link propagation delays remain constant between the two variants. Thus, any observed difference in performance will be isolated to the protocol itself.

A.4 Simulator Additions

To perform the evaluation of Sync-TCP in *ns* with a recent HTTP traffic model, I had to add features not currently available in *ns*. These additions include an implementation of PackMime and a mechanism, called DelayBox, to add per-flow delays and packet drops based on given probability distributions.

A.4.1 PackMime

The PackMime traffic model describes HTTP traffic on a single link between a cloud of nearby web clients and a cloud of distant web servers. Load is specified as the number of new connections that become active per second. A connection consists of a single HTTP 1.0 request and response pair. The PackMime traffic model was developed by analyzing actual traffic traces gathered from a single link between a Bell Labs campus and the Internet. The following connection variables are used to describe traffic in the model:

- *Connection interarrival time* - The time between consecutive HTTP requests.
- *HTTP request size* - The size of the HTTP request that the client sends.
- *HTTP response size* - The size of the HTTP response that the server sends.
- *Server delay* - The time between a server receiving an HTTP request and sending the HTTP response.
- *Server-side transit time* - The round-trip time between a server in the server cloud and the monitored link (located near the client).
- *Client-side transit time* - The round-trip time between a client in the client cloud and the monitored link (located near the client).

PackMime-NS is the *ns* object that drives the simulation of the PackMime traffic model. The network simulator *ns* has objects called *Applications* that control data transfer in a simulation. These Applications communicate via *Agents* which represent the transport layer of the network (*e.g.*, TCP). Each PackMime-NS object is attached to a *ns* node and controls the operation of two types of Applications: PackMime-NS servers and PackMime-NS clients. Each of these Applications is connected to a Full-TCP[5] Agent.

PackMime-NS is meant to be used with DelayBox (to be described in section A.4.2), which controls delay and loss functions. Because DelayBox will control network delay, server-side and client-side transit times are not included in PackMime-NS.

PackMime-NS Implementation

In *ns*, each PackMime cloud is represented by a single *ns* node and is controlled by a single PackMime-NS object. This node can produce and consume multiple HTTP connections at

[5]Full-TCP, as opposed to one-way TCP, includes bi-directional TCP connections, TCP connection setup, TCP connection teardown, and variable segment sizes.

a time. For each connection, PackMime-NS creates new server and client Applications and their associated TCP Agents. After setting up and starting each connection, PackMime-NS schedules the time for the next new connection to begin.

PackMime-NS handles the re-use of Applications and Agents that have finished their data transfer. There are five *ns* object pools used to cache used Application and Agent objects – one pool for inactive TCP Agents and one pool each for active and inactive client and server Applications. The pools for active Applications ensure that all active Applications are destroyed when the simulation is finished. Active TCP Agents do not need to be placed in a pool because each active Application contains a pointer to its associated TCP Agent. New objects are only created when there are no Agents or Applications available in the inactive pools.

For each connection, PackMime-NS creates (or allocates from the inactive pool) two TCP Agents and then sets up a connection between the Agents. PackMime-NS creates (or allocates from the inactive pools) server and client Applications, attaches an Agent to each, and starts the Applications. Finally, PackMime-NS schedules a time for the next new connection to start.

PackMime-NS Client Application A PackMime-NS client Application controls the request size of the transfer. Its order of operations is as follows:

- Sample the HTTP request size from the distribution.
- Send the request to the server.
- Listen for and receive the HTTP response from the server.
- When the response is complete, stop.

PackMime-NS Server Application A PackMime-NS server Application controls the response size of the transfer. Its order of operations is as follows:

- Listen for an HTTP request from a client.
- Sample the HTTP response size from the distribution.
- Sample server delay time from the distribution and set a timer.
- After the timer expires (*i.e.*, the server delay time has passed), send the HTTP response back to the client.
- Once the response has been sent, stop.

PackMime Random Variables

This implementation of PackMime-NS provides several *ns* Random Variable objects for specifying distributions of PackMime connection variables. The implementations were taken from code provided by Bell Labs and modified to fit into the existing *ns* Random Variable framework. This allows PackMime connection variables to be specified by any type of

ns Random Variable, which now includes the PackMime-specific Random Variables. The PackMime-specific Random Variable syntax is as follows:

- `RandomVariable/PackMimeFlowArrive <rate>`
- `RandomVariable/PackMimeFileSize <rate> <type>`
- `RandomVariable/PackMimeXmit <rate> <type>`, where `<type>` is 0 for client-side and 1 for server-side (to be used with DelayBox instead of PackMime-NS).

PackMime-NS Commands

PackMime-NS takes several commands that can be specified in the TCL simulation script:

- `set-client <node>` - Associate the node with the PackMime client cloud.
- `set-server <node>` - Associate the node with the PackMime server cloud.
- `set-rate <float>` - Specifies the average number of new connections per second.
- `set-flow_arrive <RandomVariable>` - Specifies the connection interarrival time distribution.
- `set-req_size <RandomVariable>` - Specifies the HTTP request size distribution.
- `set-rsp_size <RandomVariable>` - Specifies the HTTP response size distribution.
- `set-server_delay <RandomVariable>` - Specifies the server delay distribution.

A.4.2 DelayBox

DelayBox is a *ns* node that is placed in between source and destination nodes and can also be used as a router. With DelayBox, segments from a flow can be delayed, dropped, and forced through a bottleneck link before being passed on to the next node. DelayBox requires the use of Full-TCP flows that are assigned unique flow IDs.

DelayBox maintains two tables: a rule table and a flow table. Entries in the rule table are added by the user in the TCL simulation script and specify how flows from a source to a destination should be treated. The fields in the rule table are as follows:

- source node
- destination node
- delay Random Variable (in ms)
- loss rate Random Variable (in fraction of segments dropped)
- bottleneck link speed Random Variable (in Mbps)

The loss rate and bottleneck link speed fields are optional.

Entries in the flow table are created internally and specify exactly how segments from each flow should be handled upon entering the DelayBox node. The flow table's values are obtained by sampling from the distributions given in the rule table. The fields in the flow table are as follows:

- source node
- destination node
- flow ID
- delay (in ms)
- loss rate (in fraction of segments dropped)
- bottleneck link speed (in Mbps)

Flows are defined as beginning at the receipt of the first SYN of a new flow ID and ending after the sending of the first FIN. Segments after the first FIN are not delayed or dropped.

DelayBox also maintains a set of queues to handle delaying segments. There is one queue per entry in the flow table (*i.e.*, the queues are allocated per-flow). These queues are implemented as delta queues, meaning that the actual time to transmit the segment is kept only for the segment at the head of the queue. All other segments are stored with the difference between the time they should be transmitted and the time the previous segment in the queue should be transmitted. The actual time the first bit of the packet at the tail of the queue should be transmitted is stored in the variable *deltasum*, named so because it is the sum of all delta values in the queue (including the head segment's transfer time). This value is used in computing the delta for new packets entering the queue. If the bottleneck link speed has been specified for the flow, a processing delay is computed for each segment by dividing the size of the segment by the flow's specified bottleneck link speed. The propagation delay due to the bottleneck link speed is also factored into the new packet's delta value.

When a segment is received by a DelayBox node, its transfer time (current time + delay) is calculated. (This transfer time is the time that the first bit of the segment will begin transfer. Segments that wait in the queue behind this segment must be delayed by the amount of time to transfer all bits of the segment over the bottleneck link.) There are two scenarios to consider in deciding how to set the segment's delta:

- If the segment is due to be transferred before the last bit of the last segment in the queue, its delta (the time between transferring the previous segment and transferring this segment) is set to the previous segment's processing delay. This segment has to queue behind the previous segment, but will be ready to be transmitted as soon as the previous segment has completed its transfer.
- If the segment is due to be transferred after the last bit of the last segment in the queue, its delta is difference between its transfer time and the previous segment's transfer time.

If the current segment is the only segment in the queue, DelayBox schedules a timer for the transfer of the segment. When this timer expires, DelayBox will pass the segment on to the standard packet forwarder for processing. Once a segment has been passed on, DelayBox will look for the next segment in the queue to be processed and schedule a timer for its transfer. All segments, both data and ACKs, are delayed in this manner. Segments that should be

dropped are neither queued nor passed on. All segments in a queue are from the same flow and are delayed the same amount (except for delays due to segment size) and are dropped with the same probability.

In the evaluation of Sync-TCP, no loss rate or bottleneck link speed was used. Losses in the evaluation were only caused by congestion on the actual bottleneck link.

DelayBox Commands

DelayBox takes several commands that can be specified in the TCL simulation script:

- `add-rule <src node id> <dst node id> <delay Random Variable> <loss rate Random Variable> <bottleneck link speed RandomVariable>`
  - Add a rule to the rule table, specifying delay, loss rate, and bottleneck link speed for segments flowing from *src* to *dst*.
- `add-rule <src node id> <dst node id> <delay Random Variable> <loss rate Random Variable>`
  - Add a rule to the rule table, specifying delay and loss rate for segments flowing from *src* to *dst*.
- `list-rules` - List all rules in the rule table.
- `list-flows` - List all flows in the flow table.

Appendix B

Standards-Track TCP Evaluation

In this appendix, I present the results of a study of standards-track TCP error recovery and queue management mechanisms. I consider standards-track TCP error recovery mechanisms to nclude TCP Reno and TCP with selective acknowledgments (SACK). (TCP NewReno was not considered because it is superseded by TCP SACK.) I consider standards-track queue management mechanisms to be drop-tail, Random Early Detection (RED), Adaptive RED, and versions of these queuing mechanisms that use Explicit Congestion Notification (ECN) marking instead of dropping. Descriptions of TCP SACK, RED, Adaptive RED, and ECN can be found in Chapter 2 (sections 1.3.3, 2.5.1, 2.5.3, and 2.5.2, respectively). This study of standards-track mechanisms was conducted in order to determine which combination of TCP error recovery and queue management mechanisms should be used as a comparison to Sync-TCP.

In section B.1, I describe how the evaluation was performed, including network configuration and parameter settings. In section B.2, I will present the results of the evaluation. Finally, in section B.3, I will describe experiments to determine the best parameters for Adaptive RED. These results were used in the evaluation of Sync-TCP in Chapter 4.

B.1 Methodology

I ran simulations in the *ns* network simulator[1] with varying levels of two-way HTTP 1.0 traffic [BEF+00]. These two-way traffic loads provide roughly equal levels of congestion on both the "forward" path and "reverse" path in the network. The following pairings of error recovery and queue management techniques were tested (Table B.1): TCP Reno with drop-tail queuing in routers, TCP Reno with Adaptive RED using packet drops, ECN-enabled TCP Reno with Adaptive RED using ECN packet marking, TCP SACK with drop-tail queuing, TCP SACK with Adaptive RED using packet drops, and ECN-enabled TCP SACK with Adaptive RED using packet marking.

[1] In order to use SACK and ECN in Full-TCP, I had to modify the ns-2.1b9 source code. See `http://www.cs.unc.edu/~mcweigle/ns/` for details.

Error Recovery	Queue Management
TCP Reno	drop-tail
TCP Reno	Adaptive RED
TCP Reno	Adaptive RED with ECN marking
TCP SACK	drop-tail
TCP SACK	Adaptive RED
TCP SACK	Adaptive RED with ECN marking

Table B.1: List of Experiments

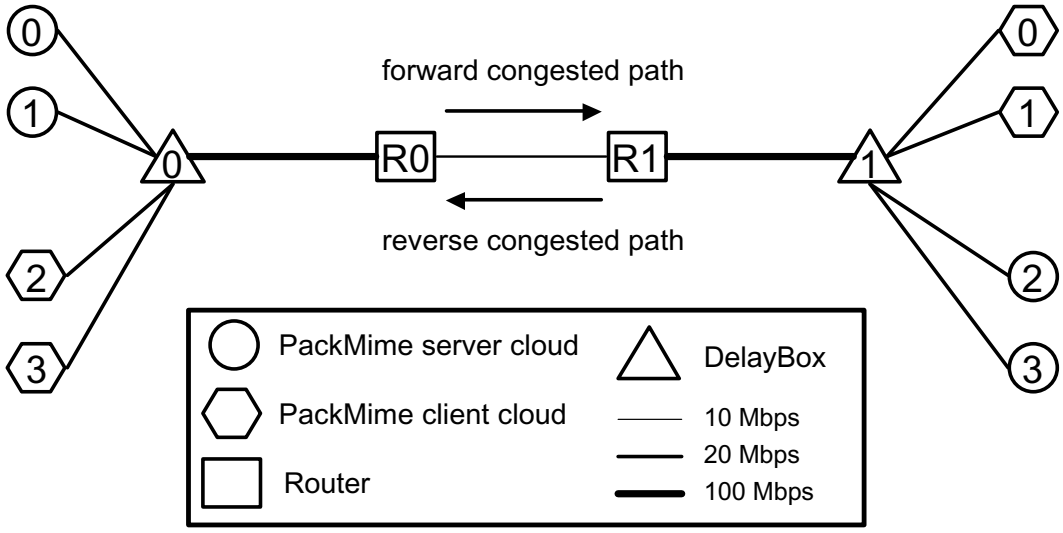


Figure B.1: Simulated Network Environment

B.1.1 Experimental Setup

The simulated network consists of two clouds of web servers and clients positioned at each end of a 10 Mbps bottleneck link (Figure B.1). There is a 10 Mbps bottleneck link between the two routers (R0 and R1), a 20 Mbps link between each PackMime cloud and its corresponding aggregation node, and a 100 Mbps link between each aggregation node and the nearest router. The 20 Mbps limit is so that transient spikes are limited to 40 Mbps and will not overload the 100 Mbps link between the aggregation node and the first router. The data presented herein comes from measurements of the traffic on the forward path in the network.

The aggregation nodes in these simulations are instances of an *ns* node that I developed called DelayBox, which can delay packets in a TCP flow. DelayBox is an *ns* analog to dummynet [Riz97], which is used in network testbeds to delay packets. With DelayBox, packets from a TCP connection can be delayed before being passed on to the next node. This allows each TCP connection to experience a different minimum delay (and hence a different round-trip time), based on random sampling from a delay distribution. In these experiments,

DelayBox uses an empirical delay distribution from the PackMime model. This results in base round-trip times (RTTs) ranging from 1 ms to 3.5 seconds. The median RTT is 54 ms, the mean is 74 ms, and the 90th percentile is 117 ms. RTTs are assigned independently of HTTP request or HTTP response size. These base RTTs represent only propagation delay, and thus do not include queuing delays.

The mean packet size for the HTTP traffic (excluding pure ACKs, but including headers) is 1,250 bytes[2]. This includes the HTTP responses for the forward path and the HTTP requests for the reverse path. For a target bottleneck bandwidth of 10 Mbps, the bandwidth-delay product (BDP) is 74 1,250-byte packets. In all cases, the maximum send window for each TCP connection is set to the BDP. In these experiments, delayed ACKs were not used.

B.1.2 HTTP Traffic Generation

The HTTP traffic generated comes from the PackMime model developed at Bell Labs [CCLS01b]. This model is based on the analysis of HTTP connections in a trace of a 100 Mbps Ethernet link connecting an enterprise network of approximately 3,000 hosts to the Internet [CLS00, CCLS01a]. The fundamental parameter of PackMime is the TCP/HTTP connection initiation rate (a parameter of the distribution of connection inter-arrival times). The model includes distributions of the size of HTTP requests and the size of HTTP responses.

I examine the behavior of traffic that consists of over 250,000 flows, with a total simulated time of 40 minutes. The distribution of response sizes has an infinite variance and hence simulations take a very long time to reach steady-state. Running the simulation for only a few minutes would take into account a small portion of the rich behavior of the traffic model. I ran the simulation for as long as the available hardware and software environments would support to capture a significant amount of this behavior.

I implemented PackMime traffic generation in *ns* using Full-TCP, which includes bi-directional TCP connection flows, connection setup, connection teardown, and variable packet sizes. In this implementation, one PackMime "node" represents a cloud of HTTP clients or servers. The traffic load is driven by the user-supplied connection rate parameter, which is the number of new connections starting per second. The connection rate corresponding to each desired link loading was determined by a calibration procedure described in the next section. New connections begin at their appointed time, whether or not any previous connection has completed.

B.1.3 Levels of Offered Load and Data Collection

The levels of offered load used in these experiments are expressed as a percentage of the capacity of a 10 Mbps link. I initially ran experiments with a 100 Mbps link between the two

[2] I assume an Ethernet MSS of 1,500 bytes and use a maximum TCP data size of 1,420 bytes, counting for 40 bytes of base TCP/IP header and 40 bytes maximum of TCP options.

routers and determined the PackMime connection rates (essentially, the HTTP request rates) that will result in average link utilizations (in both forward and reverse directions) of 5, 6, 7, 8, 8.5, 9, 9.5, 10, and 10.5 Mbps. The connection rate that results in an average utilization of 8% of the (clearly uncongested) 100 Mbps link will be used to generate an offered load on the 10 Mbps link of 8 Mbps, or 80% of the 10 Mbps link. Note that this "80% load" (*i.e.*, the connection rate that results in 8 Mbps of traffic on the 100 Mbps link) will not actually result in 8 Mbps of traffic on the 10 Mbps link. The bursty HTTP sources will cause congestion on the 10 Mbps link, and the actual utilization of the link will be a function of the protocol and router queue management scheme used. (And the link utilization achieved by a given level of offered load is a metric for comparing protocol/queue management combinations. See Figure B.3a.)

Given the bursty nature of our HTTP traffic sources, I used a 120-second "warm-up" interval before collecting any data. After the warm-up interval, the simulation proceeds until 250,000 HTTP request-response pairs have been completed. I also require that at least one 10 MB response has started a transfer before 1,000 seconds after the warm-up period. This ensures that the simulation will have some very long transfers in the network along with the typical short-lived web transfers.

B.1.4 Queue Management

Drop Tail Settings Christiansen *et al.* [CJOS01] recommend a maximum queue size between 1.25xBDP and 2xBDP for reasonable response times for drop-tail queues. The maximum queue buffer sizes tested in the drop-tail experiments were 1.5xBDP (111 packets) and 2xBDP (148 packets).

Adaptive RED Settings I ran sets of Adaptive RED experiments using the default settings in *ns* ($target_{delay} = 5$ ms) and with parameters similar to those suggested by Christiansen ($min_{th} = 30$ and $max_{th} = 90$) giving a target delay of 60 ms. Note that in both cases, min_{th} and max_{th} are computed automatically in *ns* based on the mean packet size, target delay, and link speed. The maximum router queue length was set to 5xBDP (370 packets). This ensured that there would be no tail drops, in order to isolate the effects of Adaptive RED.

B.1.5 Performance Metrics

In each experiment, I measured the following network-level metrics:

- Average loss rate - The percentage of packets that are dropped at the bottleneck link.
- The average percentage of flows that experienced any packet loss.
- The average queue size at the bottleneck router.

Abbreviation	Description
DT-111q	drop-tail with 111 packet queue (1.5xBDP)
DT-148q	drop-tail with 148 packet queue (2xBDP)
ARED-5ms	Adaptive RED with 5 ms target delay
ARED-30ms	Adaptive RED with 30 ms target delay
ECN-5ms	Adaptive RED with ECN marking and 5 ms target delay
ECN-30ms	Adaptive RED with ECN marking and 30 ms target delay

Table B.2: Summary of Labels and Abbreviations

- Link utilization - The percentage of the bottleneck link that is utilized by the traffic.
- Throughput - The number of bytes per second that enter the bottleneck link.

I look at HTTP response time, an application metric, as the main metric of performance. The response time of an HTTP request-response pair is the time from sending HTTP request to receiving entire HTTP response. I report the cumulative distribution functions (CDFs) of response times for responses that complete in 1,500 ms or less. When discussing the CDFs, I discuss the percentage of flows that complete in a given amount of time. It is not the case that only small responses complete quickly and only large responses take a long time to complete. For example, between a 500 KB response that has a RTT of 1 ms and a 1 KB response that has a RTT of 1 second, the 500 KB response will likely complete before the smaller response.

B.2 Results

I first present the results for different queue management algorithms when paired with TCP Reno end-systems. Next, results for queue management algorithms paired with TCP SACK end-systems are presented and compared to the TCP Reno results. Finally, the two best scenarios are compared. Table B.2 lists the labels I use to identify experiments.

Figures B.2-B.5 give the network-level summary statistics (as described in section B.1.5) and the median HTTP response time for each experiment.

In the following response time CDF plots, the response times obtained in a calibration experiment with an uncongested 100 Mbps link (labeled as "calibration") are included for a baseline reference as this represents the best possible performance. For completeness, I include the response time CDFs for all of the experiments that were performed (Figures B.6-B.45) at all load levels. However, I will focus on the results for offered loads of 80-105%. This is motivated by the observation that while generally HTTP response time performance decreases as load increases, for loads less than 80%, there is no difference in link utilization (Figures B.3a and B.5a) for any of the protocol/queue management combinations I considered. Moreover, the differences in response times are more significant in the 80-100% load range.

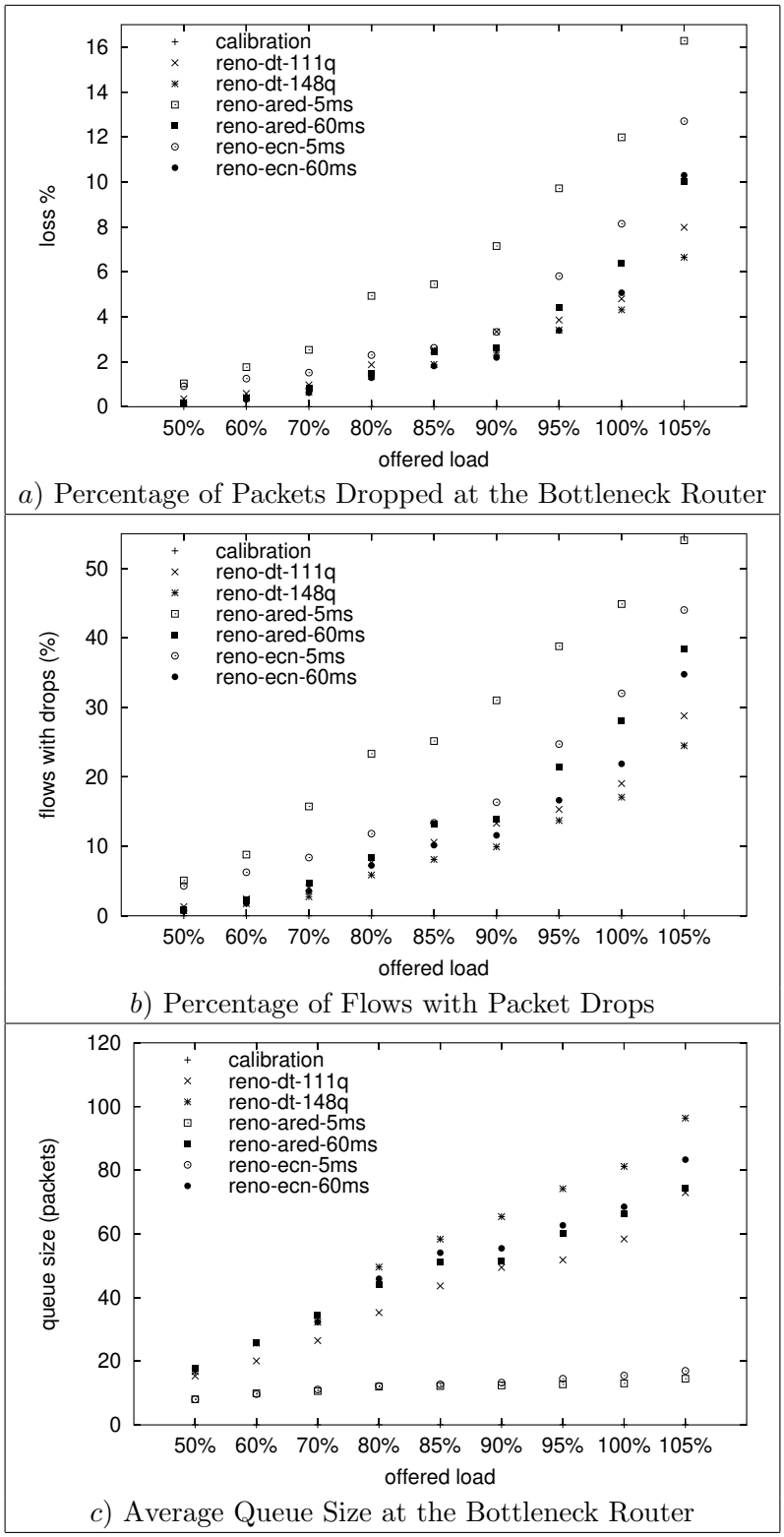


Figure B.2: Summary Statistics for TCP Reno (drops, flows with drops, queue size)

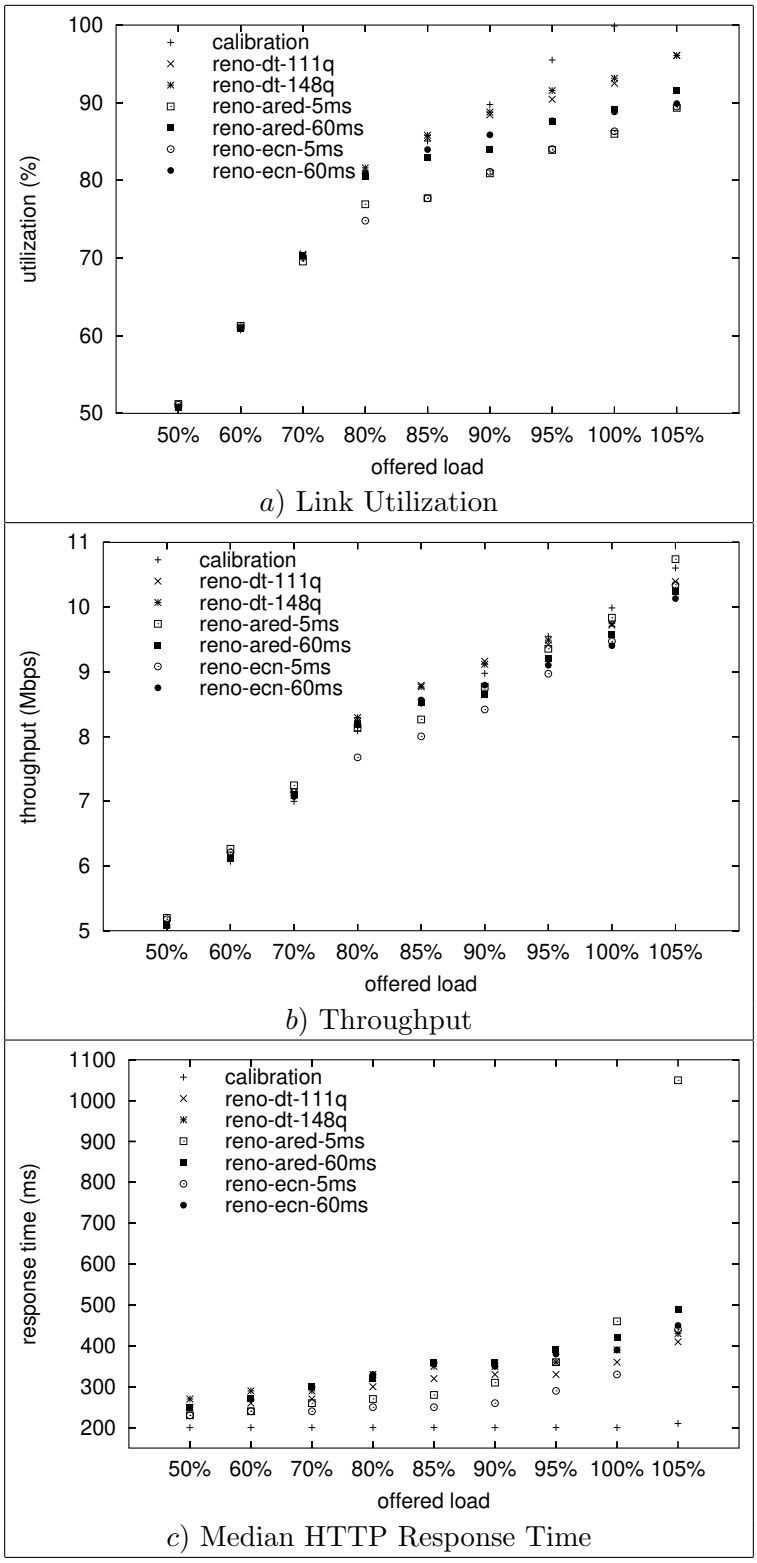


Figure B.3: Summary Statistics for TCP Reno (utilization, throughput, response time)

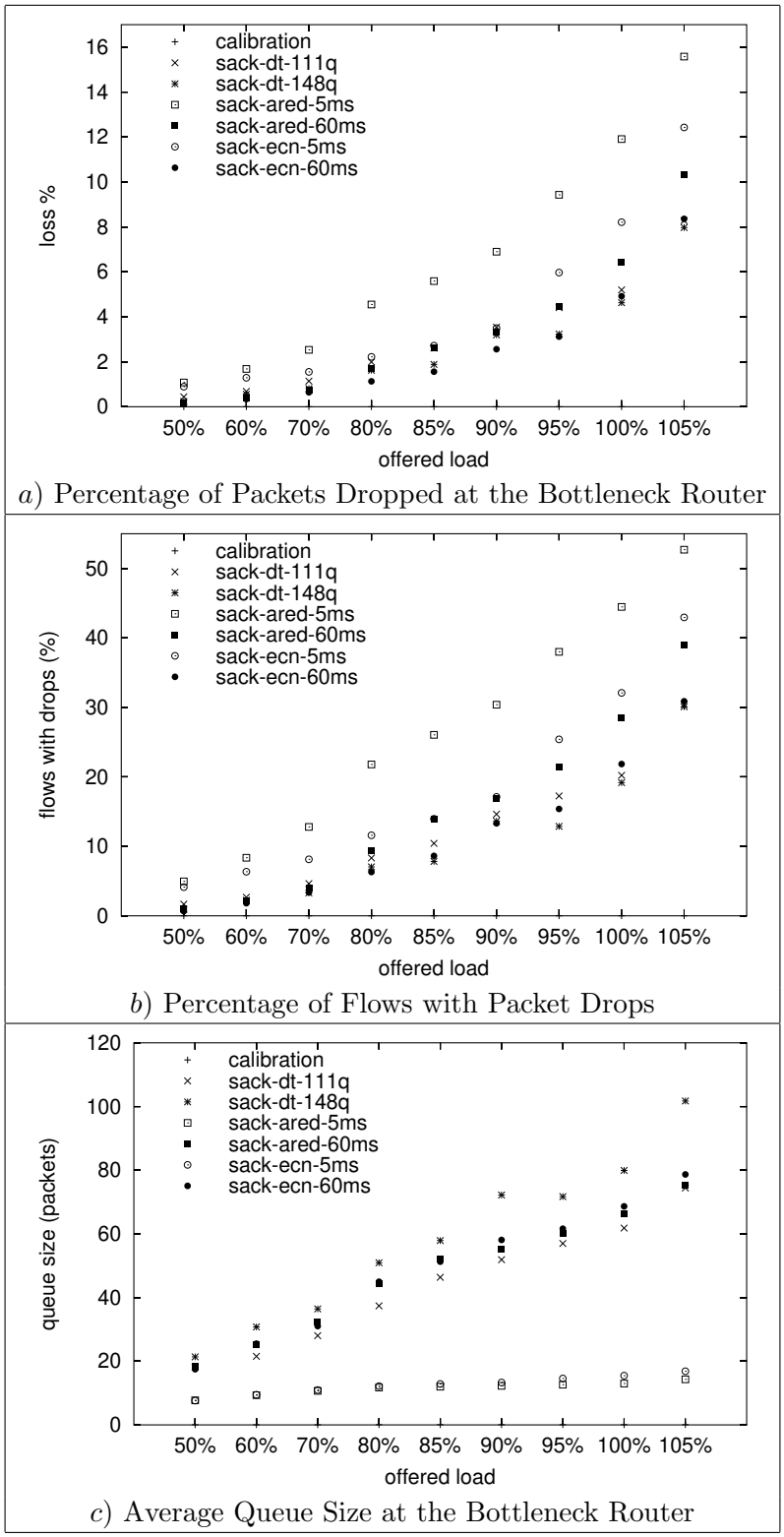


Figure B.4: Summary statistics for TCP SACK (drops, flows with drops, queue size)

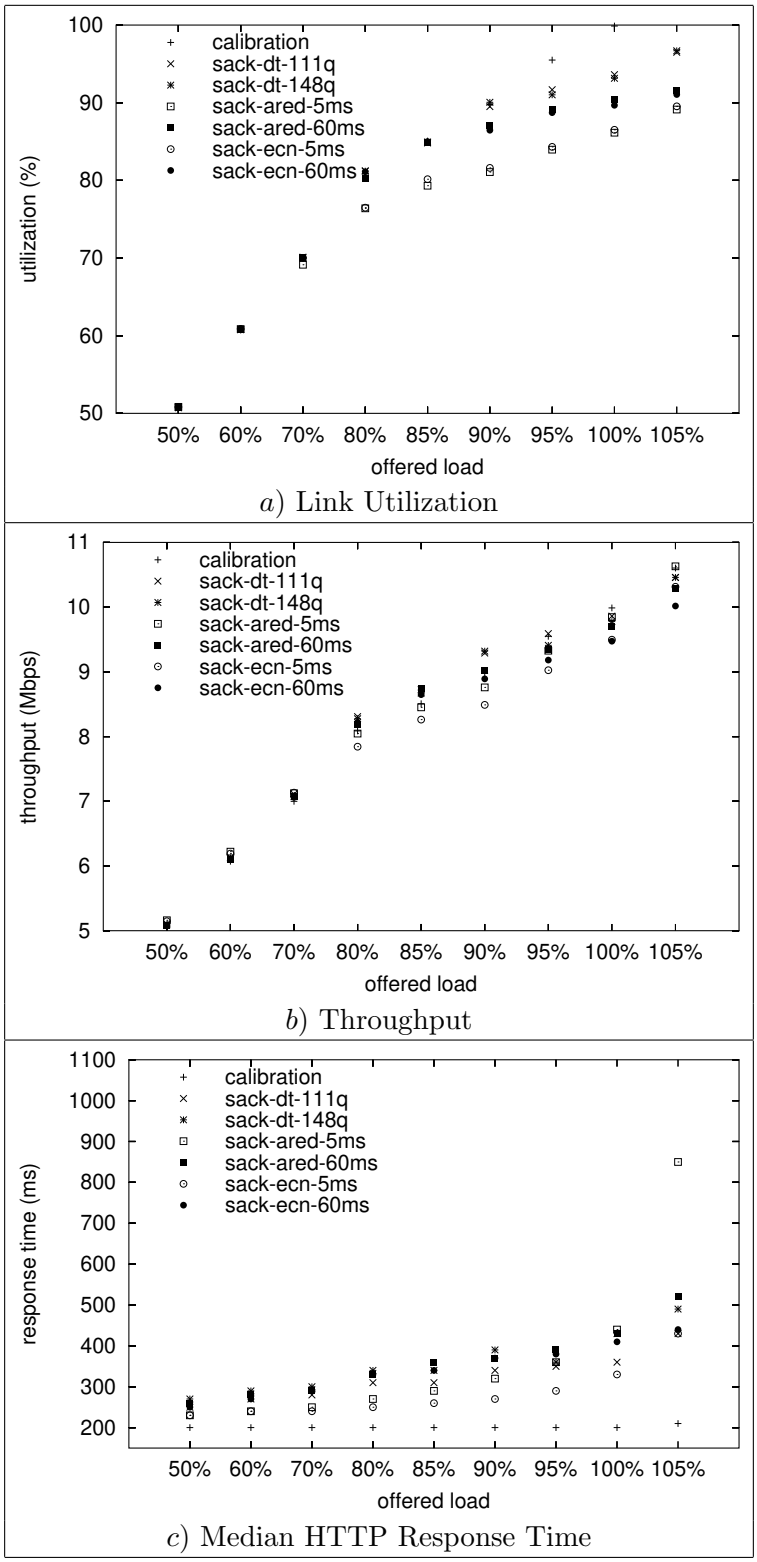


Figure B.5: Summary Statistics for TCP SACK (utilization, throughput, response time)

B.2.1 TCP Reno with Drop-Tail Queuing

Figures B.6-B.8 show the CDFs of response times for TCP Reno with drop-tail queuing at the two queue buffer sizes (Reno-DT-111q, Reno-DT-148q). There is little performance difference between the two queue sizes, though there is a crossover point in the response times. The crossover is described here only for illustration purposes, since the difference is minimal. At 80% load, the crossover is at coordinate (700 ms, 80%). This means that for both Reno-DT-111q and Reno-DT-148q, 80% of the HTTP responses completed in 700 ms or less. For a given response time less than 700 ms, Reno-DT-111q produces a slightly higher percentage of responses that complete in that time or less than does Reno-DT-148q. For a given response time greater than 700 ms, Reno-DT-148q yields a slightly higher percentage of responses that complete in that time or less than Reno-DT-111q does.

B.2.2 TCP Reno with Adaptive RED Queuing

Response time CDFs for TCP Reno with drop-tail queuing (Reno-DT-111q, Reno-DT-148q) and TCP Reno with Adaptive RED queuing (Reno-ARED-5ms, Reno-ARED-60ms) are shown in Figures B.9-B.11. At almost all loads, both of the drop-tail queues perform no worse than Reno-ARED-60ms. At 70% load, there is a distinct crossover point between Reno-ARED-5ms and Reno-ARED-60ms (and Reno-DT-148q and Reno-DT-111q) at 400 ms. This points to a tradeoff between improving response times for some flows and causing worse response times for others. For responses that complete in less than 400 ms, Reno-ARED-5ms offers better performance. For responses that complete in over 400 ms, Reno-ARED-60ms, Reno-DT-111q, or Reno-DT-148q are preferable. As the load increases, the crossover remains near a response time of 400 ms, but the percentage of completed responses in that time or less decreases. Also, as load increases, the performance of Reno-ARED-5ms for longer responses is poor.

Reno-ARED-5ms keeps a much shorter average queue than Reno-ARED-60ms (Figure B.2c), but at the expense of longer response times for many responses. With Reno-ARED-5ms, many flows experience packet drops. Many of these connections are very short-lived, often consisting of a single packet (60% of responses consist of only one packet and 80% consist of five or fewer packets) and when they experience packet loss, they are forced to suffer a retransmission timeout, increasing their HTTP response times. For Reno-ARED-5ms, the CDF of response times levels off after the crossover point and does not increase much until after 1 second, which is the minimum RTO in the *ns* simulations. This indicates that a significant portion of the flows suffered timeouts.

As congestion increased toward severe levels, the response time benefits from the drop-tail queue became substantially greater. At 105% load, about 60% of responses (those taking longer than 300-400 ms to complete) have better response times with the drop-tail queue while

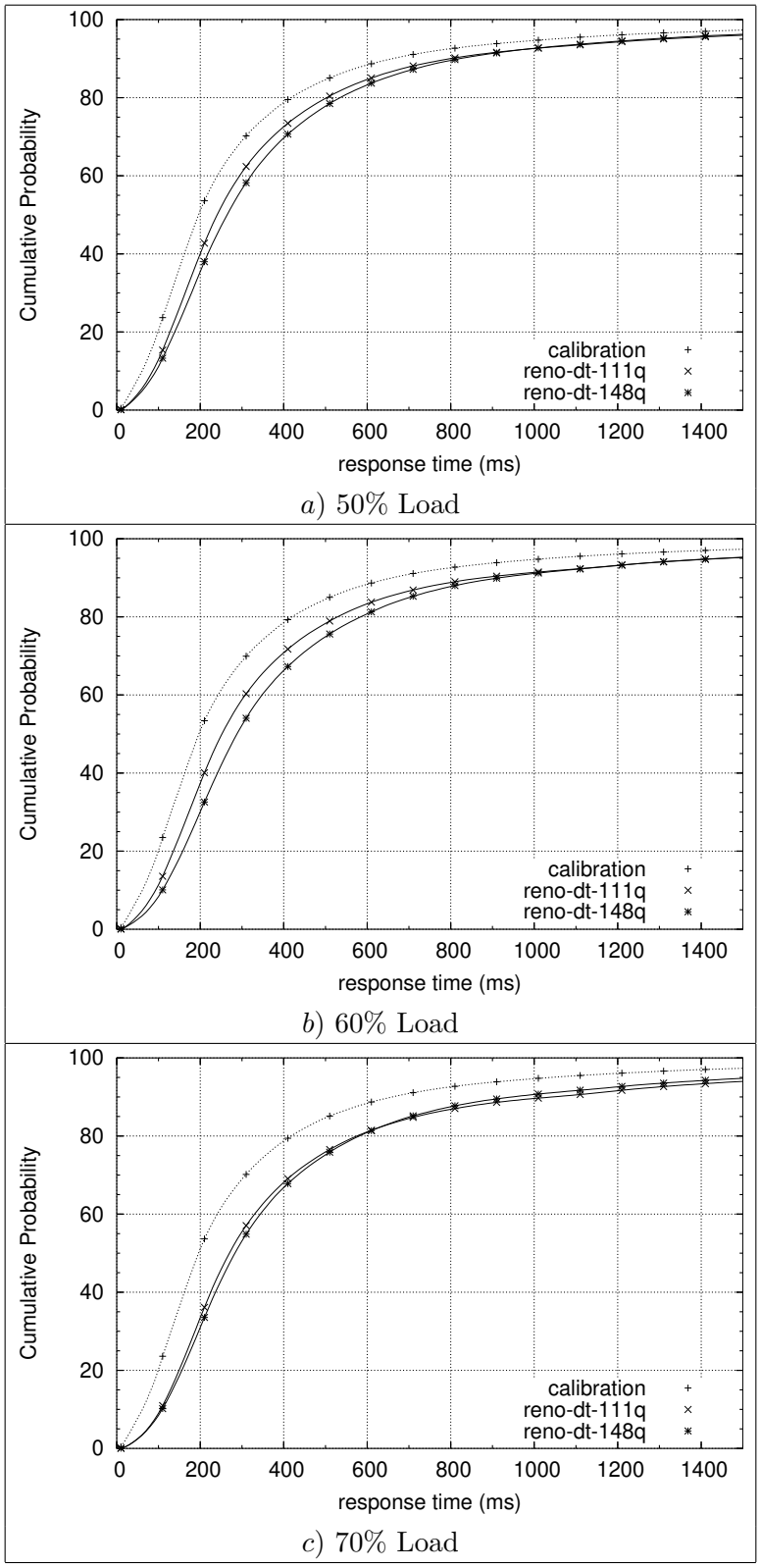


Figure B.6: Response Time CDFs, TCP Reno with Drop-Tail Queuing, Light Congestion

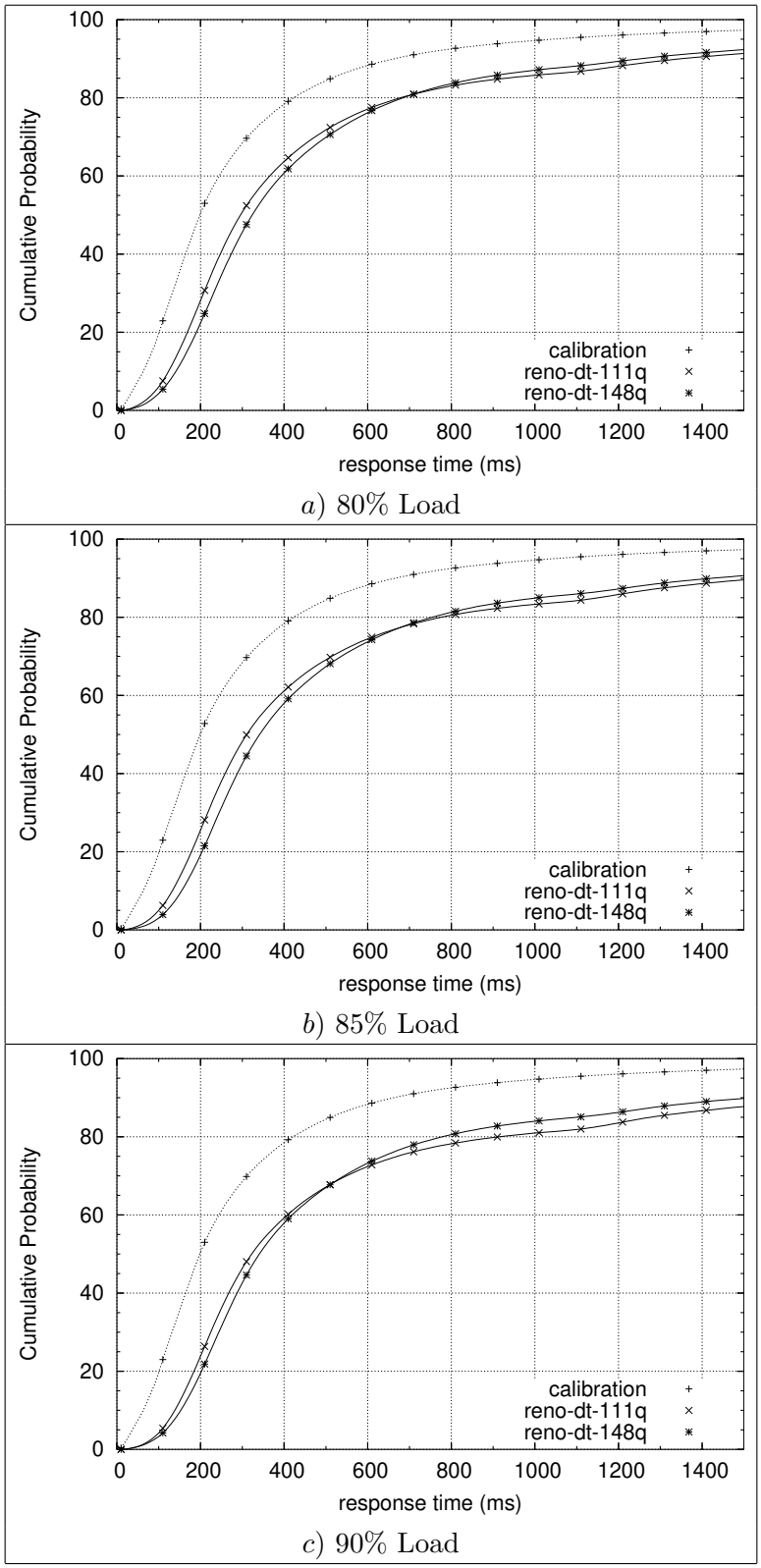


Figure B.7: Response Time CDFs, TCP Reno with Drop-Tail Queuing, Medium Congestion

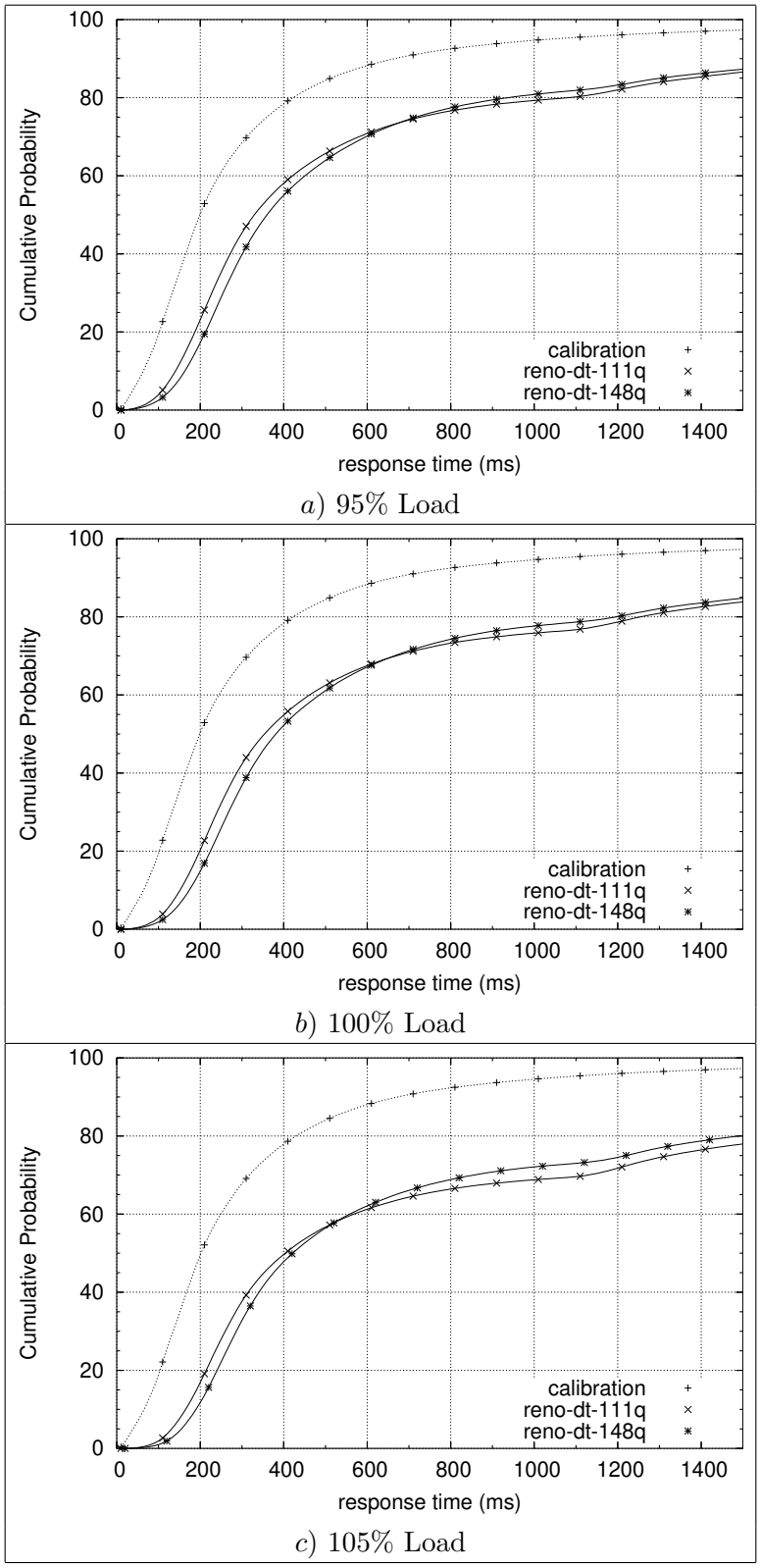


Figure B.8: Response Time CDFs, TCP Reno with Drop-Tail Queuing, Heavy Congestion

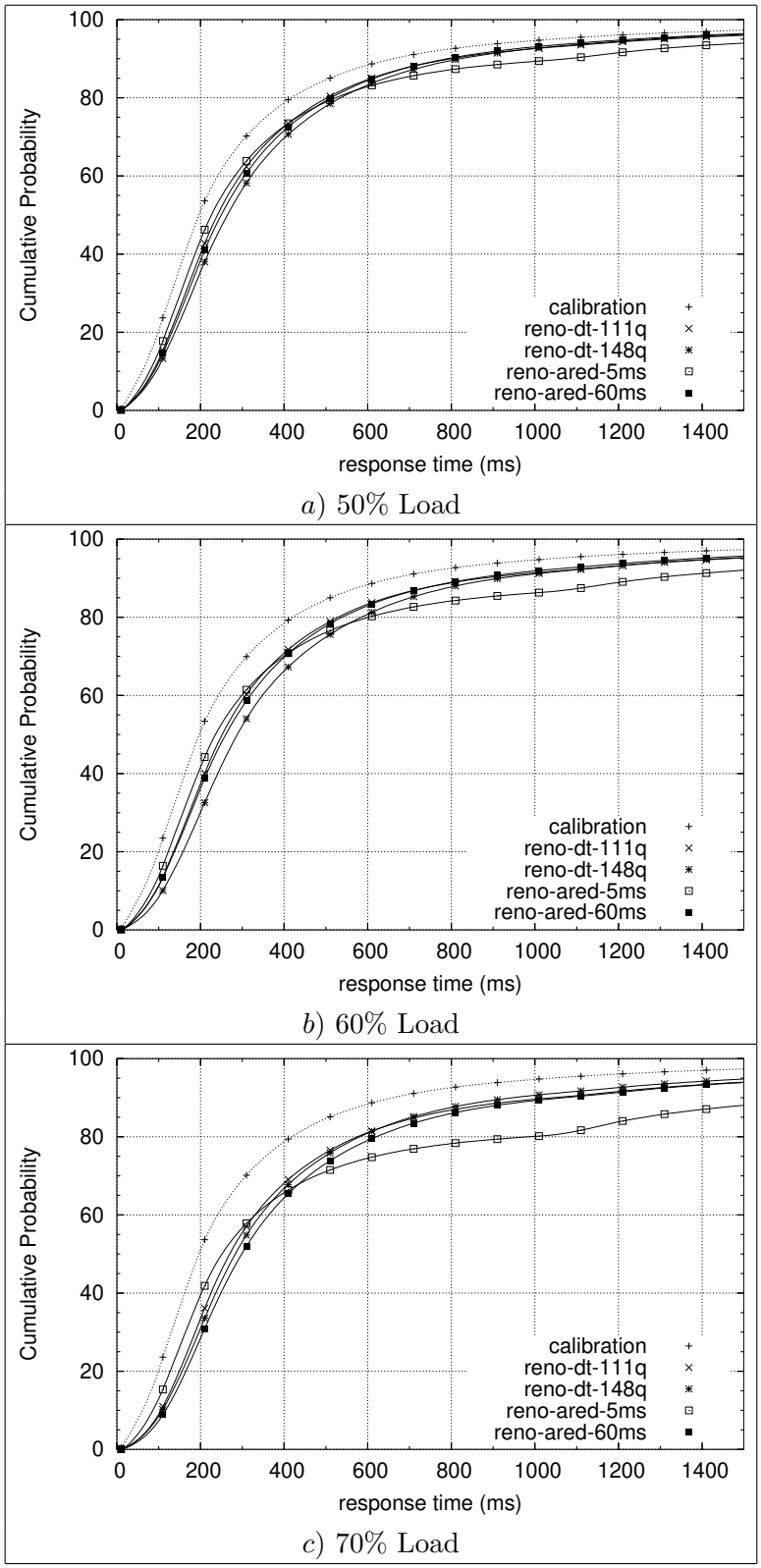


Figure B.9: Response Time CDFs, TCP Reno with Drop-Tail Queuing and with Adaptive RED Queuing, Light Congestion

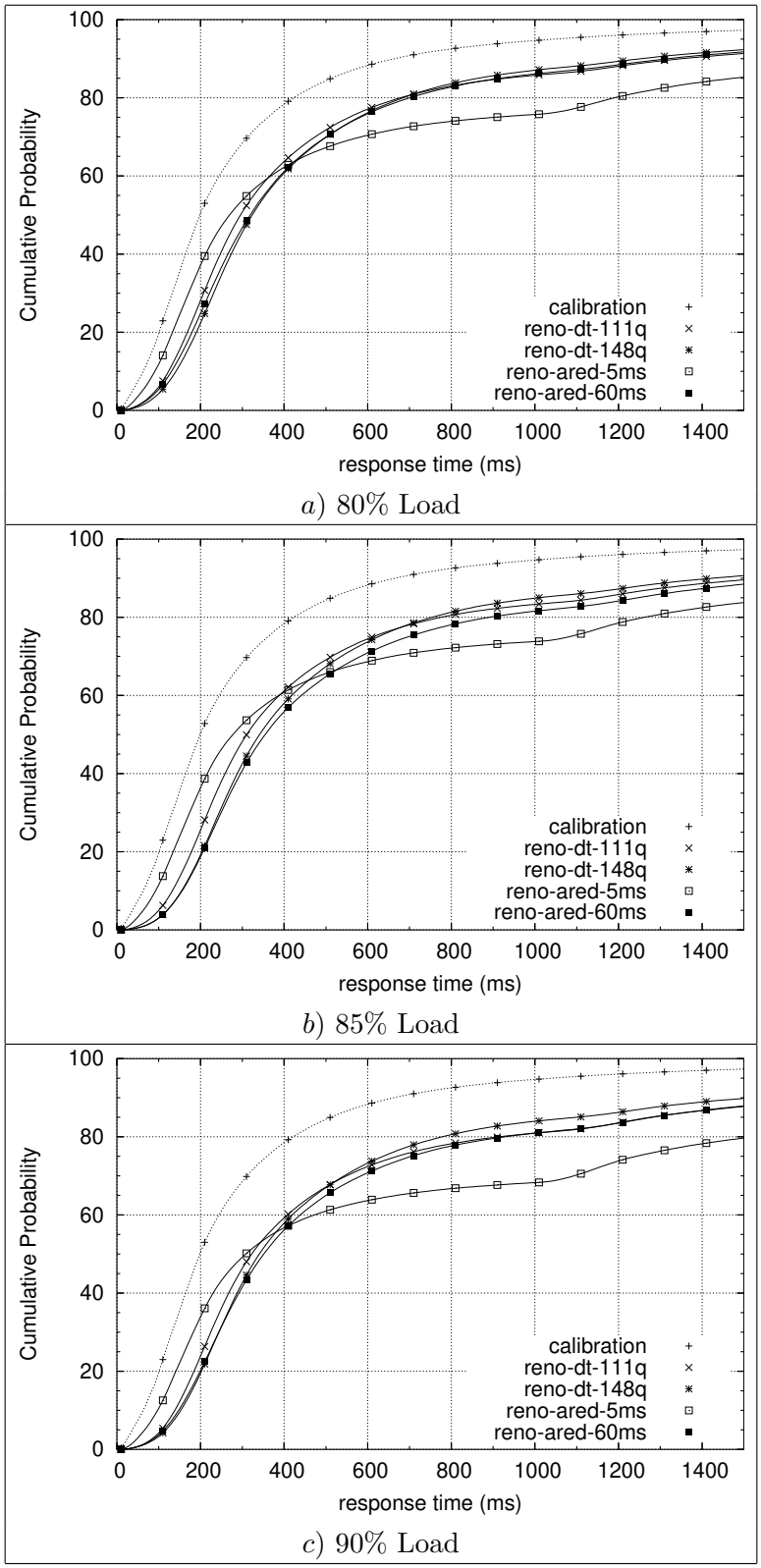


Figure B.10: Response Time CDFs, TCP Reno with Drop-Tail Queuing and with Adaptive RED Queuing, Medium Congestion

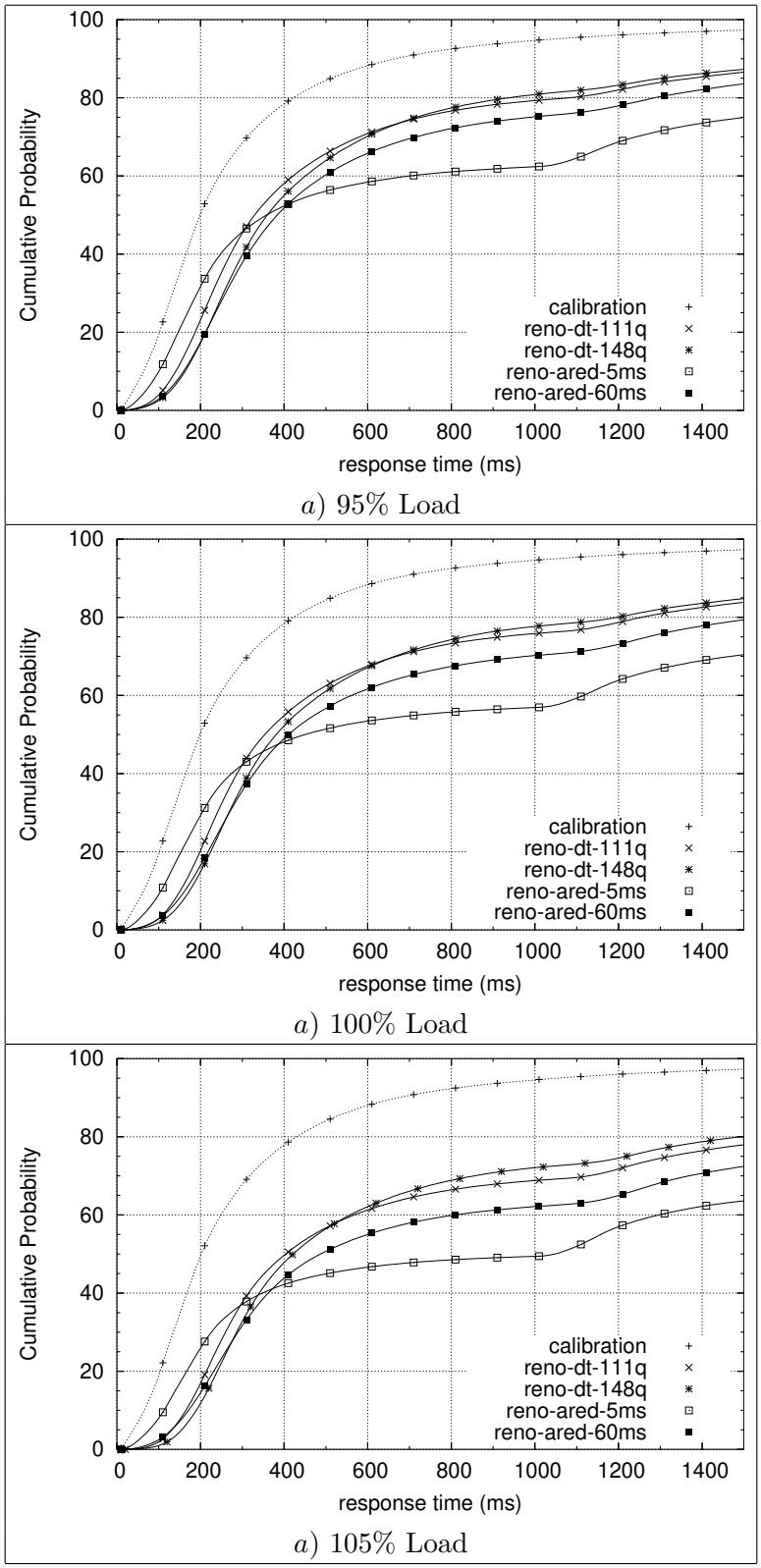


Figure B.11: Response Time CDFs, TCP Reno with Drop-Tail Queuing and with Adaptive RED Queuing, Heavy Congestion

only 40% of responses are better with Reno-ARED-5ms. For this same 60% of responses, Reno-ARED-60ms is also superior to Reno-ARED-5ms.

Adaptive RED should give better performance than the original RED design without the "gentle" option. At high loads, a significant number of packets arrive at the queue when the average queue size is greater than max_{th}. Without the "gentle" option, these packets would all be dropped, rather than being dropped probabilistically. To see the full differences between Adaptive RED including the gentle option and the original RED design I compared the results between the two designs at loads of 80%, 90%, and 100% (Figure B.12). In these comparisons, two configurations of the original RED minimum and maximum thresholds were used: (5, 15) and (30, 90) which correspond roughly to the 5 ms and 60 ms target queue lengths used for Adaptive RED. For the original RED experiments, I used the recommended settings of $w_q = 1/512$ and $max_p = 10\%$. For the Adaptive RED experiments, w_q was set automatically based on link speed to 0.001, and max_p was adapted between 1-50%. Figure B.12 shows that the adaptation of max_p and the linear increase in drop probability from max_p to 1.0 in Adaptive RED is an improvement over the original RED design.

B.2.3 TCP Reno with Adaptive RED Queuing and ECN Marking

The full value of Adaptive RED is realized only when it is coupled with a more effective means of indicating congestion to the TCP endpoints than the implicit signal of a packet drop. ECN is the congestion signaling mechanism intended to be paired with Adaptive RED. Figures B.13-B.14 present the response time CDFs for Reno-ARED-5ms, Reno-ARED-60ms, Reno-ECN-5ms, and Reno-ECN-60ms. Up to 90% load, Reno-ECN-5ms delivers superior or equivalent performance for all response times. Further, ECN has a more significant benefit when the Adaptive RED target queue is small. As before, there is a tradeoff where the 5 ms target delay setting performs better before the crossover point and the 60 ms setting performs slightly better after the crossover point. The tradeoff when ECN is paired with Adaptive RED is much less significant. Reno-ECN-5ms does not see as much drop-off in performance after the crossover point. For the severely congested case, ECN provides even more advantages, especially when coupled with a small target delay.

At loads of 80% and higher, Reno-ECN-5ms produces lower link utilization than Reno-ECN-60ms (Figure B.3a). This tradeoff between response time and link utilization is expected. Reno-ECN-5ms keeps the average queue size small so that packets see low delay as they pass through the router. Flows that experience no packet loss should see very low queuing delays, and therefore, low response times. On the other hand, larger flows may receive ECN notifications and reduce their sending rates so that the queue drains more often. As expected, drop-tail results in the highest link utilization and the lowest drop rates.

Finally, I ran a set of experiments with one-way traffic to see how well ECN and Adaptive RED would perform in a less complex environment. One-way traffic has HTTP response traffic

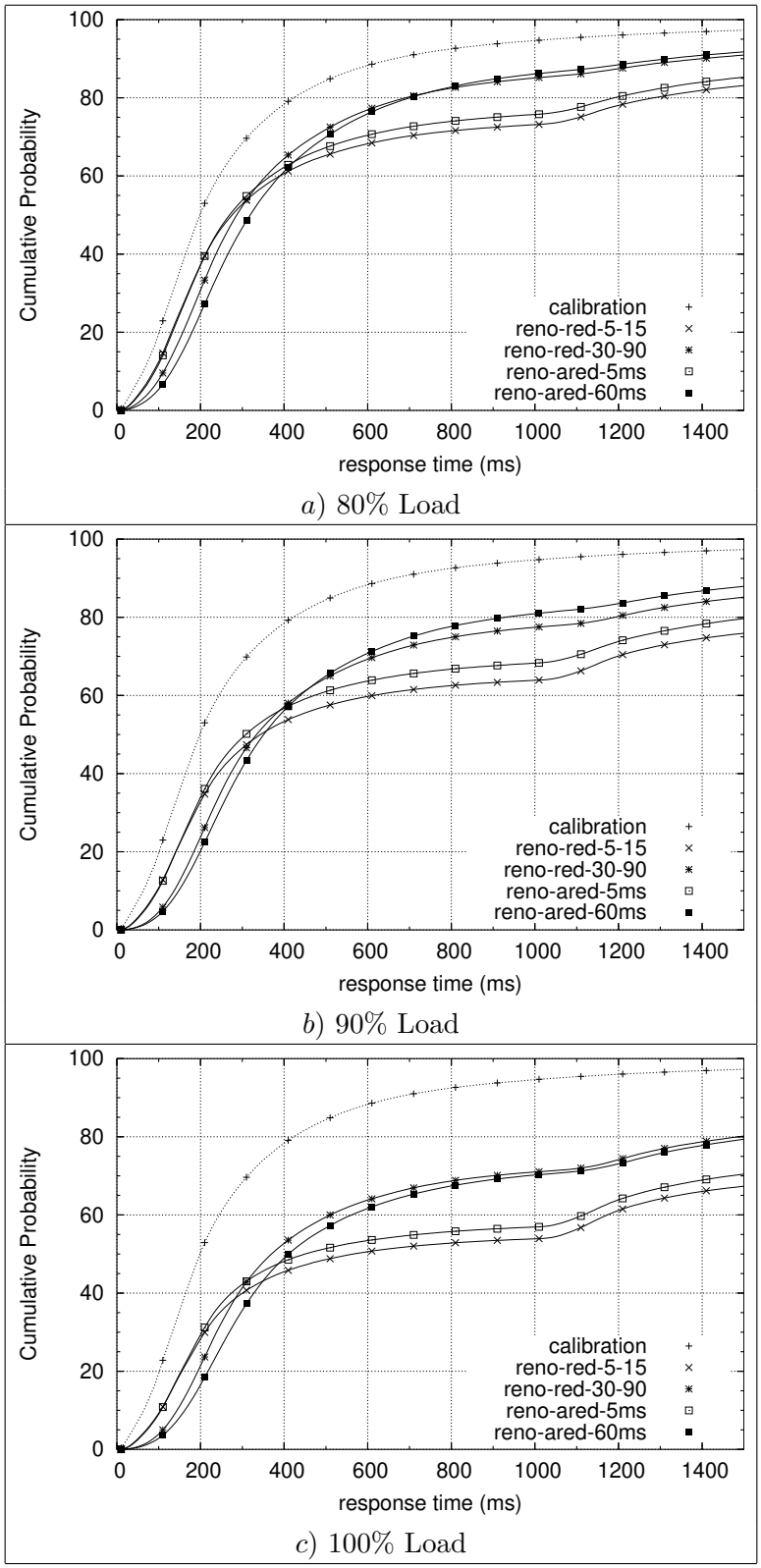


Figure B.12: Response Time CDFs, TCP Reno with "original" RED Queuing and Adaptive RED Queuing

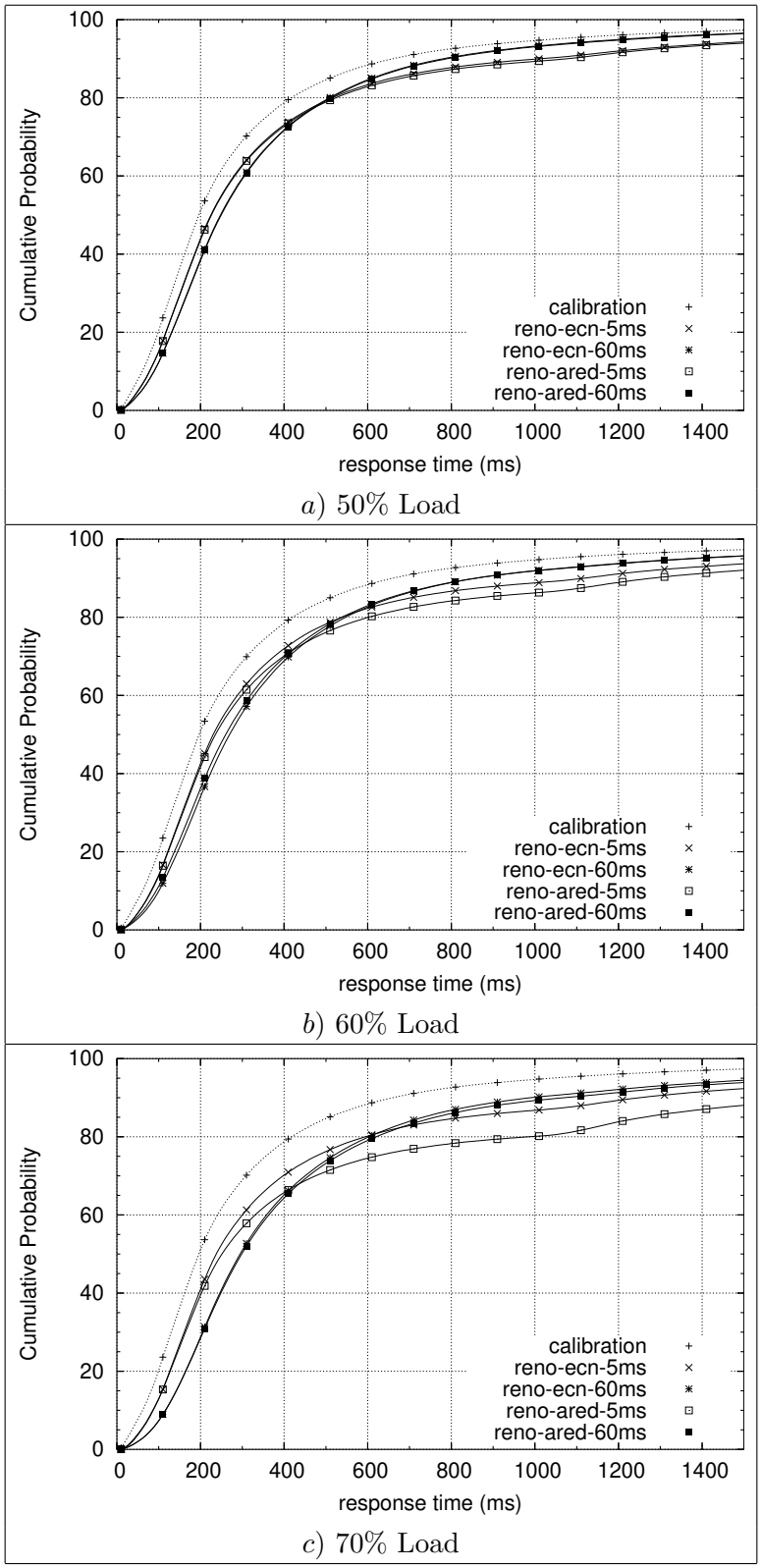


Figure B.13: Response Time CDFs, TCP Reno with Adaptive RED Queuing and with Adaptive RED Queuing and ECN Marking, Light Congestion

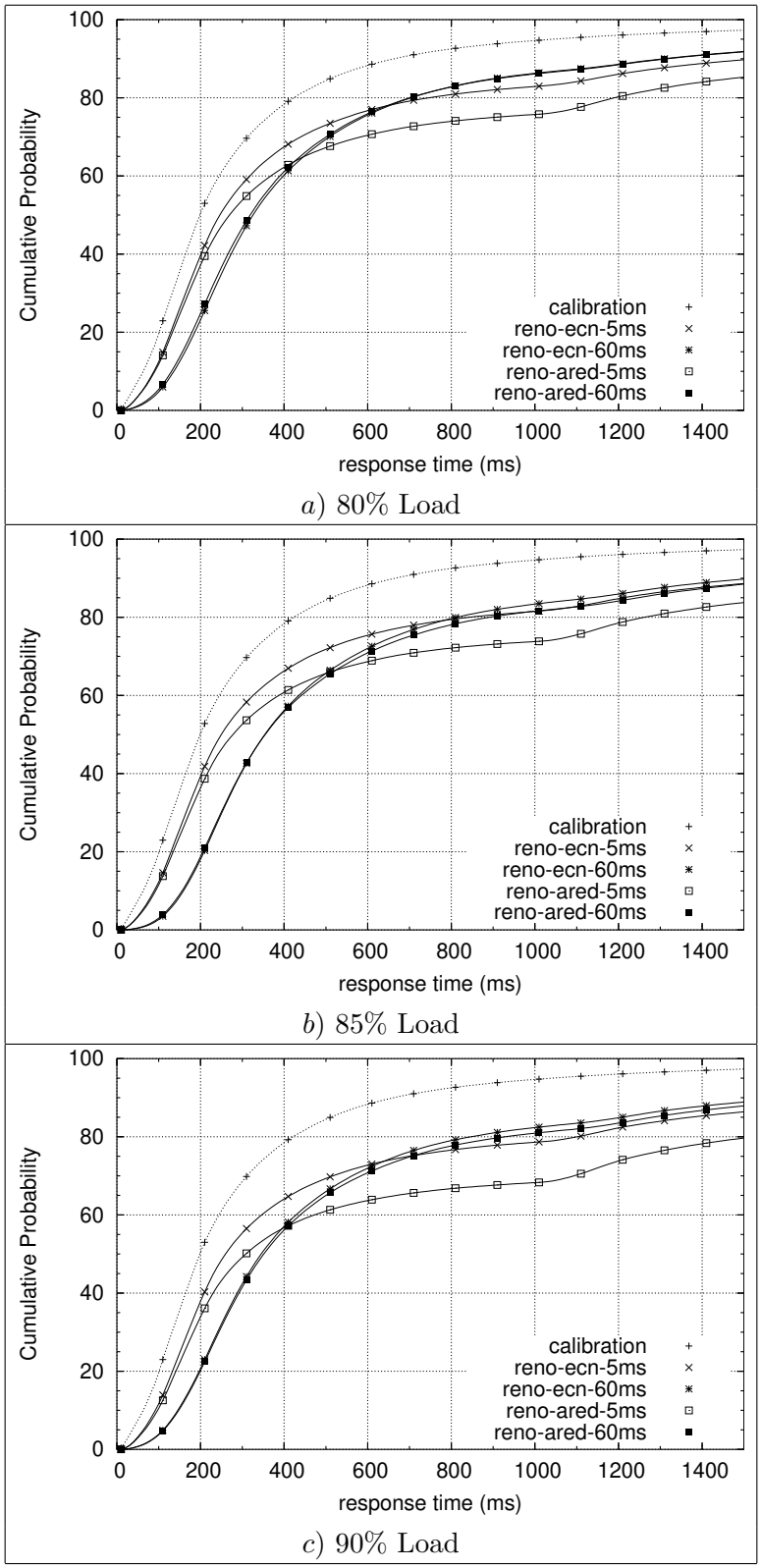


Figure B.14: Response Time CDFs, TCP Reno with Adaptive RED Queuing and with Adaptive RED Queuing and ECN Marking, Medium Congestion

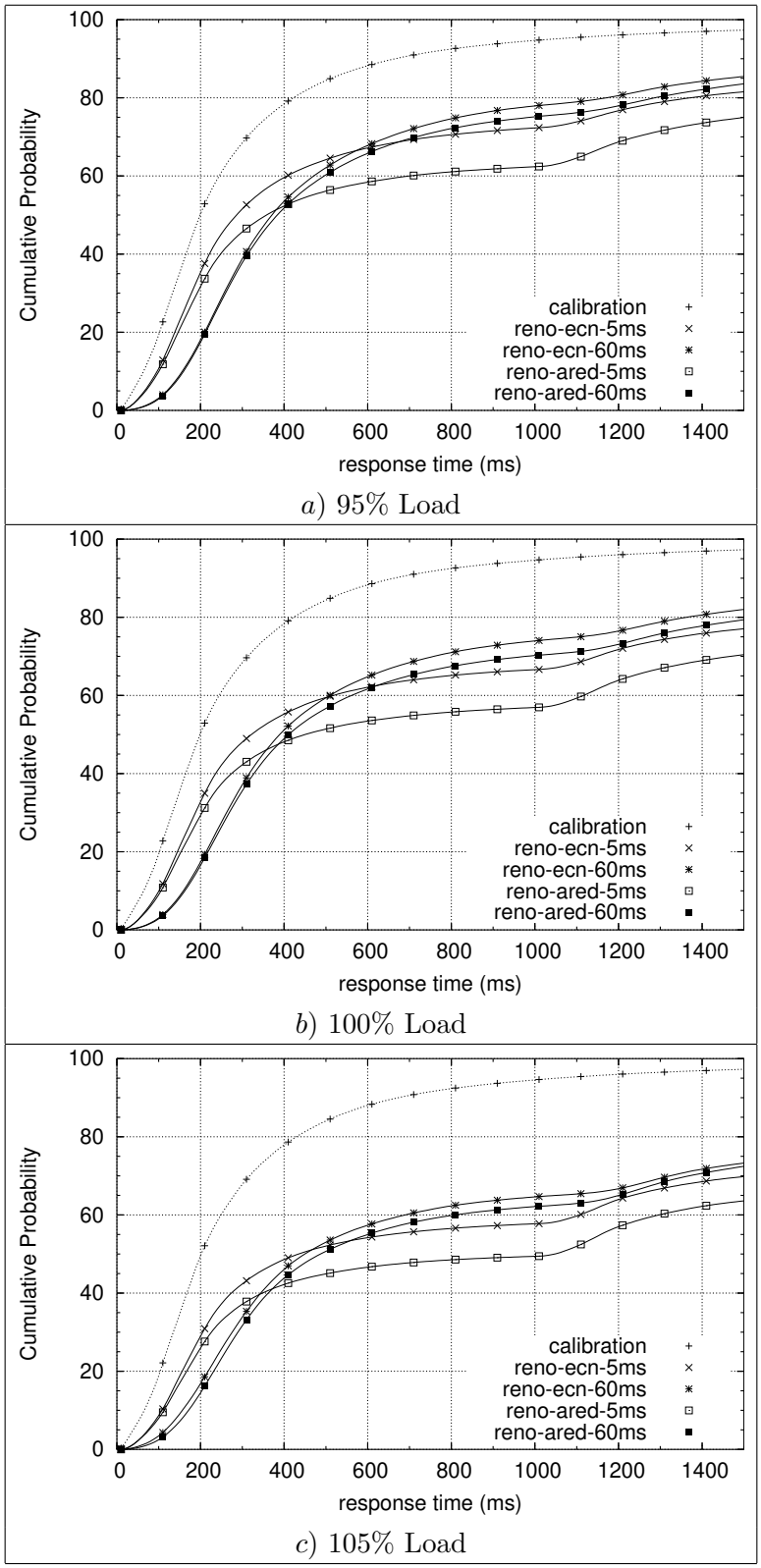


Figure B.15: Response Time CDFs, TCP Reno with Adaptive RED Queuing and with Adaptive RED Queuing and ECN Marking, Heavy Congestion

flowing in only one direction on the link between routers. Thus, the reverse-path carrying ACKs and HTTP request packets was very lightly loaded. This means that ACKs are much less likely to be lost or "compressed" at the router queue and that (1) TCP senders will receive a smoother flow of ACKs to "clock" their output segments and (2) ECN congestion signals will be more timely. In some sense, this is a best case scenario for Adaptive RED with ECN. The results for this case are given in Figures B.16-B.17. With two-way traffic, performance is significantly reduced over the one-way case especially for the severe congestion at 105% offered load. These results clearly illustrate the importance of considering the effects of congestion in both directions of flow between TCP endpoints.

B.2.4 TCP SACK

The experiments described above were repeated by pairing TCP SACK with the different queue management mechanisms in place of TCP Reno. Figures B.18-B.26 show a comparison between TCP Reno and TCP SACK based on response time CDFs when paired with drop-tail, Adaptive RED, and Adaptive RED with ECN. Overall, TCP SACK provides no better performance than TCP Reno. When paired with Adaptive RED with ECN, TCP SACK and TCP Reno are essentially identical independent of load. When paired with drop-tail, TCP Reno appears to provide somewhat superior response times especially when congestion is severe. I expected to see improved response times with TCP SACK over TCP Reno with drop-tail queues. TCP SACK should prevent some of the timeouts that TCP Reno would have to experience before recovering from multiple drops in a window. Instead, the response time CDFs are very similar. Recall that with HTTP, most TCP connections never send more than a few segments and, for loss events in these short connections, the SACK option never comes into play. For the relatively small number of connections where the SACK algorithm is invoked, the improvement in response time by avoiding a timeout is modest relative to the overall length of the responses that experience losses recoverable with TCP SACK.

B.2.5 Drop-Tail vs. Adaptive RED with ECN Marking

Here, I compare the performance of the two "best" error recovery and queue management combinations. Figures B.30-B.32 present the response time CDFs for Reno-DT-148q and Reno-ECN-5ms. With these scenarios, the fundamental tradeoff between improving response times for some responses and making them worse for other responses is clear. Further, the extent of the tradeoff is quite dependent on the level of congestion. At 80% load, Reno-ECN-5ms offers better response-time performance for nearly 75% of responses but marginally worse for the rest. At levels of severe congestion the improvements in response times for Reno-ECN-5ms apply to around 50% of responses while the response times of the other 50% are degraded significantly, Reno-DT-148q and Reno-ECN-5ms are on the opposite ends of the queue-management spectrum, yet they each offer better HTTP response times for different

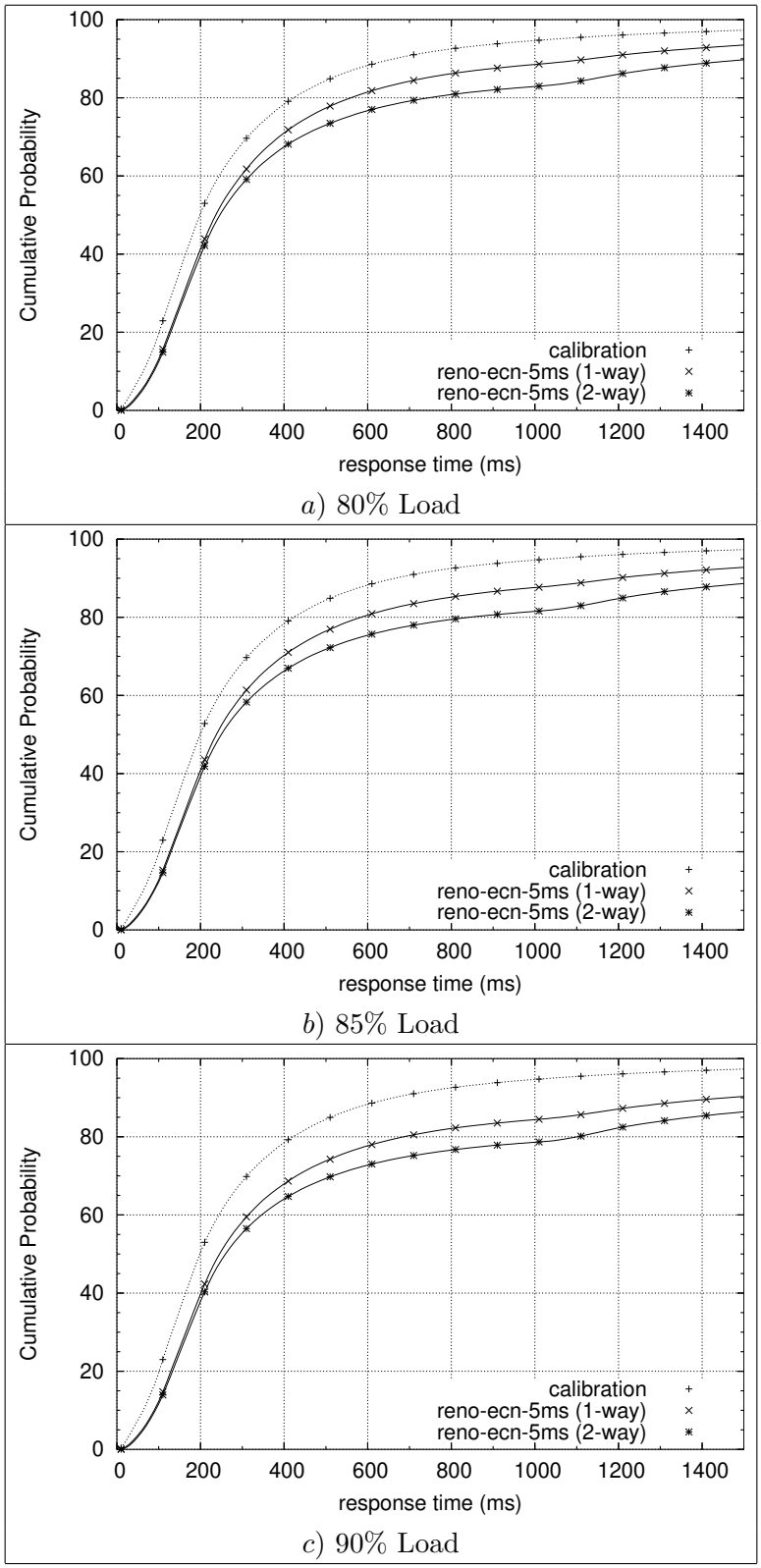


Figure B.16: Response Time CDFs, TCP Reno with Adaptive RED Queuing and ECN Marking, One-Way and Two-Way Traffic, Medium Congestion

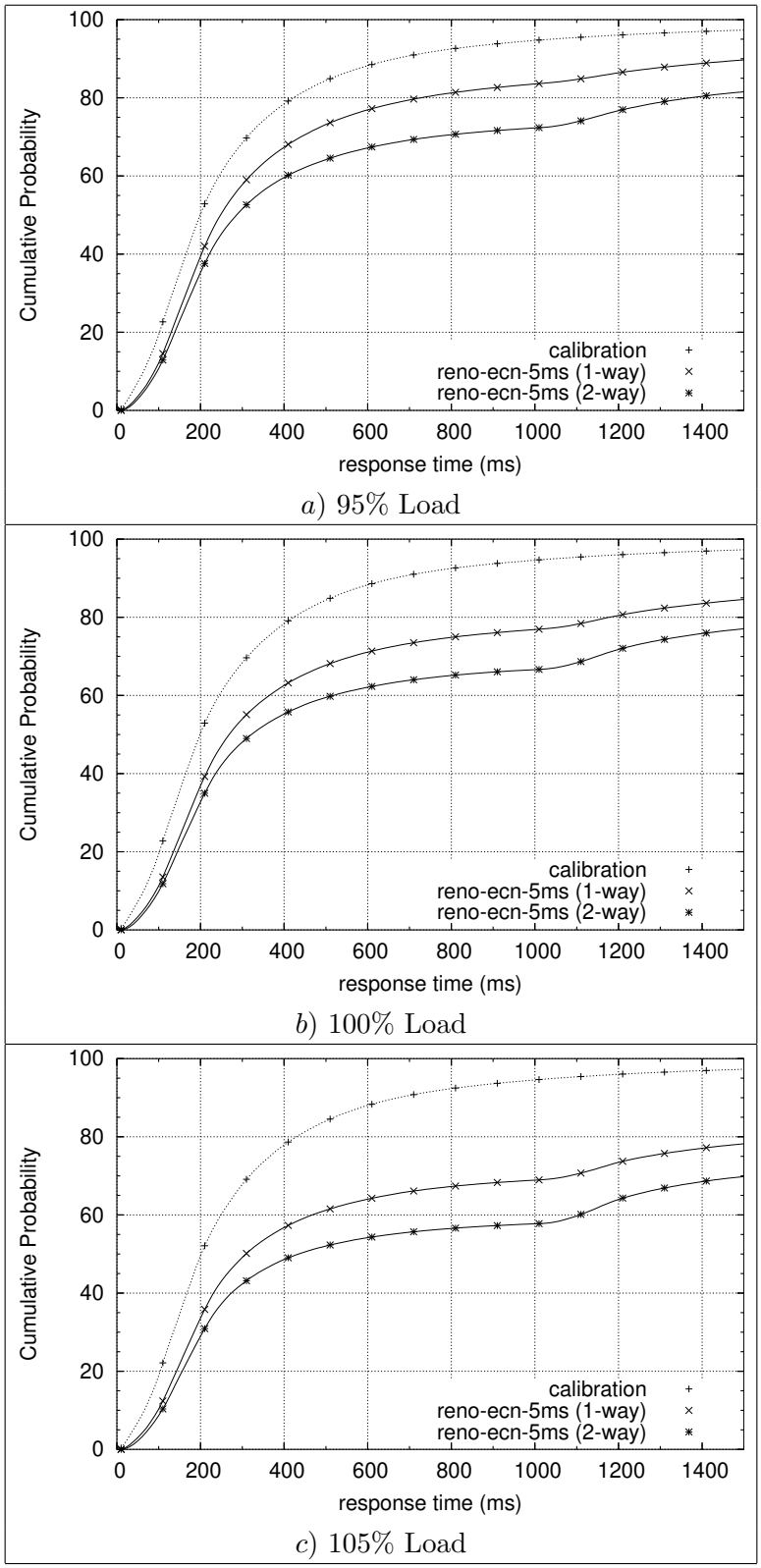


Figure B.17: Response Time CDFs, TCP Reno with Adaptive RED Queuing and ECN Marking, One-Way and Two-Way Traffic, Heavy Congestion

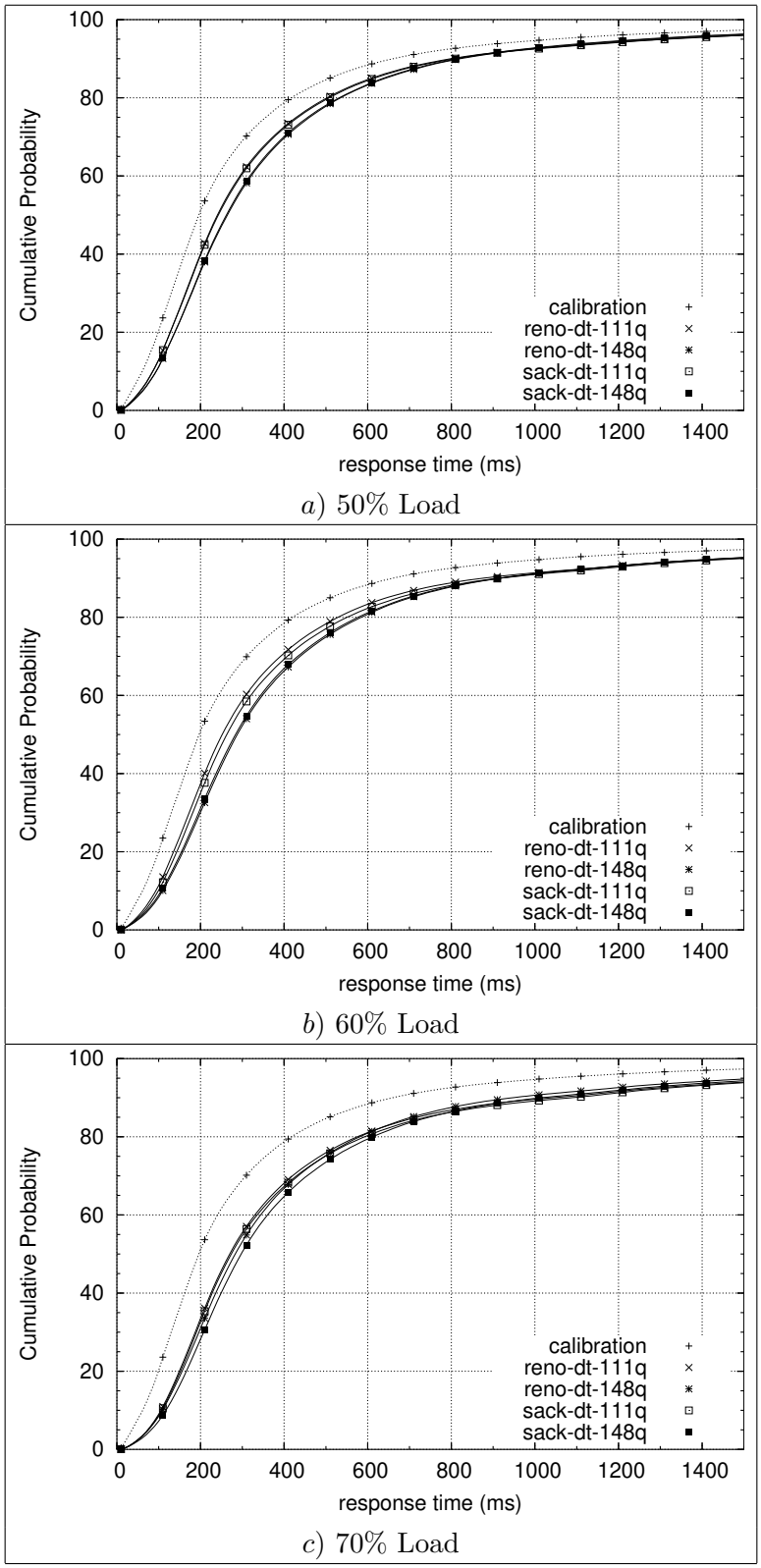


Figure B.18: Response Time CDFs, TCP Reno and TCP SACK with Drop-Tail Queuing, Light Congestion

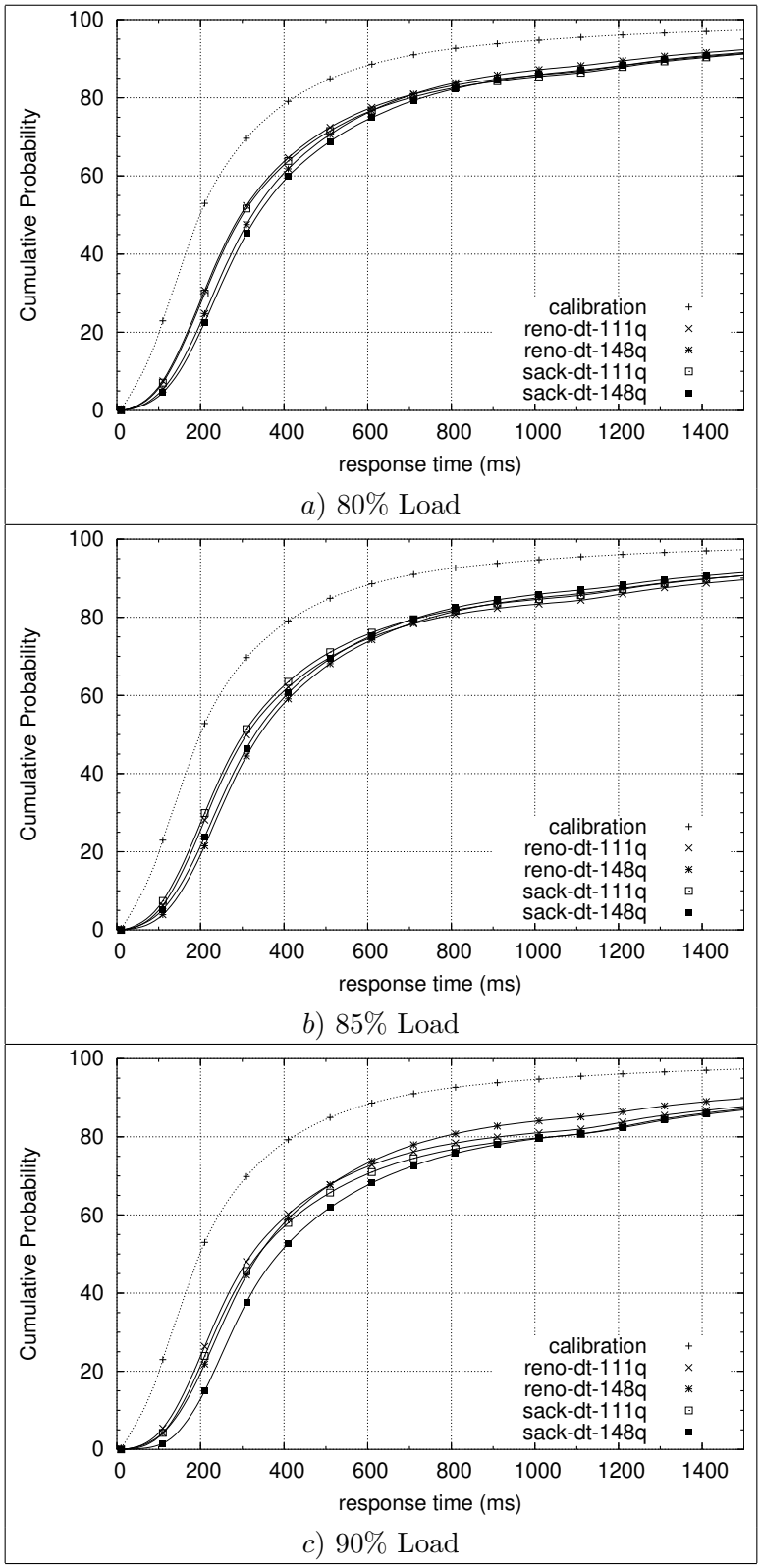


Figure B.19: Response Time CDFs, TCP Reno and TCP SACK with Drop-Tail Queuing, Medium Congestion

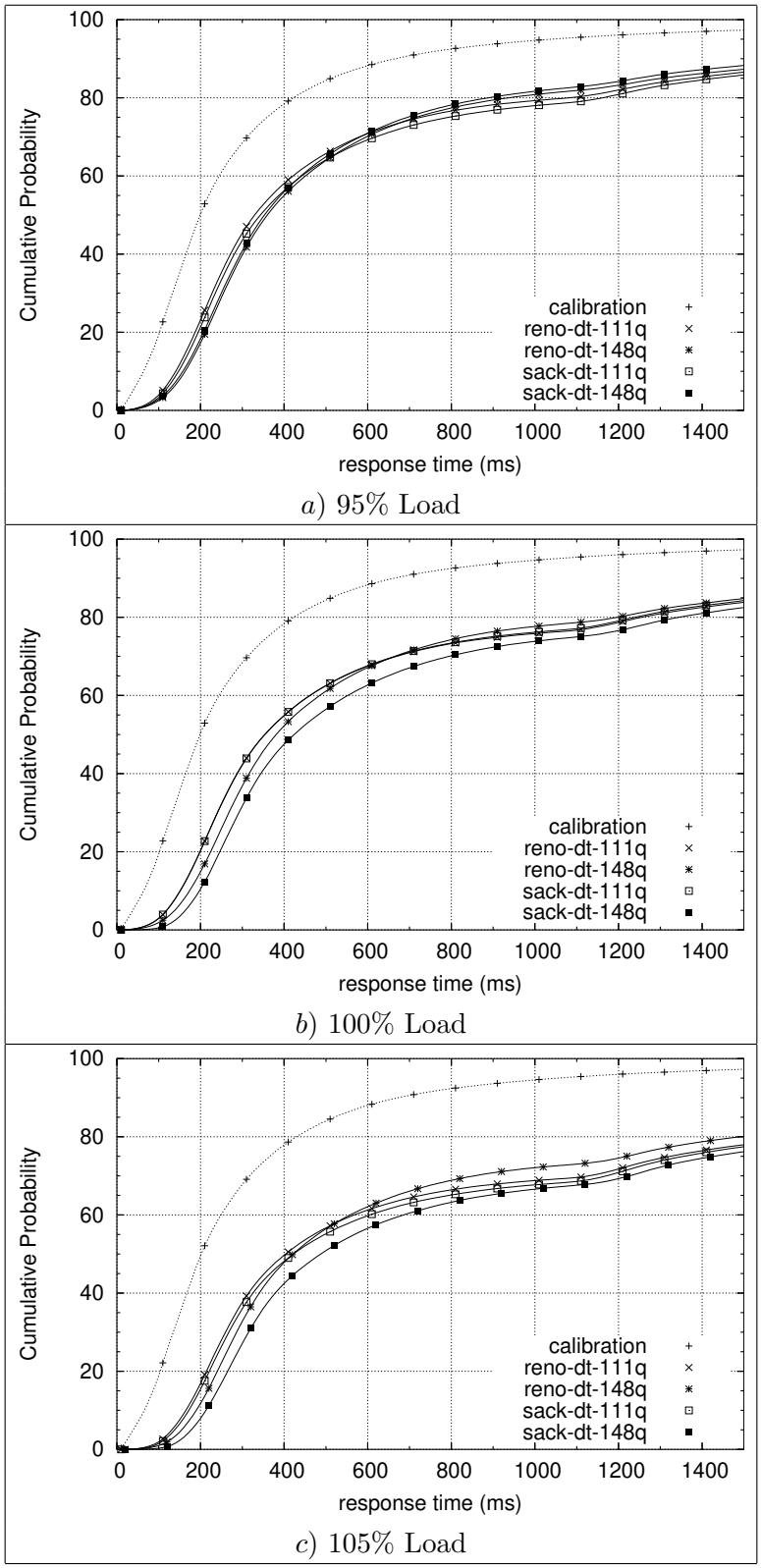


Figure B.20: Response Time CDFs, TCP Reno and TCP SACK with Drop-Tail Queuing, Heavy Congestion

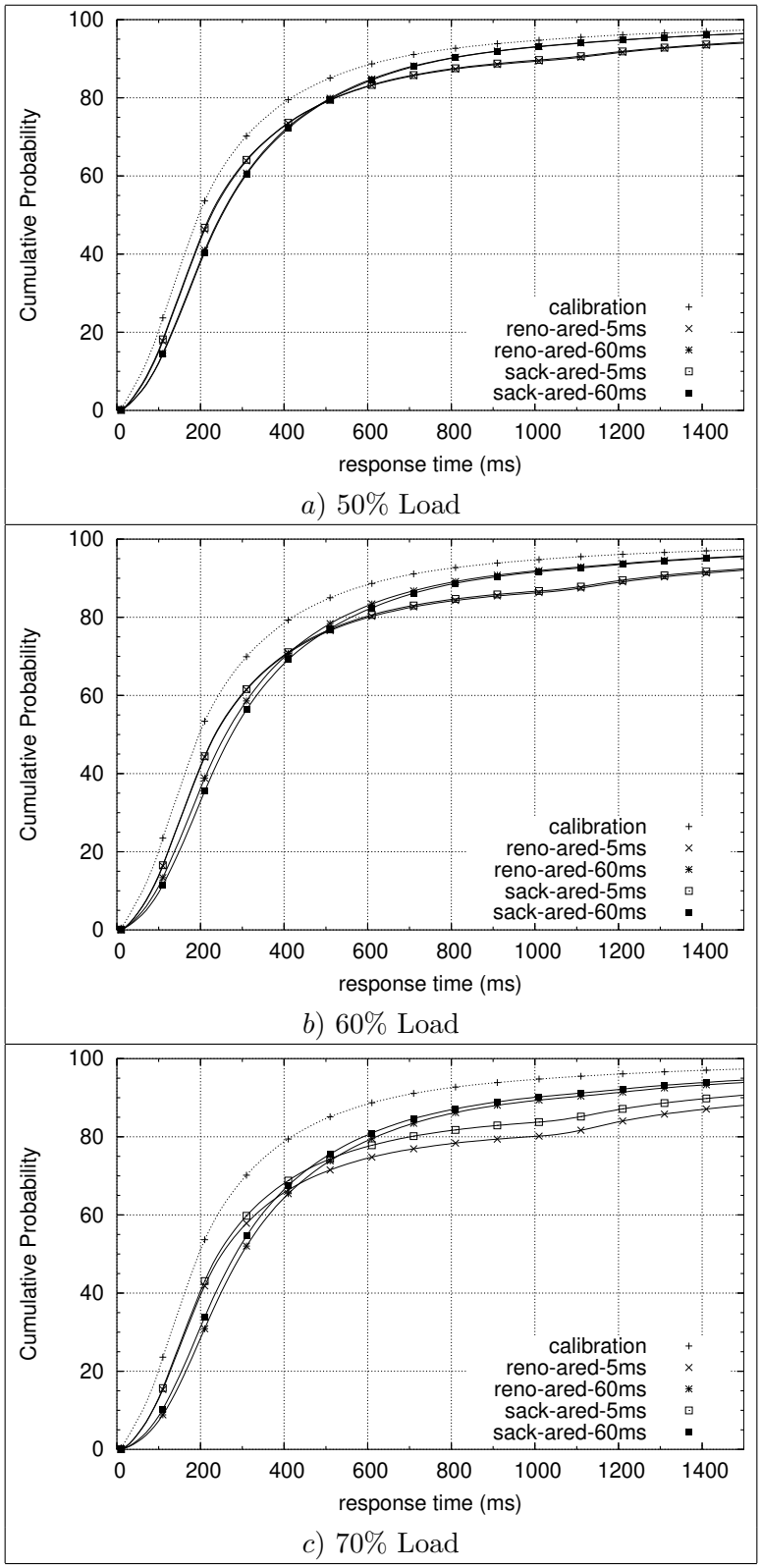


Figure B.21: Response Time CDFs, TCP Reno and TCP SACK with Adaptive RED Queuing, Light Congestion

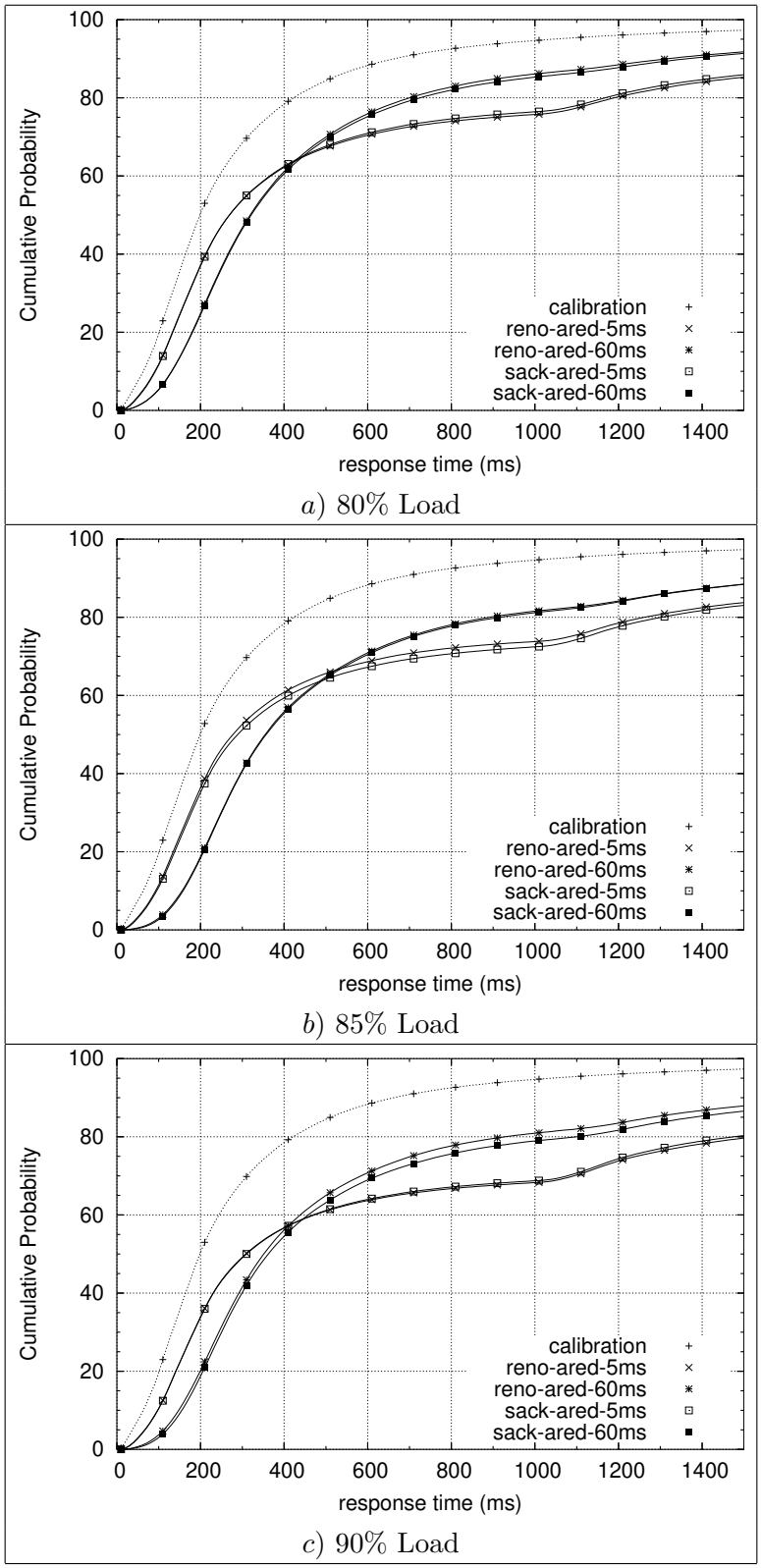


Figure B.22: Response Time CDFs, TCP Reno and TCP SACK with Adaptive RED Queuing, Medium Congestion

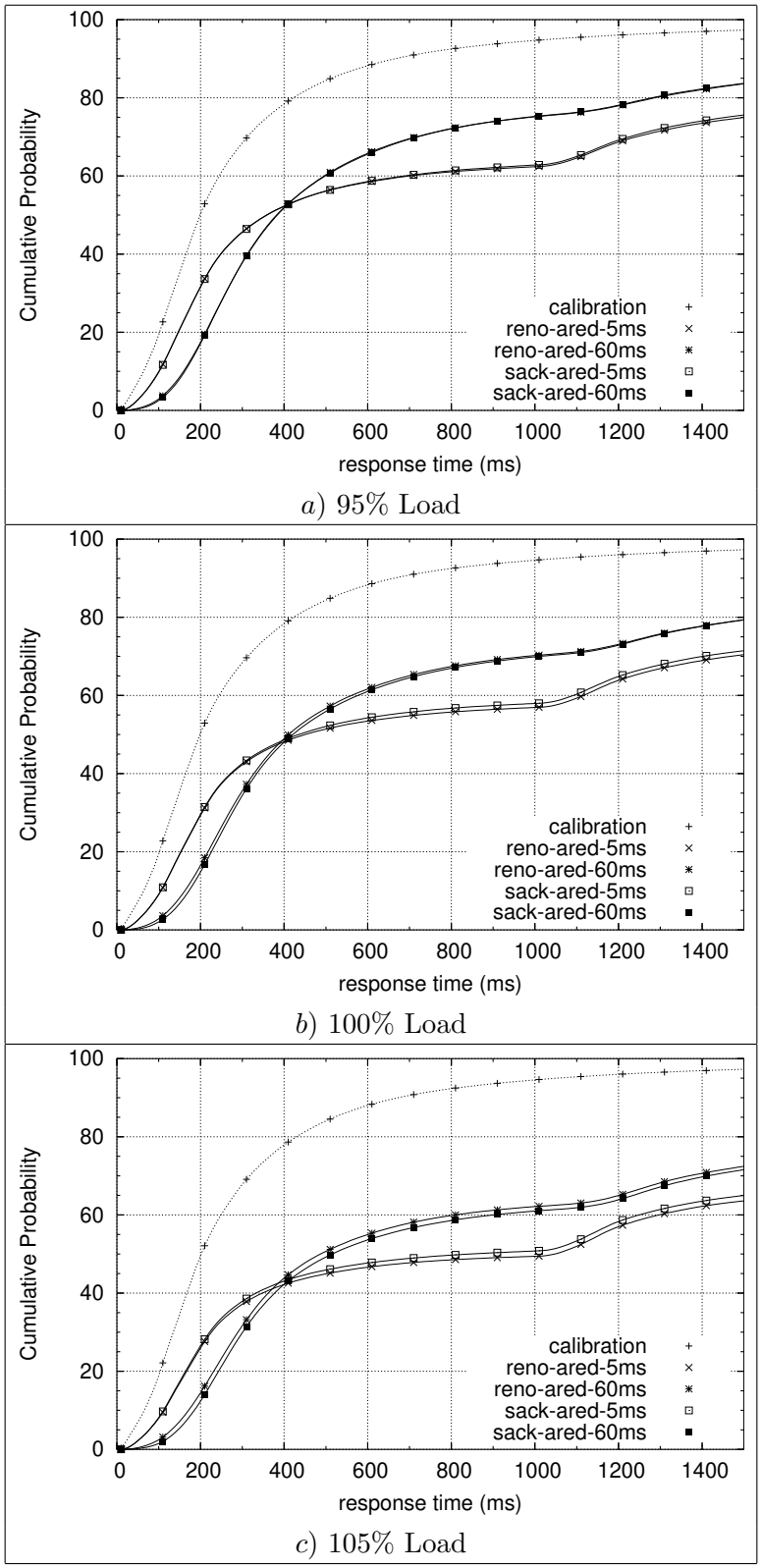


Figure B.23: Response Time CDFs, TCP Reno and TCP SACK with Adaptive RED Queuing, Heavy Congestion

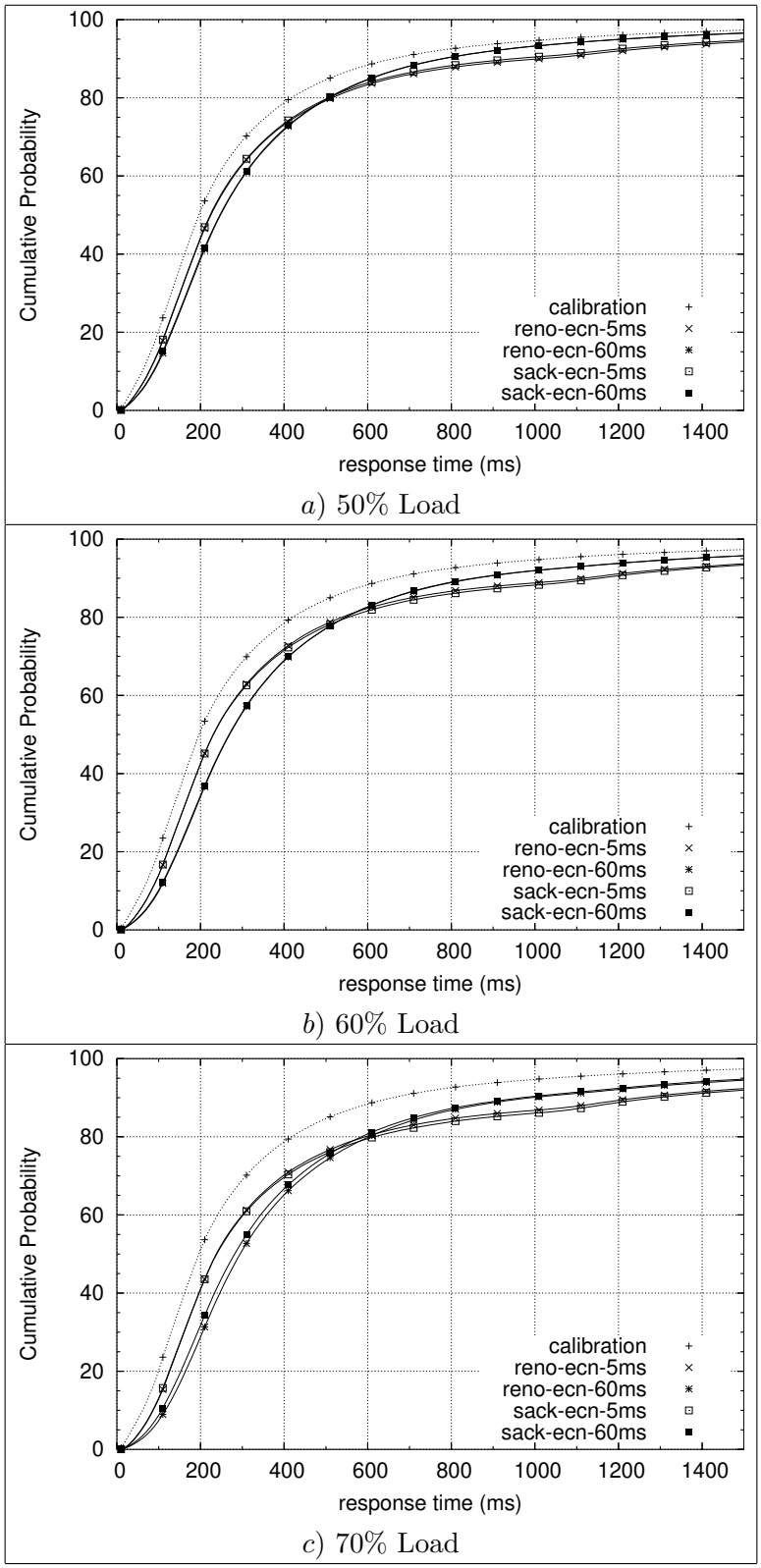


Figure B.24: Response Time CDFs, TCP Reno and TCP SACK with Adaptive RED Queuing and ECN Marking, Light Congestion

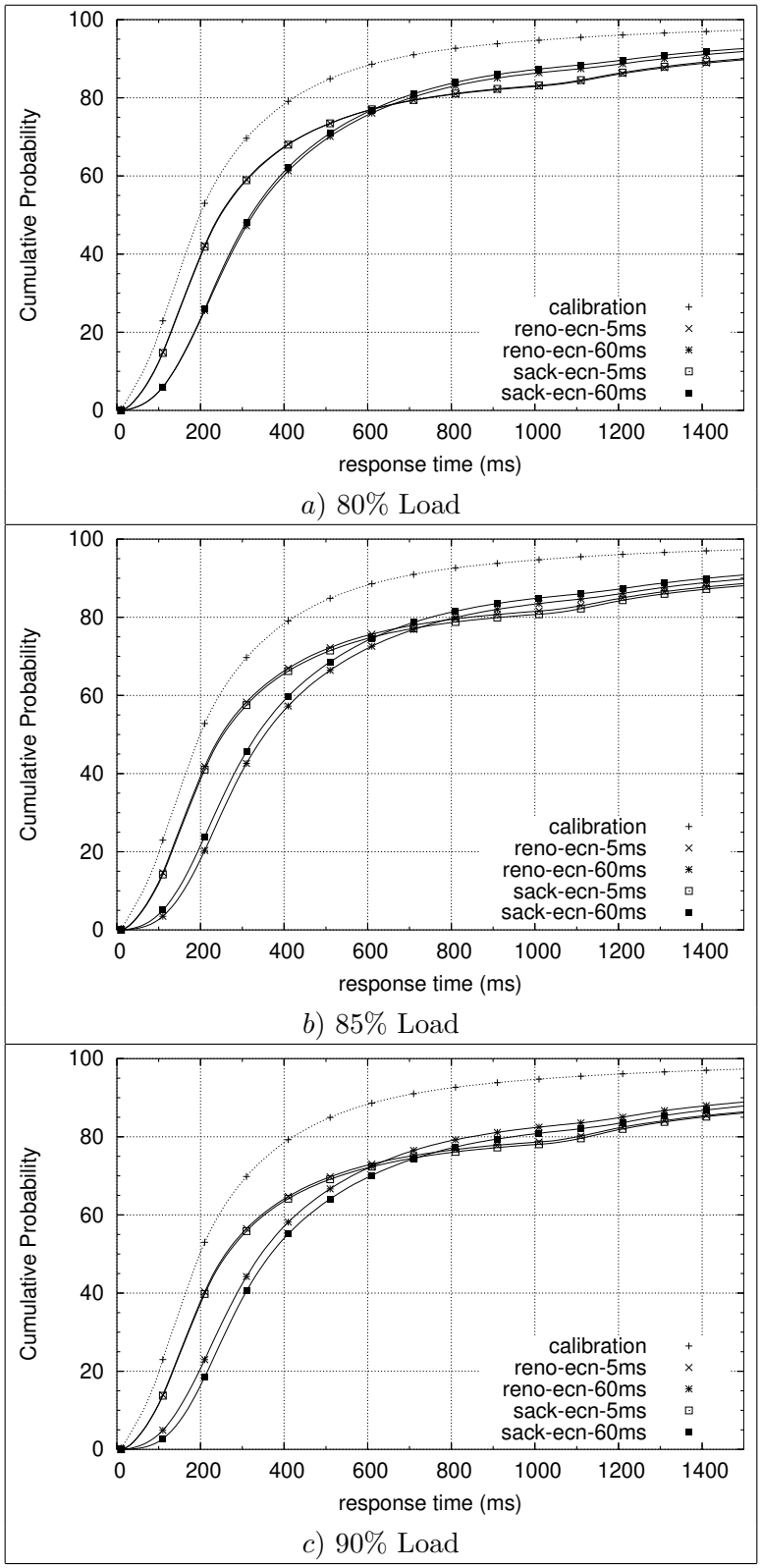


Figure B.25: Response Time CDFs, TCP Reno and TCP SACK with Adaptive RED Queuing and ECN Marking, Medium Congestion

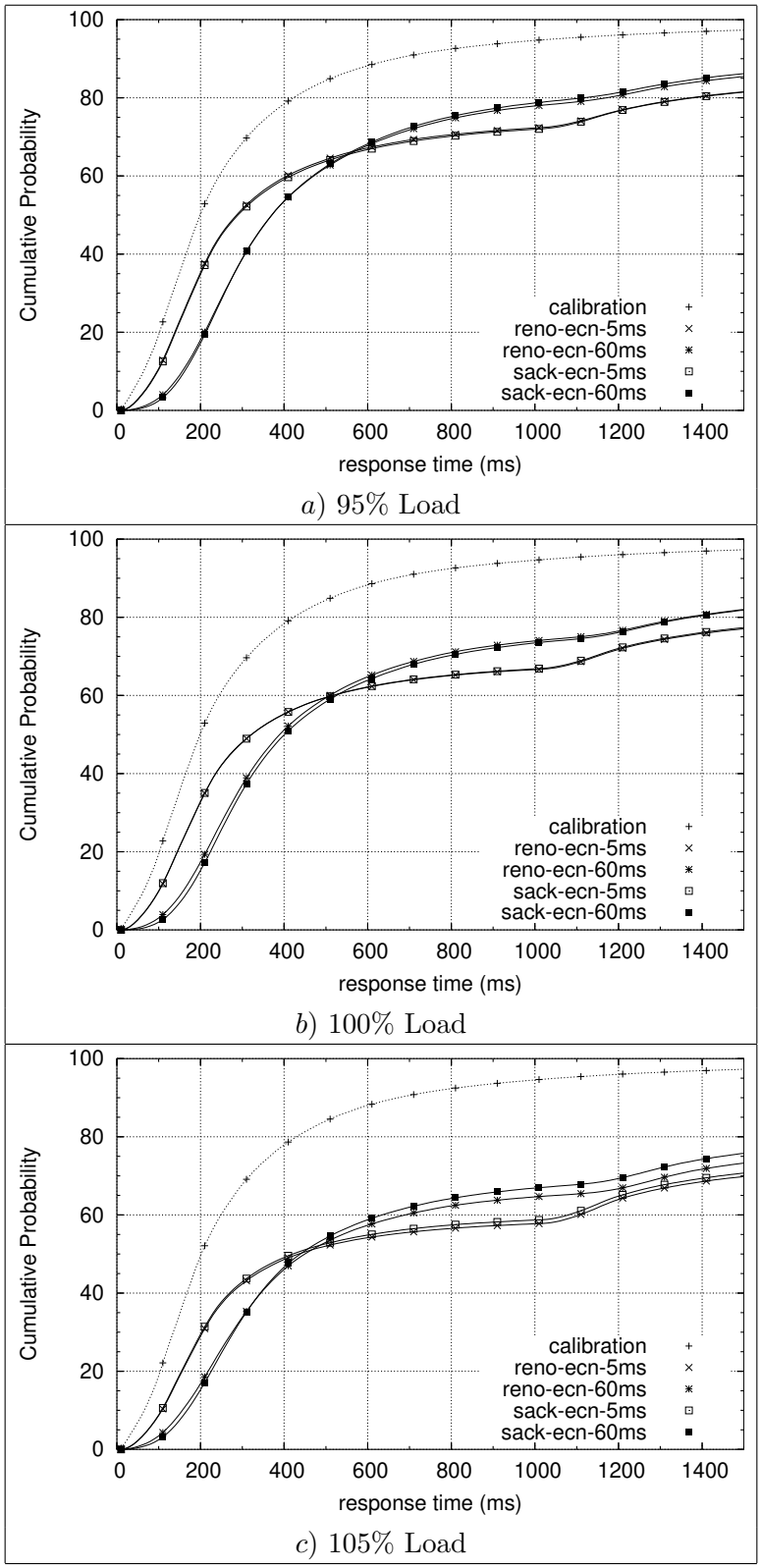


Figure B.26: Response Time CDFs, TCP Reno and TCP SACK with Adaptive RED Queuing and ECN Marking, Heavy Congestion

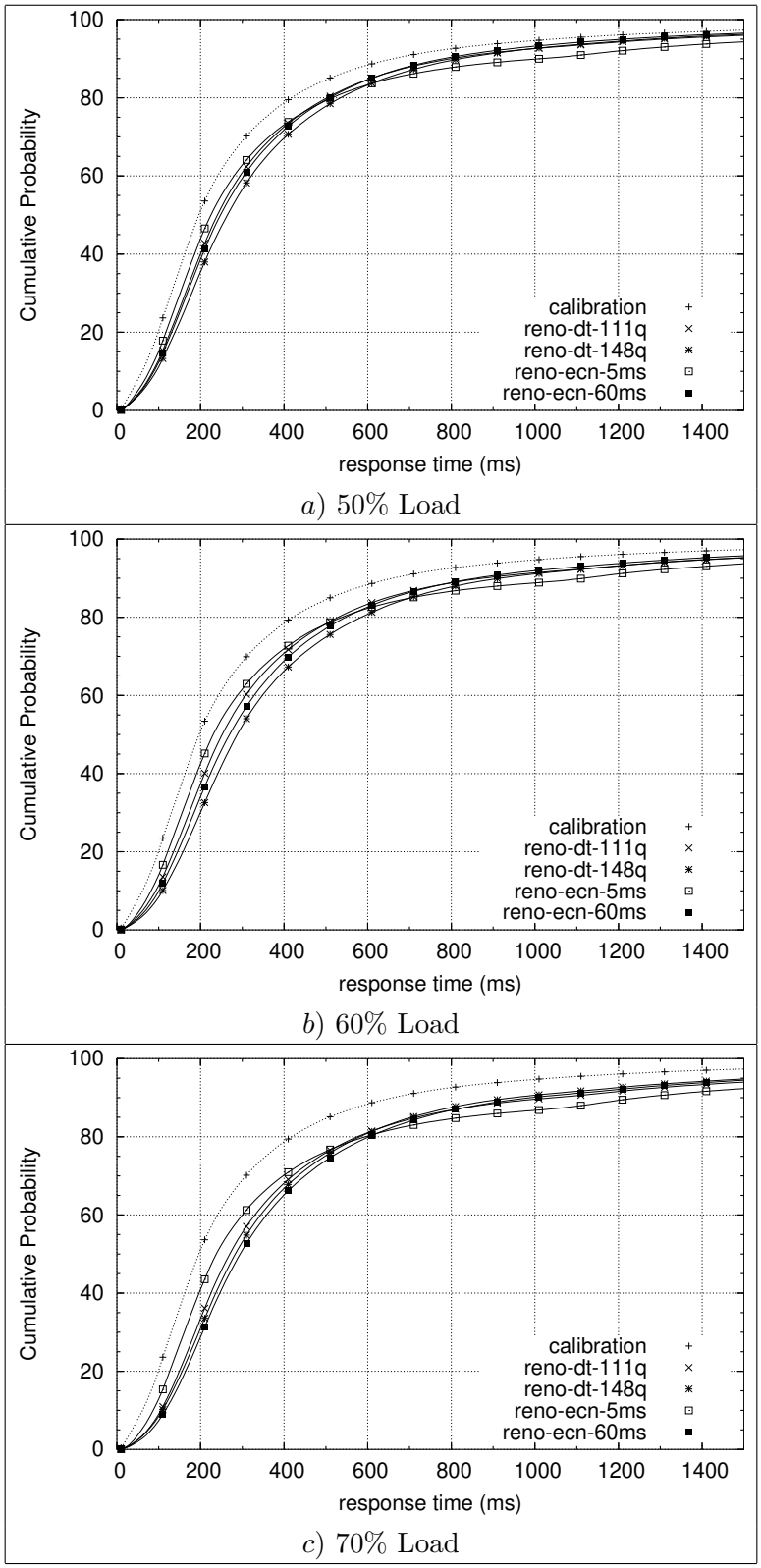


Figure B.27: Response Time CDFs, TCP Reno with Drop-Tail Queuing and with Adaptive RED Queuing and ECN Marking, Light Congestion

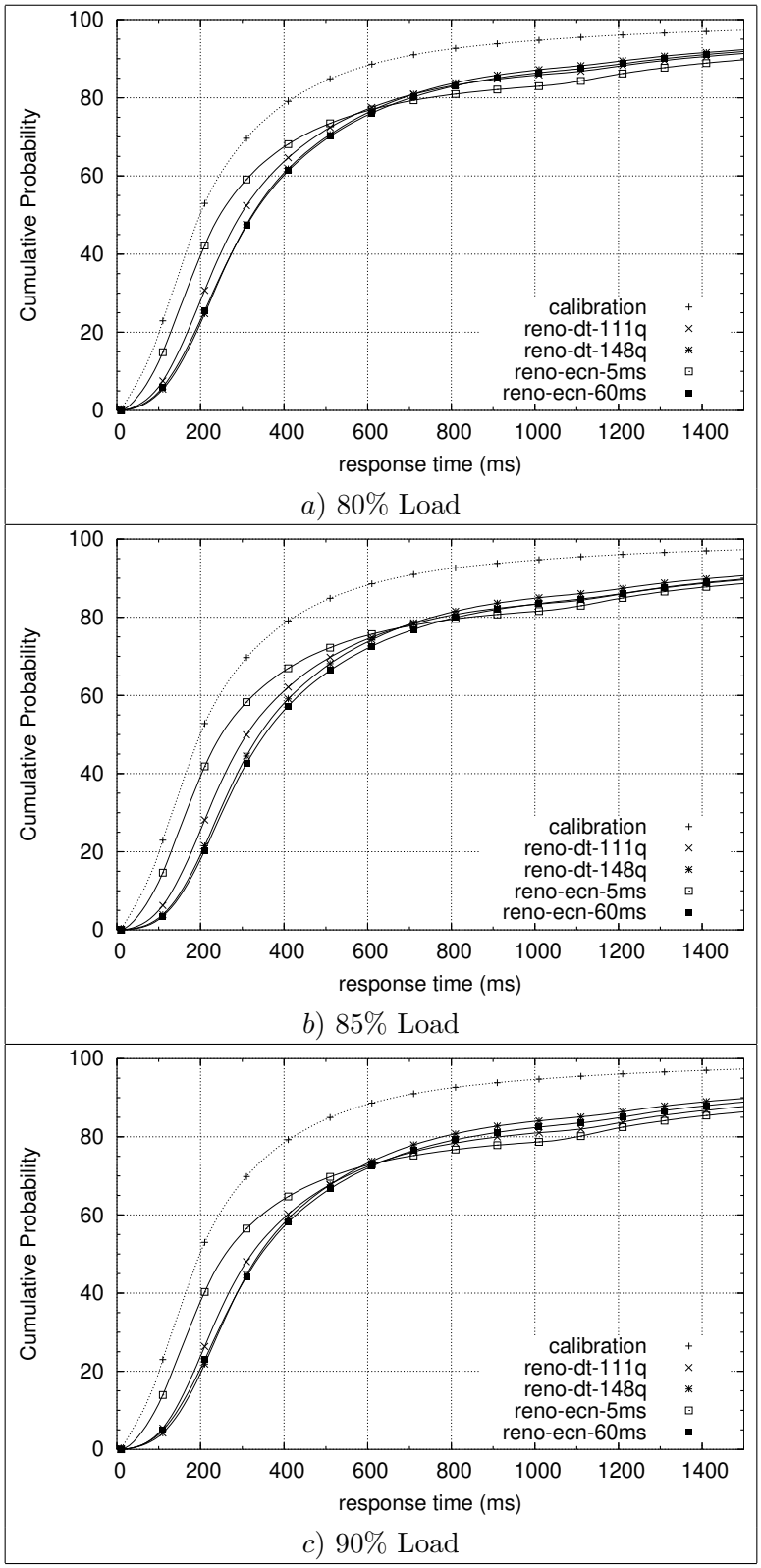


Figure B.28: Response Time CDFs, TCP Reno with Drop-Tail Queuing and with Adaptive RED Queuing and ECN Marking, Medium Congestion

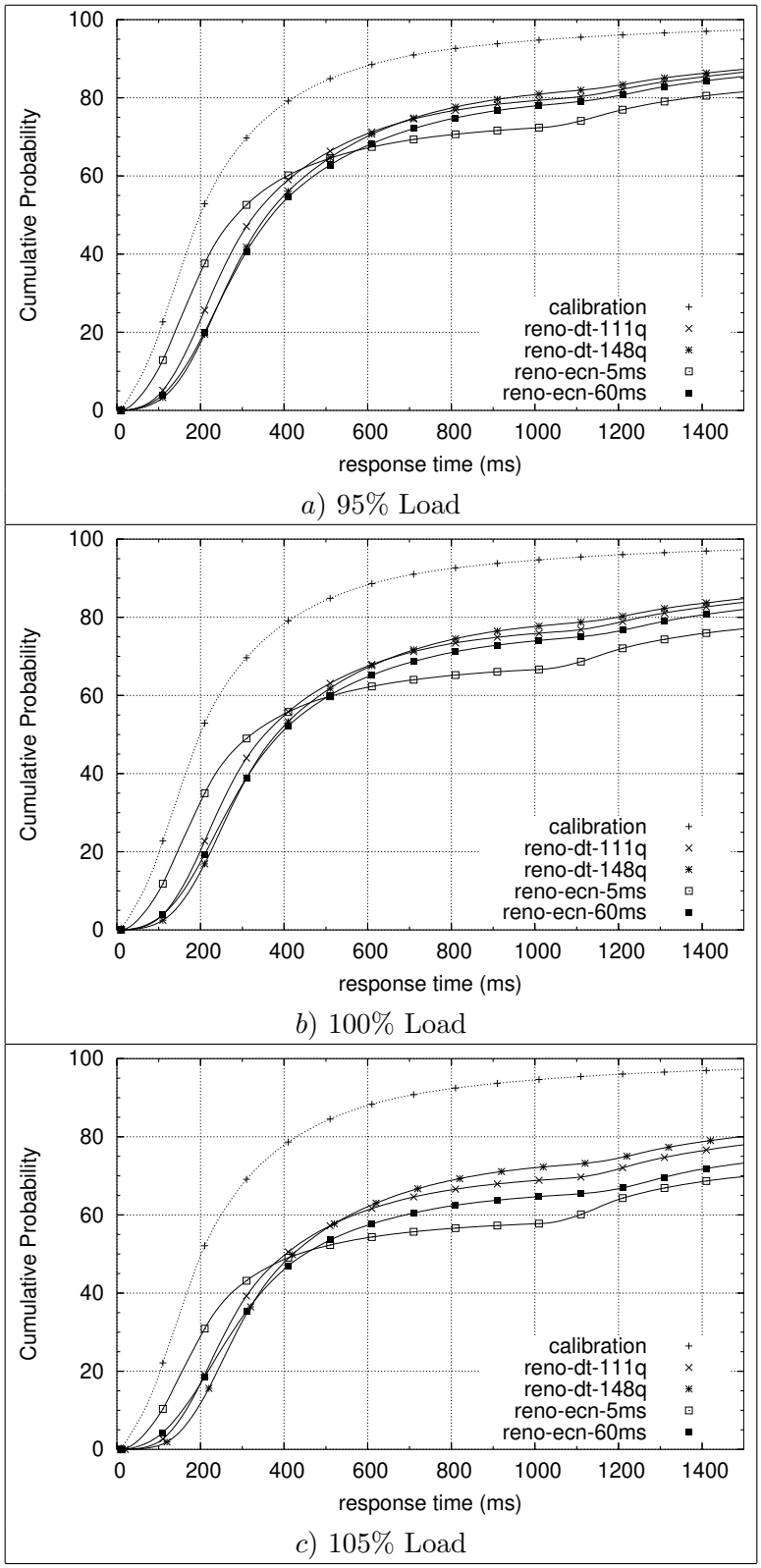


Figure B.29: Response Time CDFs, TCP Reno with Drop-Tail Queuing and with Adaptive RED Queuing and ECN Marking, Heavy Congestion

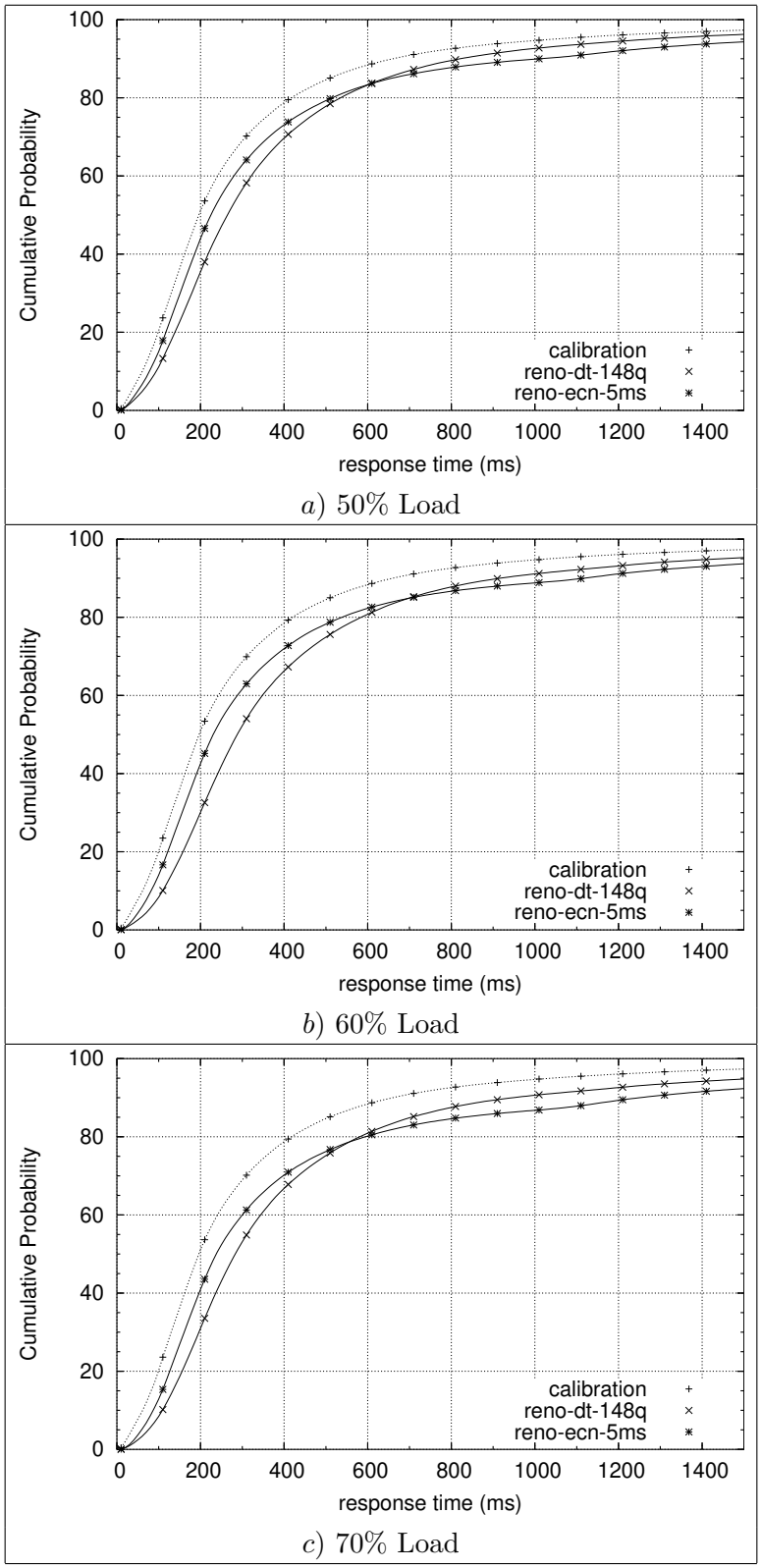


Figure B.30: Response Time CDFs, TCP Reno with Drop-Tail Queuing and with Adaptive RED Queuing and ECN Marking, Light Congestion

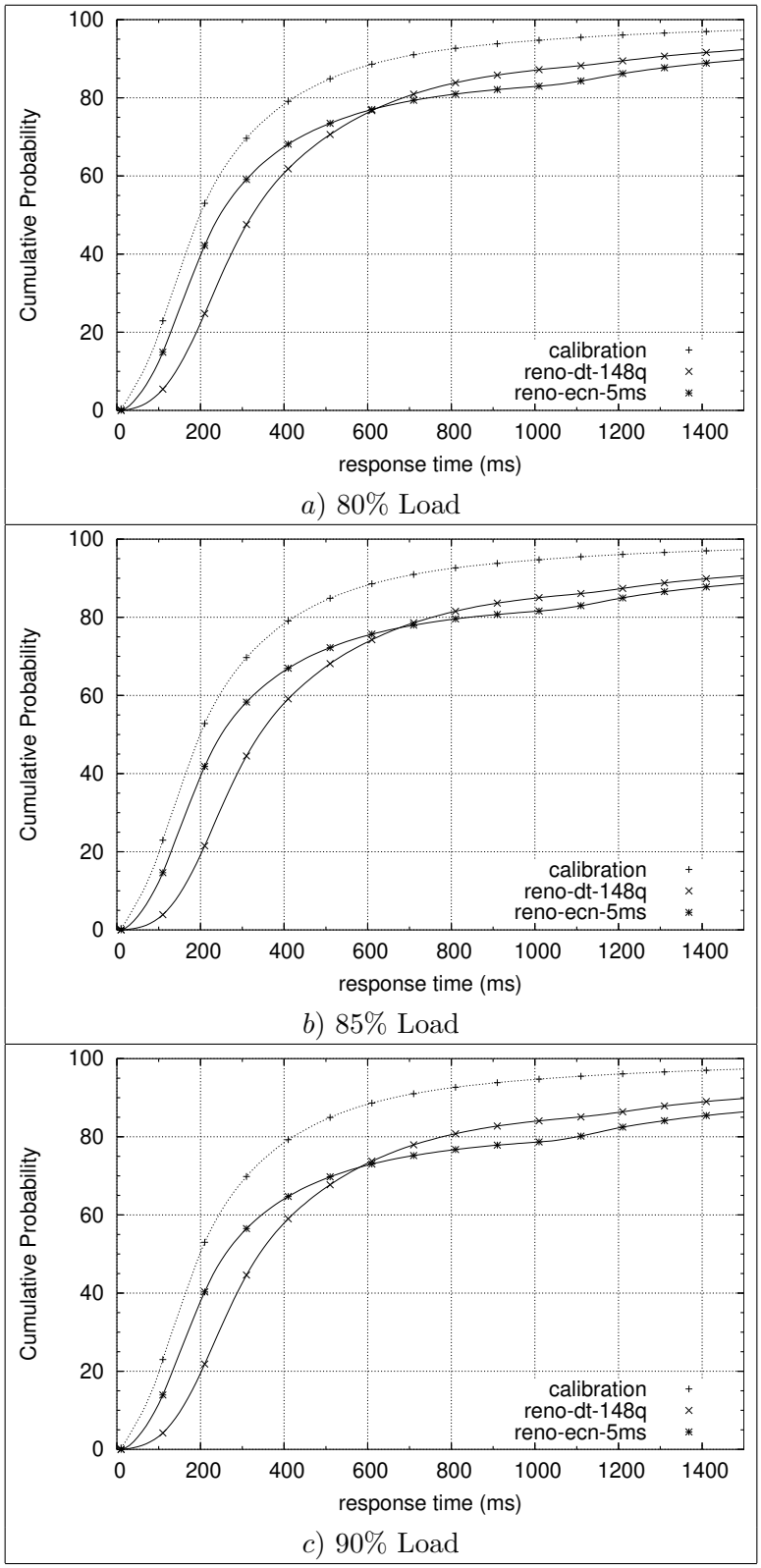


Figure B.31: Response Time CDFs, TCP Reno with Drop-Tail Queuing and with Adaptive RED Queuing and ECN Marking, Medium Congestion

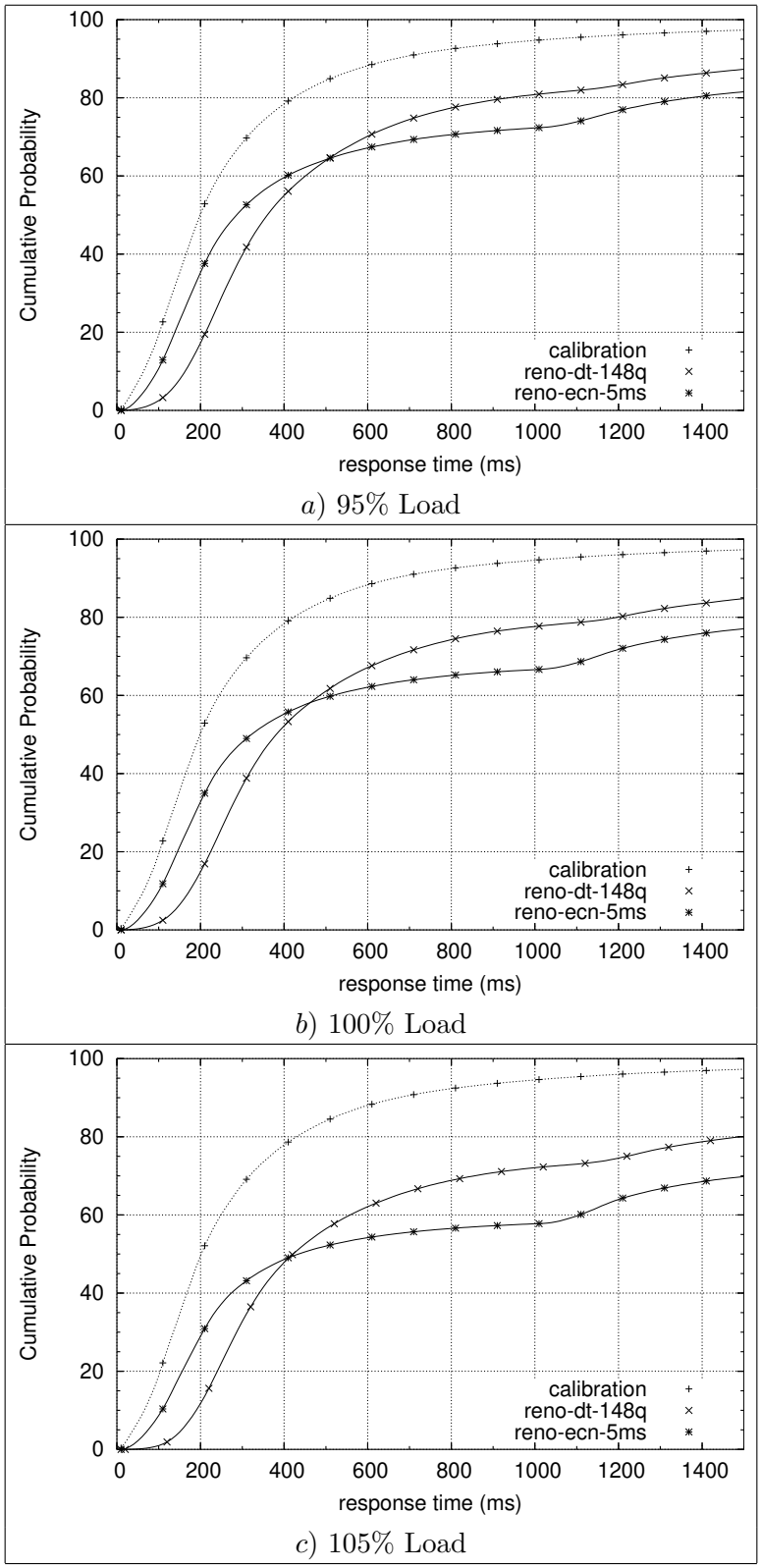


Figure B.32: Response Time CDFs, TCP Reno with Drop-Tail Queuing and with Adaptive RED Queuing and ECN Marking, Heavy Congestion

portions of the total set of responses. Further complicating the tradeoff is the result that Reno-DT-148q gives higher link utilization (Figure B.3a) along with a high average queue size (Figure B.2c), while Reno-ECN-5ms gives low average queue sizes, but also lower link utilization.

B.2.6 Performance for Offered Loads Less Than 80%

For load levels lower than 80%, there is an advantage for Reno-ECN-5ms over Reno-DT-148q for shorter responses (Figure B.30). The same tradeoff is present for 50-80% load as with loads over 80% and hence is a fundamental tradeoff. Reno-ECN-5ms has lower average queue sizes (Figure B.2c) and the corresponding better response time performance for shorter responses, with similar link utilization as Reno-DT-148q (Figure B.3a). Reno-DT-148q performs better for responses that take longer than 600 ms to complete.

Much below 80% offered load, TCP SACK and TCP Reno have identical performance (Figures B.21, B.24, and B.18) and Adaptive RED with ECN performs only marginally better than drop-tail (Figure B.27). At these loads, there is no difference in link utilization between any of the queue management techniques (Figure B.3a). There is also very little packet loss (no average loss over 3%) (Figure B.2a). For loads under 80%, given the complexity of implementing RED, there is no compelling reason to use Adaptive RED with ECN over drop-tail.

B.2.7 Summary

I found that there is a tradeoff in HTTP response times between TCP Reno with drop-tail queuing and ECN-enabled TCP Reno with Adaptive RED queuing. I found no marked difference for HTTP response times between the use of TCP Reno and TCP SACK with either drop-tail or Adaptive RED queuing. In summary, I drew the following conclusions on the performance of HTTP flows:

- There is no clear benefit in using TCP SACK over TCP Reno, especially when the complexity of implementing TCP SACK is considered. This result holds independent of load and pairing with queue management algorithm.
- As expected, Adaptive RED with ECN marking performs better than Adaptive RED with packet dropping, and the value of ECN marking increases as the offered load increases. ECN also offers more significant gains in performance when the target delay is small (5 ms).
- Unless congestion is a serious concern (*i.e.*, for average link utilizations of 80% or higher with bursty sources), there is little benefit to using RED queue management in routers.
- Adaptive RED with ECN marking and a small target delay (5 ms) performs better than drop-tail with 2xBDP queue size at offered loads having moderate levels of congestion

(80% load). This finding should be tempered with the caution that, like RED, Adaptive RED is also sensitive to parameter settings.

- At loads that cause severe congestion, there is a complex performance trade-off between drop-tail with 2xBDP queue size and Adaptive RED with ECN at a small target delay. Adaptive RED can improve the response time of about half the responses but worsens the other half. Link utilization is significantly better with drop-tail.
- At all loads there is little difference between the performance of drop-tail with 2xBDP queue size and Adaptive RED with ECN marking and a larger target delay (60 ms).

B.3 Tuning Adaptive RED

The evaluation presented above was performed on a network topology with only two PackMime server clouds and two PackMime client clouds on each end of the network (Figure B.1). The evaluation of Sync-TCP described in Chapter 4 was performed with five PackMime server clouds and five PackMime client clouds on each end of the network (Figure B.46). In order to compare the performance of Adaptive RED to Sync-TCP, I ran some of the Adaptive RED experiments using the same topology as was used for the Sync-TCP evaluation.

My goal was to compare the performance of Sync-TCP against the best standards-track TCP error-recovery and queue management combination. Since I found no difference between TCP Reno and TCP SACK for HTTP traffic, I ran experiments using TCP Reno with drop-tail queuing. These were used as my baseline comparison. For further comparison, I chose to look at ECN-enabled TCP SACK with Adaptive RED queuing and ECN marking.

I wanted to choose the best parameters for Adaptive RED for comparison with Sync-TCP. From the experiments described in section B.2, I found that a small target delay, such as 5 ms, gives better response time performance than drop-tail for some flows, but worse performance for the rest of the flows. If the target delay of the Adaptive RED router was set so that the target queue size was much smaller than the average achieved by the drop-tail router, there was a tradeoff. If the average queue sizes were similar, so was HTTP response time performance. So, in the Sync-TCP evaluation, I tested both Adaptive RED with a target delay of 5 ms and a target delay tuned to have a target queue size the same as the Sync-TCP average queue size at each load level. Below 80%, this target delay was less than 5 ms, so I only ran this "tuned" Adaptive RED for loads 80-105%. Table B.3 shows the average queue size target and target delay for loads over 80%.

In previous evaluations of Adaptive RED, I used an essentially infinite maximum queue size of 5xBDP. In the network setup used in the evaluation of Sync-TCP, 5xBDP is 340 packets (Chapter 4). Using a large maximum queue size could allow large queues to build up. Allowing a large average queue will lead to increased queuing delays, which lead to long

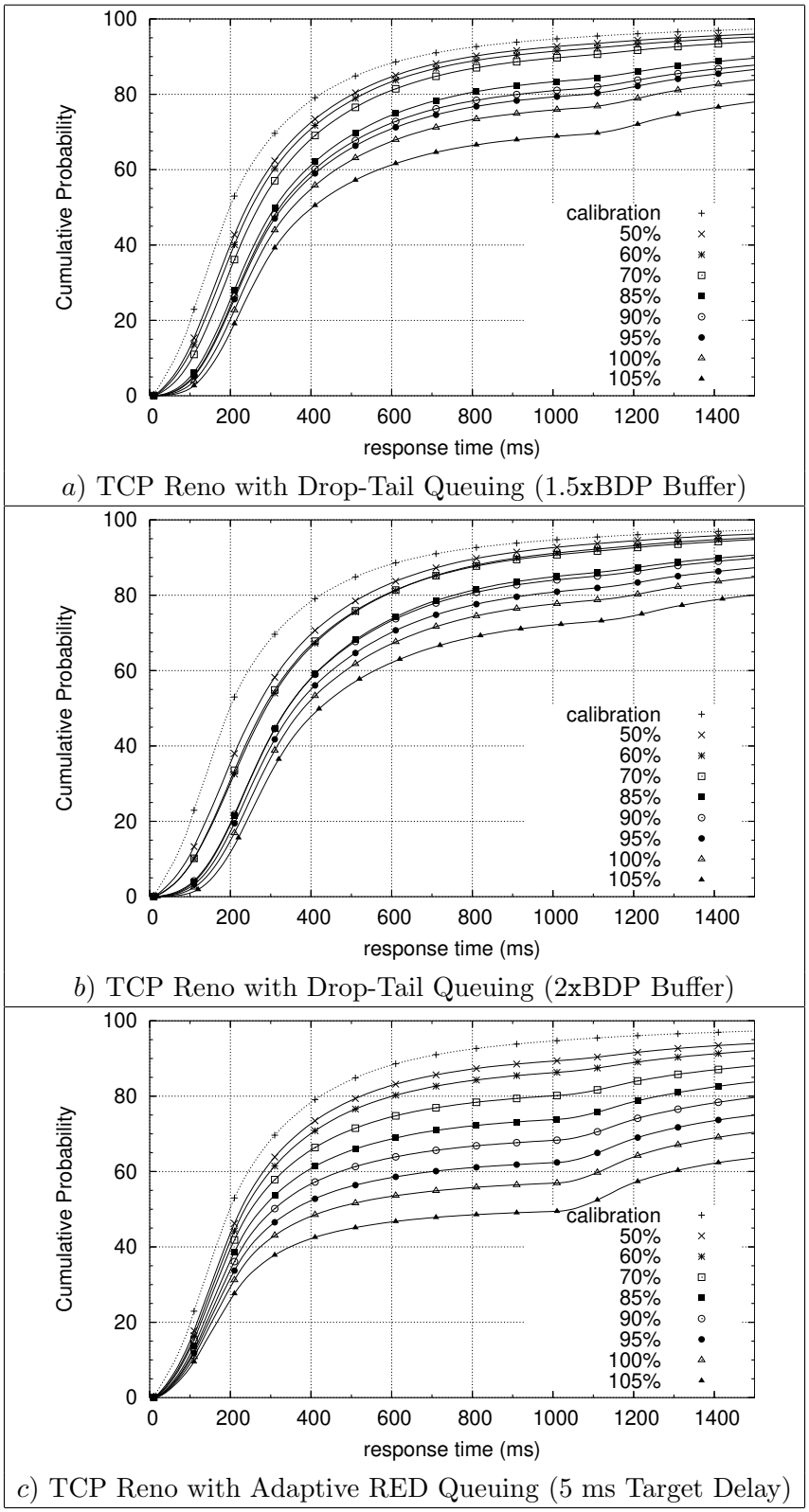
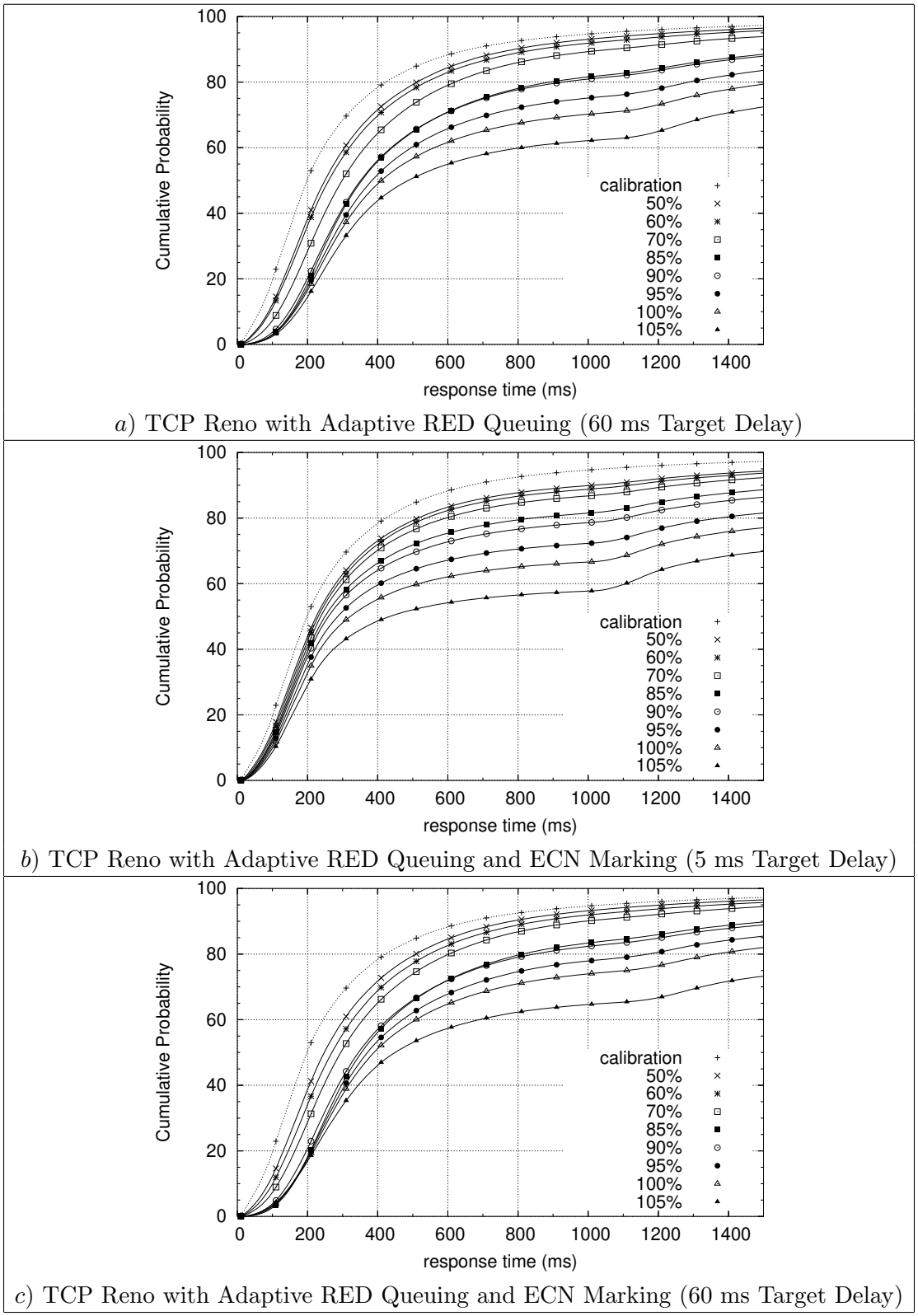


Figure B.33: Response Time CDFs, TCP Reno with Drop-Tail Queuing and with Adaptive RED Queuing, All Levels of Congestion



a) TCP Reno with Adaptive RED Queuing (60 ms Target Delay)

b) TCP Reno with Adaptive RED Queuing and ECN Marking (5 ms Target Delay)

c) TCP Reno with Adaptive RED Queuing and ECN Marking (60 ms Target Delay)

Figure B.34: Response Time CDFs, TCP Reno with Adaptive RED Queuing and with Adaptive RED Queuing and ECN Marking, All Levels of Congestion

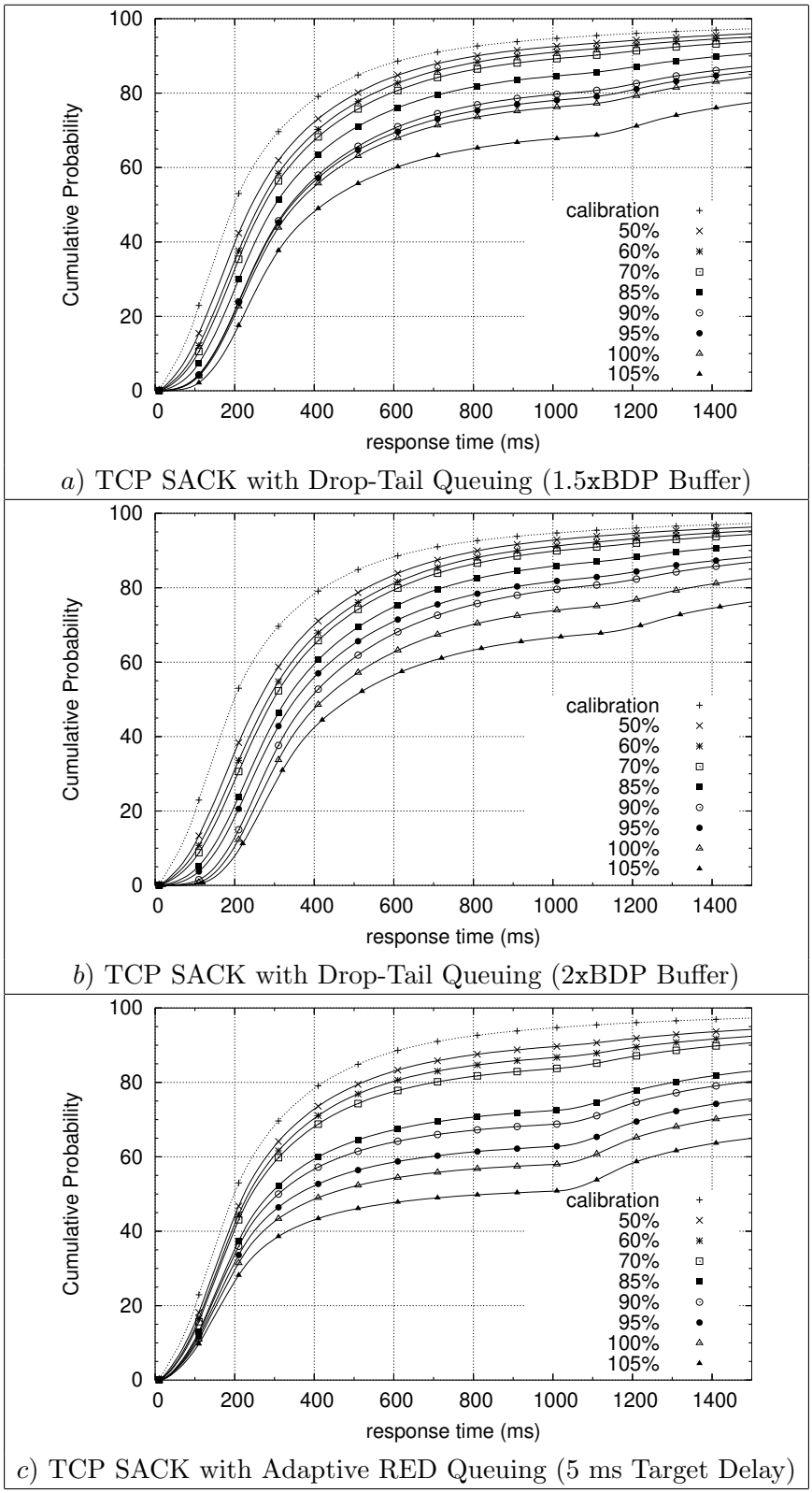
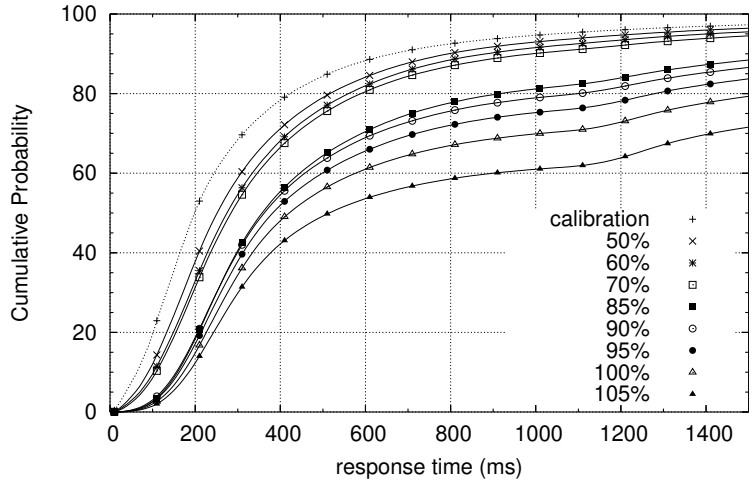
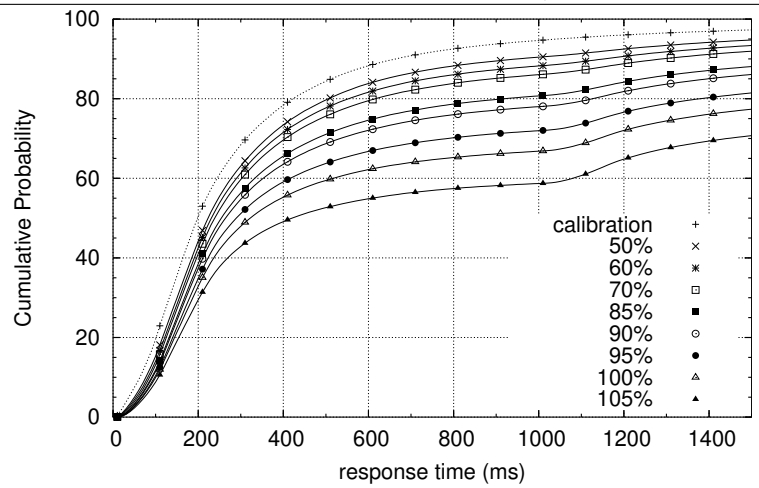


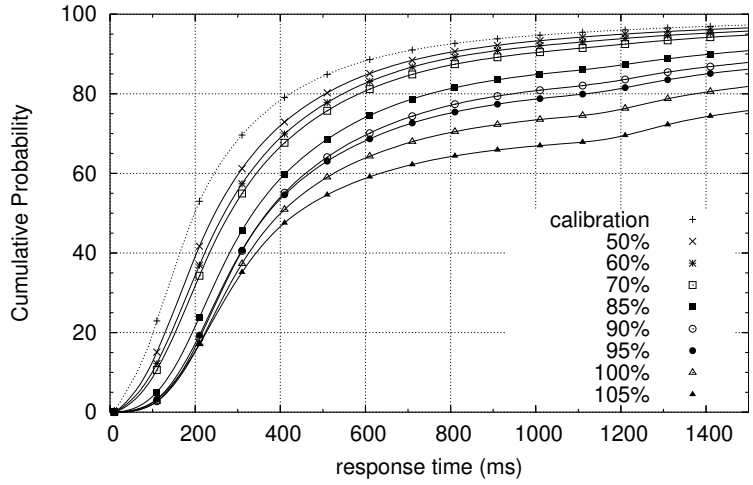
Figure B.35: Response Time CDFs, TCP SACK with Drop-Tail Queuing and with Adaptive RED Queuing, All Levels of Congestion



a) TCP SACK with Adaptive RED Queuing (60 ms Target Delay)



b) TCP SACK with Adaptive RED Queuing and ECN Marking (5 ms Target Delay)



c) TCP SACK with Adaptive RED Queuing and ECN Marking (60 ms Target Delay)

Figure B.36: Response Time CDFs, TCP SACK with Adaptive RED Queuing and with Adaptive RED Queuing and ECN Marking, All Levels of Congestion

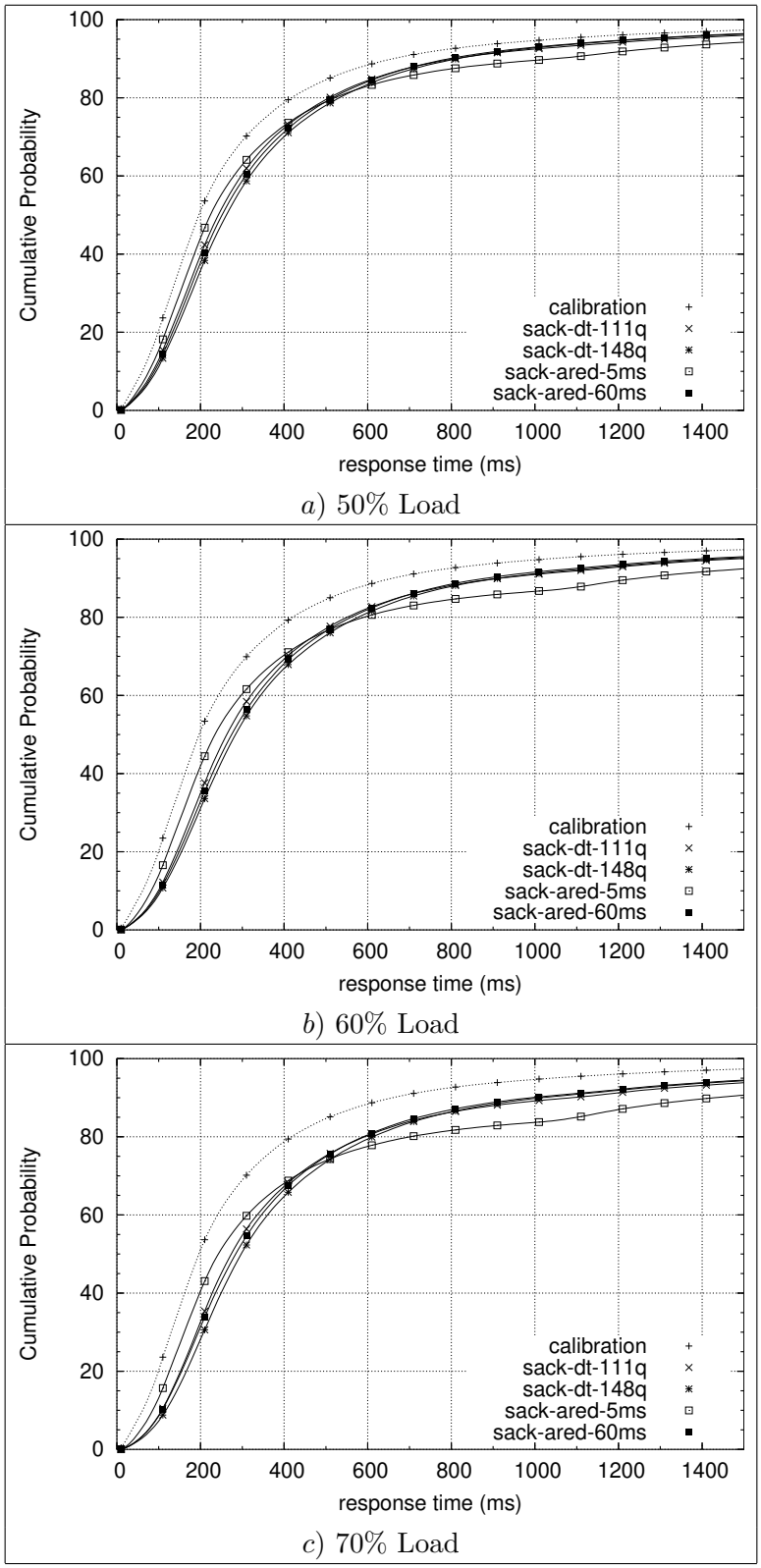


Figure B.37: Response Time CDFs, TCP SACK with Drop-Tail Queuing and Adaptive RED Queuing, Light Congestion

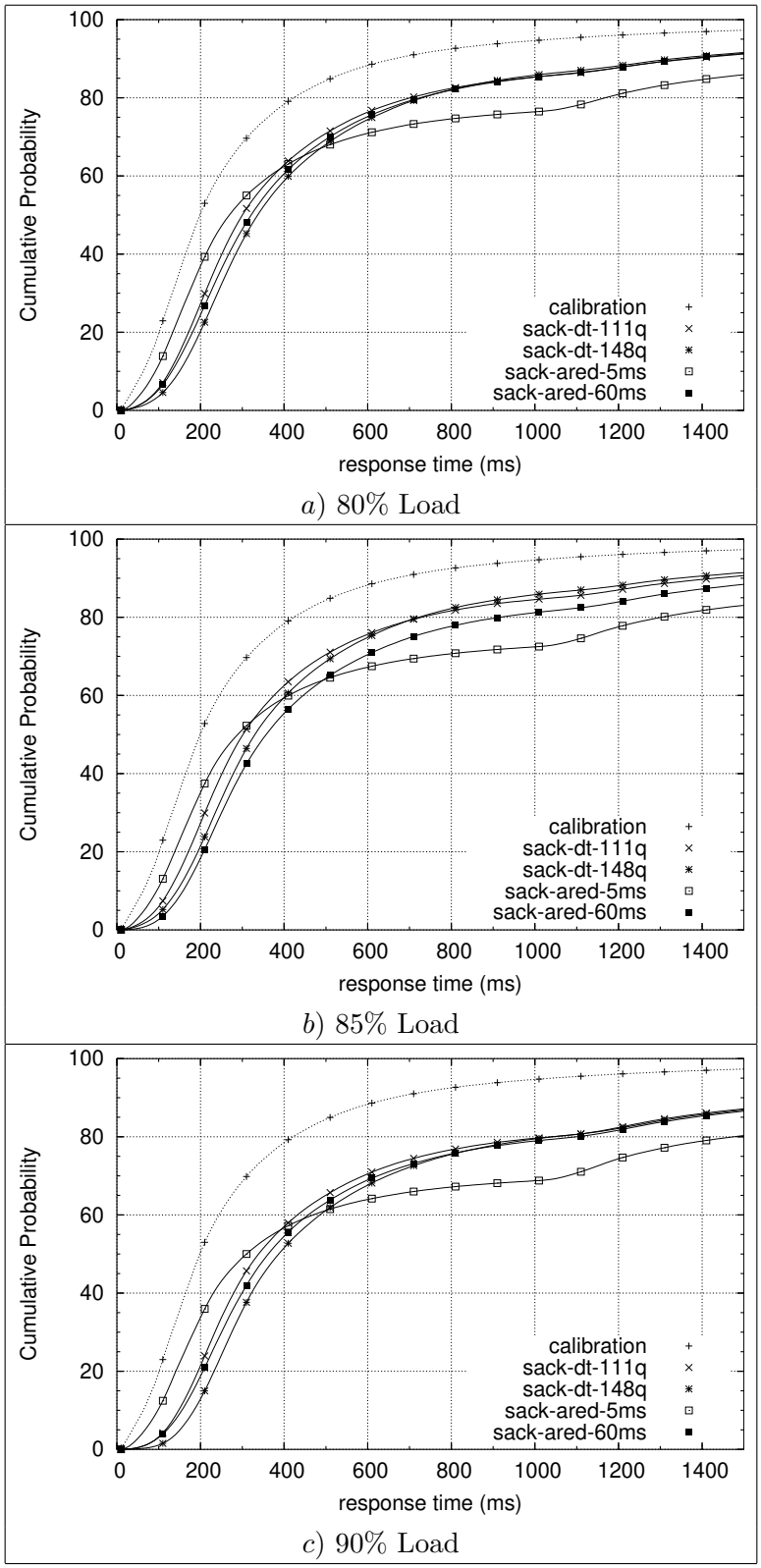


Figure B.38: Response Time CDFs, TCP SACK with Drop-Tail Queuing and Adaptive RED Queuing, Medium Congestion

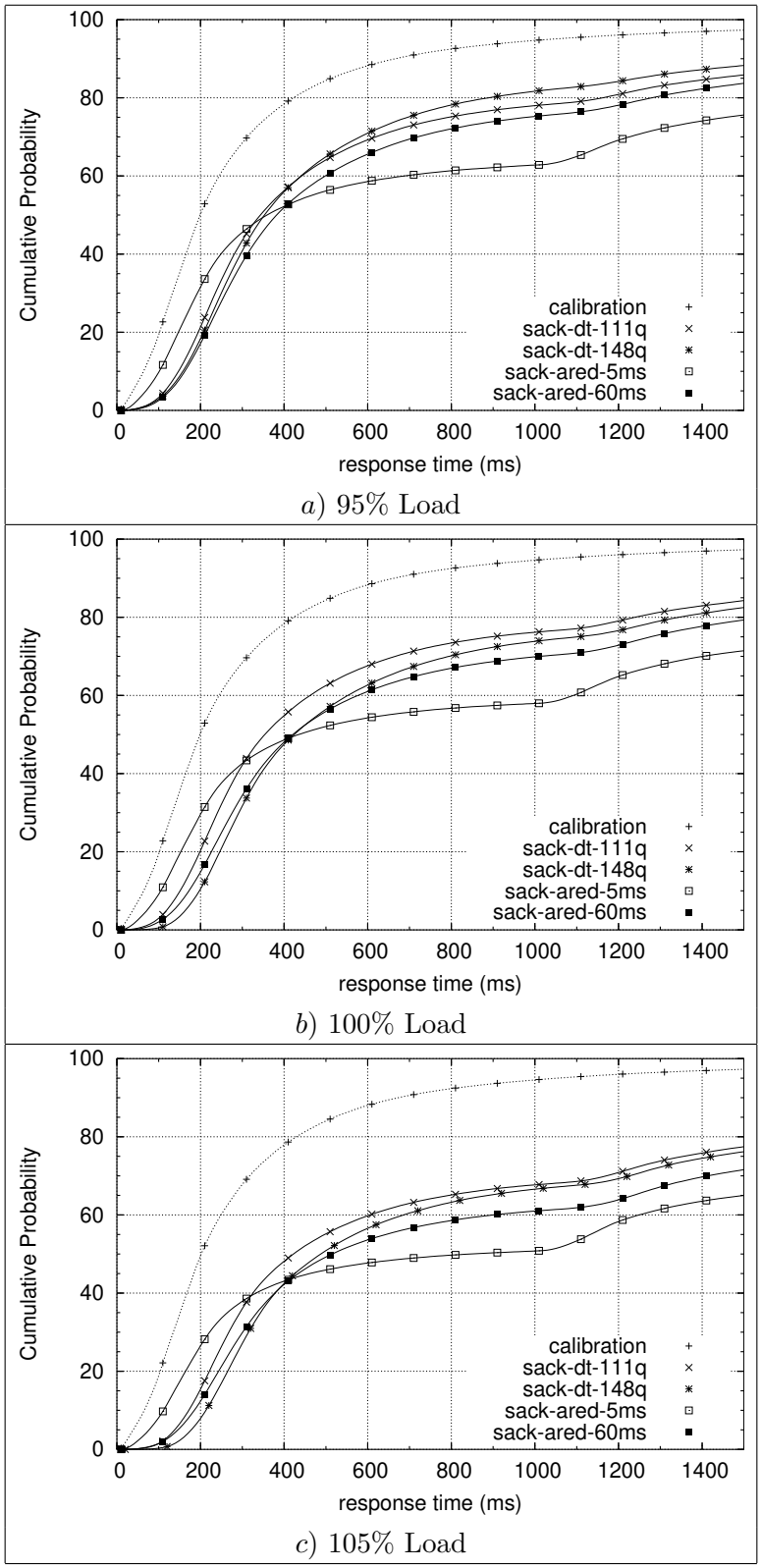


Figure B.39: Response Time CDFs, TCP SACK with Drop-Tail Queuing and Adaptive RED Queuing, Heavy Congestion

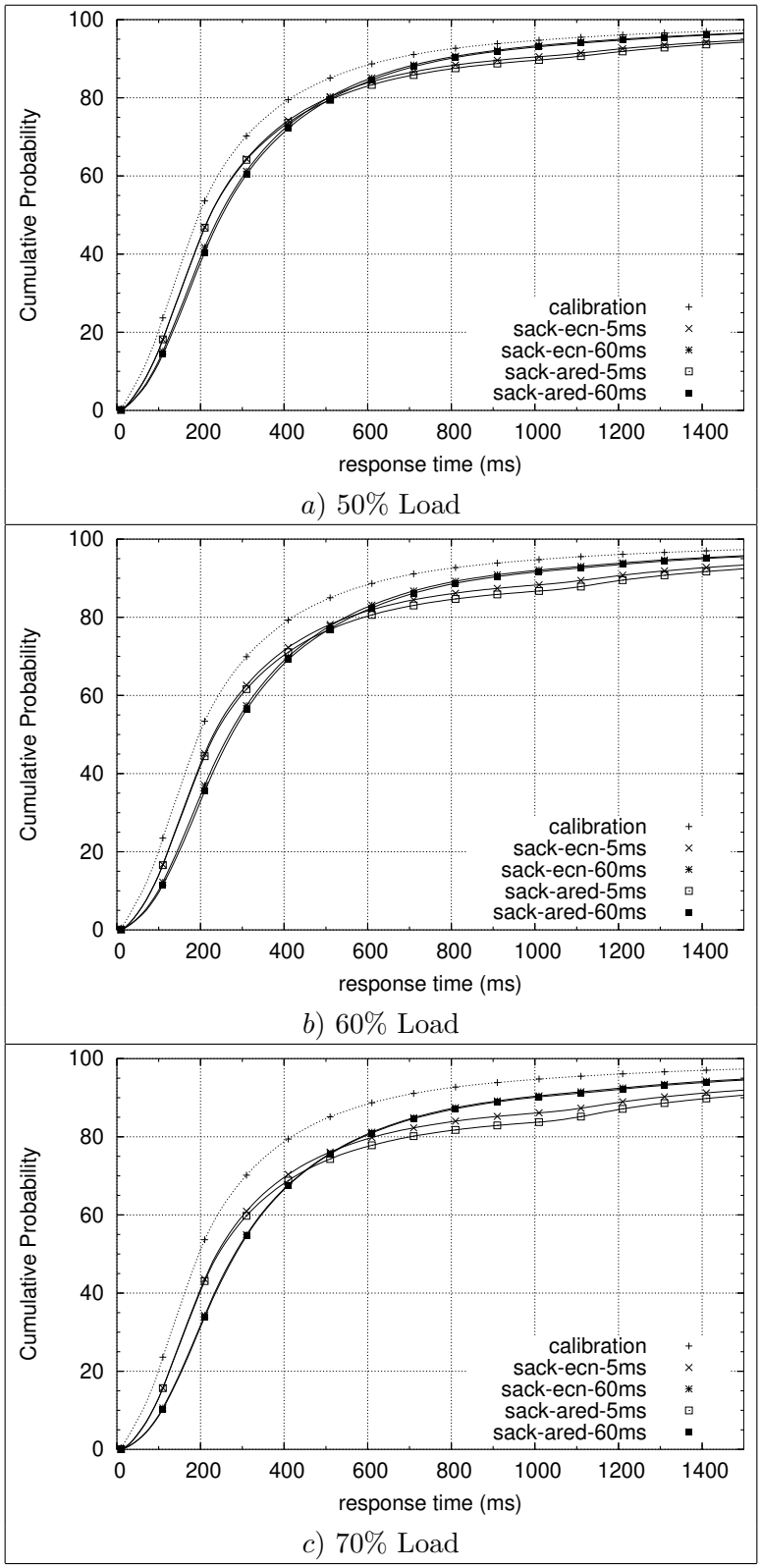


Figure B.40: Response Time CDFs, TCP SACK with Adaptive RED Queuing and with Adaptive RED Queuing and ECN Marking, Light Congestion

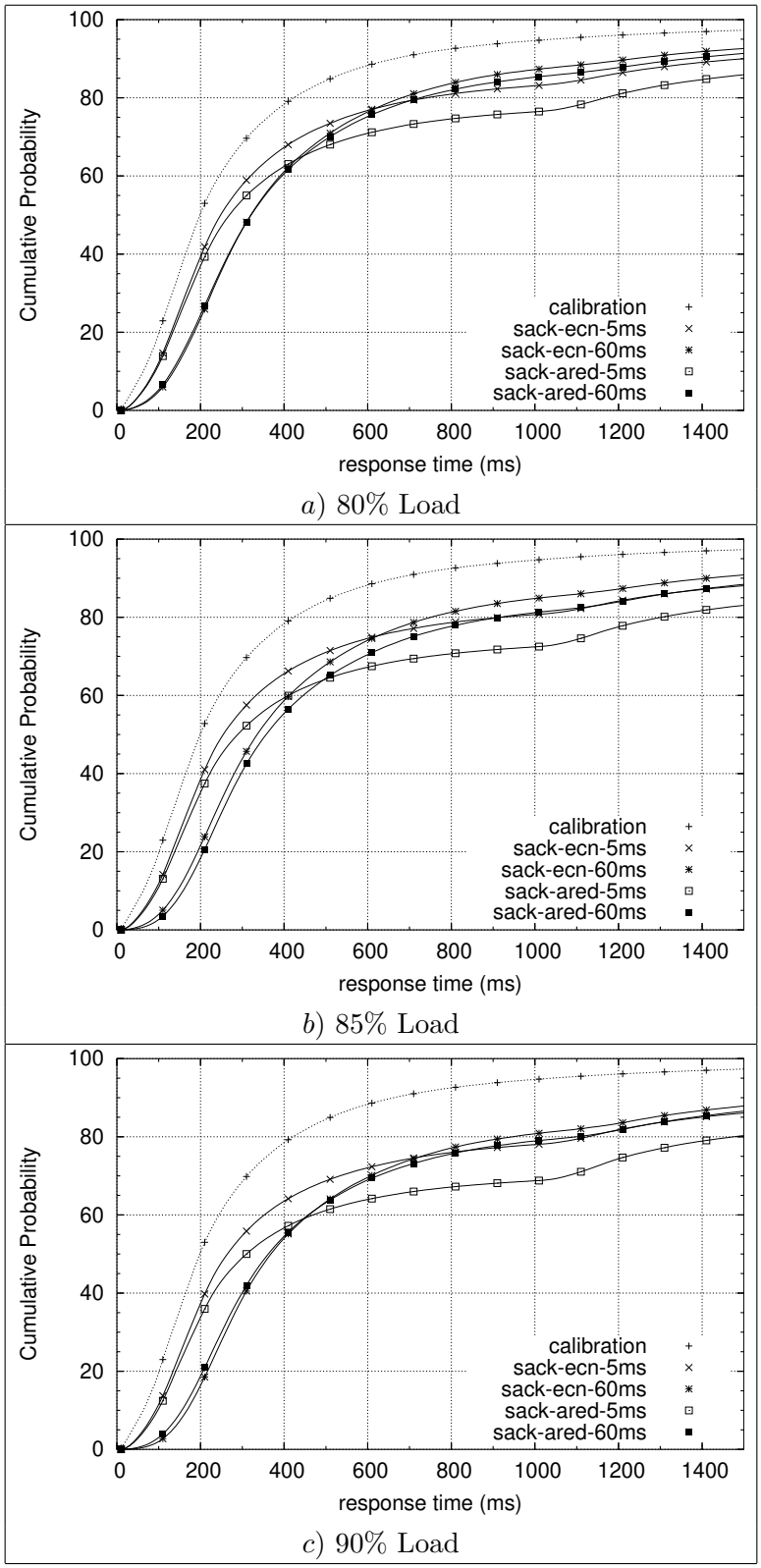


Figure B.41: Response Time CDFs, TCP SACK with Adaptive RED Queuing and with Adaptive RED Queuing and ECN Marking, Medium Congestion

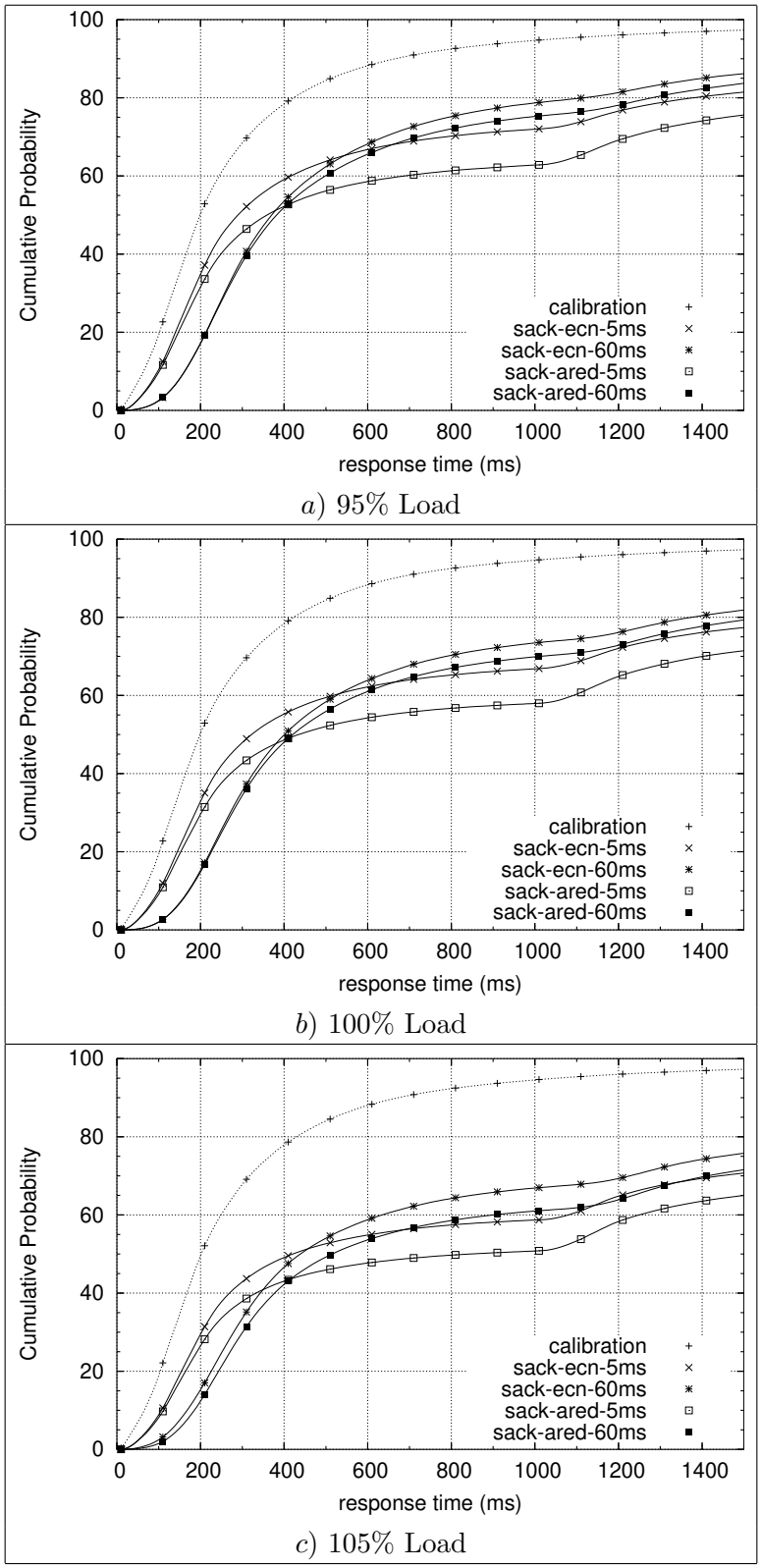


Figure B.42: Response Time CDFs, TCP SACK with Adaptive RED Queuing and with Adaptive RED Queuing and ECN Marking, Heavy Congestion

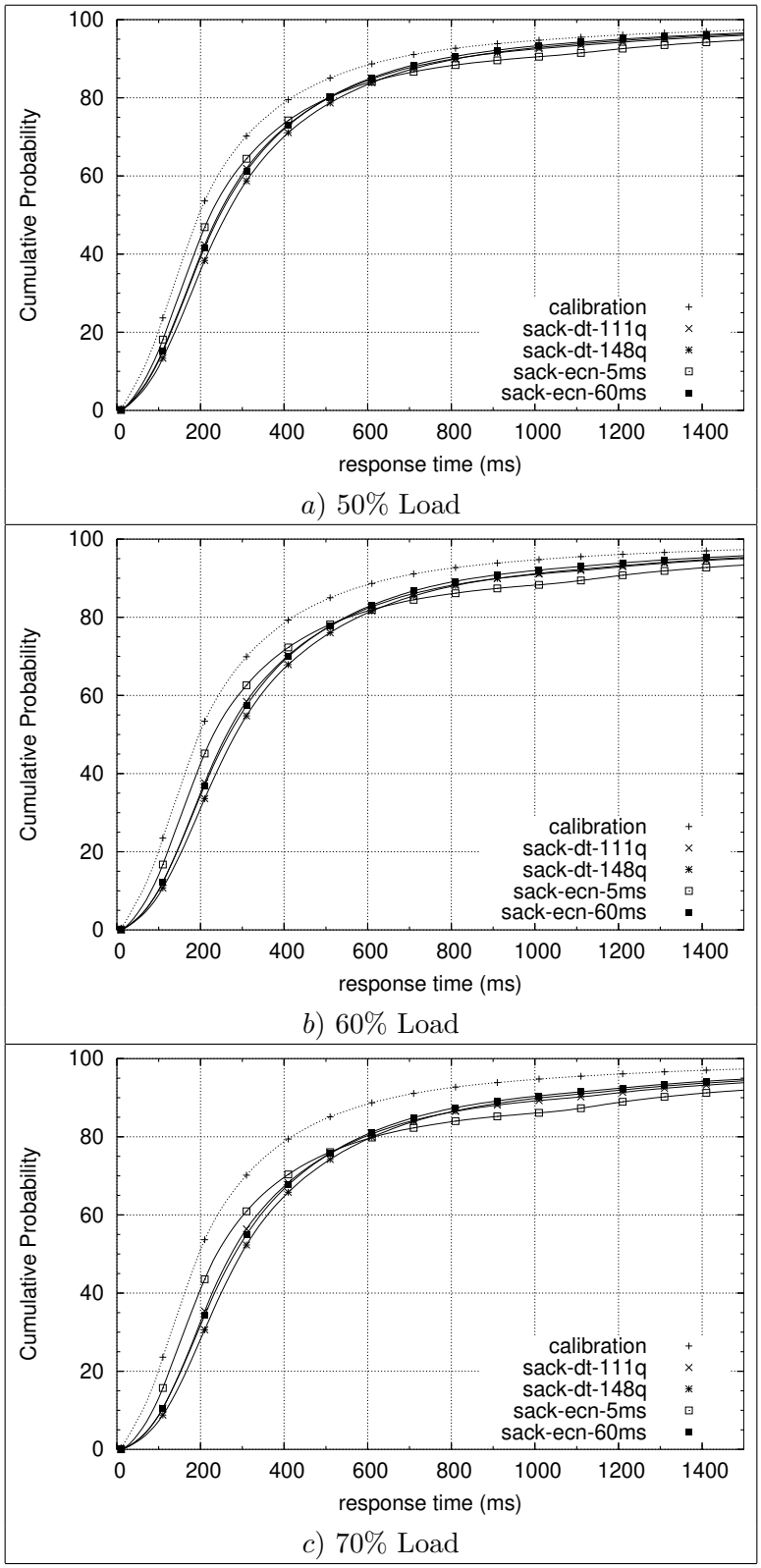


Figure B.43: Response Time CDFs, TCP SACK with Drop-Tail Queuing and with Adaptive RED Queuing and ECN Marking, Light Congestion

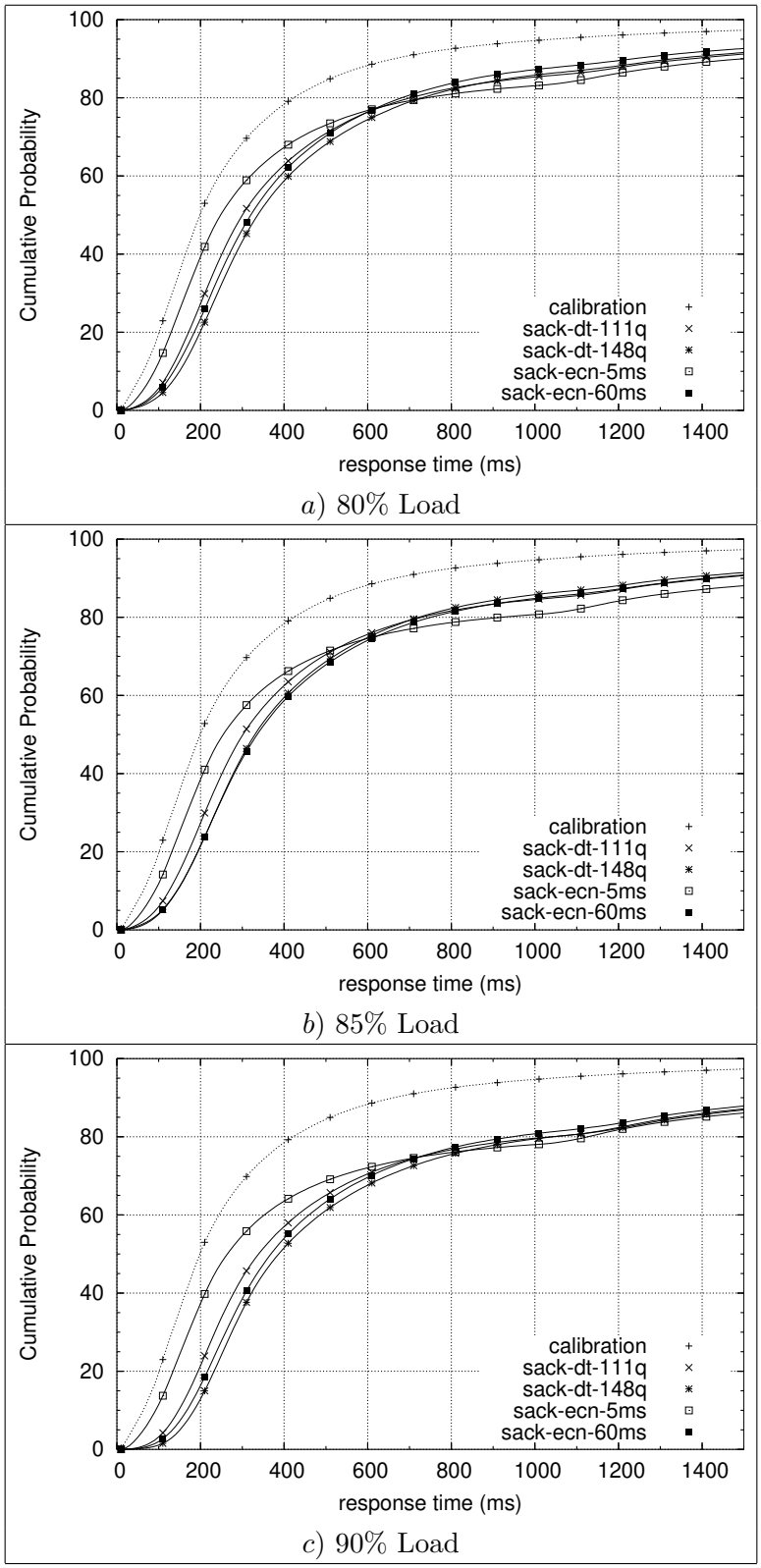


Figure B.44: Response Time CDFs, TCP SACK with Drop-Tail Queuing and with Adaptive RED Queuing and ECN Marking, Medium Congestion

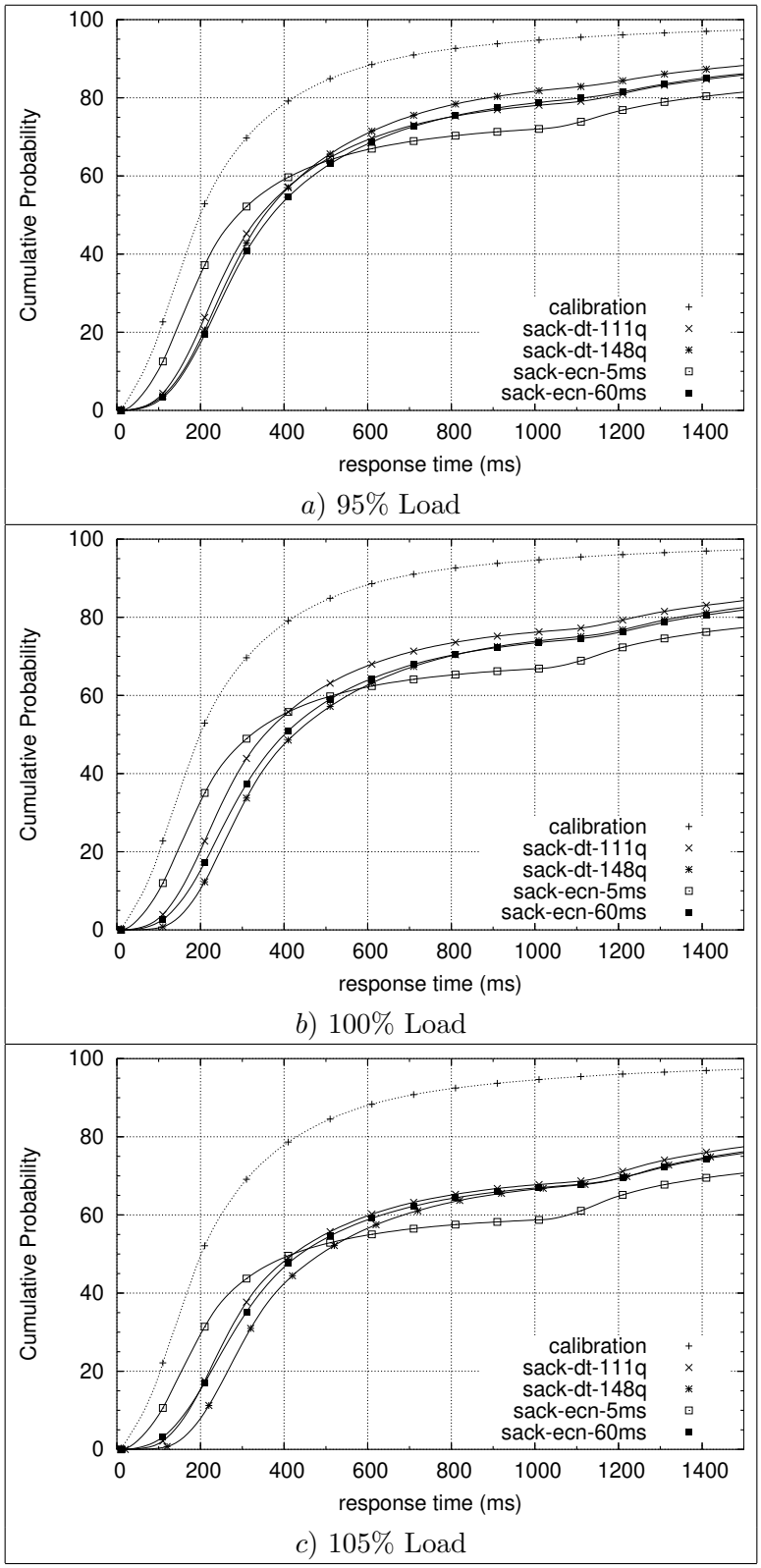


Figure B.45: Response Time CDFs, TCP SACK with Drop-Tail Queuing and with Adaptive RED Queuing and ECN Marking, Heavy Congestion

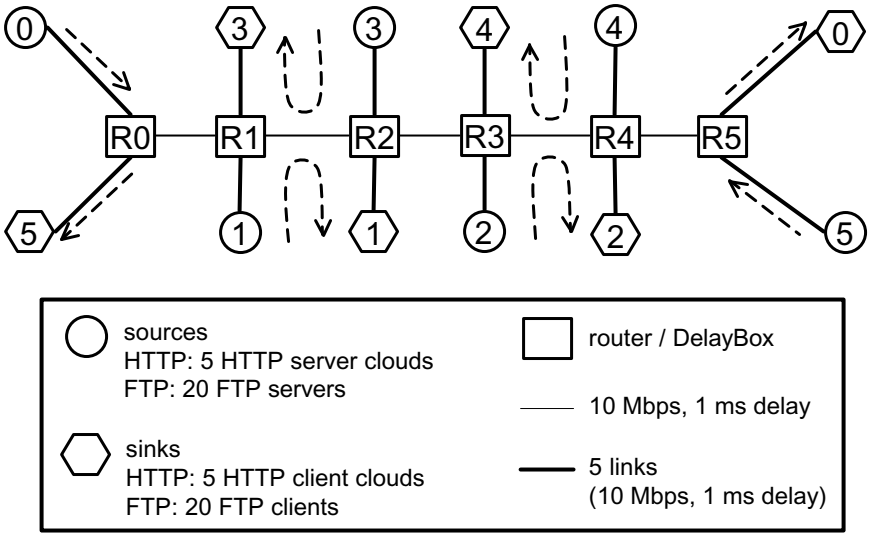


Figure B.46: Sync-TCP Evaluation Topology

Load	Average Queue Size Target (Packets)	Target Delay (ms)
80%	23	10
85%	36	15
90%	54	23
95%	73	31
100%	91	38
105%	103	44

Table B.3: Parameters for "tuned" Adaptive RED

HTTP response times. So, I also tested Adaptive RED with a maximum queue size of 2xBDP, which is the same maximum queue size as used for the Sync-TCP evaluation.

Figures B.47 and B.48 show summary statistics for the tested Adaptive RED parameter settings. Figure B.47a gives the average queue size at the bottleneck router. I included the average queue size of Sync-TCP(MixReact) for comparison. For moderate levels of congestion (80-90% load), the tuned Adaptive RED parameters produce average queue sizes that are similar to that achieved by Sync-TCP(MixReact). At higher levels of congestion (over 90% load), the 5xBDP maximum buffer size (qlen = 340) allows the queue to grow very large, which increases the average queue size. At loads of 100-105%, the average queue sizes are greater than 120 packets and, thus, are not displayed on the graph. With a maximum queue length of 2xBDP (qlen = 136), the average stays very close to the average experienced by Sync-TCP(MixReact). The tuned Adaptive RED with 2xBDP maximum queue size also performs better than tuned Adaptive RED with 5xBDP maximum queue size in terms of packet loss (Figure B.47b), median response time (Figure B.47c), link utilization (Figure B.48a), and goodput (Figure B.48c), especially at higher levels of congestion.

Figures B.49-B.52 show the HTTP response time CDFs and CCDFs for medium and heavy levels of congestion. These graphs reinforce my claim that using a smaller maximum queue size for Adaptive RED results in better performance than with an essentially infinite maximum queue. I also ran experiments with Adaptive RED with a target delay of 5 ms and a maximum queue length of 2xBDP (instead of 5xBDP, as shown in these figures). Changing the maximum queue length with a target delay as low as 5 ms did not have a large effect on performance. To maintain a target delay of 5 ms, Adaptive RED must drop packets aggressively if the average queue size reaches over $2max_{th}$ (about 30 packets). With a maximum queue length of 5xBDP, the actual queue size seldom reached higher than the 2xBDP limit of 136 packets even at 105% load, so lowering the maximum queue length did not have a large effect on performance.

B.4 Summary

I ran experiments to determine the best combination of TCP error recovery and queue management mechanisms for comparison with Sync-TCP. I found little difference between the performance of TCP Reno and TCP SACK for HTTP traffic. I found that there are tradeoffs for HTTP performance with drop-tail queuing and Adaptive RED queuing with ECN marking. As a result, I chose to look at TCP Reno with drop-tail queuing and ECN-enabled TCP SACK with Adaptive RED queuing and ECN marking for comparison with Sync-TCP.

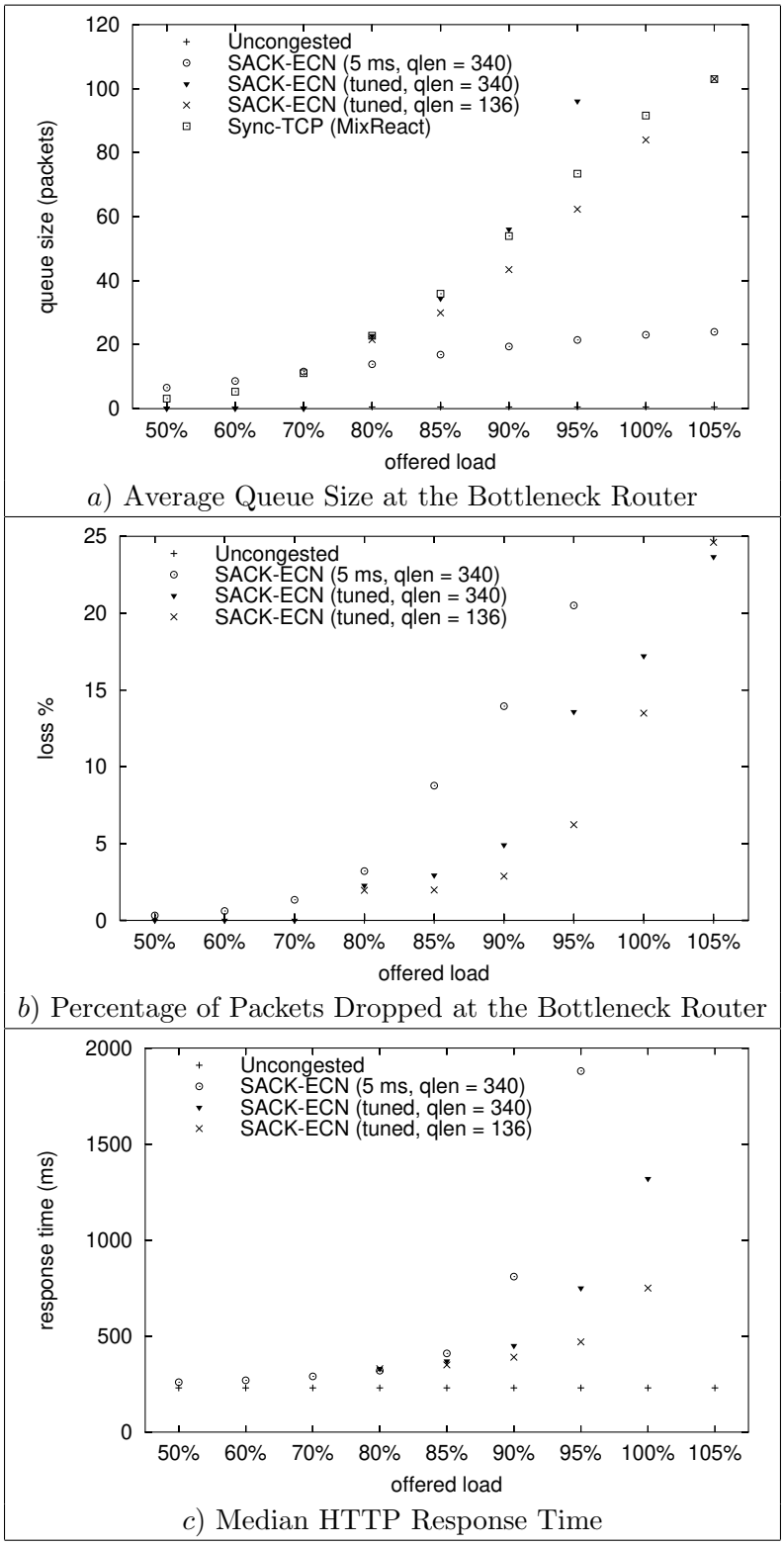


Figure B.47: Summary Statistics for TCP SACK with Adaptive RED Queuing and ECN Marking (queue size, drops, response time)

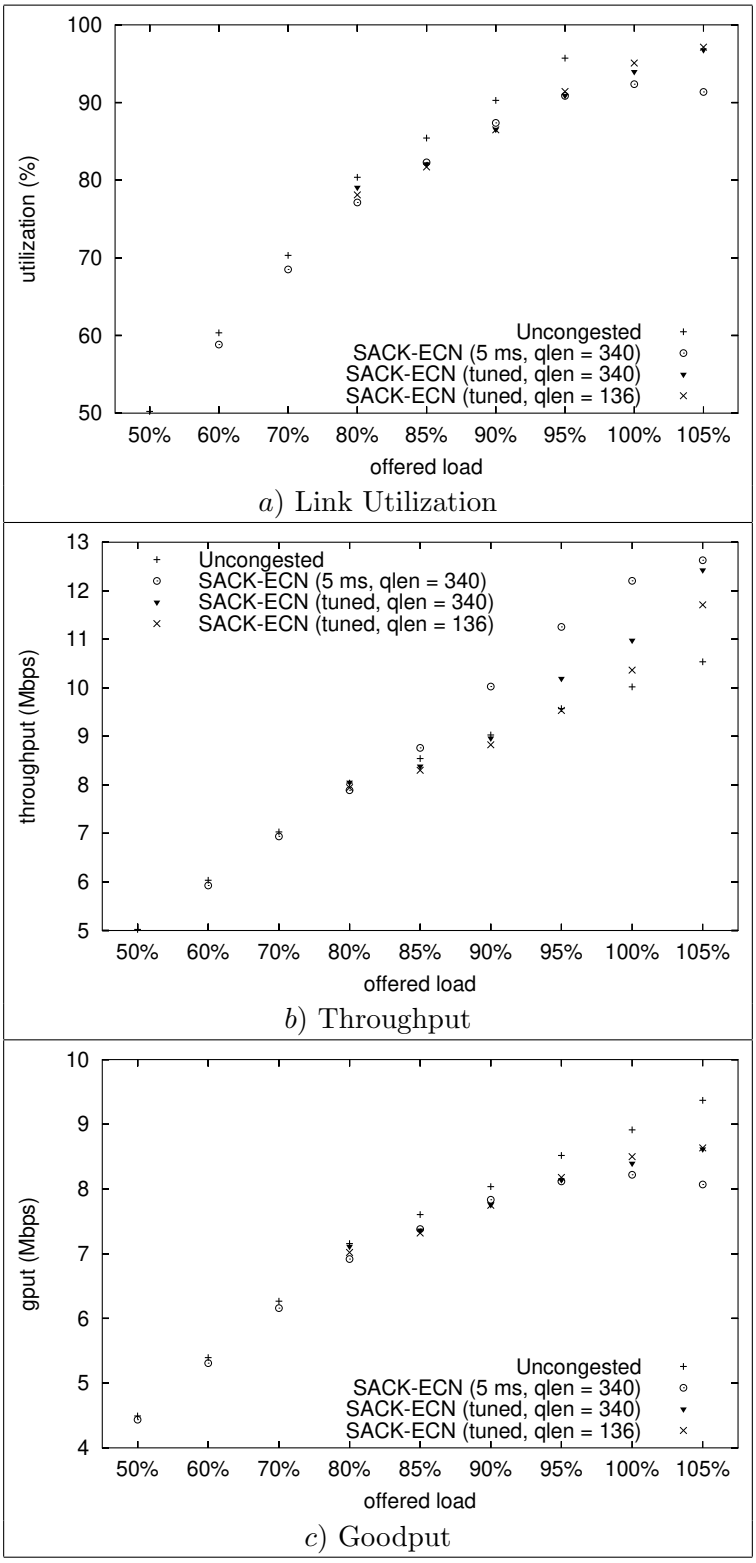


Figure B.48: Summary Statistics for TCP SACK with Adaptive RED Queuing and ECN Marking (utilization, throughput, goodput)

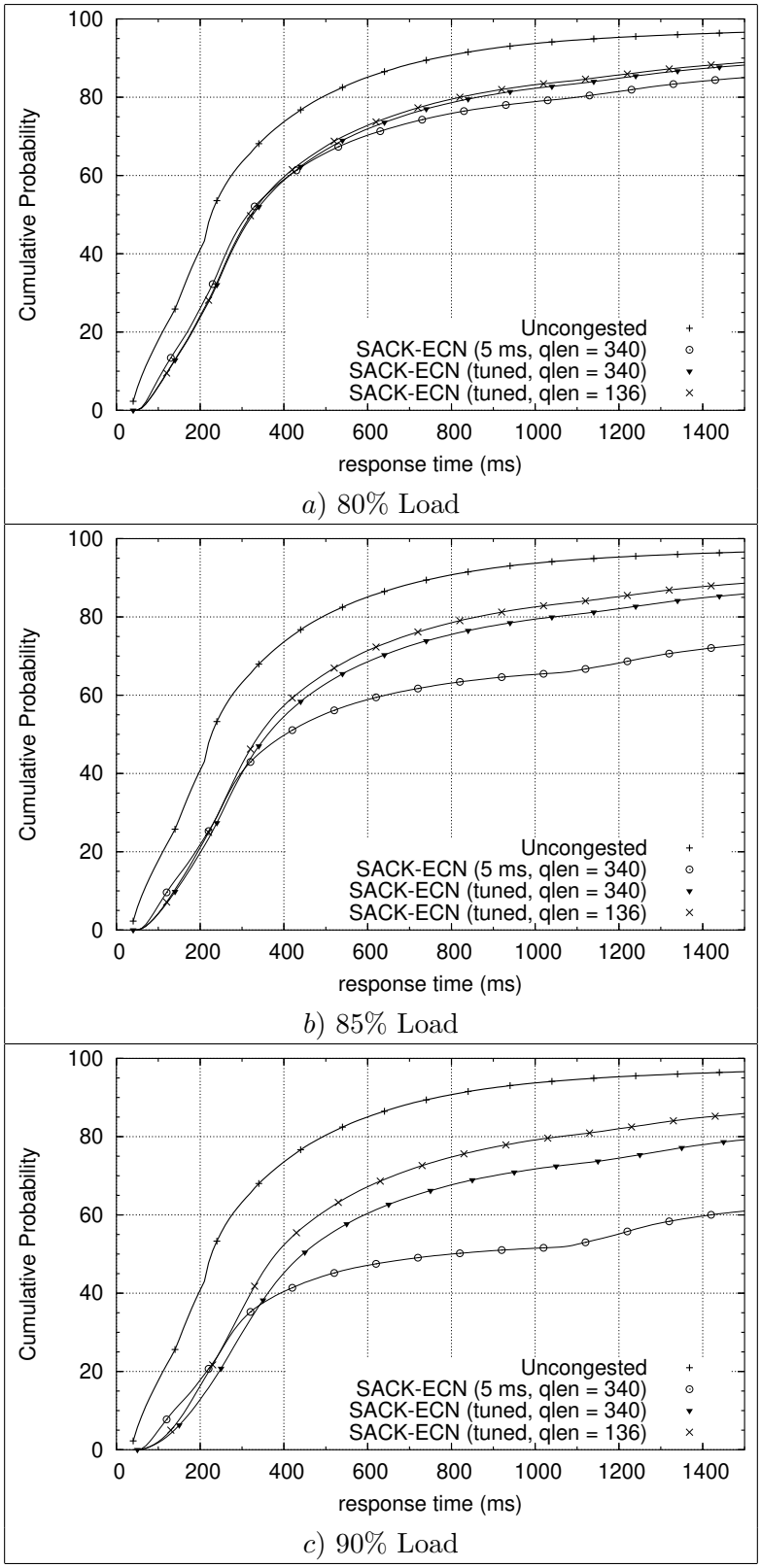


Figure B.49: Response Time CDFs, TCP SACK with Adaptive RED Queuing and ECN Marking, Medium Congestion

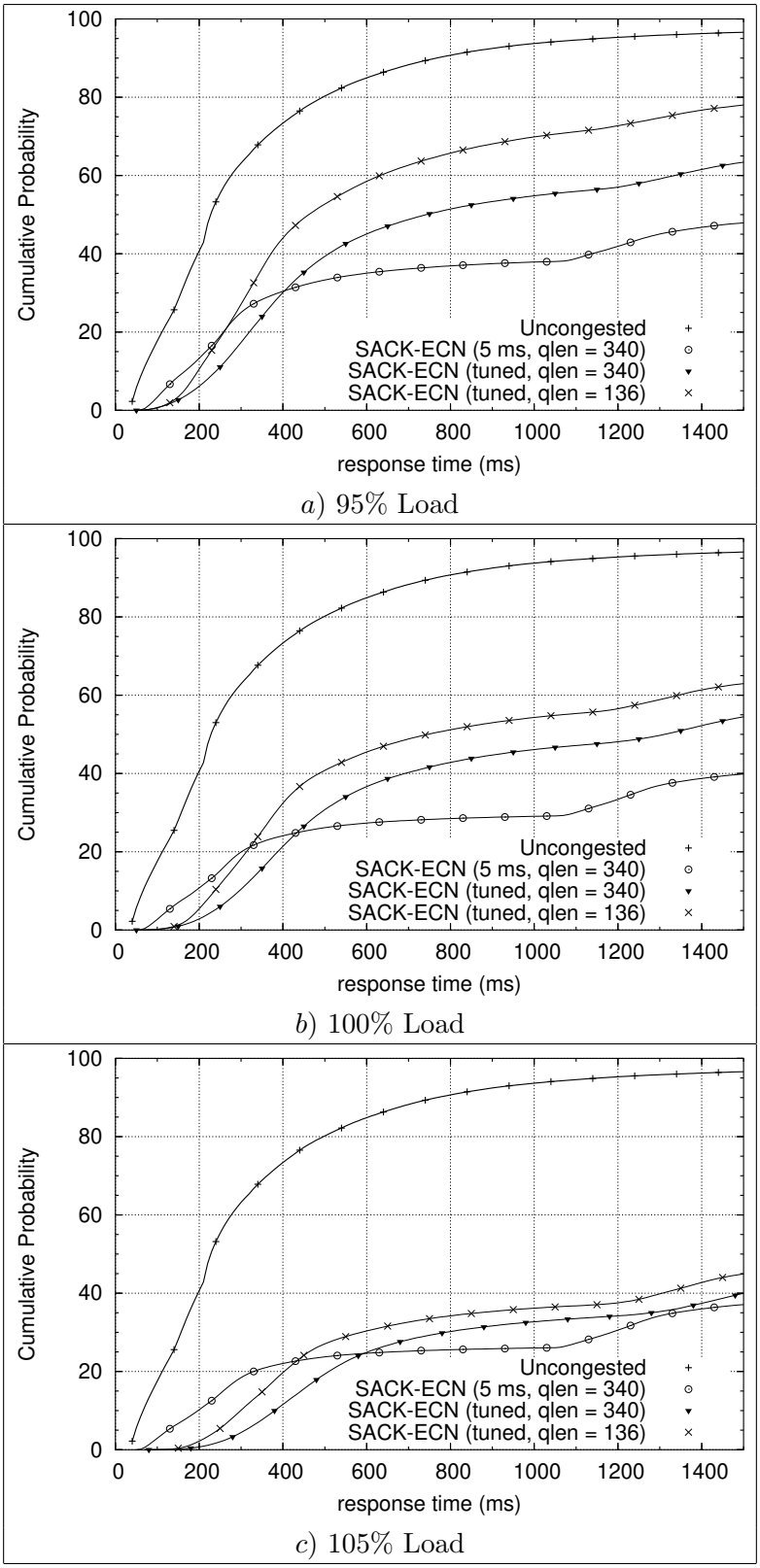


Figure B.50: Response Time CDFs, TCP SACK with Adaptive RED Queuing and ECN Marking, Heavy Congestion

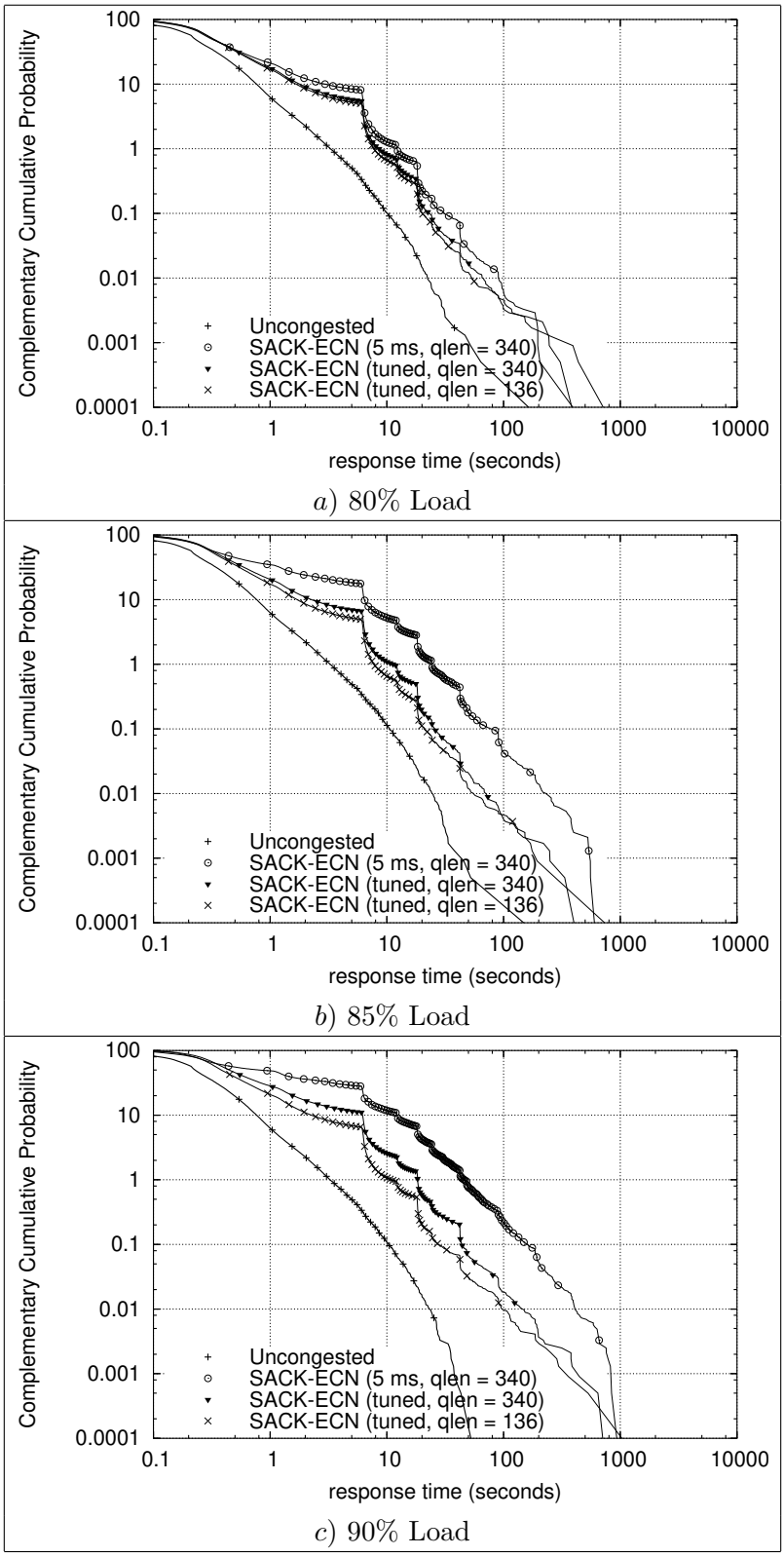


Figure B.51: Response Time CCDFs, TCP SACK with Adaptive RED Queuing and ECN Marking, Medium Congestion

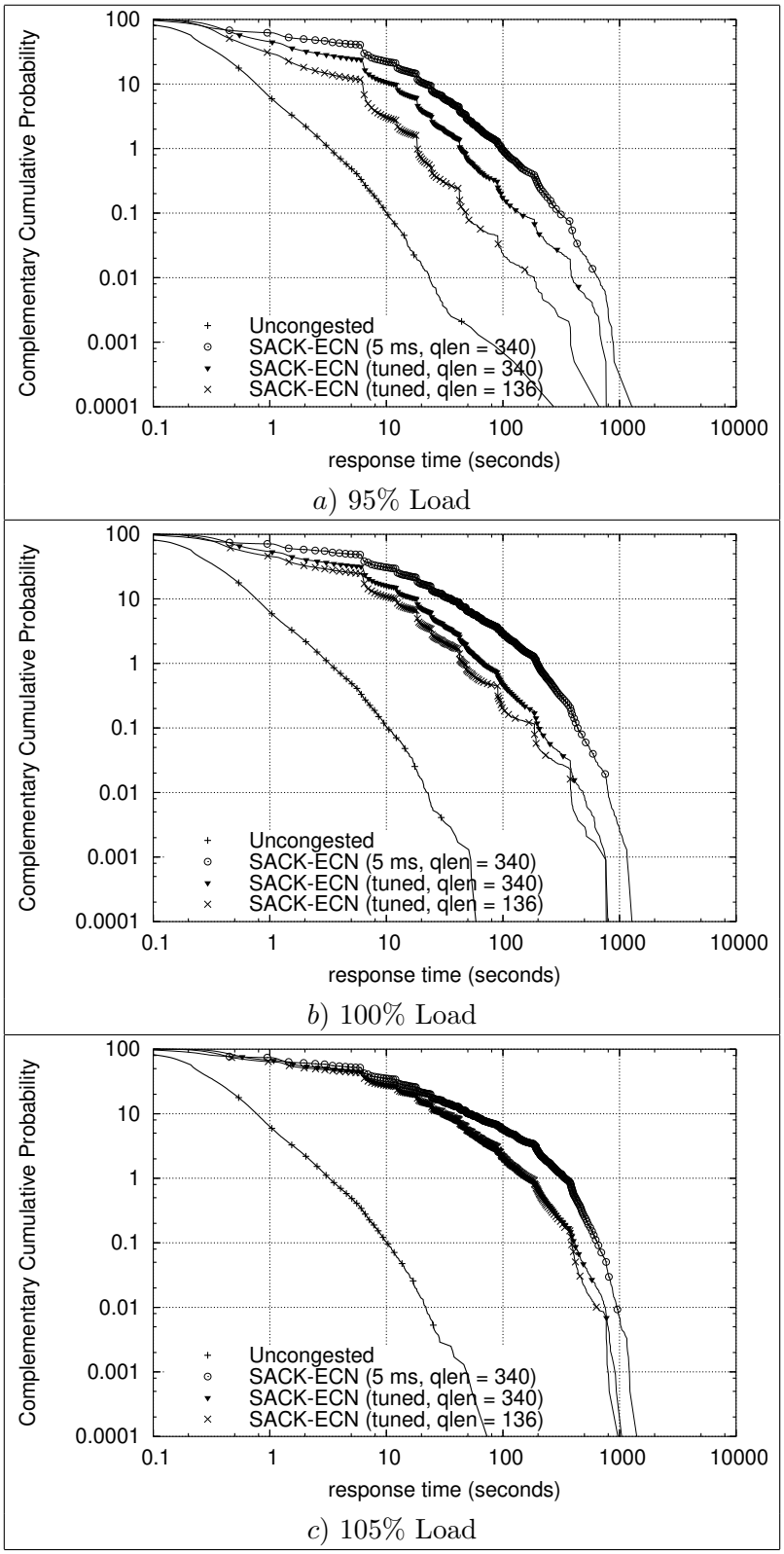


Figure B.52: Response Time CCDFs, TCP SACK with Adaptive RED Queuing and ECN Marking, Heavy Congestion

Appendix C

Experiment Summary

This appendix presents Table C.1, which lists all of the HTTP experiments that were run and where their results can be found. Several abbreviations are made to make the table more readable:

- Adaptive RED queuing is abbreviated as "ARED," and Adaptive RED queuing with ECN-marking is abbreviated as "ARED-ECN." Parameters for Adaptive RED are abbreviated as follows:
  - A target delay of 5 ms is specified as "5ms."
  - A target delay tuned to match the average queue size achieved by SyncTCP(MixReact) is specified as "tuned."
    - A maximum queue size of two times the bandwidth-delay product is specified as "2BDP."
    - A maximum queue size of five times the bandwidth-delay product is specified as "5BDP."
- "Offered Load" describes the end-to-end offered load.
- When "50-105%" is specified as the end-to-end offered load, the following offered loads were actually run: 50, 60, 70, 80, 85, 90, 95, 100, and 105% of link capacity.
- "Link Type" describes the number of congested, or bottleneck, links in the experiment. One bottleneck is denoted as "1B," multiple bottlenecks with 75% total load (small load) is denoted as "2BS," multiple bottlenecks with 90% total load (medium load) is denoted as "2BM," and multiple bottlenecks with 105% total load (large load) is denoted as "2BL."

TCP Type	Queuing Method	Offered Load (%)	Link Type	Figures	
TCP Reno	drop-tail	50-105	1B	4.12-4.14	summary stats
				4.15-4.16	summary response time CDFs
				4.17-4.19	response time CDFs
				4.20-4.22	response time CCDFs
				4.23-4.25	queue size CDFs
				4.26-4.28	response size CCDFs
				4.29-4.31	response time CDFs (responses < 25 KB)
				4.32-4.34	response time CCDFs (responses < 25 KB)
				4.35-4.37	response time CDFs (responses > 25 KB)
				4.38-4.40	response time CCDFs (responses > 25 KB)
				4.41-4.43	RTT CCDFs
				4.44-4.46	RTT CCDFs (responses < 25 KB)
				4.47-4.49	RTT CCDFs (responses > 25 KB)

continued on next page

continued from previous page

TCP Type	**Queuing Method**	**Offered Load (%)**	**Link Type**	**Figures**	
Sync-TCP (Pcwnd)	drop-tail	50-105	1B	4.12-4.14	summary stats
				4.15-4.16	summary response time CDFs
				4.17-4.19	response time CDFs
				4.20-4.22	response time CCDFs
				4.23-4.25	queue size CDFs
				4.26-4.28	response size CCDFs
				4.29-4.31	response time CDFs (responses < 25 KB)
				4.32-4.34	response time CCDFs (responses < 25 KB)
				4.35-4.37	response time CDFs (responses > 25 KB)
				4.38-4.40	response time CCDFs (responses > 25 KB)
				4.41-4.43	RTT CCDFs
				4.44-4.46	RTT CCDFs (responses < 25 KB)
				4.47-4.49	RTT CCDFs (responses > 25 KB)

continued on next page

continued from previous page

TCP Type	**Queuing Method**	**Offered Load (%)**	**Link Type**	**Figures**	
Sync-TCP (MixReact)	drop-tail	50-105	1B	4.12-4.14	summary stats
				4.15-4.16	summary response time CDFs
				4.17-4.19	response time CDFs
				4.20-4.22	response time CCDFs
				4.23-4.25	queue size CDFs
				4.26-4.28	response size CCDFs
				4.29-4.31	response time CDFs (responses < 25 KB)
				4.32-4.34	response time CCDFs (responses < 25 KB)
				4.35-4.37	response time CDFs (responses > 25 KB)
				4.38-4.40	response time CCDFs (responses > 25 KB)
				4.41-4.43	RTT CCDFs
				4.44-4.46	RTT CCDFs (responses < 25 KB)
				4.47-4.49	RTT CCDFs (responses > 25 KB)
TCP SACK	ARED-ECN (5ms)	50-105	1B	4.12-4.14 (B.47-B.48)	summary stats
				4.15-4.16	summary response time CDFs

continued on next page

continued from previous page

TCP Type	**Queuing Method**	**Offered Load (%)**	**Link Type**	**Figures**	
TCP SACK	ARED-ECN (5ms)	50-70	1B	4.17	response time CDFs
				4.20	response time CCDFs
				4.23	queue size CDFs
				4.26	response size CCDFs
				4.29	response time CDFs (responses < 25 KB)
				4.32	response time CCDFs (responses < 25 KB)
				4.35	response time CDFs (responses > 25 KB)
				4.38	response time CCDFs (responses > 25 KB)
				4.41	RTT CCDFs
				4.44	RTT CCDFs (responses < 25 KB)
				4.47	RTT CCDFs (responses > 25 KB)
TCP SACK	ARED-ECN (5ms)	80-105	1B	B.49-B.50	response time CDFs
				B.51-B.52	response time CCDFs

continued on next page

continued from previous page

TCP Type	Queuing Method	Offered Load (%)	Link Type	Figures	
TCP SACK	ARED-ECN (tuned, 2BDP)	80-105	1B	4.12-4.14 (B.47-B.48)	summary stats
				4.15-4.16	summary response time CDFs
				4.18-4.19	response time CDFs
				4.21-4.22	response time CCDFs
				4.24-4.25	queue size CDFs
				4.27-4.28	response size CCDFs
				4.30-4.31	response time CDFs (responses < 25 KB)
				4.33-4.34	response time CCDFs (responses < 25 KB)
				4.36-4.37	response time CDFs (responses > 25 KB)
				4.39-4.40	response time CCDFs (responses > 25 KB)
				4.42-4.43	RTT CCDFs
				4.45-4.46	RTT CCDFs (responses < 25 KB)
				4.48-4.49	RTT CCDFs (responses > 25 KB)
TCP SACK	ARED-ECN (tuned, 5BDP)	80-105	1B	B.47-B.48	summary stats
				B.49-B.50	response time CDFs
				B.51-B.52	response time CCDFs
TCP Reno	drop-tail	50-70	2BS	4.56a	summary response time CDFs
				4.57	response time CDFs
				4.58	response time CCDFs

continued on next page

continued from previous page

TCP Type	**Queuing Method**	**Offered Load (%)**	**Link Type**	**Figures**	
Sync-TCP (MixReact)	drop-tail	50-70	2BS	4.56b	summary response time CDFs
				4.57	response time CDFs
				4.58	response time CCDFs
TCP Reno	drop-tail	50-85	2BM	4.59a	summary response time CDFs
				4.60-4.61	response time CDFs
				4.62-4.63	response time CCDFs
Sync-TCP (MixReact)	drop-tail	50-85	2BM	4.59b	summary response time CDFs
				4.60-4.61	response time CDFs
				4.62-4.63	response time CCDFs
TCP Reno	drop-tail	50-100	2BL	4.64a	summary response time CDFs
				4.65-4.67	response time CCDFs
Sync-TCP (MixReact)	drop-tail	50-100	2BL	4.64b	summary response time CDFs
				4.65-4.67	response time CCDFs

Table C.1: List of HTTP Experiments

BIBLIOGRAPHY

[ADLY95] Jong Suk Ahn, Peter B. Danzig, Zhen Liu, and Limin Yan. Evaluation of TCP Vegas: Emulation and Experiment. In *Proceedings of ACM SIGCOMM*, 1995.

[AF99] Mark Allman and Aaron Falk. On the effective evaluation of TCP. *ACM Computer Communication Review*, 29(5):59–70, October 1999.

[AMP01] Ian F. Akyildiz, Giacomo Morabito, and Sergio Palazzo. TCP-Peach: A new congestion control scheme for satellite IP networks. *IEEE/ACM Transactions on Networking*, 9(3):307–321, June 2001.

[AP99] Mark Allman and Vern Paxson. On estimating end-to-end network path properties. In *Proceedings of ACM SIGCOMM*, pages 263–274, September 1999.

[APS99] Mark Allman, Vern Paxson, and W. Richard Stevens. TCP congestion control. RFC 2581, April 1999.

[BB01] Deepak Bansal and Hari Balakrishnan. Binomial congestion control algorithms. In *Proceedings of IEEE INFOCOM*, April 2001.

[BBFS01] Deepak Bansal, Hari Balakrishnan, Sally Floyd, and Scott Shenker. Dynamic behavior of slowly-responsive congestion control algorithms. In *Proceedings of ACM SIGCOMM*, August 2001.

[BC98] Paul Barford and Mark E. Crovella. Generating representative web workloads for network and server performance evaluation. In *Proceedings of ACM SIGMETRICS*, pages 151–160, 1998.

[BEF+00] Lee Breslau, Deborah Estrin, Kevin Fall, Sally Floyd, John Heidemann, Ahmed Helmy, Polly Huang, Steven McCanne, Kannan Varadhan, Ya Xu, and Haobo Yu. Advances in Network Simulation. *IEEE Computer*, 33(5):59–67, May 2000.

[BMB00] Thomas Bonald, Martin May, and Jean-Chrysostome Bolot. Analytic evaluation of RED performance. In *Proceedings of IEEE INFOCOM*, pages 1415–1424, 2000.

[Bol93] Jean-Chrysostome Bolot. End-to-end packet delay and loss behavior in the Internet. In *Proceedings of ACM SIGCOMM*, 1993.

[BOP94] Lawrence Brakmo, Sean O'Malley, and Larry Peterson. TCP Vegas: New techniques for congestion detection and avoidance. In *Proceedings of ACM SIGCOMM*, pages 24–35, October 1994.

[BV98] Saad Biaz and Nitin H. Vaidya. Distinguishing congestion losses from wireless transmission losses: A negative result. In *Proceedings of IC3N*, 1998.

[CCLS01a] Jin Cao, William S. Cleveland, Dong Lin, and Don X. Sun. On the nonstationarity of Internet traffic. In *Proceedings of ACM SIGMETRICS*, 2001.

[CCLS01b] Jin Cao, William S. Cleveland, Dong Lin, and Don X. Sun. PackMime: an Internet traffic generator. In *National Institute of Statistical Sciences Affliates Workshop on Modeling and Analysis of Network Data*, March 2001.

[CDS74] Vint Cerf, Yogen Dalal, and Carl Sunshine. Specification of Internet Transmission Control Program. RFC 675, December 1974.

[CJ89] Dah-Ming Chiu and Raj Jain. Analysis of the increase and decrease algorithms for congestion avoidance in computer networks. *Computer Networks and ISDN Systems*, 17:1–14, 1989.

[CJOS01] Mikkel Christiansen, Kevin Jeffay, David Ott, and F. Donelson Smith. Tuning RED for web traffic. *IEEE/ACM Transactions on Networking*, 9(3):249–264, June 2001.

[CK74] Vint Cerf and Bob Kahn. A protocol for packet network intercommunication. *IEEE Transactions on Communications*, 22(5), May 1974.

[CL97] Mark Crovella and Lester Lipsky. Long-lasting transient conditions in simulations with heavy-tailed workloads. In *Proceedings of the 1997 Winter Simulation Conference*, pages 1005–1012, 1997.

[Cla82] David D. Clark. Window and acknowledgement strategy in TCP. RFC 813, July 1982.

[CLS00] William S. Cleveland, Dong Lin, and Don X. Sun. IP packet generation: Statistical models for TCP start times based on connection-rate superposition. In *Proceedings of ACM SIGMETRICS*, 2000.

[DP90] Peter H. Dana and Bruce M. Penrod. The role of GPS in precise time and frequency dissemination. *GPS World*, July/August 1990.

[FF96] Kevin Fall and Sally Floyd. Simulation-based comparisons of Tahoe, Reno, and SACK TCP. *ACM Computer Communication Review*, July 1996.

[FGS01] Sally Floyd, Ramakrishna Gummadi, and Scott Shenker. Adaptive RED: An algorithm for increasing the robustness of RED's active queue management. under submission, August 2001.

[FH99] Sally Floyd and Tom Henderson. The NewReno modification to TCP's fast recovery algorithm. RFC 2582, Experimental, April 1999.

[FJ93] Sally Floyd and Van Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, August 1993.

[Flo91a] Sally Floyd. Connections with multiple congested gateways in packet-switched networks, part 1: One-way traffic. *ACM Computer Communication Review*, 21(5):30–47, October 1991.

[Flo91b] Sally Floyd. Connections with multiple congested gateways in packet-switched networks part 2: Two-way traffic. `ftp://ftp.ee.lbl.gov/papers/gates2.ps.Z`, 1991.

[Flo94] Sally Floyd. TCP and explicit congestion notification. *ACM Computer Communication Review*, 24(5):10–23, October 1994.

[Flo97] Sally Floyd. RED: Discussions of setting parameters. `http://www.icir.org/floyd/REDparameters.txt`, November 1997.

[Flo00a] Sally Floyd. Congestion control principles. RFC 2914, September 2000.

[Flo00b] Sally Floyd. Recommendation on using the gentle variant of RED. `http://www.icir.org/floyd/red/gentle.html`, March 2000.

[FP92] Sally Floyd and Vern Paxson. On traffic phase effects in packet-switched gateways. *Internetworking: Research and Experience*, 3(3):115–156, Sepember 1992.

[HBG00] Urs Hengartner, Jurg Bolliger, and Thomas Gross. TCP Vegas revisited. In *Proceedings of IEEE INFOCOM*, 2000.

[Hoe95] Janey Hoe. Startup dynamics of TCP's congestion control and avoidance schemes. Master's thesis, MIT, `http://ana-www.lcs.mit.edu/anaweb/ps-papers/hoe-thesis.ps`, 1995.

[Hoe96] Janey Hoe. Improving the start-up behavior of a congestion control scheme for TCP. In *Proceedings of ACM SIGCOMM*, 1996.

[HSMK98] Thomas H. Henderson, Emile Sahouria, Steven McCanne, and Randy H. Katz. On improving the fairness of TCP congestion avoidance. In *Proceedings of the IEEE Globecomm Conference*, Sydney, 1998.

[Hua01] Polly Huang. Simulation using self-similar traffic. Email to end2end-interest mailing list, April 2001.

[Jac94] Van Jacobson. Problems with Arizona's Vegas. Note to end2end-interest mailing list, March 1994.

[Jai89] Raj Jain. A delay-based approach for congestion avoidance in interconnected heterogeneous computer networks. *ACM Computer Communication Review*, pages 56–71, 1989.

[JBB92] Van Jacobson, Robert Braden, and D. Borman. TCP extensions for high performance. RFC 1323, May 1992.

[JD02] Manish Jain and Constantinos Dovrolis. End-to-end available bandwidth: Measurement methodology, dynamics, and relation with TCP throughput. In *Proceedings of ACM SIGCOMM*, August 2002.

[JK88] Van Jacobson and Michael Karels. Congestion avoidance and control. In *Proceedings of ACM SIGCOMM*, pages 314–329, 1988.

[Kes91] Srinivsan Keshav. A control-theoretic approach to flow control. In *Proceedings of ACM SIGCOMM*, September 1991.

[LL99] David Lapsley and Steven Low. Random early marking: An optimization approach to Internet congestion control. In *Proceedings of IEEE International Conference on Networks*, September 1999.

[LPW01] Steven Low, Larry Peterson, and Limin Wang. Understanding TCP Vegas: A duality model. In *Proceedings of ACM SIGMETRICS*, June 2001.

[Mah97] Bruce A. Mah. An empirical model of HTTP network traffic. In *Proceedings of IEEE INFOCOM*, pages 592–600, 1997.

[MCG+01] Saverio Mascolo, Claudio Casetti, Mario Gerla, M. Y. Sanadidi, and Ren Wang. TCP Westwood: Bandwidth estimation for enhanced transport over wireless links. In *Proceedings of ACM Mobicom*, 2001.

[Mil90] David L. Mills. On the accuracy and stability of clocks synchronized by the network time protocol in the Internet system. *ACM Computer Communication Review*, 20(1):65–75, January 1990.

[Mil92] David L. Mills. Network time protocol (version 3): Specification, implementation and analysis. RFC 1305, March 1992.

[MLAW99] J. Mo, R.J. La, V. Anantharam, and J. Walrand. Analysis and comparison of TCP Reno and Vegas. In *Proceedings of IEEE INFOCOM*, pages 1556–1563, March 1999.

[MM96] Matthew Mathis and Jamshid Mahdavi. Forward acknowledgement: Refining TCP congestion control. In *Proceedings of ACM SIGCOMM*, August 1996.

[MMFR96] Matthew Mathis, Jamshid Mahdivi, Sally Floyd, and Allyn Romanow. TCP selective acknowledgement options. RFC 2018, October 1996.

[MNR00] Jim Martin, Arne Nilsson, and Injong Rhee. The incremental deployabilty of RTT-based congestion avoidance for high speed TCP Internet connections. In *Proceedings of ACM SIGMETRICS*, pages 134–144, June 2000.

[Nag84] John Nagle. Congestion control in IP/TCP internetworks. RFC 896, January 1984.

[Par00] Kihong Park. *Self-Similar Network Traffic and Performance Evaluation*, chapter 1: Self-similar network traffic: An overview. Wiley-Interscience, 2000.

[Pax97] Vern Paxson. End-to-end routing behavior in the Internet. *IEEE/ACM Transactions on Networking*, 5(5):601–615, October 1997.

[Pax98] Vern Paxson. On calibrating measurements of packet transit times. In *Proceedings of ACM SIGMETRICS*, 1998.

[Pax99] Vern Paxson. End-to-end Internet packet dynamics. *IEEE/ACM Transactions on Networking*, 7(3):277–292, June 1999.

[PF97] Vern Paxson and Sally Floyd. Why we don't know how to simulate the Internet. In *Proceedings of the 1997 Winter Simulation Conference*, pages 1037–1044, 1997.

[PF01] Jitendra Padhye and Sally Floyd. On inferring TCP behavior. In *Proceedings of ACM SIGCOMM*, pages 287–298, September 2001.

[PGLA99] Christina Parsa and J. J. Garcia-Luna-Aceves. Improving TCP congestion control over internets with heterogeneous transmission media. In *Proceedings of the IEEE International Conference on Network Protocols (ICNP)*, 1999.

[PKC96] Kihong Park, Gitae Kim, and Mark Crovella. On the relationship between file sizes, transport protocols, and self-similar network traffic. In *Proceedings of the IEEE International Conference on Network Protocols (ICNP)*, 1996.

[RF99] K.K. Ramakrishnan and Sally Floyd. A proposal to add explicit congestion notification (ECN) to IP. RFC 2481, Experimental, January 1999.

[Riz97] Luigi Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *ACM Computer Communication Review*, January 1997.

[RJ90] K. K. Ramakrishnan and Raj Jain. A binary feedback scheme for congestion avoidance in computer networks. *ACM Transactions on Computer Systems*, 8(2):158–181, May 1990.

[SA00] Jamal Hadi Salim and Uvaiz Ahmed. Performance evaluation of explicit congestion notification (ECN) in IP networks. RFC 2884, July 2000.

[SF98] Nihal K.G. Samaraweera and Godred Fairhurst. Reinforcement of TCP error recovery for wireless communication. *ACM Computer Communication Review*, 28(2), April 1998.

[Ste94] W. Richard Stevens. *TCP/IP Illustrated, Volume 1: The Protocols*. Addison-Wesley, 1994.

[Taq] Murad Taqqu. Aggregated variance. `http://math.bu.edu/people/murad/methods/var/`.

[WP98] Walter Willinger and Vern Paxson. Where mathematics meets the Internet. *Notices of the American Mathematical Society*, pages 961–970, September 1998.

[YL00] Yang Richard Yang and Simon S. Lam. General AIMD congestion control. In *Proceedings of the IEEE International Conference on Network Protocols (ICNP)*, 2000.

[ZQ00] Yin Zhang and Lili Qiu. Understanding the end-to-end performance impact of RED in a heterogeneous environment. Technical Report 2000-1802, Cornell CS, July 2000.

[ZSC91] Lixia Zhang, Scott Shenker, and David Clark. Observations on the dynamics of a congestion control algorithm: The effects of two-way traffic. In *Proceedings of ACM SIGCOMM*, 1991.