A Performance Analysis of TCP and STP Implementations and Proposals for New QoS Classes for TCP/IP

by

David Holl, Jr.

A Thesis
Submitted to the Faculty
of the
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements for the
Degree of Master of Science
in
Electrical and Computer Engineering
by

May 2003

APPROVED:

Professor David Cyganski, Major Advisor

Professor Brian King

Professor William Michalson

Abstract

With a new United States Army initiative to exploit commercially developed information technology, there is a heightened interest in using Internet protocols over the military's geosynchronous satellite links. TCP is the dominant Internet protocol used for reliable data exchange, but its own design limits performance when used over long delay network links such as satellites. Initially this research set out to compare TCP with another proposed protocol, the Satellite Transport Protocol (STP). However through a series of tests, we found that STP does not fulfill its claims of increased throughput over TCP and uncovered a flaw in STP's founding research. In addition, this thesis proposes and demonstrates novel performance enhancing techniques that significantly improve transport protocol throughput.

Acknowledgements

First, I would like to thank my advisor, Prof. David Cyganski, for sharing his insights in physics, electrical engineering, and the art behind successful presentations. Next, I would like to thank my thesis committee, Prof. Brian King and Prof. William Michalson, for their time and helpful suggestions, and I would like to thank Raytheon's Network Centric Systems Division for sponsoring the research project which provided ample thesis material and funded my pursuit of the Degree of Masters of Science.

I would also like to thank my lab-mates & coworkers, Chris Boumenot, Nick Hatch, Pavan Reddy, and Mike Thurston, for the fun and professional atmosphere in and around our lab and department. Finally and most importantly, I thank Nita Madhav for her patience and support through my countless weekends of programming, simulations and ramblings about TCP. I wish her the best of advisors as she starts working on her graduate degree.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this research, we set out to optimize the flow of stream-based network traffic while providing multiple Quality of Service (QoS) levels and Dynamic Bandwidth Allocation over geosynchronous satellite links. We compared different router queuing disciplines and several implementations of stream-oriented data transport protocols. We also investigated the possibility of splitting connections at satellite uplinks[8], where splitting involves transparently terminating best-effort connections at the uplink routers to allow for the use of an intermediary optimized protocol.

This thesis will review two stream oriented protocols and four router queuing disciplines, and present our evaluation of which combination of protocol and discipline meets QoS requirements for satellite mediated Internet connectivity for military services within the context of the WIN-T initiative described in chapter 2. In the course of this evaluation, we developed new techniques to maximize link utilization and attain QoS levels analogous to those in Asynchronous Transfer Mode (ATM) networks which were previously unavailable in TCP/IP networks.

We have organized this thesis as follows: Chapter 2 presents the WIN-T military initiative, the Transmission Control Protocol (TCP), and the Satellite Transport Protocol (STP). We discuss our simulation environment in chapter 3. In chapter 4, we compare TCP with STP while seeking to optimize throughput while maintaining required QoS for different traffic classes. Chapter 5 continues our protocol comparison as we measure the effects of

bit errors on application throughput. In chapter 6, we measure how the protocols respond to changes in available bandwidth and explore a new strategy for fair bandwidth sharing in TCP. Then in chapter 7, we present three new throughput optimization strategies based on our previous chapters of observations. Lastly, we summarize our findings and provide conclusions in chapter 8.

Also, this thesis uses many abbreviations and defines many of them only once. Table 1.1 lists these abbreviations for the reader's convenience.

ABR	Available Bit Rate
ACK	TCP Acknowledgment Packet
ATM	Asynchronous Transfer Mode
BE	Best-Effort
BER	Bit Error Rate
CBQ	Class Based Queuing
CBR	Constant Bit Rate
FACK	Forward Acknowledgments
MSS	Maximum Segment Size
MTU	Maximum Transmission Unit
ns	Network Simulator
OS	Operating System
OTcl	MIT Object Tcl - an object-oriented extension to Tcl/TK
PEP	Performance Enhancing Proxy
QoS	Quality of Service
RED	Random Early Detection
RTO	Round-Trip Time Out
RTT	Round-Trip Time
SACK	Selective Acknowledgments
STP	Satellite Transport Protocol
VBR	Variable Bit Rate
VoIP	Voice over IP

Table 1.1: Common abbreviations used throughout this thesis.

Chapter 2

Background

In an effort to equip their forces with the latest information technology, the United States Army proposed a new concept of operation named the Warfighter Information Network-Tactical (WIN-T). The military wishes to integrate voice, data, and video services into a single fault-tolerant network while keeping costs low. For integrating voice, data, and video services, WIN-T requires guaranteed QoS levels while making the most efficient use of available bandwidth. To reduce implementation and management costs, WIN-T stresses compatibility with commercial off-the-shelf (COTS) equipment and reusing existing military information infrastructure where possible. In WIN-T implementation, COTS implies compatibility with the protocol suite used on the Internet, TCP/IP, and infrastructure reuse means using TCP/IP over the military's existing links, especially their geosynchronous satellites.

A typical WIN-T traffic profile will consist of voice and video data, which by nature require high QoS such as that provided by a constant bit rate (CBR) service with guaranteed delivery. In addition to this, WIN-T will also include best-effort (BE) service traffic of varying priority levels such as email versus database queries. For compatibility with the TCP/IP protocol suite, we examine transporting CBR traffic over the User Datagram Protocol (UDP) and BE traffic over the Transmission Control Protocol (TCP) as well as the Satellite Transport Protocol (STP).

The comingling of CBR and BE traffic types in WIN-T mandate some sort of reserva-

tion system with special attention to protocols and queuing disciplines. The remainder of this chapter will briefly summarize key operational characteristics, defining references, and common applications of the UDP, TCP, and STP protocols, while the queuing disciplines will be discussed as they arise in chapter 4.

2.1 User Datagram Protocol (UDP)

CBR traffic commonly encompasses voice, video, and multimedia traffic, and in TCP/IP networks, CBR data is commonly sent using the User Datagram Protocol (UDP)[14] which provides a low-overhead, connectionless, unreliable data transport mechanism for applications. In a CBR application, the sending computer encapsulates bytes into fixed sized UDP packets and transmits each packet over the network. At the receiving computer, the UDP packets are not checked for missing data or even for data arriving out of order; all data is merely passed to the application. For example, a telephone application may send a UDP packet every 8 ms with 64 bytes in each in order to obtain the 64 kbps rate commonly used of the public switched telephone network. However, since UDP does not correct for missing data, audio quality degradation may occur in the application unless the underlying network may assist and offer QoS guarantees. For CBR traffic, the necessary QoS typically implies guaranteed delivery of all UDP packets without retransmission.

2.2 Transmission Control Protocol (TCP)

A best-effort application typically requires a connection-oriented, reliable protocol that allows one to send and receive as little as 1 byte at a time, similar to streaming file input and output. All bytes are guaranteed to be delivered in order to the destination, and the application is not exposed to the packet nature of the underlying network. On the Internet, the Transmission Control Protocol (TCP) is the most widely use protocol for BE traffic.

TCP is unsuitable for most CBR applications, because the protocol needs extra time to verify packets and request retransmissions. If a packet is lost in a CBR audio telephone call, it is more acceptable to allow a skip in the audio, instead of pausing audio for a period

of time while TCP requests retransmission of the missing data.

When TCP is packaging bytes into packets, it includes a sequence number in the packet header to assist the receiver in reordering data for the application. For every packet the destination receives in order, an acknowledgment packet is sent back to the source indicating successful receipt. If the receiver receives a sequence number out of order, the receiver may conclude the network lost a prior packet and inform the source by sending an acknowledgment (ACK) for the last sequence number received in order. Whether the receiver keeps or discards the latest out of order packet is implementation dependent.

RFC 793	RFC 1122	Tahoe	Reno	New Reno	Vegas	Win2k	
	X	X	X	X	X	X	Dynamic Window Sizing
	X	X	X	X	X	X	Exponential RTO Back-off
	X	X	X	X	X	X	Karn's Algorithm
	X	X	X	X	X	X	RTT Variance Estimation
	X	X	X	X		X	Slow Start
					X		Slow Start/RTT
					X		Constant Rate + "Extra Data"
		X	X	X	X	X	Fast Retransmit
			X	X	X	X	Fast Recovery
				X			" " Exit Fix (ACK highest)
						X	SACK (Reno+SACK)[12]
							Forward ACKs (SACK++)[11]

Table 2.1: Features in Different TCP Implementations.

The various TCP implementations which are in use today are deviations from the original protocol specified in RFC 793[15] and refined in RFC 1122[4], and are based upon attempts to optimize how senders and receivers communicate through ACKs to identify available bandwidth, congestion, and line errors. After RFC 1122, some newer techniques to improve TCP's performance were introduced such as Fast Retransmits and Fast Recovery and later documented in RFC 2001[17]. Table 2.1 highlights a few TCP implementation families such as those based upon BSD Tahoe and BSD Reno, and also includes the implementation found in the Windows 2000 operating system distribution[10].

New Reno TCP is included in the simulation package used in this research, and is based on Reno TCP except that "to exit fast recovery, the sender must receive an ACK for the highest sequence number sent."[18] Additionally, TCP Vegas[5] is also based upon Reno except it takes a rate limiting approach to congestion window management.

2.3 Satellite Transport Protocol (STP)

For BE traffic, we also examined a newer protocol designed specifically for use over satellite links, the Satellite Transport Protocol (STP)[8]. Unlike TCP which uses acknowledgments to communicate link statistics, an STP receiver will not send an ACK for every arriving packet. Instead, the receiver sends a status packet (STAT) when it detects missing packets, or when it receives a status request from the sender. This reduces the amount of status traffic from the receiver to the sender, and because of this, STP's founding research claimed significant throughput advantages over TCP. However as we found in our own tests, STP did not show any significant advantages over TCP across a suite of performance tests related to QoS for BE and CBR streams.

Chapter 3

Simulation Tools

To investigate network behavior, we could use hardware realizations, but they are too expensive when testing even a small network of 128 computers, 34 UDP-adapted telephones, and a geosynchronous satellite link. We could also employ mathematical models, but this would over simplify protocol mechanisms. In software based simulations, though, our experiments may be as detailed as desired and permit us to log any level of data detail, such as traces of individual packet histories.

3.1 Network Simulator

To speed test development, we chose to use an existing package named the Network Simulator, or "ns" for short. Ns is a software package for simulating networks with discrete events. It started as a "variant of the REAL network simulator in 1989" and is presently developed by the University of Southern California's Information Sciences Institute with support from DARPA and the NSF.

Ns's event simulator and network components are coded in C++, but tests are set up through an Object-Tcl interpreter. The simulation engine can handle up to 64000 node networks and comes with a variety of error models, link types, protocols, and queuing disciplines. Also, given that ns's source is freely distributed, it is highly extensible to add needed functionality on the fly.

3.2 Our Modifications & Scripts

Ns's extensibility comes at the cost of a higher learning curve resulting from the need for the user to modify the underlying C++ code, learning the Object-Tcl scripting language, and wrapping the simulator within shell scripts to run simulations, record data, and process or graph the results. Our ns C++ source modifications involved reincorporating components written for STP's founding research, the STP implementation itself and another object to emulate split connections. We wrote shell scripts to re-run ns for all combinations of test variables, and we employed "awk" and "gnuplot" scripts to extract measurements, such as packet delay variations, and graph the results for analysis.

We kept most workarounds for ns inconsistencies and limitations sequestered in the OTcl simulation setup scripts. For example, ns provides two styles of some of the best-effort protocols. "Full" protocol implementations simulate complete bidirectional data flow as in a real network, and "partial" protocol implementations only simulate one direction of data flow. The inconsistency is that the full implementations measure highest acknowledged data in bytes, and partial implementations measure highest acknowledged data in sequential packet numbers. Also, ns's partial TCP implementations do not deduct TCP+IP header bytes before reporting to the application.

However there were two problems we encountered in ns we were not able to correct through the OTcl scripts. Our first difficulty was that the existing STP implementation did not support an application layer for counting received bytes. We were able to fix this deficiency though by modifying STP's C++ code in ns. However after attempting to simulate connection splitting in ns, we found a design assumption within ns hindered our progress. Ns assumes an application is always ready to receive data. Internally, transport protocols within ns call an application's receive call-back function for any incoming bytes. Unfortunately, this mechanism does not provide any means for the application to inform the protocol when it is busy or that it has not finished processing prior data. This is a major hurdle for connection splitting, because there are no convenient means for an application to signal an upstream protocol when a downstream protocol is still sending data. Later in our research, we abandoned the topic of connection splitting due to the inherent sacrifice

of network fault tolerance which is introduced by this approach.

Chapter 4

Application Throughput & QoS

In the following experiments, we set out to investigate the QoS when running both CBR audio traffic and BE file transfer traffic over military geosynchronous satellite links. This chapter presents our QoS and throughput observations for different router queuing disciplines, BE protocol implementations and amount of CBR traffic present on the network.

4.1 Test Setup

In these experiments, we sought to measure QoS parameters such as throughput and packet delay variation in a network with a military satellite link. We modeled this network as a single satellite link with a bandwidth of 1.536 Mbps and a delay of 450 ms with uplink routers at each end. Then over this link, we ran 64 BE file transfers and between 0 and 17 CBR phone calls.

We defined each file transfer or phone call as two separate traffic flows, one for each direction of the satellite link, and we connected a computer node at each uplink. For a given flow of phone data, we encapsulated the 64 kbps telephone traffic as 64 data bytes per UDP packet sent every 8 ms. To model file transfer flows, we used a BE protocol, either TCP or STP, and simulated transmitting an unlimited size file as fast at the network permits.

To complete the definition of our simulated network, we set the bandwidth of all links from the uplink routers to their attached computer nodes to 10 Mbps. Propagation delays for all CBR node links are 5 ms, and those of BE node links are evenly distributed from 4

to 100 ms. Fig. 4.1 portrays a network with only 2 CBR (blue) and 3 BE (green) sources at each uplink, and table 4.1 summarizes which nodes communicate with each other, their protocol, and their one-way propagation delay.

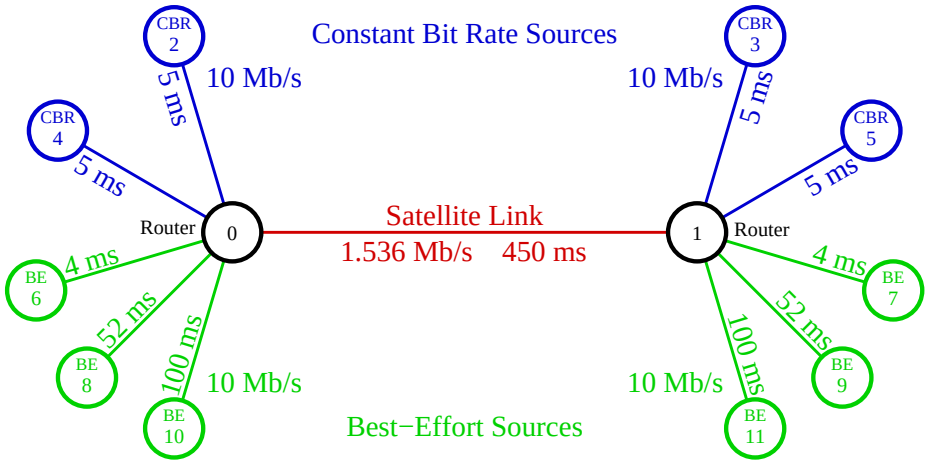

Figure 4.1: An example ns configuration involving 3 Best Effort and 2 Constant Bit Rate pairs of sources.

Node Pair	Traffic	Protocol	Delay
2, 3	CBR	UDP	450+5+5 = 460 ms
4, 5	CBR	UDP	450+5+5 = 460 ms
6, 7	BE	STP/TCP	450+4+4 = 458 ms
8, 9	BE	STP/TCP	450+52+52 = 554 ms
10, 11	BE	STP/TCP	450+100+100 = 650 ms

Table 4.1: Network node pairs, protocols & delays used for the example ns configuration with 3 BE and 2 CBR source pairs.

The actual test topology uses 64 pairs of BE sources instead of 3, and the link delays linearly range from 4 to 100 ms at each router. Fig. 4.2 shows the resulting crowded topology with 64 BE and 17 CBR sources as a blur of superimposed links and sources.

T1 links also have a 1.536 Mbps bandwidth, and normally carry 24 phone calls of 64 kbps each in the public switched telephone network. However in our network configuration, we only tested up to 17 simultaneous calls. When we encapsulate CBR audio into UDP, the UDP header adds an extra 18 bytes of information for every 64 audio bytes. So when a

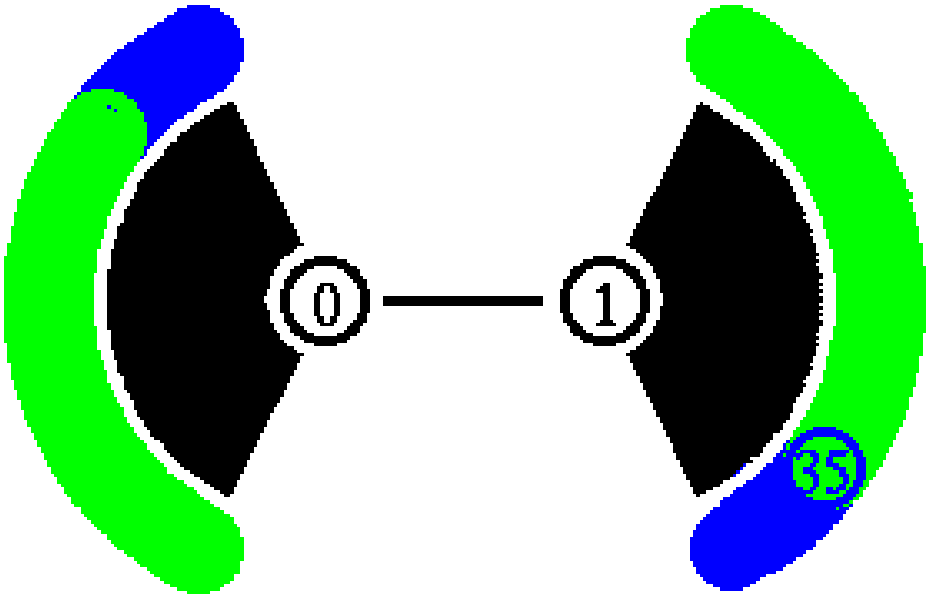

Figure 4.2: The ns configuration used in the test evaluated in chapter 4 involving 64 Best Effort and 17 Constant Bit Rate pairs of sources.

phone transmits 64 audio bytes every 8 ms over UDP, the packets are really 92 bytes, and the resulting flow consumes 92 kbps of bandwidth. With a total satellite link bandwidth of 1.536 Mbps, our network can only support 16.70 calls when encapsulating them in UDP. An evaluation for a specific implementation of VoIP would require accounting for the variable packet size and data rate associated with that system.

4.1.1 Test Timing Parameters

Instead of starting all CBR and BE flows at the beginning of each test, we start the CBR sources at 0 seconds and staggered start groups of 8 BE pairs at a time to partially randomize network traffic patterns. As shown in Fig. 4.3 and table 4.2, the first group of 8 BE pairs starts at 0.1 seconds, the next group at 0.6 seconds, the next at 1.1 seconds, and so on until the eighth group starts at 3.6 seconds. As each pair of BE nodes starts, the nodes initiate connections to each other and proceed to send limitless amounts of data.

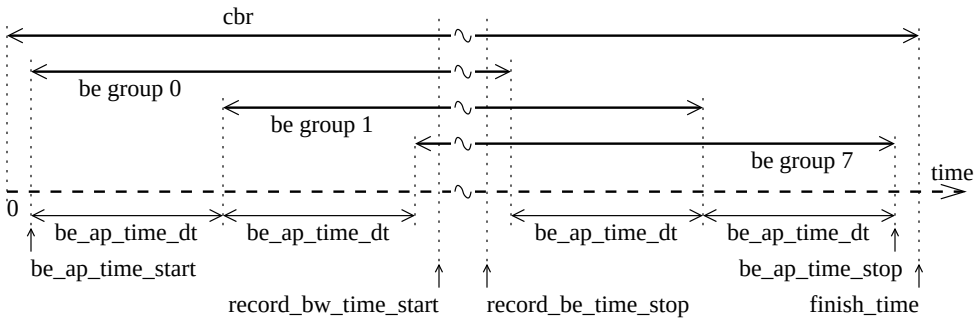

Figure 4.3: Timing diagram for the test evaluated in chapter 4.

be_ap_time_start	0.1 s	cbr_ap_time_start	0.0 s
be_ap_time_stop	127.1 s	cbr_ap_time_stop	128.1 s
be_ap_time_dt	0.5 s	cbr_delay	5 ms
be_delay_min	4 ms	cbr_size	92 bytes
be_delay_max	100 ms	cbr_interval	0.008 s
be_ap_groups_num	8	finish_time	128.6 s
sat_bw	1536000 bps	record_bw_time_start	63.6 s
sat_delay	0.450 s	record_bw_time_stop	123.6 s

Table 4.2: Timing parameters chosen for the test evaluated in chapter 4.

4.1.2 Test Implementation Constants

In each test, we identified a number of input parameters and held several of these constant while varying others between tests. All tests used a single satellite link with 2 uplink routers, and each router had an input queue size of 64 MTU[1]-sized packets. Every test has 64 pairs of BE nodes at each side of the network. The MTU for TCP and STP packets is 576 bytes, and the size of each UDP packet is 92 bytes to include 64 bytes of audio and 18 bytes of UDP/IP headers. The link bandwidths, delays, test timing parameters, and UDP packet transmission rate are also fixed to the values discussed in the previous sections.

4.1.3 Test Implementation Variables

Between tests, we varied the number of CBR source pairs, the routers' queuing discipline, and the protocol used by BE sources. The number of CBR pairs ranged between 0 to 17, in steps of 1. The routers used one of four queuing disciplines from table 4.3, which will be discussed in the next sections.

Drop-Tail	RED
CBQ	CBQ+RED

Table 4.3: Router queuing disciplines used in our tests.

For the BE sources, we wanted to compare TCP and STP, but TCP has many different implementations. So rather than testing with only 2 protocol implementations, we tested one STP implementation and six TCP implementations available in ns. Table 4.4 summarizes the 7 protocols tested for BE traffic.

However as mentioned in section 3.2, ns contains two styles of TCP stack implementations. "Full" protocol implementations simulate complete bidirectional data flow as in a real network, and "partial" protocol implementations only simulate one direction of data flow. Ns provides partial protocol modules for all of our tested TCP and STP implementations, and in addition to this, ns comes with full protocol modules for four of the TCP implemen-

[1]A Maximum Transmission Unit (MTU) is the maximum size of an IP packet allowed in a network including packets headers and data payload.

Protocol	Implementation
TCP	Tahoe
TCP	Reno
TCP	New Reno
TCP	Vegas
TCP	SACK
TCP	FACK
STP	STP

Table 4.4: Best-Effort protocols available in ns.

tations as well. Even though ns's documentation states the full protocol implementations are not as well tested, we still included them in our research for completeness. So we tested a total of eleven protocol implementations as summarized in table 4.5.

Protocol	TCP	TCP	TCP	TCP	TCP	TCP	STP
Implementation	Tahoe	Reno	New Reno	Vegas	SACK	FACK	STP
"Partial"	X	X	X	X	X	X	X
"Full"	X	X	X		X		

Table 4.5: Best-Effort protocol implementations tested in this evaluation.

4.1.4 Collected Data

In each test, we measured BE data throughput as well as the CBR packet losses and delays for each router queuing discipline, and we captured these measures for each individual connection. Then using post simulation processing, we calculated the mean data throughput and standard deviation across all connections to evaluate the *fairness* of bandwidth division.

4.2 Queuing Throughput & QoS Tests

For our first level evaluation of TCP versus STP performance, we compared the throughput and throughput variance experienced by BE sources, and the CBR Quality of Service (QoS) parameters. This section introduces the router queuing disciplines listed in section 4.1.3 and presents the results of our tests obtained with each discipline.

4.2.1 Drop-Tail Simple Router

Drop-Tail is a basic router queuing discipline where a router drops incoming packets if the router's packet input queue is full. In this test, we set the uplink routers to use Drop-Tail queuing, and we measured BE throughput and CBR packet loss and delay, versus the number of CBR sources. Fig. 4.4 shows the results from this test, and in it, we find that STP distinguishes itself with reduced variance of bandwidth between BE connections. And when more than 9 CBR source pairs were present, consuming over half the link bandwidth, STP demonstrates the best average throughput.

However, when looking at the quality of service obtained for CBR traffic in Fig. 4.5, we found the CBR packet losses and delays were very high due to congestion from any of the best-effort protocols. On closer inspection, we found 2000 UDP packets were lost per minute when BE traffic was using New Reno TCP, and 3600 packets were lost per minute when using STP. So STP's throughput advantage came at a cost of higher CBR packet loss. For perspective, a 64 kbps telephone call encapsulated in 64 data byte UDP packets requires 7500 packets per minute. So using New Reno resulted in a loss of 27% of all packets, while using STP destroyed 48%.

Also, why did the delay experienced by CBR traffic decrease by 10 ms when the number of CBR sources increased from 0 to 17? Should not increased CBR traffic, only add to the congestion and slow things further? With some mental analysis, it becomes apparent that CBR performance improved slightly because of the increased number of *short* (92 octet) CBR packets filling the router queues that yielded smaller scheduling delays versus the longer (MTU=576 octet) packets.

4.2.2 Random Early Detection

Next, we evaluated the results again with a change in queuing discipline: both routers using Random Early Detection (RED). Sally Floyd and Van Jacobson introduced RED with the goals of keeping a low average queue size, allowing occasional packet bursts in the queue, and slowing TCP sources before hitting congestion. By prematurely throttling back TCP sources, RED seeks to reduce retransmissions by avoiding full congestion, and also

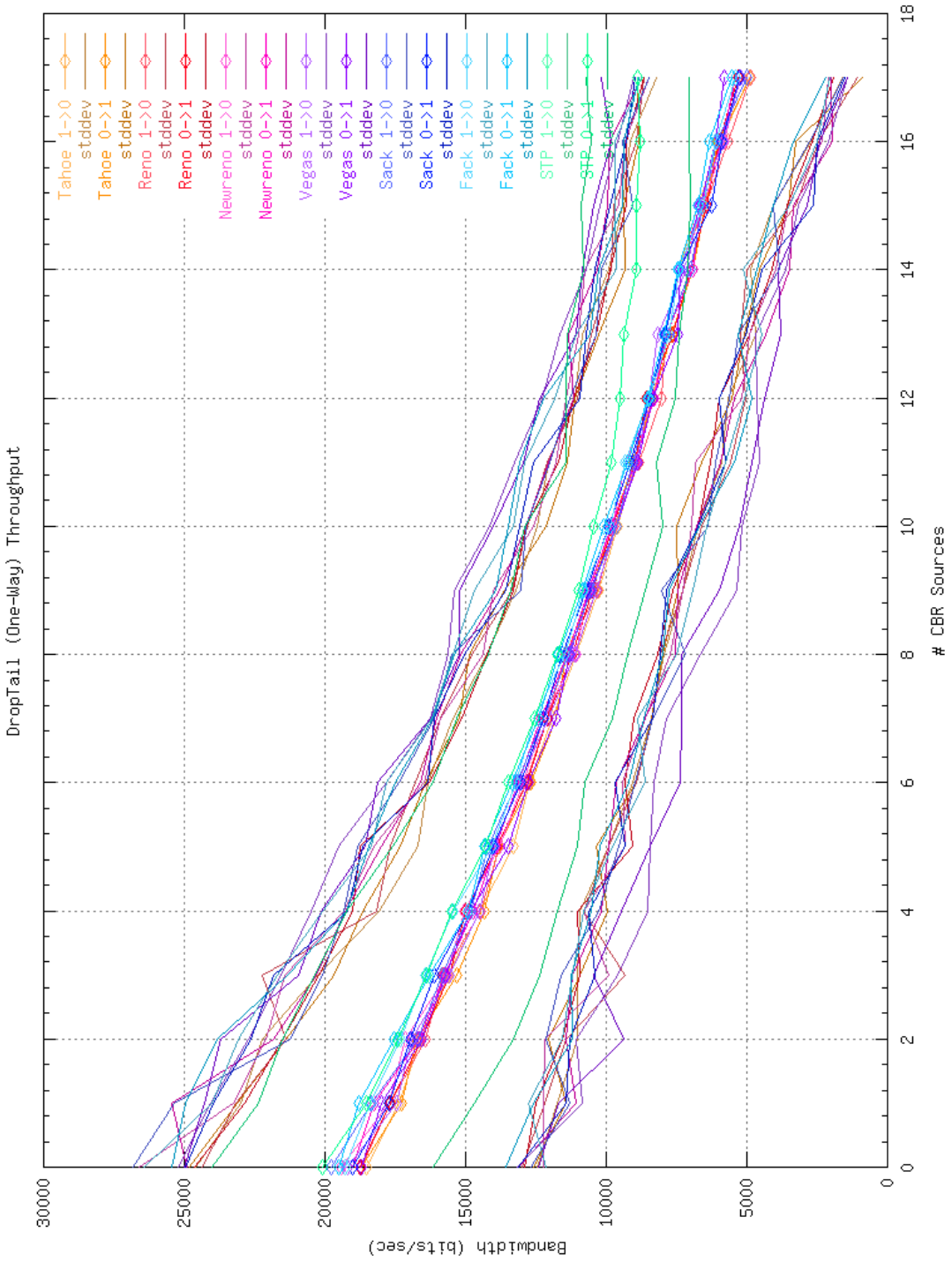

Figure 4.4: BE throughput when using Drop-Tail simple routers.

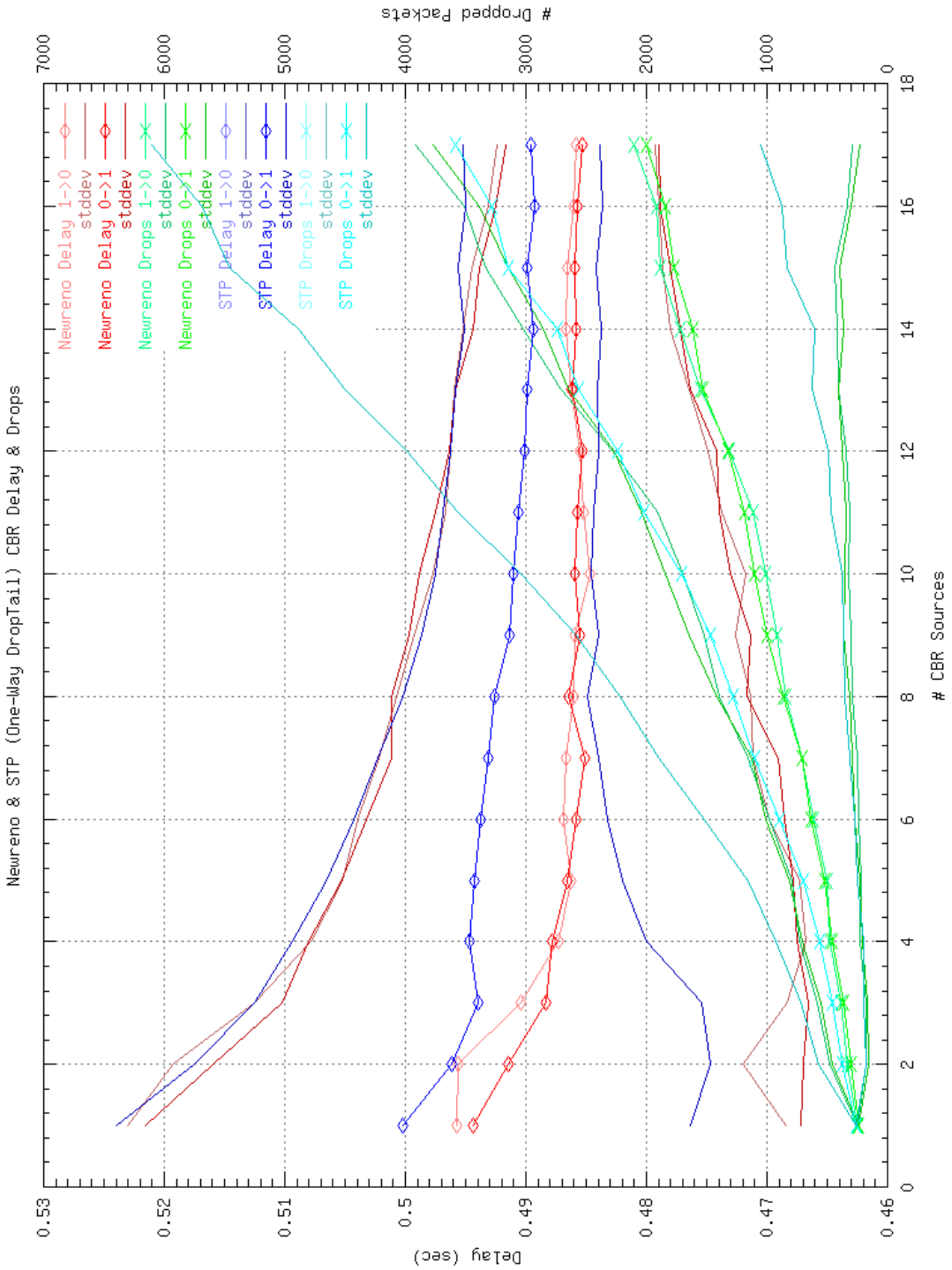

Figure 4.5: CBR QoS when using Drop-Tail simple routers.

to avoid global synchronization of many TCP/STP stacks simultaneously decreasing their windows.

We used "gentle RED" in this test which works in two stages. It first computes the average router input queue length using an exponentially decreasing weighted sum. Then RED drops or marks packets with increasing probability as the average queue length increases.

More specifically, RED does not drop any packets until the average queue length reaches a minimum threshold, $thresh$. Then it begins dropping with a probability that linearly increases from 0 to p as the average queue length approaches a second threshold, $maxthresh$. Once the queue length exceeds $maxthresh$, the drop probability linearly increases from p to 1 as the queue approaches $2\,maxthresh$. Equation 4.1 expresses this relationship.

$$p_{drop} = \begin{cases} 0 & 0 \le q \le thresh \\ p\frac{q-thresh}{maxthresh-thresh} & thresh < q \le maxthresh \\ (1-p)\frac{q-maxthresh}{maxthresh} + p & maxthresh < q \le 2\,maxthresh \end{cases} \tag{4.1}$$

Tested RED Configuration

Using suggestions from one of RED's creators[7], we used these parameters in each of our tests: $thresh = 5$, $maxthresh = 32$, and $p = 0.1$. Fig. 4.6 graphs the resulting relationship between average queue size and incoming packet drop probability.

RED: Observations

When using Drop-Tail queuing, the average throughput for best-effort traffic was 5357 bps when 17 Constant Bit Rate source pairs were using the link. However when using Random Early Detection at the routers, this common *performance* solution dropped the average throughput by 25% to 4000 bps as shown in Fig. 4.7. Also, STP still has a throughput advantage, though now the throughput variance of TCP is closer to that of STP.

In the results in Fig. 4.8, we found that though CBR packet loss is reduced, it is still incredibly high. For the tests with New Reno, the loss of 1300 packets per CBR flow over the simulation's 1 minute interval is a 17% packet loss, and for the STP tests, the loss of 2500 packets per minute is a 33% loss of audio. So RED helped reduce audio traffic losses,

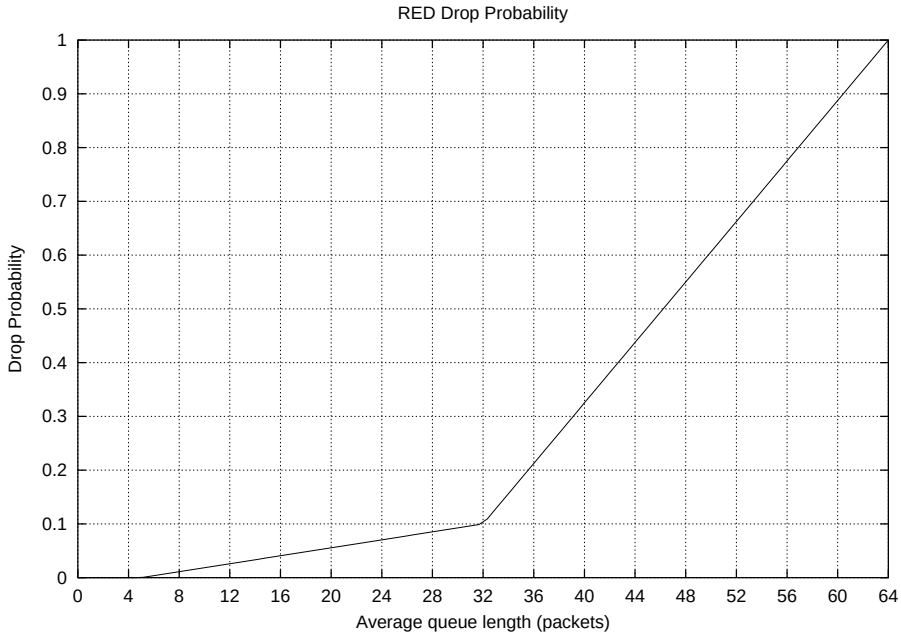

Figure 4.6: RED's packet drop probability as the average queue length ranged from 0 to 64 packets.

but we still have not achieved telephone quality of service for CBR traffic.

4.2.3 Class Based Queuing (CBQ)

David Clark & Van Jacobson introduced another router enhancement, named Class Based Queuing, that categorizes and places incoming packets into separate queues, instead of a single queue like RED or Drop-Tail[6]. These queues may be prioritized, have bandwidth limits, and share bandwidth with other queues. Different algorithms may be used to manage each queue such as Drop-Tail, RED, or even a nested CBQ implementation to create hierarchies.

Tested CBQ Configuration

In our CBQ tests, we placed CBR and BE traffic into separate queues, as summarized in table 4.6. We managed each queue with Drop-Tail and did not place bandwidth constraints. Though we did give the CBR traffic queue a higher priority to cause its packets to always

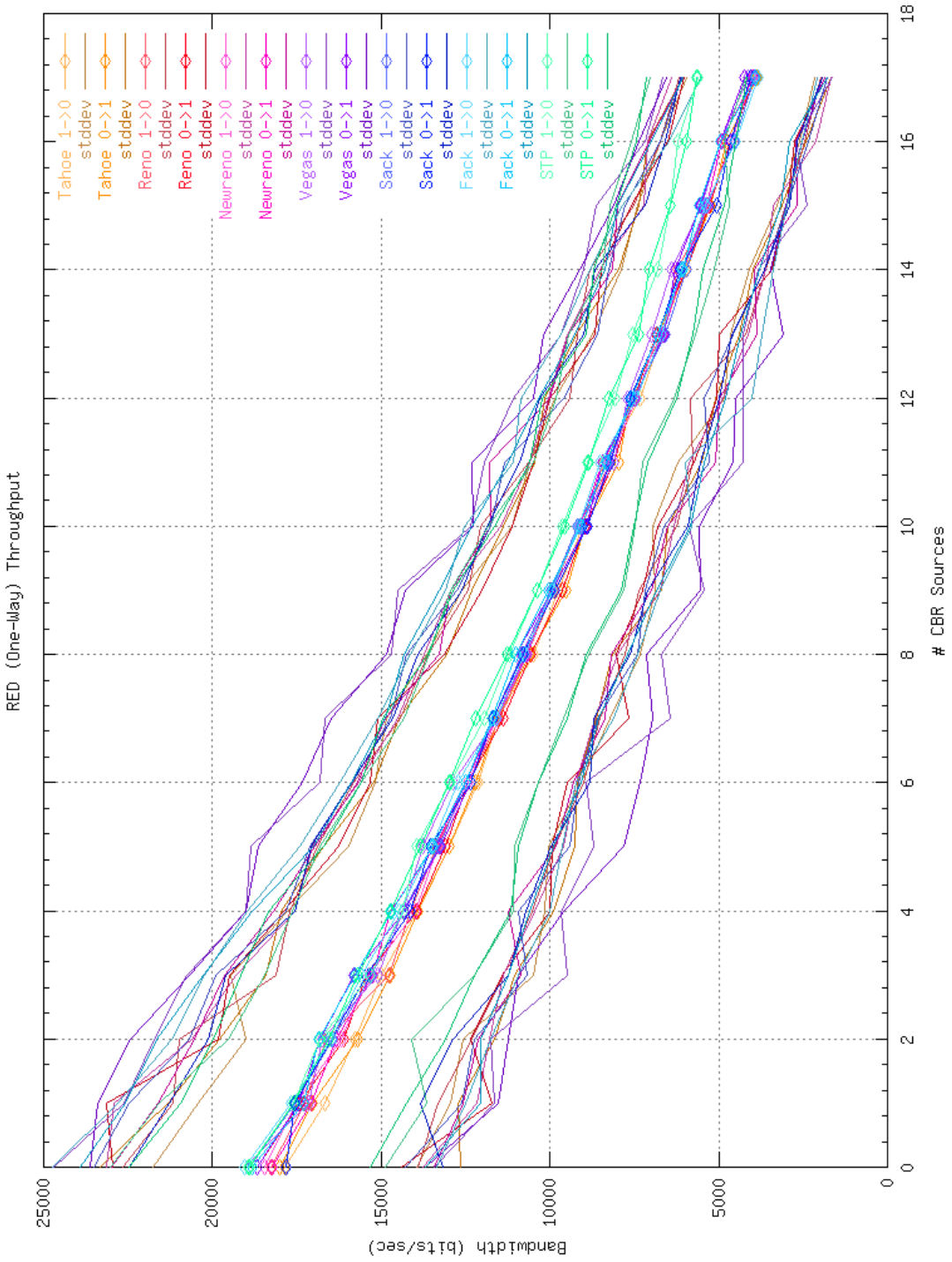

Figure 4.7: BE Throughput when using RED simple routers.

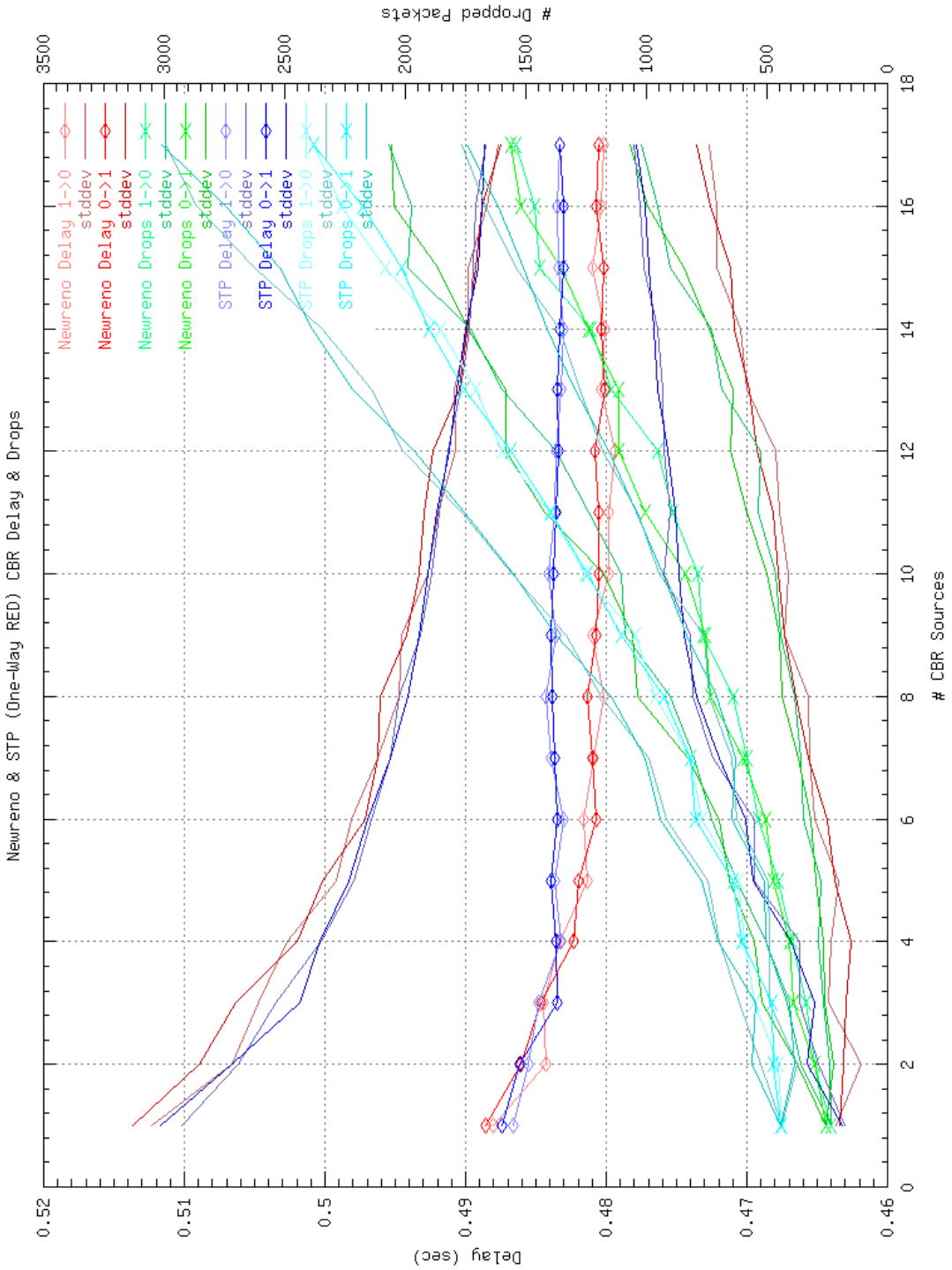

Figure 4.8: CBR QoS when using RED simple routers.

be sent before any BE traffic.

Queue	Priority	Size	Discipline	Packet Class
cbqCBR	1	(eq 4.2)	Drop-Tail	Constant Bit Rate
cbqBE	2	64	Drop-Tail	Best Effort

Table 4.6: Our CBQ configuration.

We set the BE queue size to 64 MTU sized packets, and for the CBR queue, we calculated its size to hold the worst case number of CBR packets simultaneously in transit for each test. Equation 4.2 describes our approximation for calculating the CBR queue length. We first determine how many CBR packets may arrive while waiting to send one BE packet, and then we add how many more CBR packets may arrive while still servicing the previous surge of CBR packets. Table 4.7 summarizes the CBR queue lengths used during each tested CBR load.

$$
\begin{aligned}
A &= \underbrace{cbr_{num} \cdot \lceil \frac{8 \cdot be_{size}}{sat_{bw} \cdot cbr_{interval}} \rceil}_{\text{CBR packets arriving while sending a BE packet}} \\
cbr_{qlen} &= A + \underbrace{cbr_{num} \cdot \lceil \frac{8 \cdot cbr_{size} \cdot A}{sat_{bw} \cdot cbr_{interval}} \rceil}_{\text{CBR packets arriving while sending previous CBR packets}}
\end{aligned}
\tag{4.2}
$$

#CBR pairs	Queue Size	#	Q	#	Q	#	Q	#	Q	#	Q
0	0	3	6	6	12	9	18	12	24	15	30
1	2	4	8	7	14	10	20	13	26	16	32
2	4	5	10	8	16	11	22	14	28	17	51

Table 4.7: BE queue sizes chosen in our CBQ configuration.

CBQ: Observations

Fig. 4.9 shows our measured BE throughput when using CBQ at the routers. Here, we found the best-effort throughput drops proportionally and appropriately as the number of CBR sources increases, and we found STP's average throughput is comparable to that of

TCP. However, STP still has the lowest bandwidth variance across the 64 flows indicating better fairness than TCP to flows with differing Round Trip Times (RTTs).

Also we noticed that with only a few CBR phone calls on the network, STP never achieved the throughput advantages over TCP that were originally claimed in STP's background research, and this observation also includes our previous tests with RED and Drop-Tail queuing. Also in this CBQ test, we found that STP performed slightly worse than TCP at high CBR loads as well. We followed up on this peculiarity in later tests, and section 8.1 presents our conclusions.

In the data presented in Fig. 4.10, we found that CBR packet loss was completely avoided until the link was stressed beyond capacity with 17 CBR flows. As as noted in section 4.1, the link can only handle 16.7 flows.

Also, the mean delay encountered by CBR packets traveling across the network increased proportionally with the number of CBR calls. With only one active call, CBR packets experienced a 462 ms delay, but with 16 calls, these packets were delayed 466 ms. From the test configuration in table 4.1, straight propagation delay for CBR traffic was 460 ms, so here, CBR UDP packets experienced an average 2 to 6 ms in queuing/transmission time with a standard deviation of 2 ms.

4.2.4 Testing CBQ & RED Together

This section presents our findings when using RED in conjunction with CBQ. For these tests, we continued to use the CBQ configuration presented in section 4.2.3, but instead of both queues using Drop-Tail, we configured the best-effort traffic queue with Random Early Detection using the parameters from section 4.2.2. Table 4.8 summarizes the new class based queuing discipline with RED for BE traffic.

Queue	Priority	Size	Discipline	Packet Class
cbqCBR	1	(eq 4.2)	Drop-Tail	Constant Bit Rate
cbqBE	2	64	RED	Best Effort

Table 4.8: Our CBQ+RED configuration.

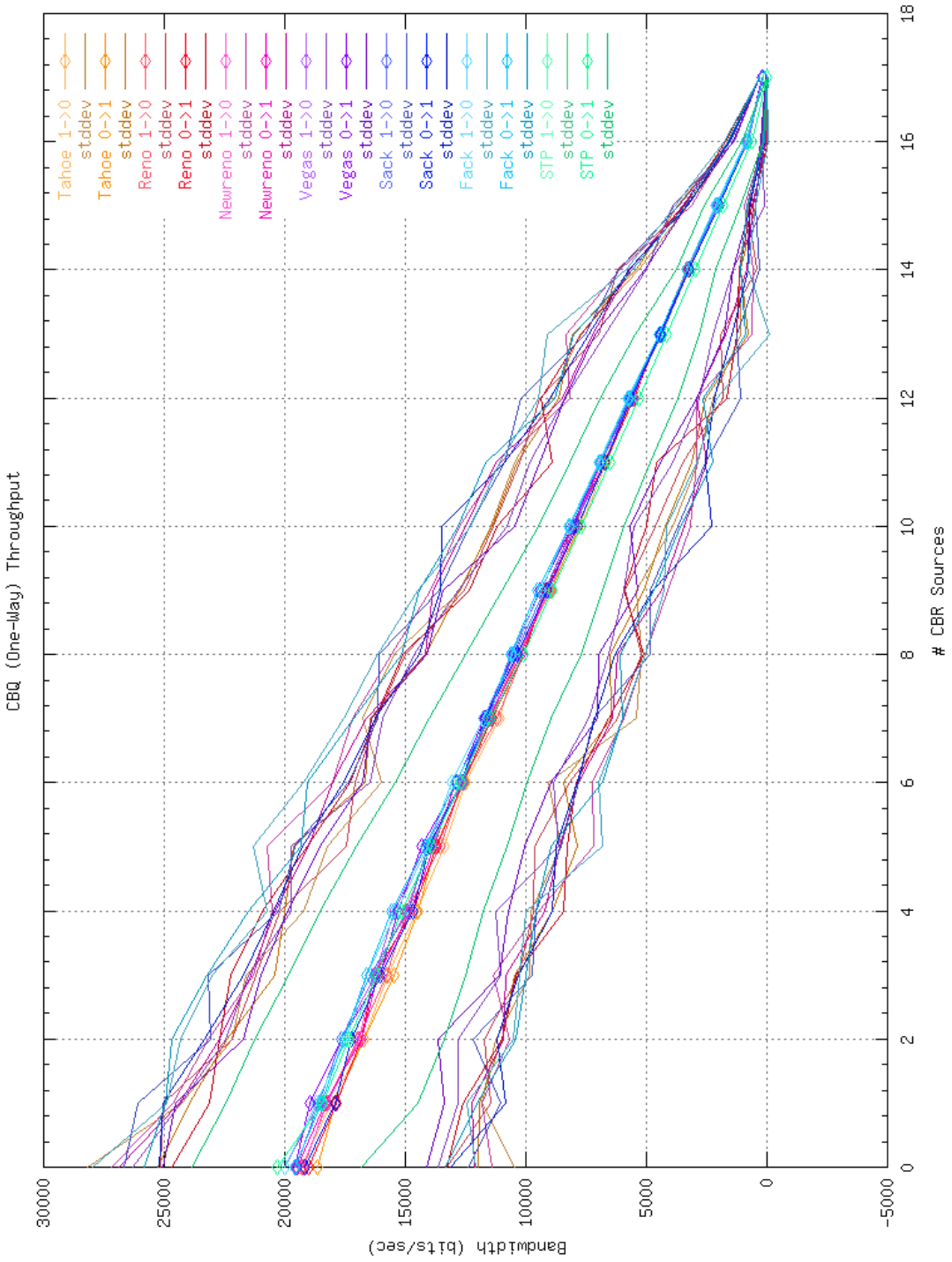

Figure 4.9: BE Throughput when using CBQ simple routers.

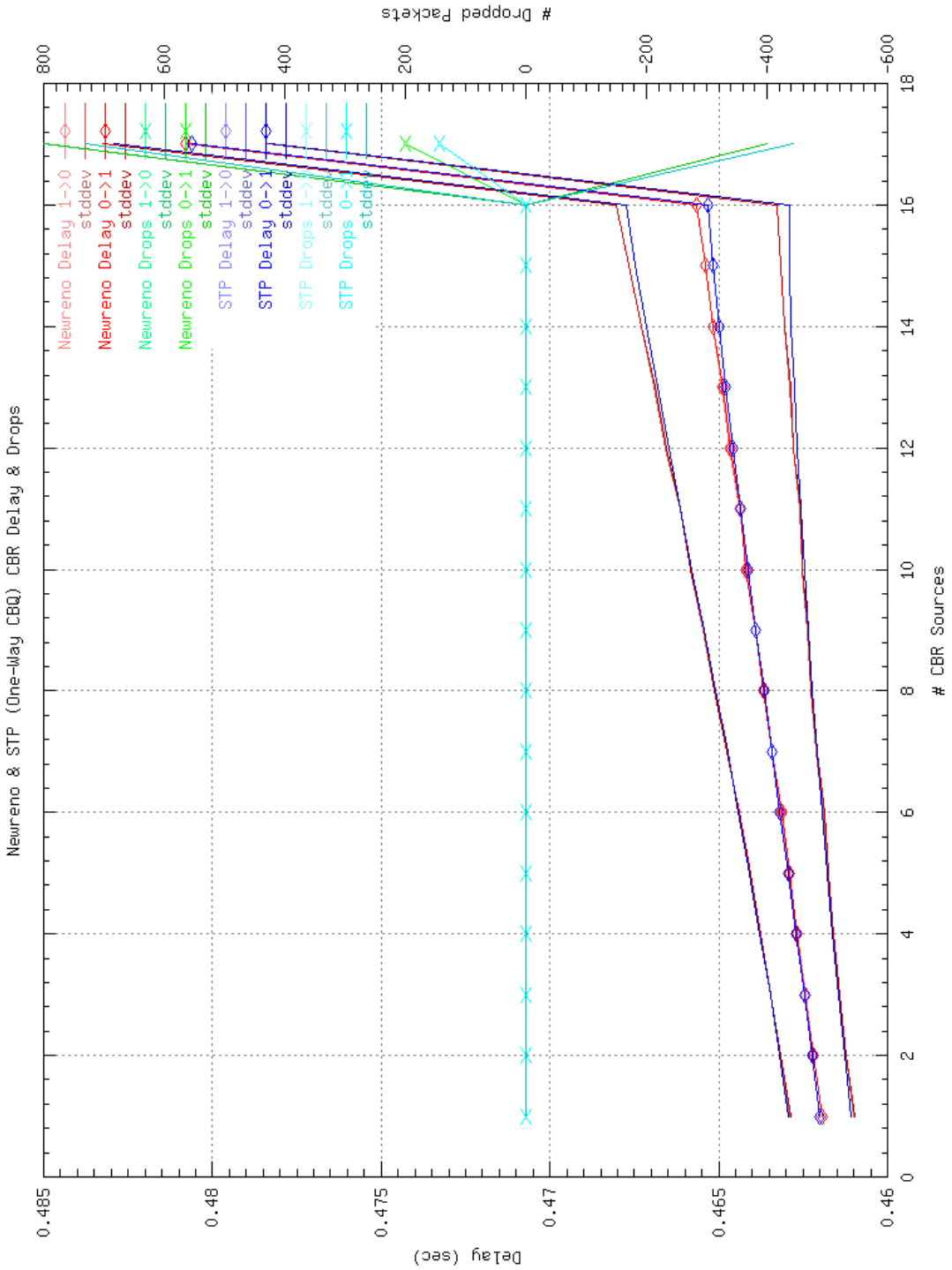

Figure 4.10: CBR QoS when using CBQ simple routers.

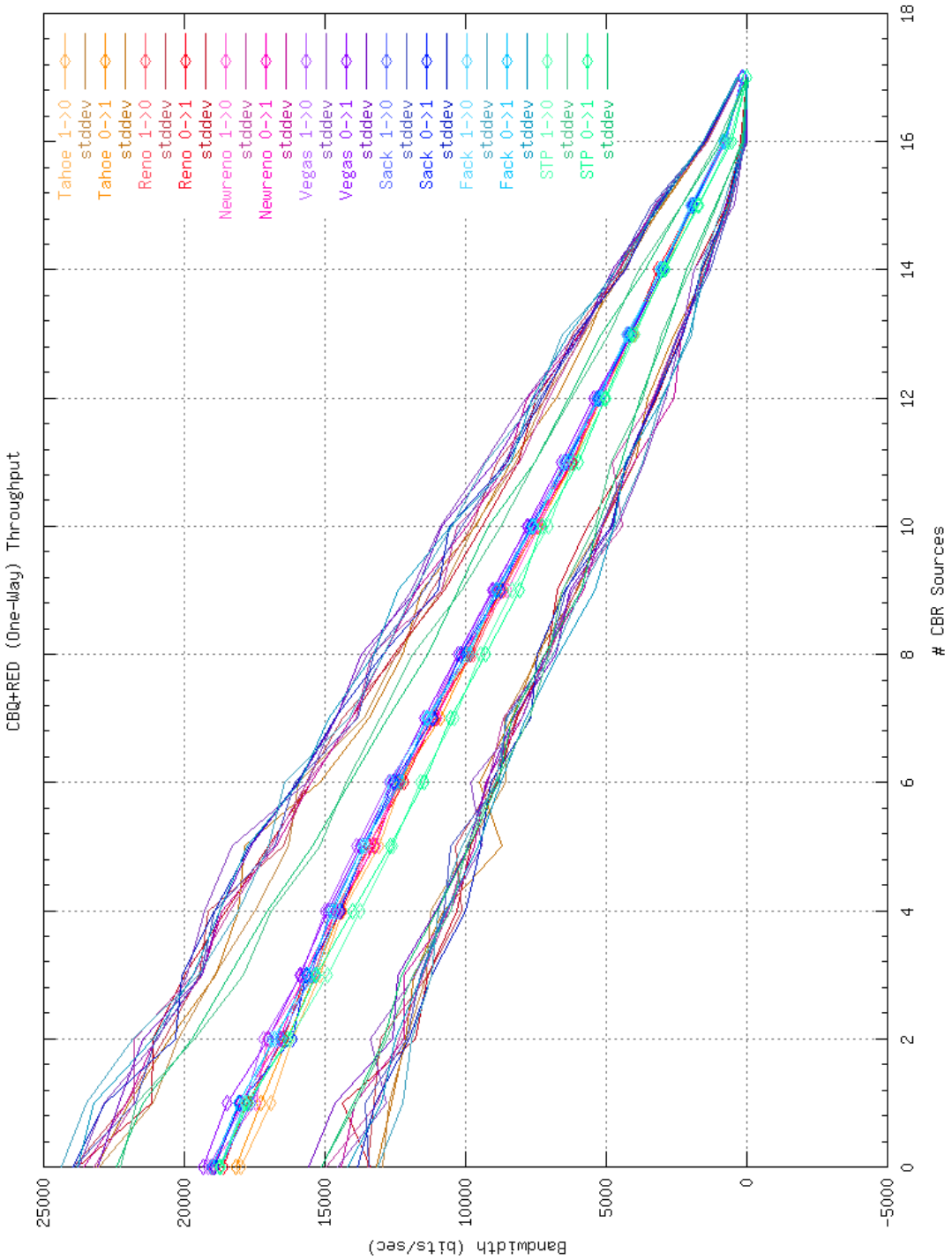

Figure 4.11: BE Throughput when using CBQ+RED simple routers.

When comparing the CBQ+RED results in Fig. 4.11 to the CBQ results in Fig. 4.9, we found CBQ+RED decreased TCP's standard deviation of flow throughput by ≈3 kbps when there were zero to twelve CBR calls on the network. For STP, CBQ+RED induced a slight decrease in best-effort throughput compared to CBQ, and did not affect throughput variance significantly. **The result is: TCP and STP realized near identical results for this queuing discipline.**

As seen in Fig. 4.12, adding RED to CBQ did not affect the Quality of Service for Constant Bit Rate traffic. This is expected, because RED was added only to the BE class queue, not to the CBR queue, and CBQ has already provided good QoS for CBR traffic. RED is only intended for use on transport protocols that react to packet loss by reducing throughput - specifically TCP.

4.3 Analysis

In this chapter, we presented our tests to measure how CBR load and uplink router queuing discipline affects BE performance and CBR QoS. From the obtained data, we made comparisons between TCP and STP, and we determined how to obtain the desired QoS for CBR traffic encapsulated in UDP packets.

4.3.1 STP Summary

Without CBQ routers, STP obtained higher throughput than TCP at high loads at the expense of losing CBR packets. With the CBQ discipline and at high CBR loads, STP had a slightly lower throughput than TCP. When using CBQ routers with RED, the performance of TCP & STP were essentially identical, achieving the best throughput and delay variance performance across all tested approaches.

STP often had a smaller throughput variance across all sources indicating it is more fair across BE sources with differing round trip times. Unless a minor improvement in best-effort throughput variance only at high CBR load levels (at a cost of lower throughput at low CBR levels) is important, there appear to be no reasons to use STP over TCP when CBQ+RED routers are used.

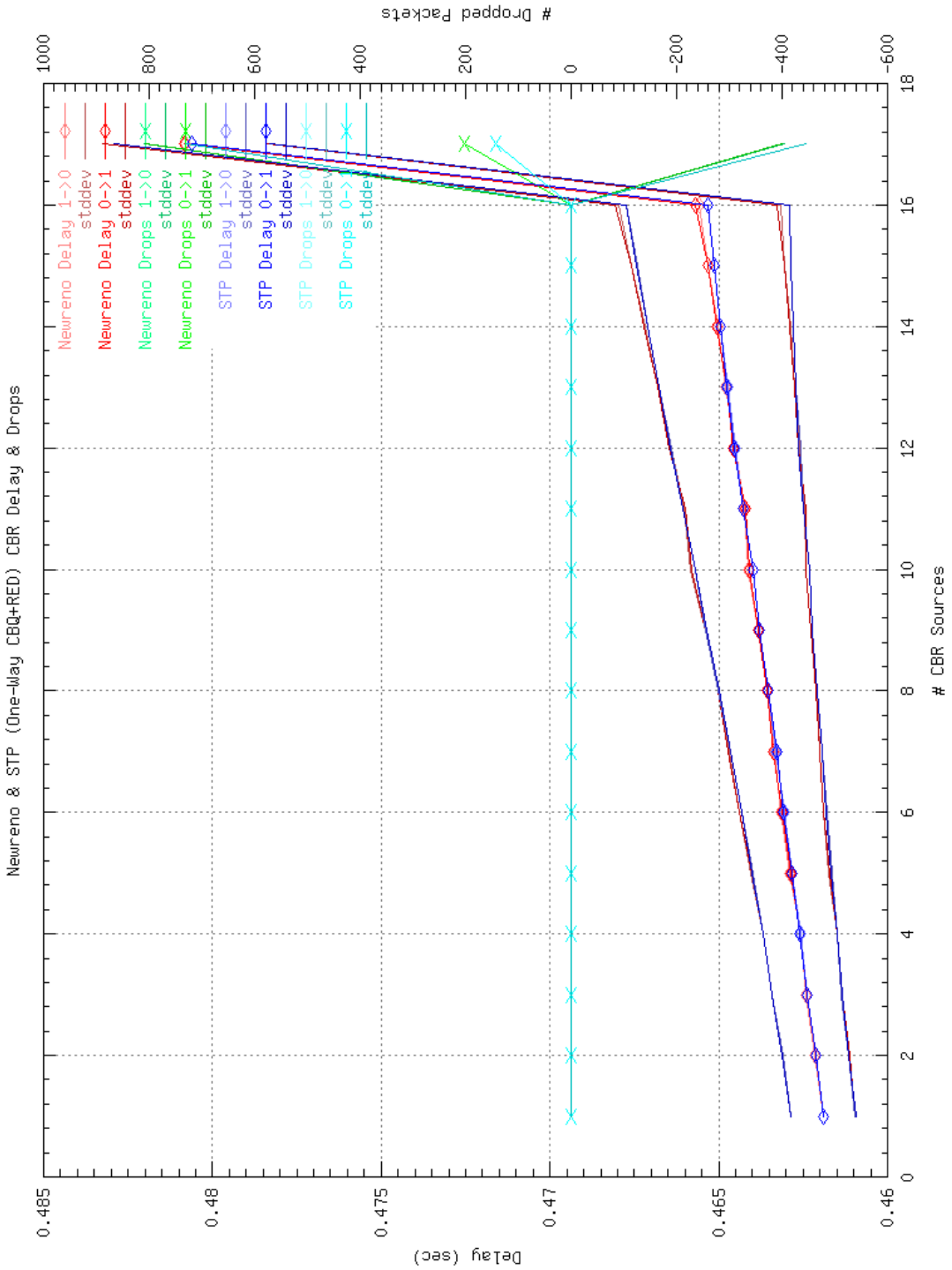

Figure 4.12: CBR QoS when using CBQ+RED simple routers.

Also as we noted earlier, STP does not demonstrate significant throughput advantages over TCP. This finding is contrary to STP's background research, and after further testing, we give our analysis of this discrepancy in section 8.1.

4.3.2 TCP Summary

We tested six derivatives of TCP, and of these, ns offered an alternative "full" two-way implementation for four of these TCP derivatives in addition to a "partial" one-way implementation. In total, we tested 10 TCP implementations in ns but later decided to omit the results for the four "full" TCP stacks. These implementations were considered untested by ns's authors, and the data we obtained for those implementations closely resembled that of the "partial" implementations anyhow.

We found the 6 "partial" TCP protocol implementations performed similarly to each other for each given queuing discipline. Comparing TCP and STP, the CBQ and CBQ+RED disciplines closed TCP's throughput gap to STP, and when using CBQ+RED, TCP obtained similar throughput variance to STP as well.

4.3.3 Queuing Discipline Summary

Of the four tested queuing mechanisms, CBQ and CBQ+RED satisfied our QoS requirement of guaranteed delivery for CBR phone packets. No CBR packets were lost until the link was overwhelmed, and CBR traffic experienced a low additional mean delay of 2-6 ms and standard deviation of 2 ms.

We also observed that, regardless of whether CBQ was used or now, using RED decreased TCP's throughput standard deviation by $\approx$3 kbps across flows with differing RTT when zero to twelve CBR calls occupied the network. This signifies that RED increased TCP's bandwidth allocation fairness to flows of differing RTT.

Chapter 5

Bit Error Rate and Performance

Real network links have some non-zero probability of bit corruption given by a bit error rate (BER), but our previous tests assumed perfect network links without bit errors. How does bit error rate affect protocol performance? Do we see advantages for protocols such as STP and TCP/IP with FACK/SACK? In this chapter, we explore how bit error rate affects transport protocol throughput by sweeping BER and recording the average application data throughput for each protocol.

5.1 Test Implementation

For our base topology, we started with that described in section 4.1, with 2 satellite uplink gateways, a 1.536 Mbps satellite link, and 64 BE users at each gateway. However in these tests, we varied BER from 10^{-8} to 10^{-3} and did not test with CBR sources. Since we excluded CBR sources from these tests, we did not need Class Based Queuing to protect CBR traffic and therefore only tested with Drop-Tail and RED routers.

Fig. 5.1 shows the simulated network with no CBR sources and only 3 BE sources at each router, and table 5.1 lists the communicating node pairs with one-way path delay times. Again, the actual test topology used 64 BE source pairs instead of 3.

Similar to our previous tests, the best-effort transport protocol (table 4.4), router implementation (table 5.2), and Bit-Error-Rate (table 5.3) do not change during one test trial. Instead many test trials were used to iterate over all combinations of input variables.

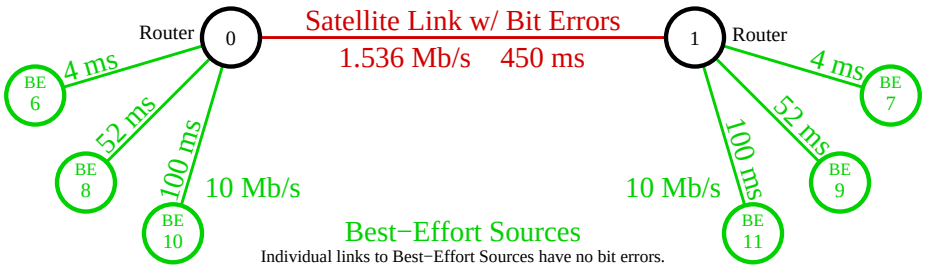

Figure 5.1: An example ns configuration involving 3 Best Effort and no Constant Bit Rate pairs of sources.

Node Pair	Traffic	Protocol	Delay
6, 7	BE	STP/TCP	450+4+4 = 458 ms
8, 9	BE	STP/TCP	450+52+52 = 554 ms
10, 11	BE	STP/TCP	450+100+100 = 650 ms

Table 5.1: Network node pairs, protocols & delays used for the example ns configuration with 3 BE and no CBR source pairs.

Fig. 5.2 and table 5.4 enumerate the exact timing parameters which primarily differ from our previous Throughput and QoS tests (section 4.1.1) through the omission of CBR sources.

Drop-Tail RED

Table 5.2: Router queuing disciplines used in our tests evaluating bit error rate impact on performance.

5.1.1 RED Configuration

We used the same RED configuration from section 4.2.2 and Fig. 4.6 for these tests. To summarize, our simulated routers drop packets instead of mark them, have a drop probability of 0 when the average queue length is between 0 and 5 packets which linearly increases to .1 from 5 to 32 packets and increases to 1 from 32 to 64 packets.

$$0 \quad 10^{-8} \quad 10^{-7} \quad 10^{-6} \quad 10^{-5} \quad 10^{-4} \quad 10^{-3}$$

Table 5.3: Bit error rates chosen for the test evaluated in chapter 5.

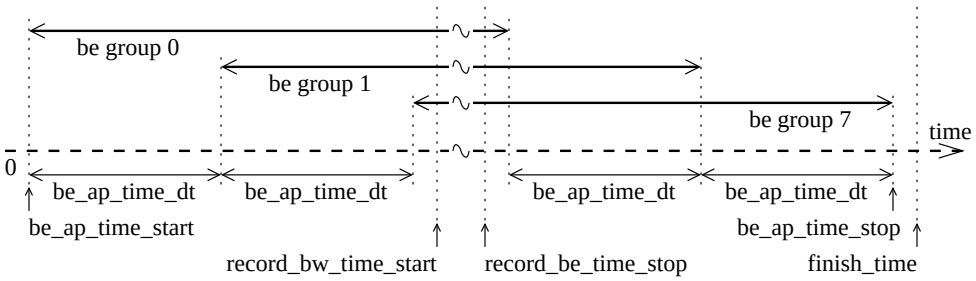

Figure 5.2: Timing diagram for the bit error rate test evaluated in chapter 5.

be_ap_time_start	0.1 s	sat_bw	1536000 bps
be_ap_time_stop	127.1 s	sat_delay	0.450 s
be_ap_time_dt	0.5 s	finish_time	128.6 s
be_delay_min	4 ms	record_bw_time_start	63.6 s
be_delay_max	100 ms	record_bw_time_stop	123.6 s
be_ap_groups_num	8		

Table 5.4: Timing parameters chosen for the test evaluated in chapter 5.

5.2 Data Collected in BER Tests

For each BER test, we measured the Best Effort throughput for each Bit-Error-Rate and queuing discipline, and individually captured each connection's results. Then, using post simulation processing, we found the mean and standard deviation of bandwidth across all connections to determine the *fairness* of bandwidth division.

5.2.1 Drop-Tail with Bit Errors

Using Drop-Tail routers, we obtained Fig. 5.3 which plots the mean and standard deviation of each BE transport protocol's throughput versus the bit error rate. For bit error rates less than $2 \cdot 10^{-6}$, mean flow throughputs were distributed between 18 to 20 kbps. However at an error rate of $4 \cdot 10^{-6}$, STP demonstrates a slight throughput advantage and progresses to a 5 kbps lead over the TCP stacks at $3 \cdot 10^{-5}$. At $1 \cdot 10^{-4}$, TCP's average throughput is 750 bps and STP's is 5250 bps; this 5250 bps throughput is approximately 23% of the original per flow bandwidth. For comparison, the bit error rate of $1 \cdot 10^{-4}$ corresponds to a frame error rate of 37%; about one out of every three packets is discarded.

5.2.2 RED with Bit Errors

When using RED on the uplink routers, we obtain Fig. 5.4 which shows the same relationship again of BE transport protocol throughput and standard deviation versus bit error rate. For BER less than $2 \cdot 10^{-6}$, flows are distributed between 17.5 and kbps, and at $4 \cdot 10^{-6}$, STP again demonstrates a slight throughput advantage that broadens to nearly a 5 kbps lead over the TCP stacks at $3 \cdot 10^{-5}$. At $1 \cdot 10^{-4}$, TCP's average throughput is 500 bps, and STP's is 5250 bps, still 23% of the original per flow bandwidth.

5.3 Overall BER Study Conclusion

From comparing both Fig. 5.3 and Fig. 5.4, we note that RED reduced the mean throughputs from 19 kbps to 18.25 kbps (4%), and that all BE protocols performed similarly until the bit error rate exceeded $2 \cdot 10^{-6}$. Beyond that point, STP demonstrated a throughput

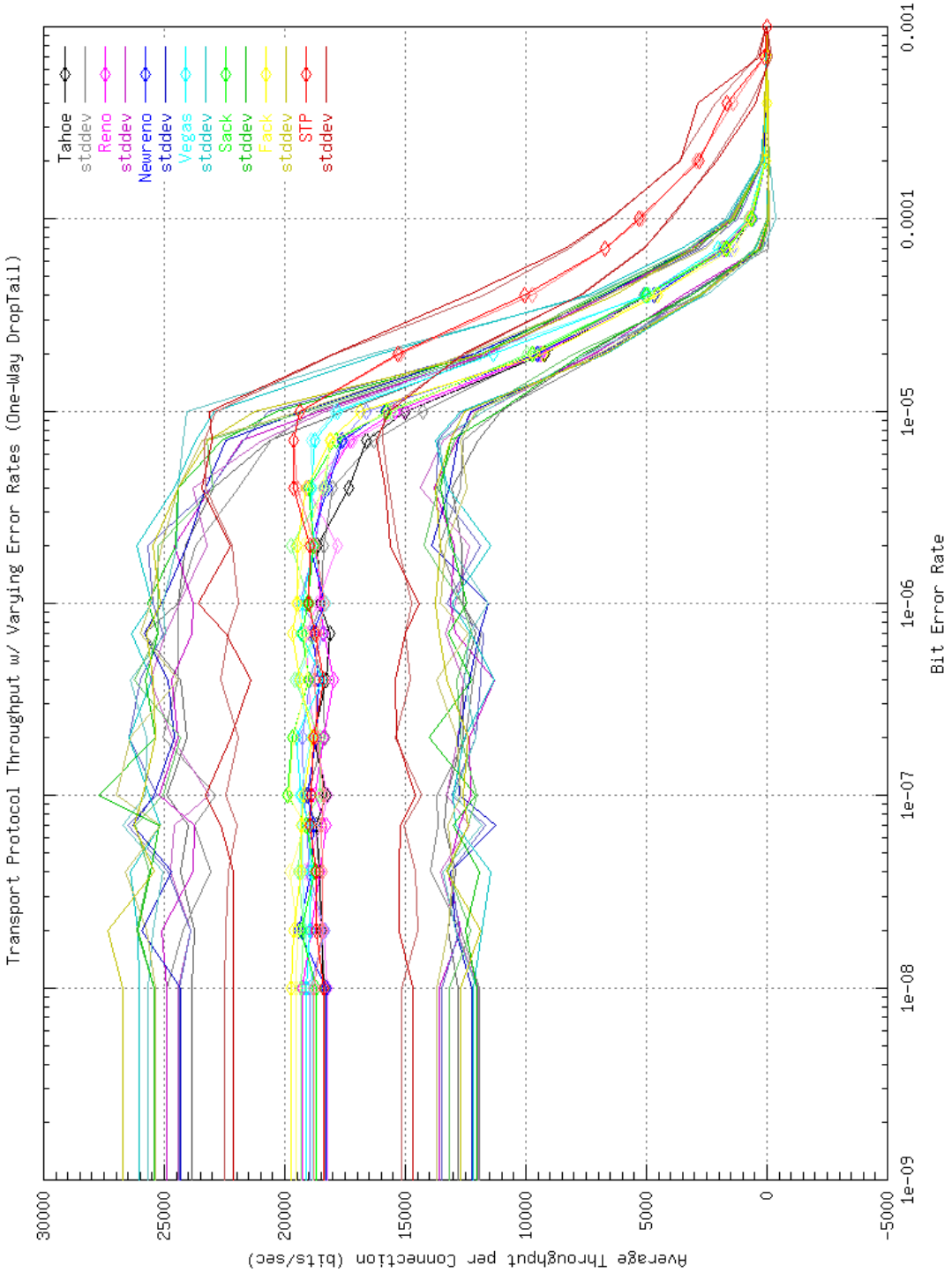

Figure 5.3: BE Throughput versus BER when using Drop-Tail simple routers.

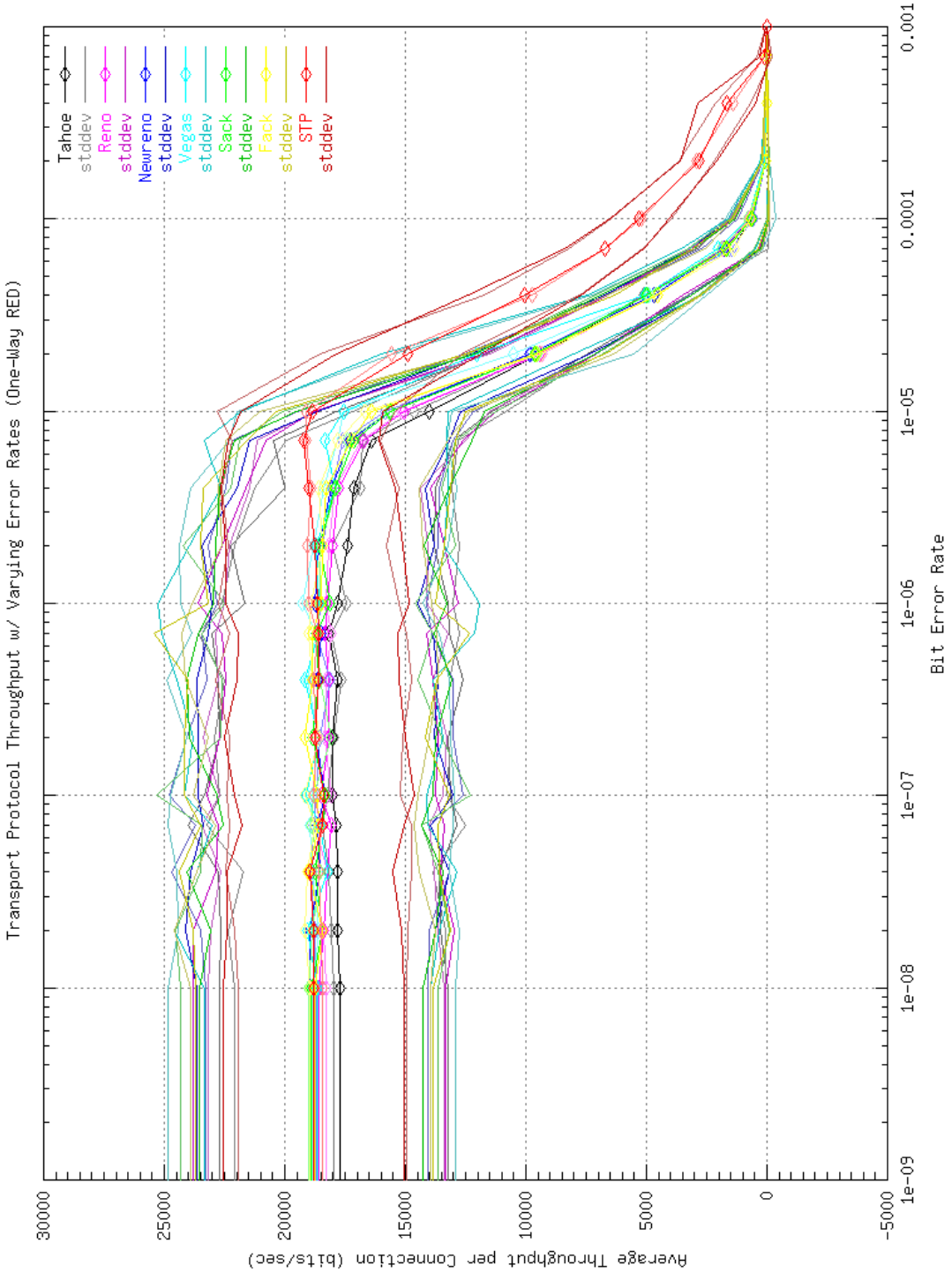

Figure 5.4: BE Throughput versus BER when using RED simple routers.

advantage for bit error rates from $4 \cdot 10^{-6}$ to $7 \cdot 10^{-4}$. When the BER was $2 \cdot 10^{-4}$, TCP throughput dropped to zero, and STP was not quenched until the BER reached $7 \cdot 10^{-4}$. From this, we conclude STP demonstrated no advantage over TCP except at the edge of usable service breakdown, where one out of three packets were being dropped.

It is perhaps surprising to note the lack of advantage displayed by variants of TCP such as SACK and FACK which are intended for applications with higher BERs and recover from multiple packet losses. With Drop-Tail routers, SACK and FACK demonstrate a $\approx$1 kbps throughput lead over other TCP implementations at bit err rates from $1 \cdot 10^{-8}$ to $2 \cdot 10^{-6}$, but lag behind Vegas TCP and STP performance at error rates greater than $7 \cdot 10^{-6}$. When using RED, SACK and FACK lose the lead after $2 \cdot 10^{-7}$. We believe this is due to the protocols' implementations falling back to New Reno TCP behavior when encountering too many packet losses[11].

Chapter 6

Step-Response to Load Variation

Throughout our queuing throughput and QoS tests in chapter 4, we assumed a constant amount of CBR traffic during each trial. However in any network, many CBR phone calls may be started or stopped at any time. Our previous tests did not evaluate how TCP and STP respond to transients in CBR load, so this chapter presents our odyssey into how the BE transport protocols react to sharp changes in available bandwidth.

6.1 Test Implementation

For these tests, we used the same topology, parameters and protocols as the test presented in section 4.1 with BE and CBR traffic. However, using knowledge gained from our tests in chapters 4 and 5, we assumed perfect links with no bit errors, reduced the selections of queuing discipline, and dynamically changed the bandwidth available to BE traffic.

Typical network links have bit error rates ranging from $1 \cdot 10^{-8}$ to $1 \cdot 10^{-7}$, and through our BER test results, we found the best-effort transport protocols behaved similarly for perfect links as well as links with error rates up to $2 \cdot 10^{-6}$. In this test, we excluded faulty network links and assumed perfect data links with no bit errors.

CBQ	CBQ+RED

Table 6.1: Router queuing disciplines used in our tests evaluating impact of CBR load transients on BE performance.

From section 4.2.3, we found that when using CBQ to protect and prioritize CBR traffic over BE, a linear increase in CBR calls will proportionately scale down the bandwidth available to BE traffic. Unlike our tests in the preceding chapters, we only tested with either the CBQ or CBQ+RED disciplines at the uplink routers (table 6.1), and this permitted us to dynamically change the bandwidth available to BE traffic by changing the number of active CBR phone calls mid-simulation.

At the start of each test, the best-effort sources proceed in a staggered-start fashion identical to that described in section 4.1.1. Then 2 minutes after the last BE sources start, 8 CBR source pairs start sending thereby reducing the available bandwidth for BE traffic by ≈50%. After the CBR sources start, the test continues for an additional 5 minutes to permit observing how BE sources settle over many periods of RTT. Fig. 6.1 and table 6.2 enumerate the tests' timing parameters.

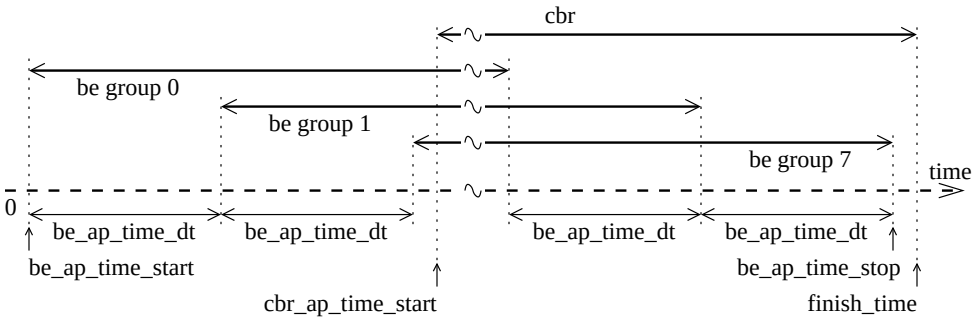

Figure 6.1: Step response test timing diagram.

be_ap_time_start	0.1 s	cbr_ap_time_start	123.6 s
be_ap_time_stop	427.1 s	cbr_ap_time_stop	428.1 s
be_ap_time_dt	0.5 s	cbr_delay	5 ms
be_delay_min	4 ms	cbr_size	92 bytes
be_delay_max	100 ms	cbr_interval	0.008 s
be_ap_groups_num	8	sat_bw	1536000 bps
finish_time	428.6 s	sat_delay	0.450 s

Table 6.2: Step response test timing parameters.

6.2 Data Collected in Step Response Tests

In each test, we monitored the application throughput over time for each BE transport protocol and queuing discipline by logging individual application-level data arrivals for each connection. Then, using post simulation processing, we found the instantaneous throughput over time across all connections, and we found the mean and standard deviation of these instantaneous throughputs across all connections in a test indicating the *fairness* of bandwidth division.

Fig. 6.2 shows for each flow the number of bytes delivered by the TCP stack to the receiving application[1] versus the time of delivery, but the figure is hard to read! Part of the problem is that we are trying to view 128 flows at the same time. For a better look, we select one flow and a smaller time interval (zoom in) in Fig. 6.3.

In the period from 88 to 100 seconds, we might be able to deduce the transmission rate is increasing due to the increasingly larger clumps of arrival bytes. This is clearly due to the action of the TCP slow start mechanism which is increasing the TCP window by one packet for each packet that is acknowledged; this results in a doubling of packets "on the line" every round trip time.

We needed a new tool to better visualize the dynamics of these flows, though possibly at some loss of packet-wise resolution of traffic. For each data arrival, we divided the number of bytes by the time difference from the time of arrival of data arriving just before it. This gave us a scale in bytes over difference in time as plotted over time in Fig. 6.4.

Due to slow-start dynamics, packets tend to arrive in bursts that are separated by RTT-like intervals. As a result, instantaneous measures show extremely high bandwidths with great variation within the bursts[2] and very low bandwidths between bursts[3].

6.2.1 New Tool: Windowed Time Averaging

Instead of finding the instantaneous throughput over time, we wanted to find a running time-average of throughput for each flow. However, we wanted the throughput to be "flex-

[1] which due to TCP buffer policies may include one or more delivered segments
[2] thanks to small random intervals between packets of a burst
[3] due to large RTT

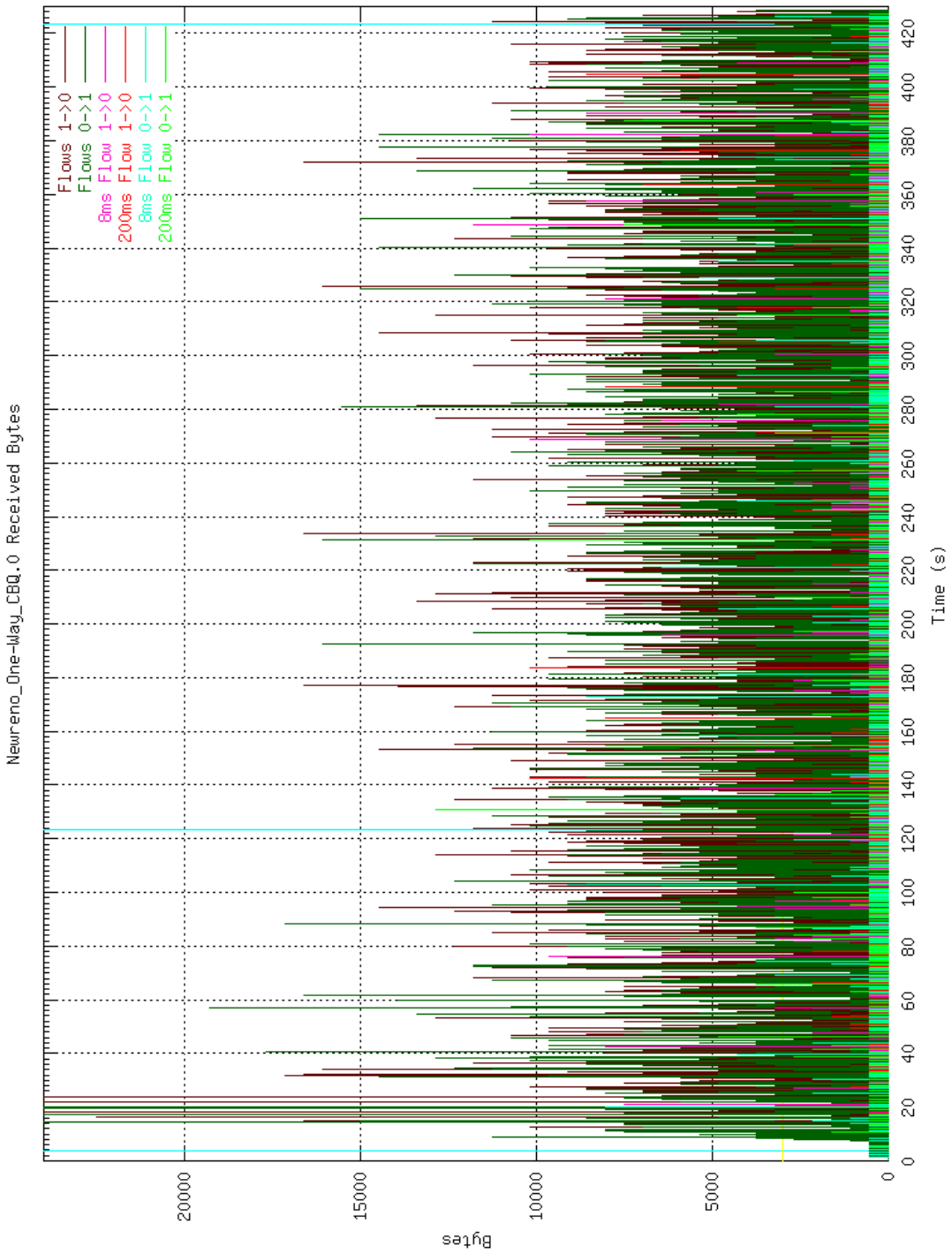

Figure 6.2: New Reno's received bytes plotted over time.

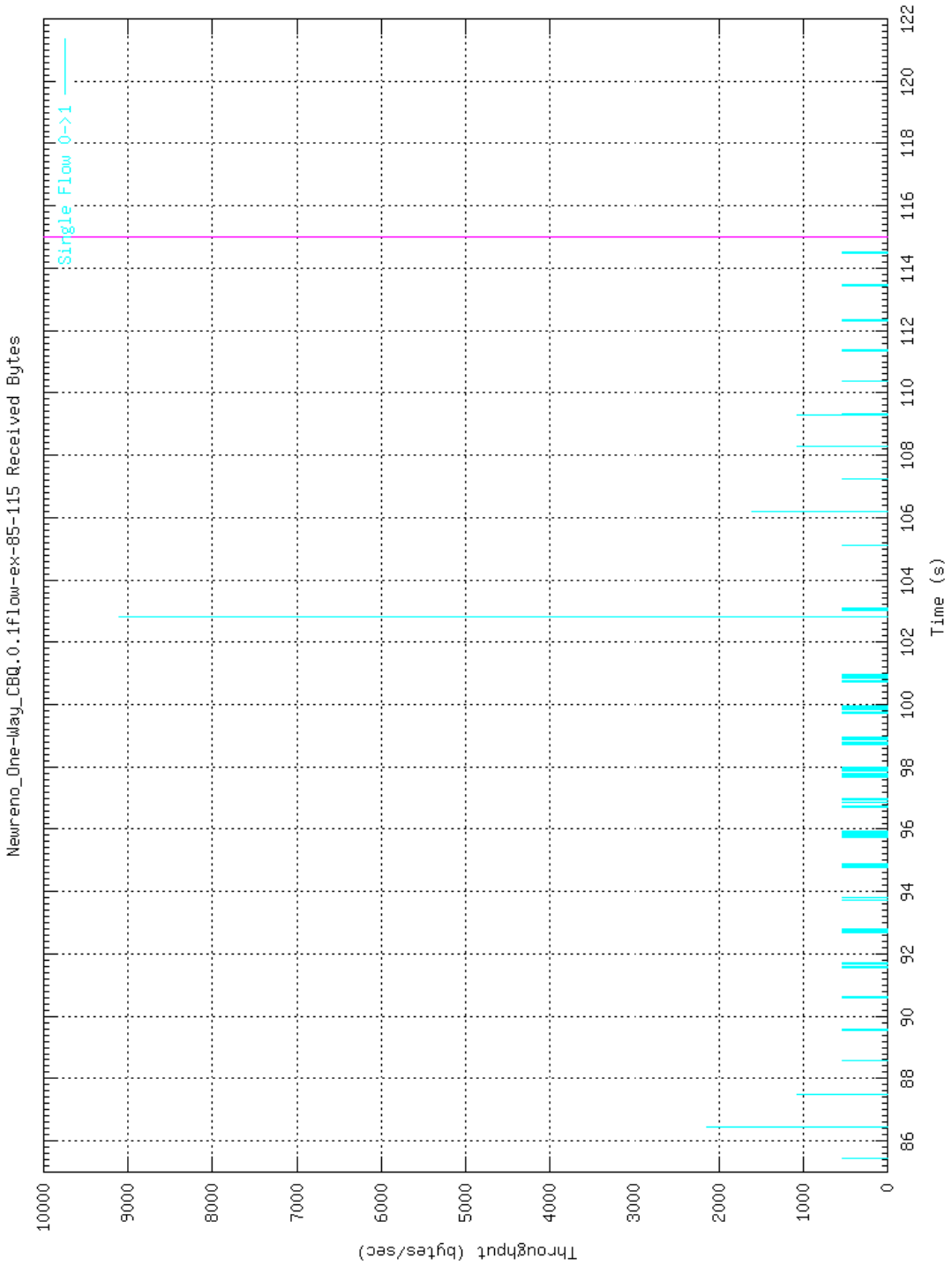

Figure 6.3: Plot of 1 New Reno TCP flow's received bytes over time from 85 to 115 seconds.

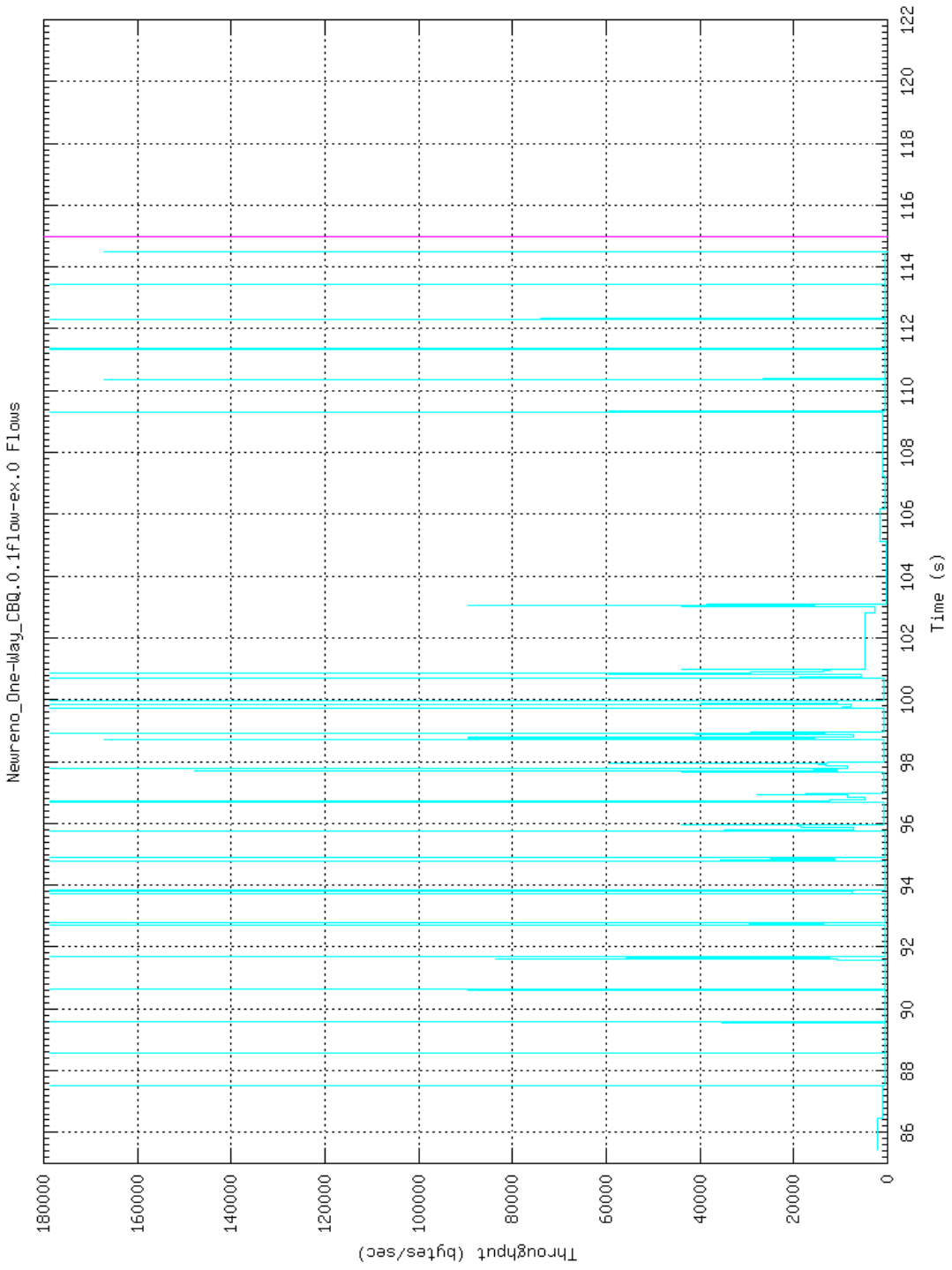

Figure 6.4: New Reno's instantaneous rate at each segment arrival.

ible" and not keep a long term memory of all received bytes. We chose to use an approach wherein instead of averaging all byte receive ns trace events, we calculated the throughput at any given time by only averaging a flow's received bytes that arrived *recently*. Here we define *recently* as any bytes that arrived within a specific amount of time, and we name this time our averaging window.

Using a 500 ms averaging window in Fig. 6.5, the flow's rising data rate becomes evident when neglecting the periodic lows. Fig. 6.6, Fig. 6.7 and Fig. 6.8 plot the same data again but using 1, 3 and 5 second time windows, respectively.

With this technique, the averaging window determines how smooth the plot will be, but at the expense of blurring sharp throughput changes as seen in Fig. 6.9. The 3 second window as shown in Fig. 6.7 offered the best view of our data in that it summarized packet arrivals without obscuring significant throughput changes.

6.2.2 New Reno Behavior

Using a 3 second averaging window, Fig. 6.10 is the first of our graphs to transparently portray the behavior of a transport protocol (New Reno) in response to a sharp bandwidth change. Here, the throughputs of all 128 flows are color coded according to table 6.3 and plotted over time. The horizontal yellow lines mark $\frac{1}{64}$ of the available bandwidth which is the ideal bandwidth each BE flow should occupy if the total available bandwidth was shared evenly. The vertical cyan colored lines mark three significant events in the test: By 3.6s, all best-effort sources are started. At 136.6s, the CBR sources are started, and at 423.6s, the TCP sources begin staggered shut down.

Color	Uplink Delays	Direction	Comment
Cyan	4 ms	Forward	shortest RTT forward flow
Magenta	4 ms	Reverse	shortest RTT reverse flow
Bright Green	100 ms	Forward	longest RTT forward flow
Bright Red	100 ms	Reverse	longest RTT reverse flow
Dark Green	between 4 & 100 ms	Forward	all other forward flows
Dark Red	between 4 & 100 ms	Reverse	all other reverse flows

Table 6.3: Color coding of step response flows.

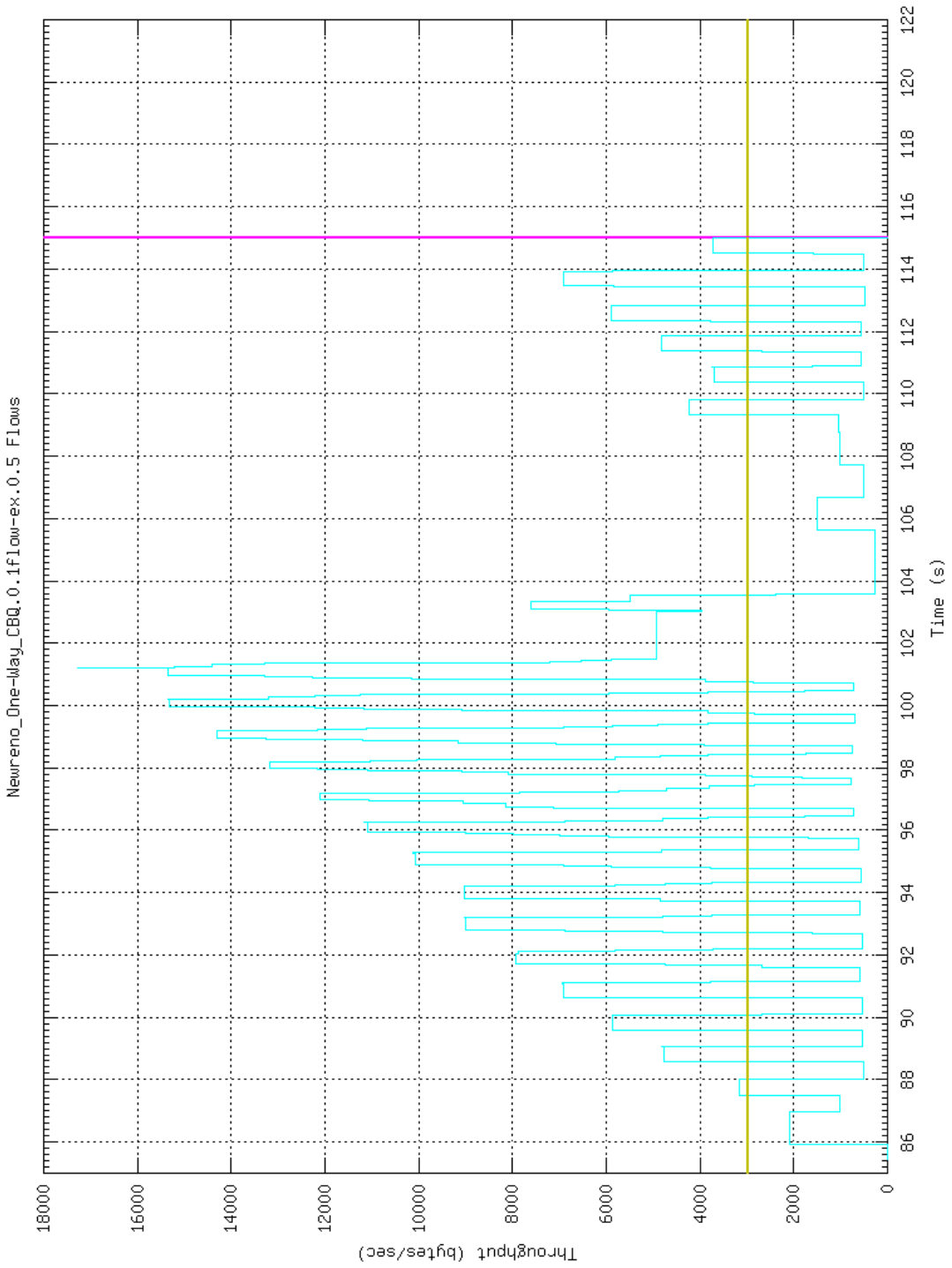

Figure 6.5: New Reno's rate averaged over 500 ms.

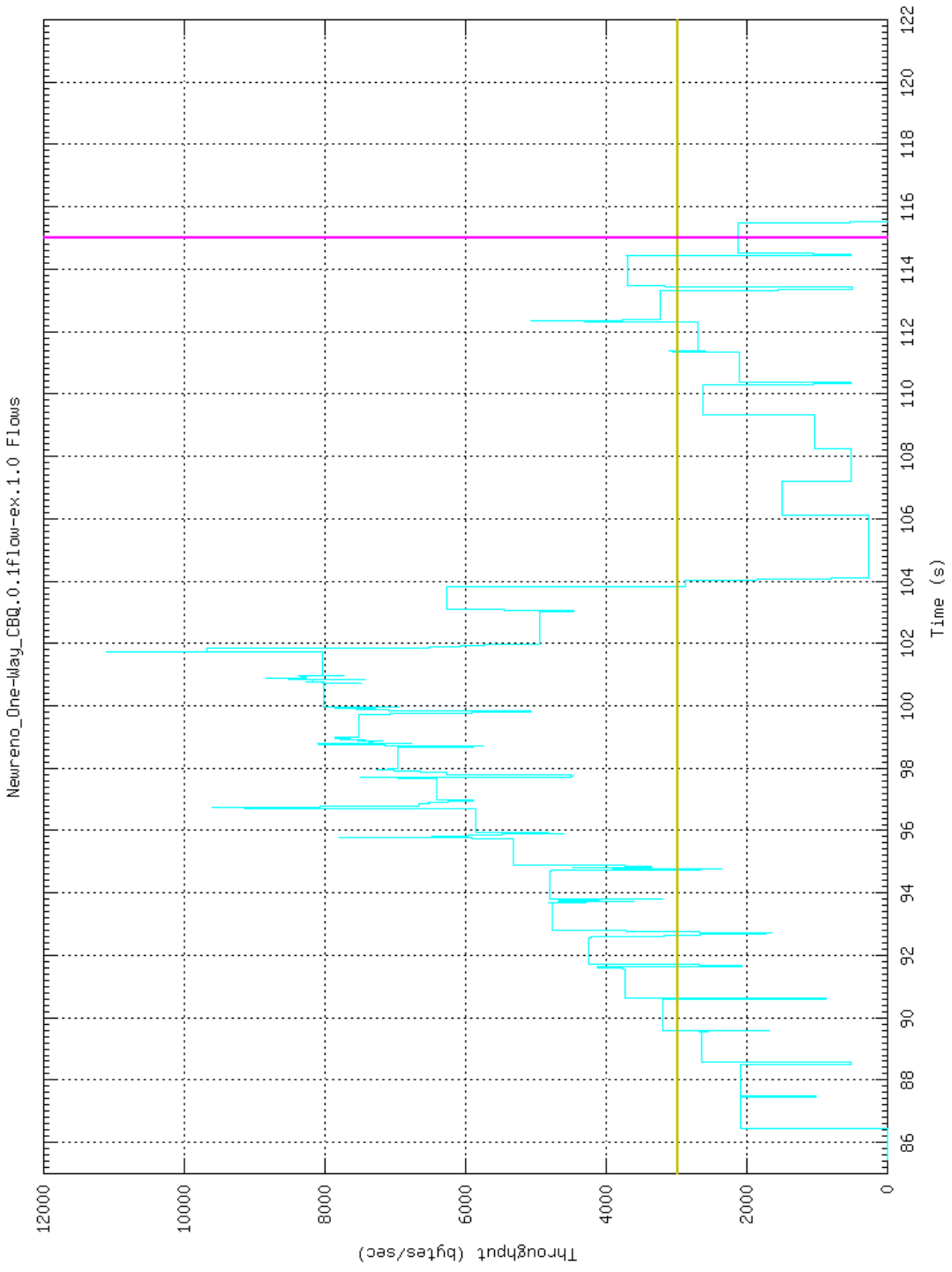

Figure 6.6: New Reno's rate averaged over 1s.

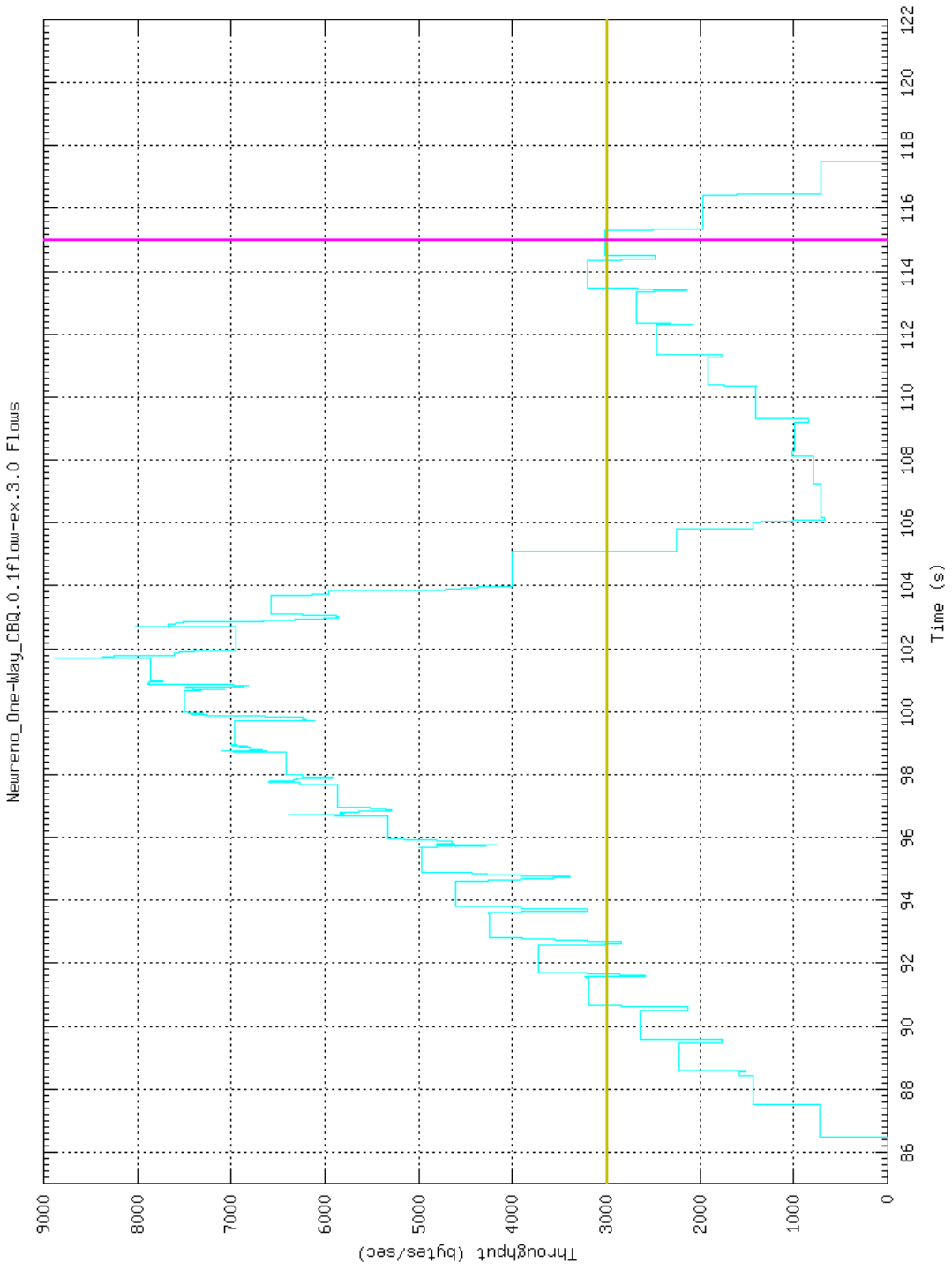

Figure 6.7: New Reno's rate averaged over 3s.

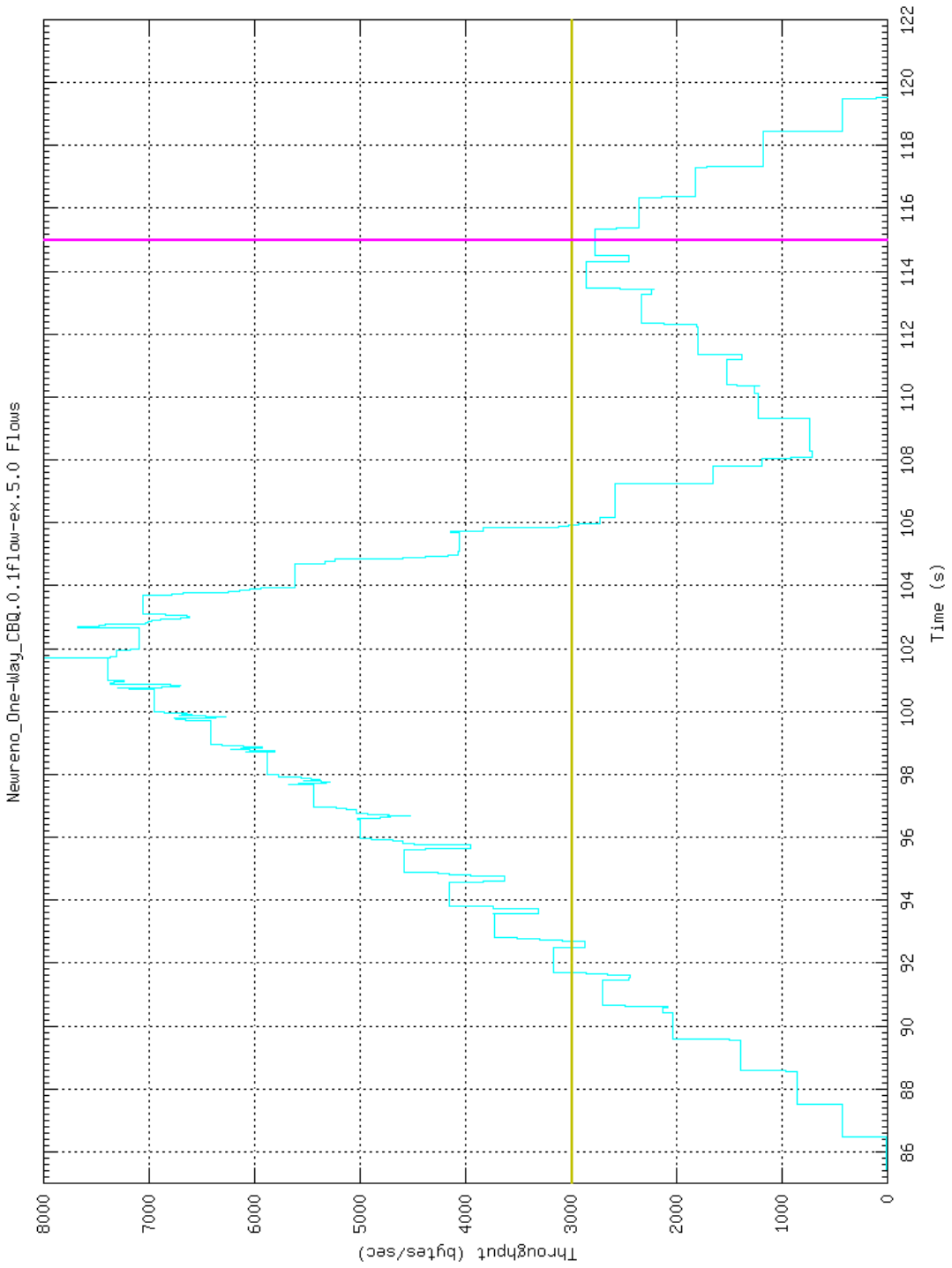

Figure 6.8: New Reno's rate averaged over 5s.

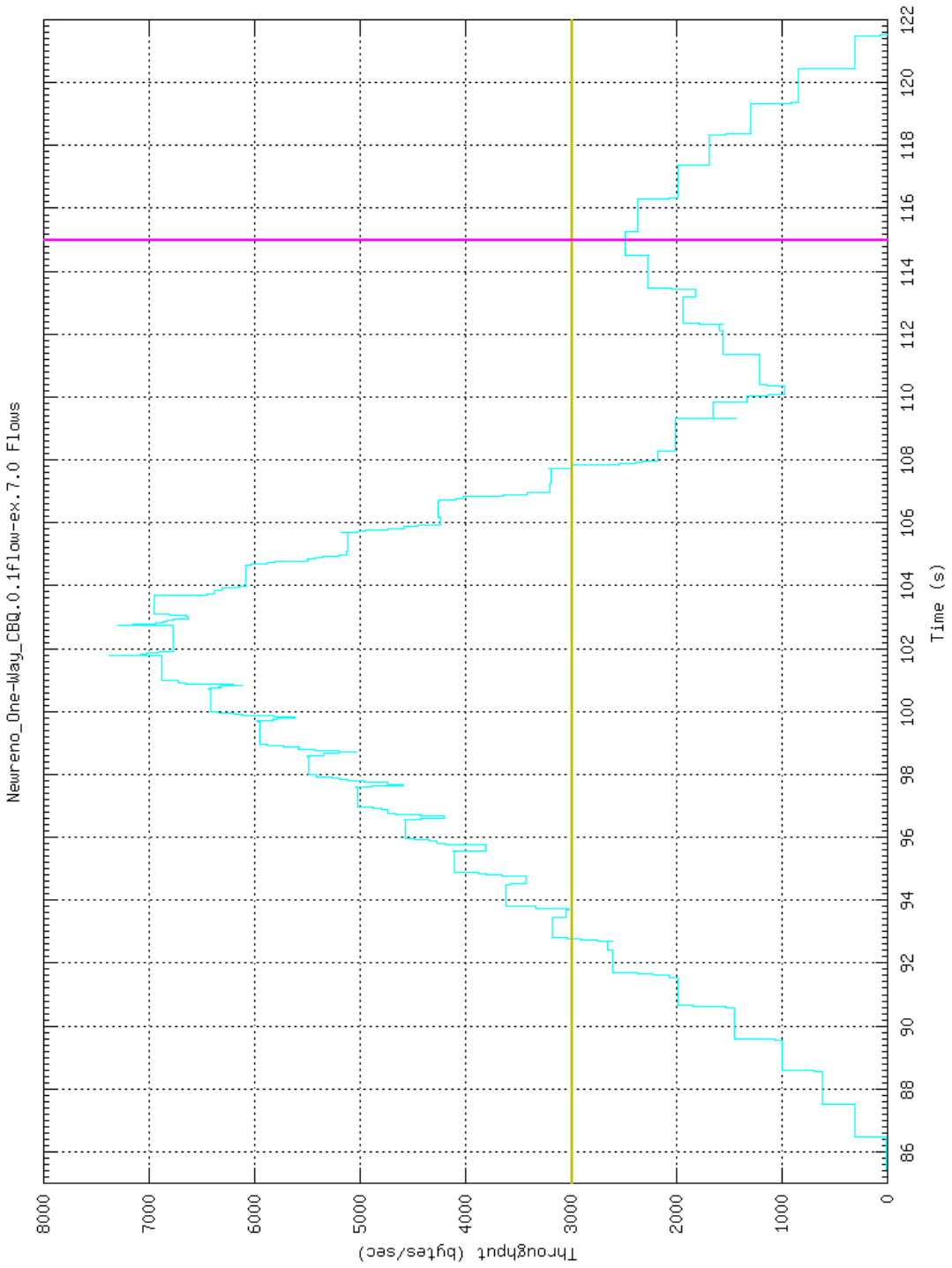

Figure 6.9: New Reno's rate averaged over 7s.

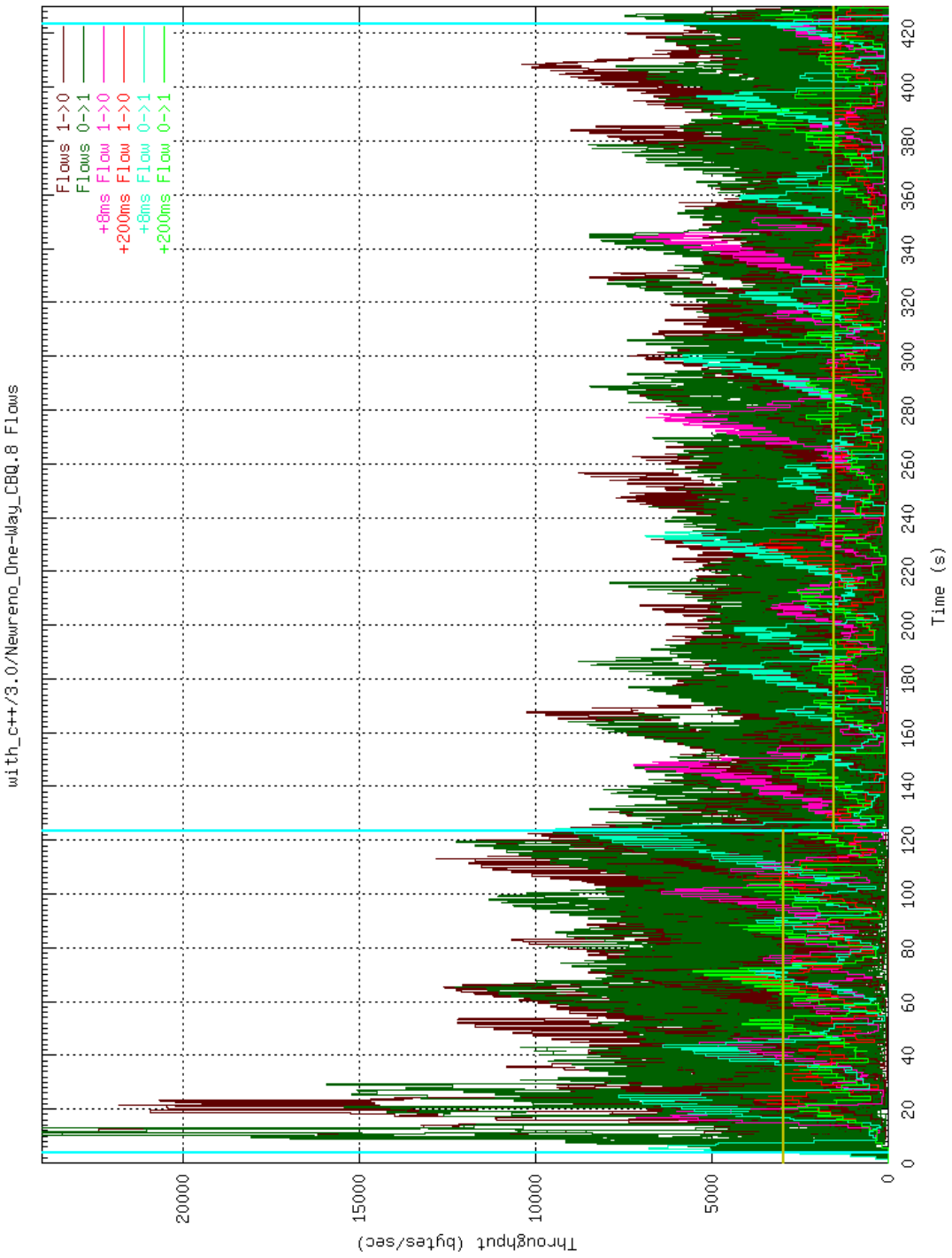

Figure 6.10: New Reno's behavior in response to starting 8 CBR sources.

In operation, TCP passes through several phases which are presented here for later identification in our results:

- "slow start" - Exponentially increase window with each RTT.
- "self pacing" - After a while, the TCP stack settles into a mode where it sends 1 data segment for every received ACK.
- "fast retransmit" - Resend missing segments when 3 duplicate ACKs are detected before the round-trip timeout.
- "fast recovery" - When 3 duplicate ACKs are detected[16],
  - reduce the congestion window by half (set ssthresh $\leftarrow$ cwnd/2),
  - retransmit the missing segment and increment the congestion window by, one MSS for each duplicate ACK (to transmit new data to replace data that was ACKed),
  - and when an accumulated ACK arrives, start the linear growth mode of Jacobson's congestion avoidance algorithm.

Fig. 6.11 examines a single New Reno TCP flow, and Fig. 6.12 zooms in on the area from 158 to 200 seconds to permit us to identify examples of these modes.

Fig. 6.12 provides an opportunity to show that this means of packet flow visualization does not obscure[4] our ability to identify key protocol behaviors as enumerated in table 6.4.

The flow in Fig. 6.12 never attained self pacing during this 7 minute test interval. Looking back at all the flows in Fig. 6.10, we see many of the flows express a similar pattern of repeated throughput oscillations.

When many sources cyclicly idle back and ramp up throughput again, due to interactions with each other, it is called "Global Synchronization." RED was not used on the up-link routers during this simulation but is designed to reduce or avoid global synchronization. When using RED for the test in Fig. 6.13, we found RED reduced New Reno TCP's

[4]We would claim our method of packet flow visualization enhances our ability to identify key protocol behaviors.

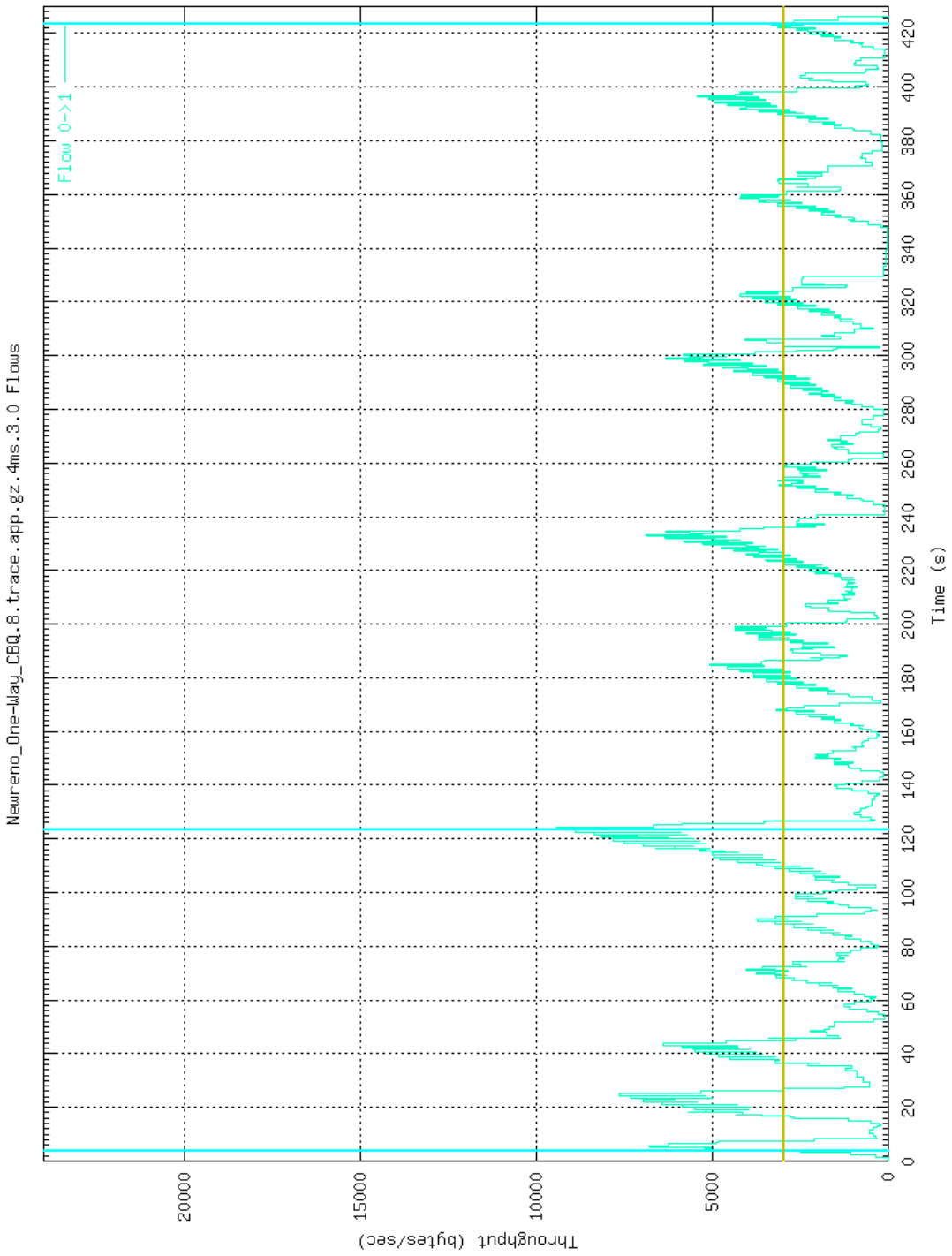

Figure 6.11: New Reno's behavior: single flows.

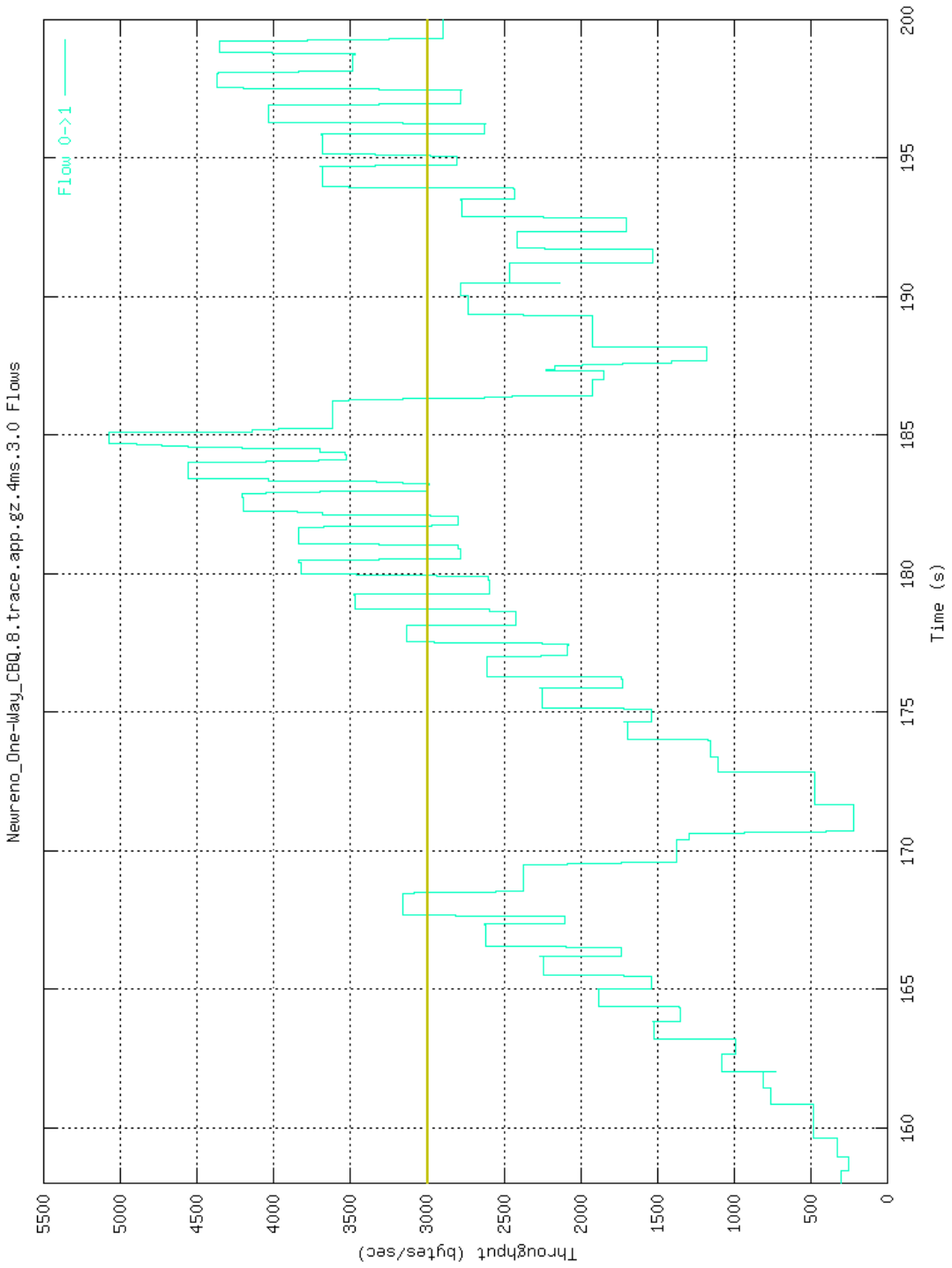

Figure 6.12: New Reno's behavior: single flow from 158 to 200 seconds.

Time Range	Description
171s to 175s	slow start with exponential increase
175s to 185s	linear growth & congestion avoidance
185s to 188s	Multiple ACKs occur and cwnd is set to ssthresh+3 (approximately halved) by Fast Recovery algorithm
188s to 190s	Fast recovery increases cwnd by one MSS for each received duplicate ACK
190s to 198s	cwnd is reset to ssthresh and linear growth congestion avoidance algorithm begins again

Table 6.4: Behavioral examples of New Reno TCP in Fig. 6.12.

throughput peaks. However, the flows still did not exhibit self pacing, and the throughput variance remains large.

6.2.3 Vegas Behavior

Fig. 6.14 and Fig. 6.15 illustrate the behavior of Vegas TCP in response to the reduction in available bandwidth. With or without RED, Vegas drastically reduced the sharp throughput peaks when compared to New Reno TCP, and it even appears to hold to constant throughputs as evidenced by the trace labeled "+8ms Flow $0 \rightarrow 1$" in Fig. 6.14 between 140 and 150 seconds.

6.2.4 STP Behavior

As seen in Fig. 6.16 and Fig. 6.17, STP also expressed lower throughput peaks than New Reno regardless of RED. This lowered variance of flows was visible in earlier aggregate flow statistics plots like the ones in chapters 4 or 5. However, STP still never achieved self pacing.

6.3 Self Pacing TCP

The ideal operating mode for TCP is one in which it sends one data segment for each received ACK, or "self pacing", and given that no other sources share the connection, a TCP source will eventually become self-paced as described in section 10.2 of [16]. We can induce

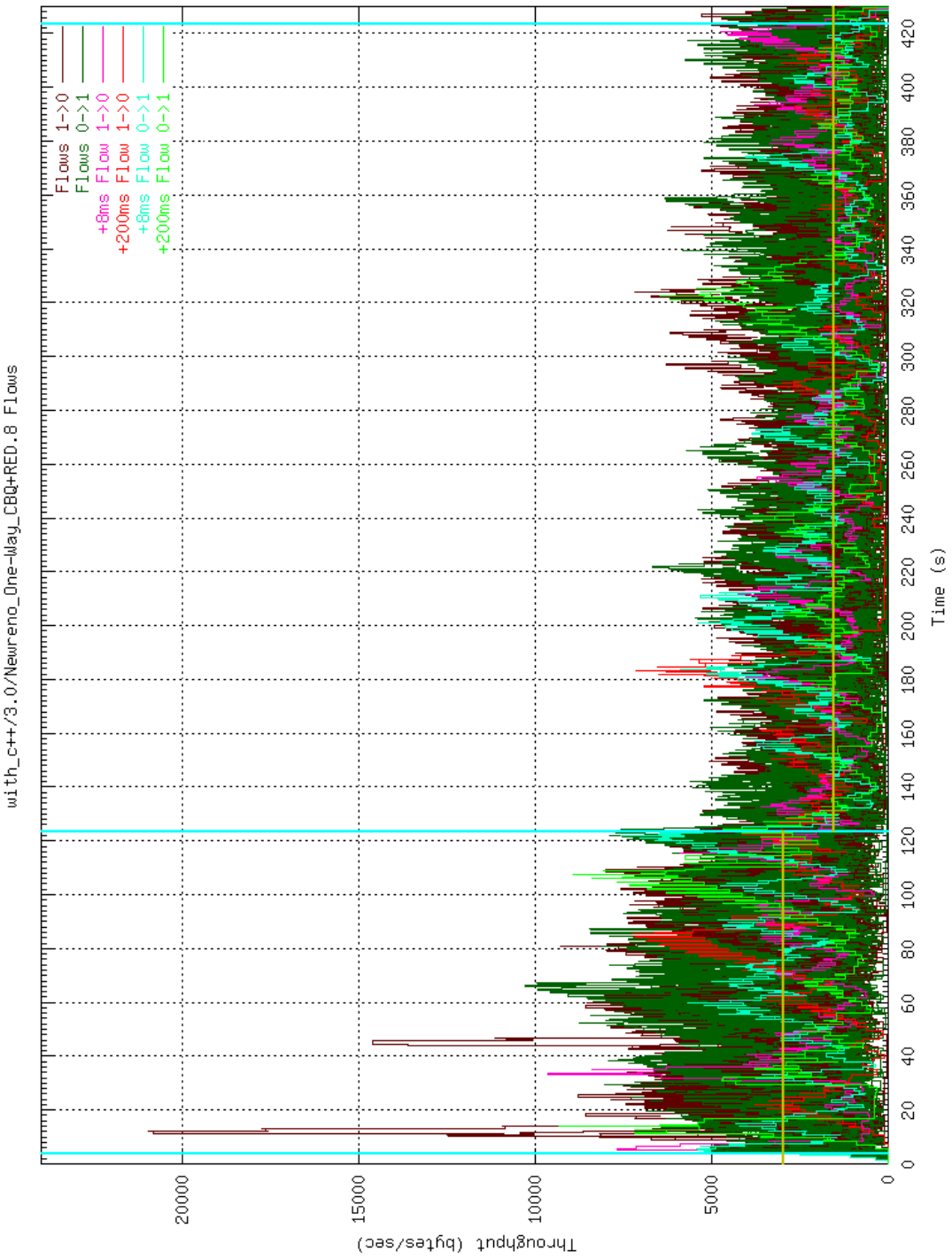

Figure 6.13: New Reno's behavior with RED.

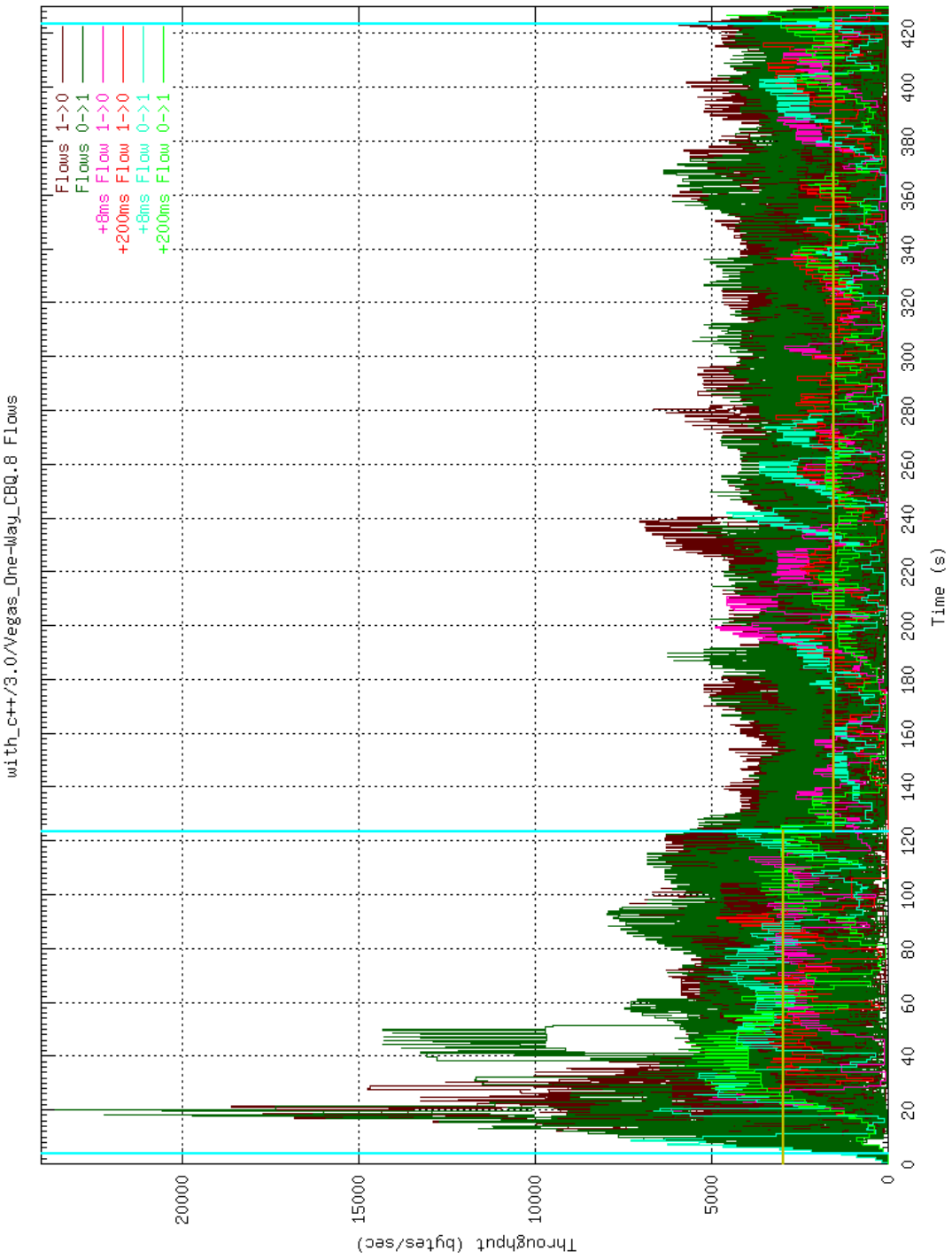

Figure 6.14: Vegas' behavior in response to starting 8 CBR sources.

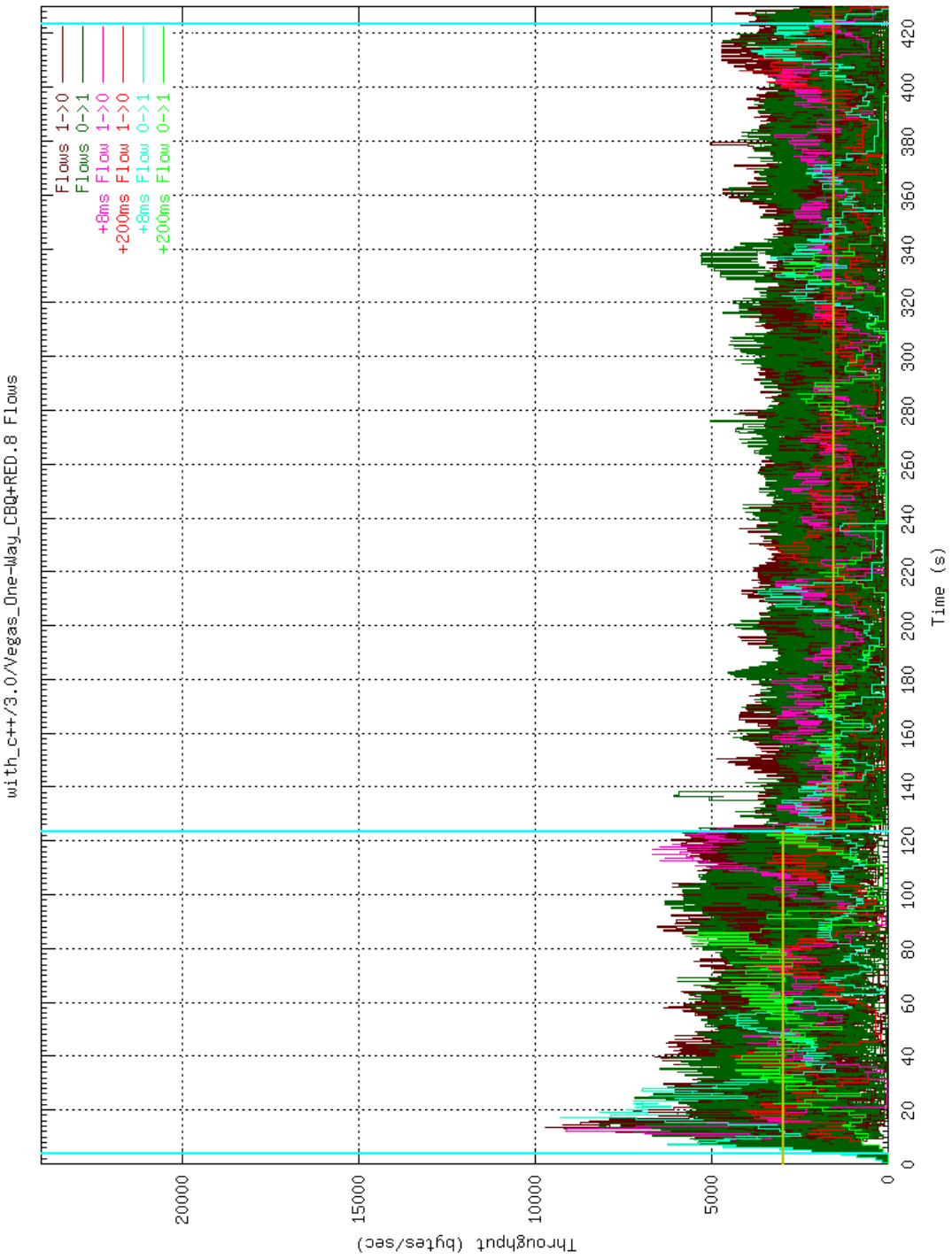

Figure 6.15: Vegas' behavior with RED.

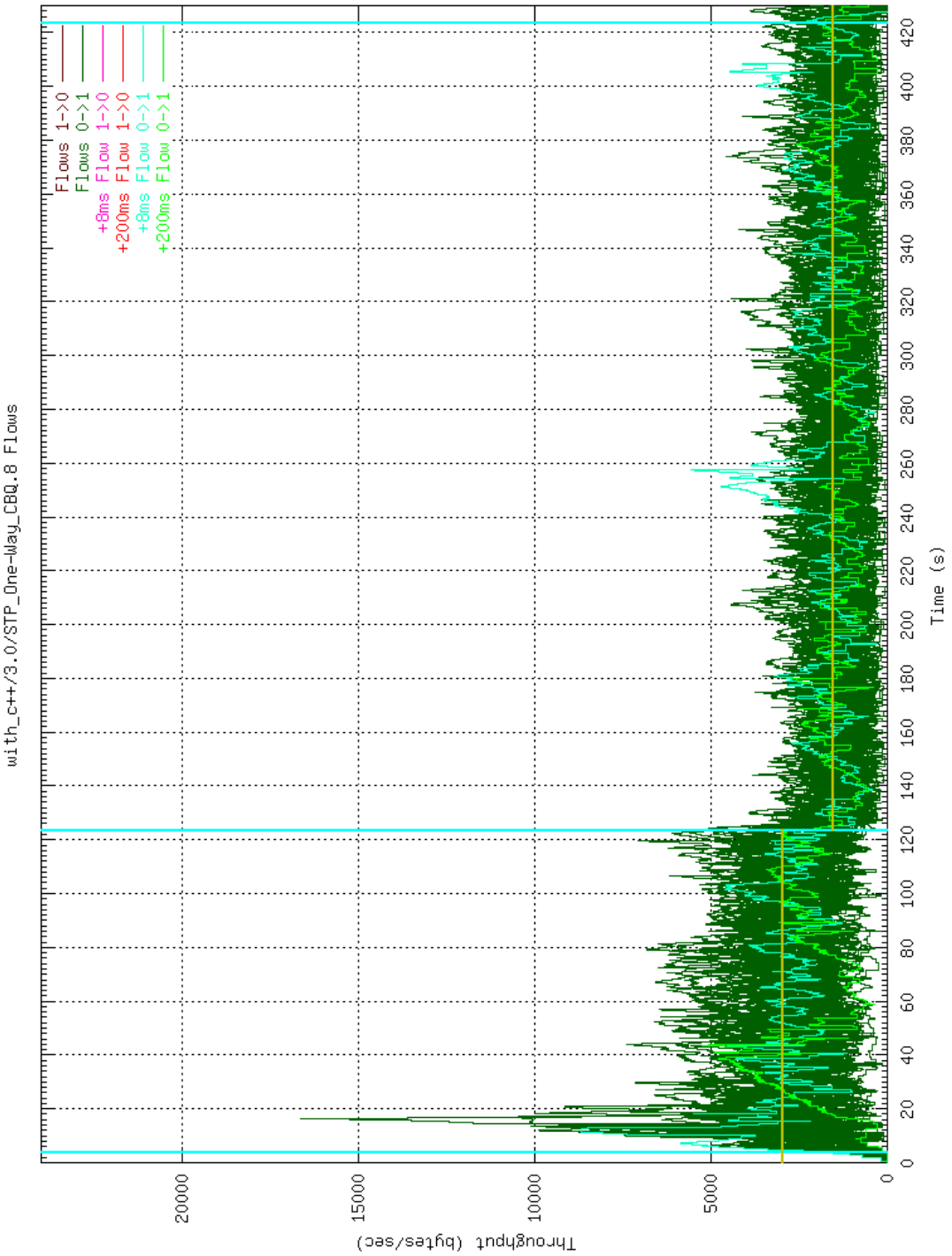

Figure 6.16: STP's behavior in response to starting 8 CBR sources. Here, only the "forward" traffic throughputs (green and blue) on the satellite link are visible, because the "reverse" traffic throughputs (red and magenta) were identical and thus, completely hidden due to the order in which the throughputs were plotted.

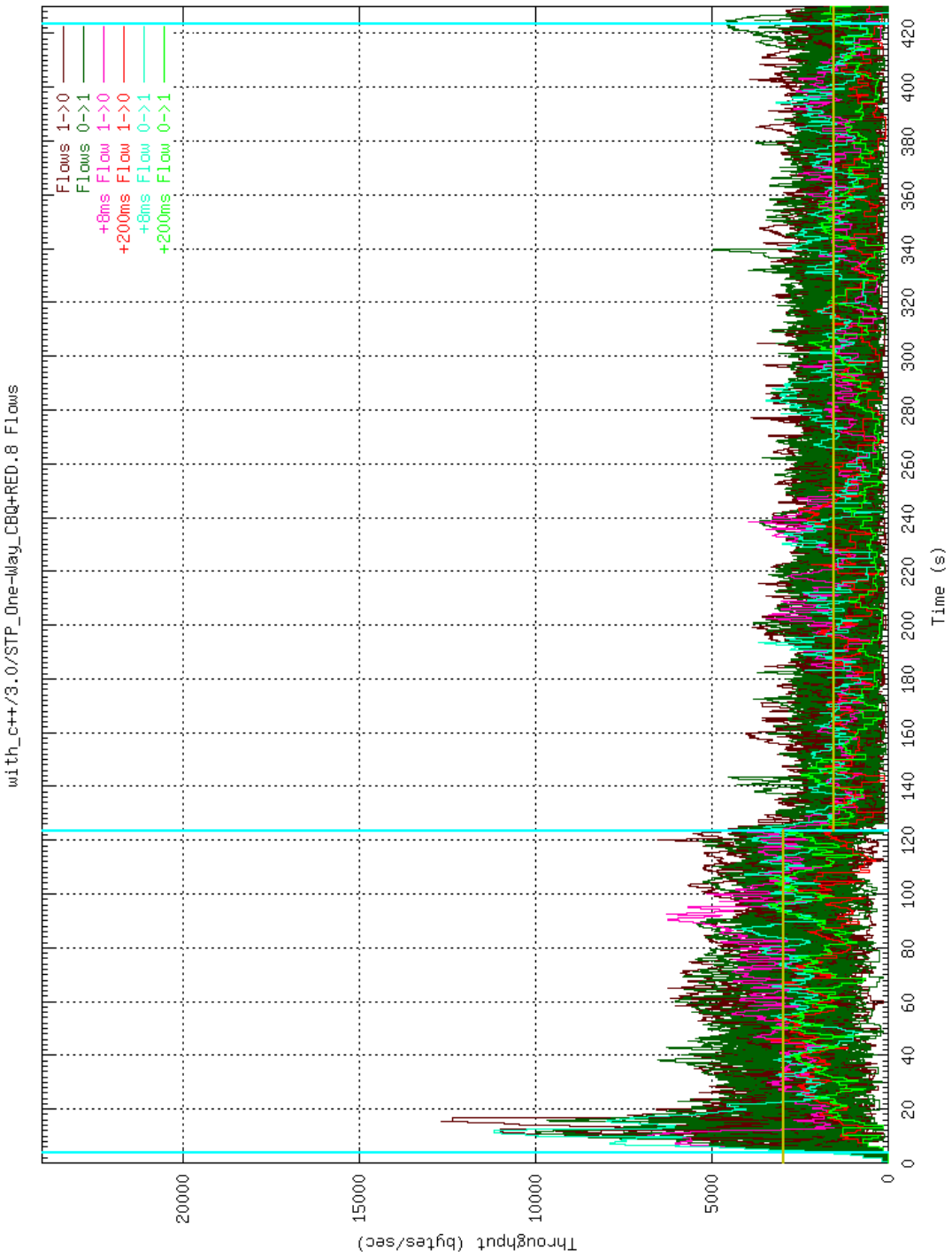

Figure 6.17: STP's behavior with RED.

self pacing by limiting the sender's window to the exact number of segments the sender may have pending acknowledgment at any time. Equation 6.1 describes how to calculate the optimal window, W_{optimal}, given we know the maximum number of segments a sender may have in transit on one side of the link, a in equation 6.2.

$$W_{\text{optimal}} = \lfloor 2 \cdot a + 1 \rfloor \tag{6.1}$$

$$a = \frac{BW_{\text{sat}} \cdot \sum_{\text{link}} delay_{\text{link}}}{8 \cdot \text{MTU}} \tag{6.2}$$

Assuming each of the 64 BE users should receive an equal share of bandwidth we can calculate a and substitute for W_{optimal} as in equations 6.3 and 6.4.

$$a = \frac{BW_{\text{sat}} \cdot (delay_{\text{sat}} + 2 \cdot delay_{\text{BE}})}{8 \cdot \text{MTU} \cdot num_{\text{BE}}} \tag{6.3}$$

$$W_{\text{optimal}} = \lfloor 2 \cdot \frac{BW_{\text{sat}} \cdot (delay_{\text{sat}} + 2 \cdot delay_{\text{BE}})}{8 \cdot \text{MTU} \cdot num_{\text{BE}}} + 1 \rfloor \tag{6.4}$$

As an experiment, we conducted the step response tests again but limited each best-effort source to their optimal window, W_{optimal}. For this network topology and the $delay_{\text{BE}}$ values from 4 to 100 ms, W_{optimal} ranged from 5 to 7 MTU segments, and now that our sources are window limited, we can determine the maximum router queue size to prevent losses should all source's transmit their full windows simultaneously. Equation 6.5 approximates the worst case by assuming the links between the routers and BE sources are infinitely fast, and the sources can send their entire window of data instantaneously.

$$\begin{aligned} burst &= \sum_{i=1}^{64} W_{\text{optimal},i} \\ &= 8 \cdot 5 + 31 \cdot 6 + 25 \cdot 7 \\ &= 401 \end{aligned} \tag{6.5}$$

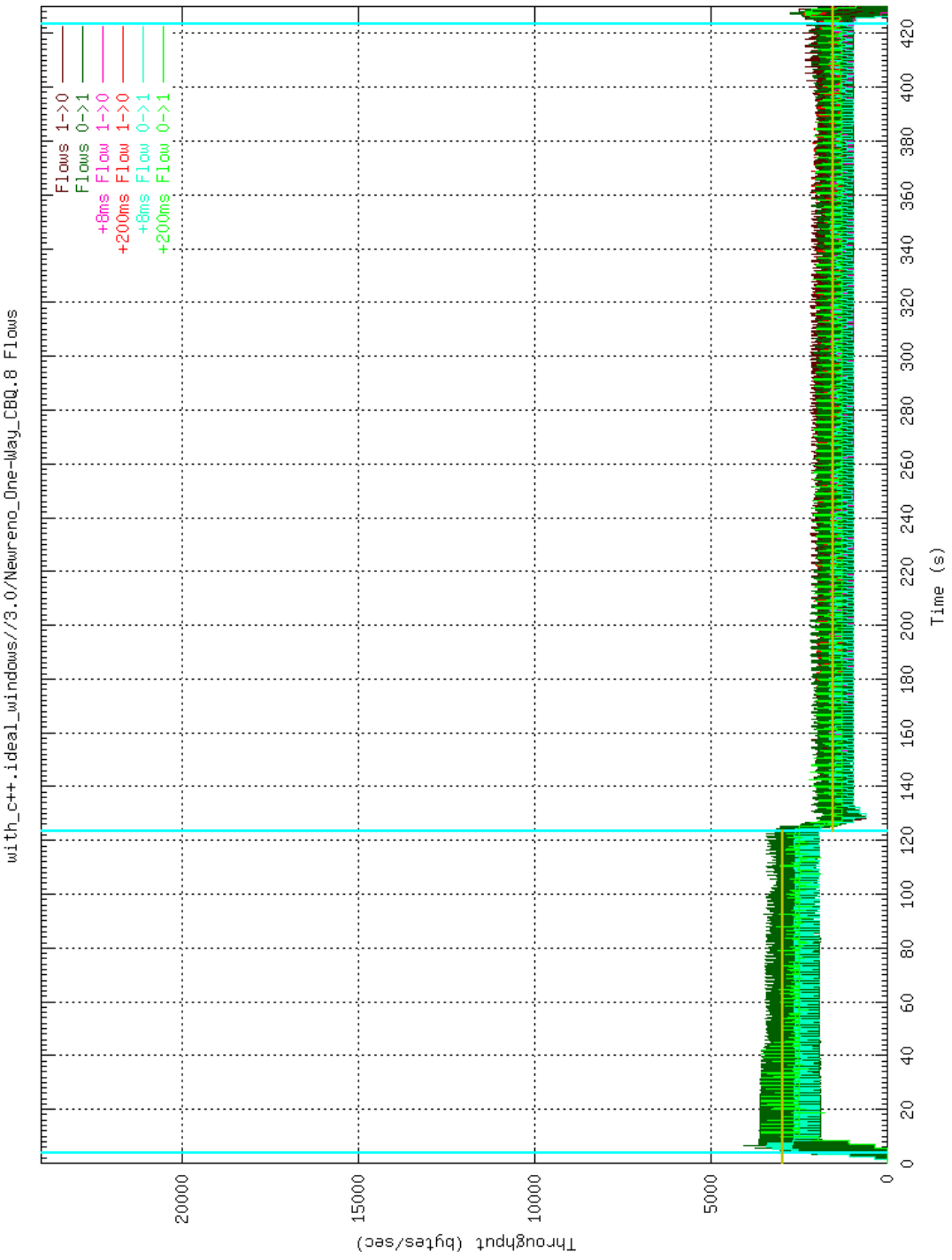

Figure 6.18: Behavior of window limited New Reno TCP in response to a change in available bandwidth.

6.3.1 Window Limited New Reno TCP

From Fig. 6.18, we see New Reno expressed clean, unchanging throughput and cleanly scaled back when CBR sources consumed half the satellite link bandwidth. We see here a kind of mode locking behavior that makes window limited rate division a useful tool even for variable bandwidth channels for which no provision is made to dynamically vary the receive-window size.

6.3.2 Window Limited Vegas TCP

Vegas also demonstrated similar uniform throughputs in Fig. 6.19 except the low delay (4 ms) BE flows are reduced to 35% of their fair bandwidth share when the link bandwidth dropped by half. This indicates that Vegas would be an inappropriate protocol for window limited fair-share operation unless some dynamic system of window sizing was utilized.

6.3.3 Window Limited STP

STP did not achieve self pacing operation under the present conditions as shown in Fig. 6.20. A design goal of STP was the reduction in the reverse channel ACK traffic so as to improve throughput for highly asymmetric connections[8]. The normal ACK activity of TCP is replaced by "unsolicited STAT" packets on lost packets, and periodic "solicited STAT" packets.

This results in delaying the flow of information from the receiver that in TCP regulates the flow of outgoing traffic through self-pacing at the sender. The result of this feedback-delay is the oscillatory behavior that is visible in the plot.

6.4 Near/Far Source Mix

With our previous tests, we only considered symmetric cases where all traffic shared the same satellite link. How would our step response test results change if half the BE flows did not use the satellite link but still competed for bandwidth in other network links?

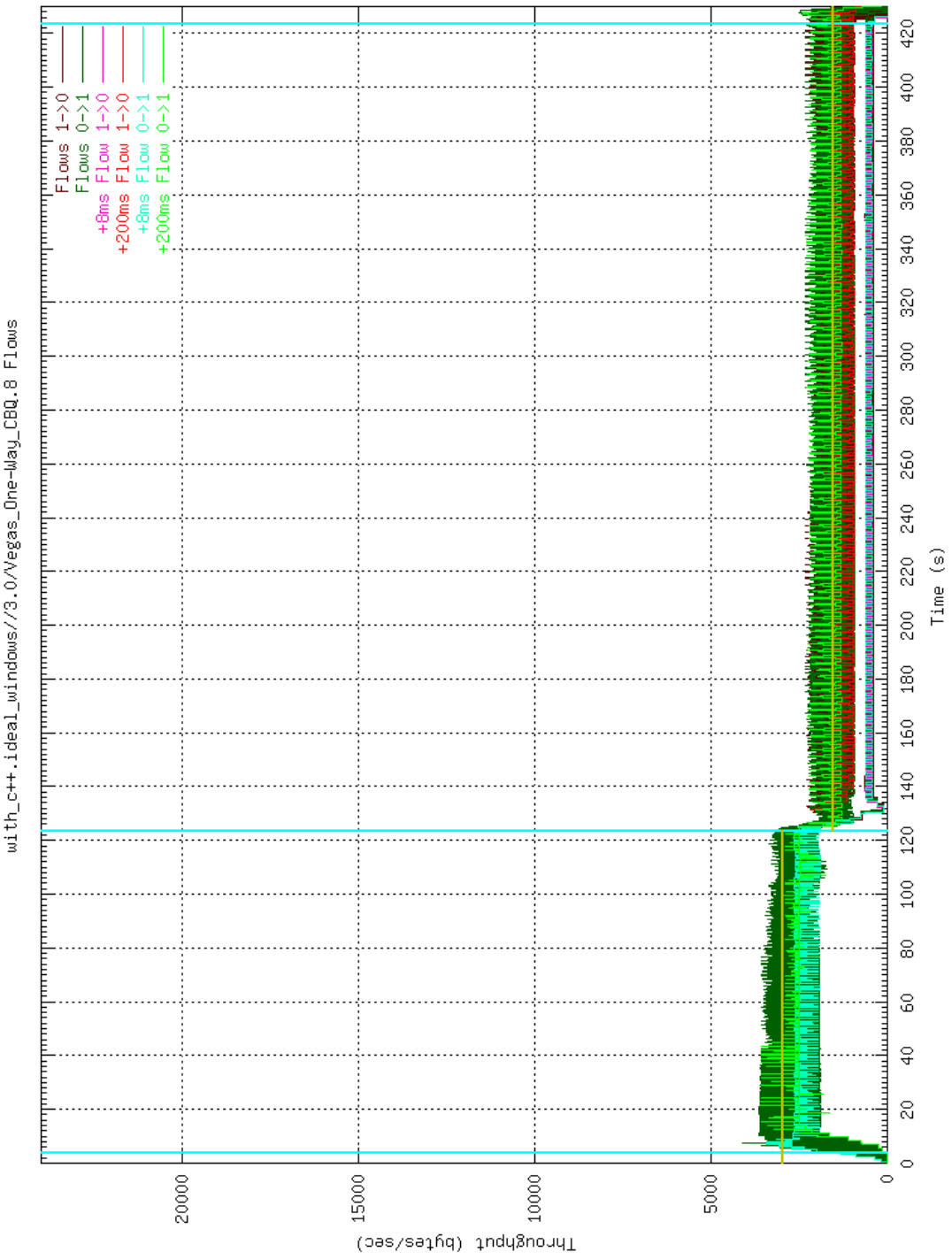

Figure 6.19: Behavior of window limited Vegas TCP in response to a change in available bandwidth.

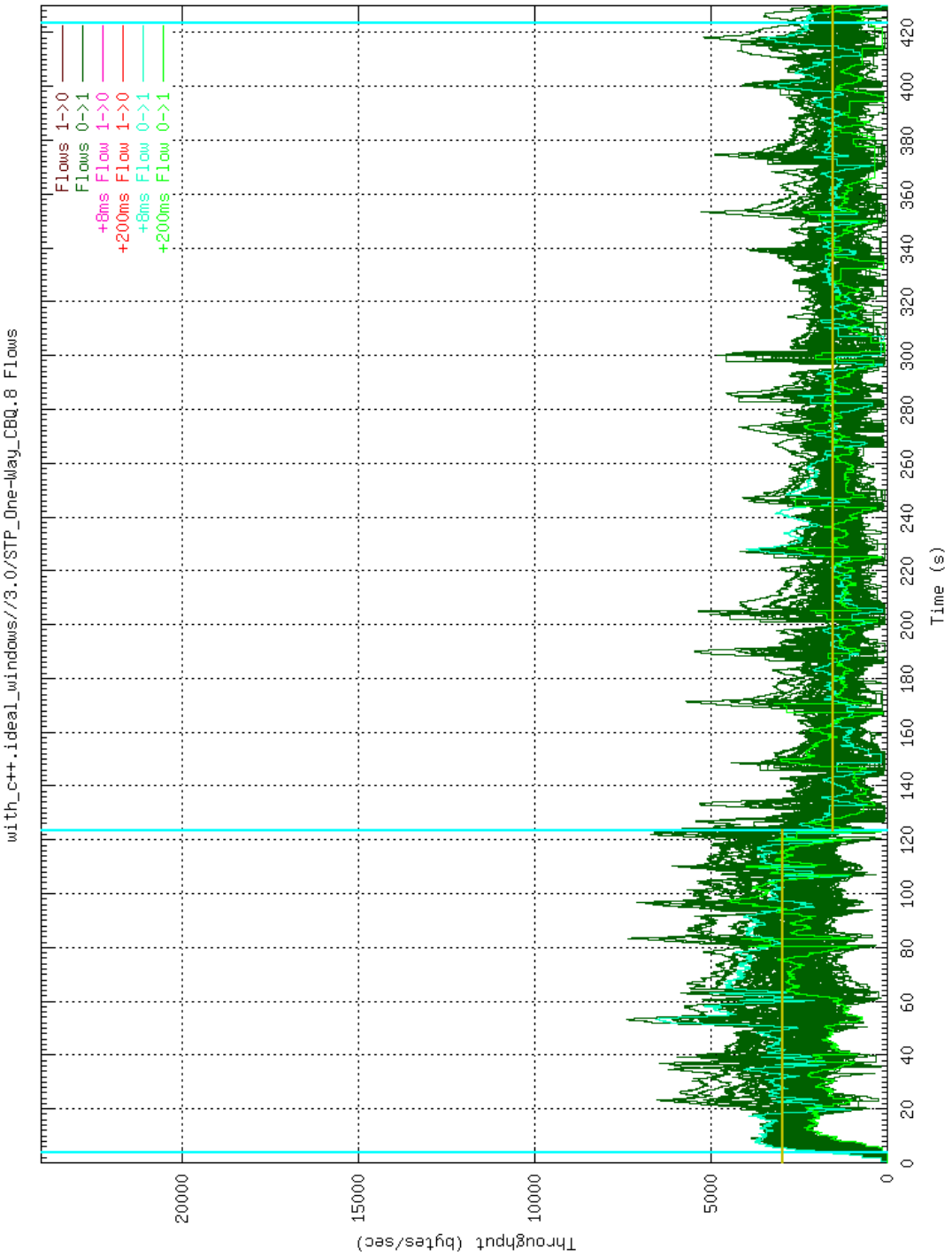

Figure 6.20: Behavior of window limited STP in response to a change in available bandwidth.

6.4.1 Near/Far Test Setup

To answer this question, we simulated the network topology in Fig. 6.21. In this configuration, all routers use either CBQ or CBQ+RED, and all links have a bandwidth of 1.536 Mbps and a 1 ms delay, except the link between nodes 0 and 1. The link between nodes 0 and 1 represents our long delay satellite link and has a 450 ms delay.

Also, we again simulated 64 BE and 8 CBR node pairs. Of the best-effort connections, half relied on the link between nodes 2 and 3 while the other half relied on the links from 0 to 3 - over the satellite link. The constant bit rate traffic only traveled over the link between nodes 2 and 3.

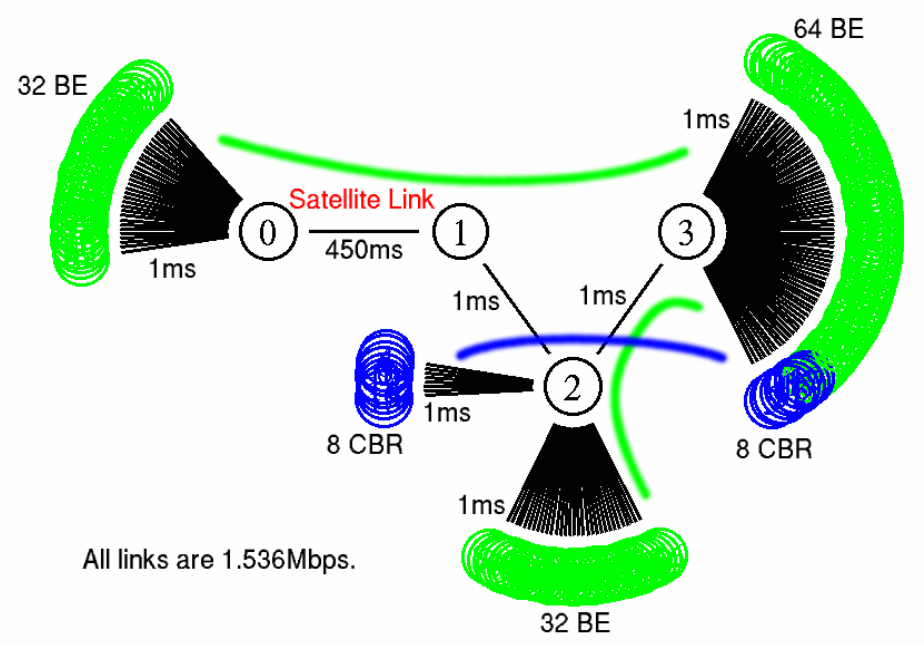

Figure 6.21: Near/far source mixing network topology with flows marked.

Test timing remained identical to the step response tests in section 6.1. In short, BE hosts were stagger-started at the beginning of the simulation, and from 3.6 to 123.6 seconds, only BE traffic traversed the links. Then from 123.6 to 432.6 seconds CBR traffic consumed half the link bandwidth between nodes 2 and 3.

6.4.2 Near/Far Collected Data & Analysis

We used the same averaging window mechanism described in section 6.2 to monitor BE application throughput over time across all flows, and in Fig. 6.22, we observe the throughputs of individual New Reno flows during the test with CBQ routers. When the CBR sources turned on, all flow throughputs dropped. In magenta, all BE flows running over the satellite link clearly have a throughput disadvantage both with and without competing CBR traffic. The difference in bandwidth available to the sources nearest to their destinations derives from the greater aggressiveness that a short RTT allows a TCP flow. With a short (6 ms) RTT, the slow start and linear congestion avoidance phases can probe and capture bandwidth between congestive events much more quickly than the sources farther away (908 ms RTT). Also we find in Fig. 6.23 that adding RED to the routers, only slightly reduced the throughput across all flows but still permitted an unfair division of bandwidth.

From Fig. 6.24, Fig. 6.25, Fig. 6.26 and Fig. 6.27, we see Vegas and STP successively demonstrated more fair bandwidth sharing than New Reno. As a check, Fig. 6.28 and Fig. 6.29 show that TCP FACK also performed similarly to New Reno.

6.5 Step-Response Conclusions

In the experiments presented in this chapter, we explored how various BE protocols respond to changes in available bandwidth, how to obtain better bandwidth sharing through inducing self-pacing, and how long-delay, satellite traffic is affected by shorter RTT traffic elsewhere in the network. Also in the course of these tests, we developed a new strategy for depicting the average BE application throughput over time.

In examining how the BE protocols respond to bandwidth changes, we were able to visually identify textbook behavioral examples (section 6.2.2), and we also noted how New Reno TCP, Vegas TCP, and STP never achieved self pacing. Instead we found the protocols were in a continual state of global synchronization, which, RED was able to reduce but not eliminate.

We also experimented with inducing self-pacing by limiting the BE protocols' window sizes. This technique successfully suppressed throughput variations for New Reno and Vegas

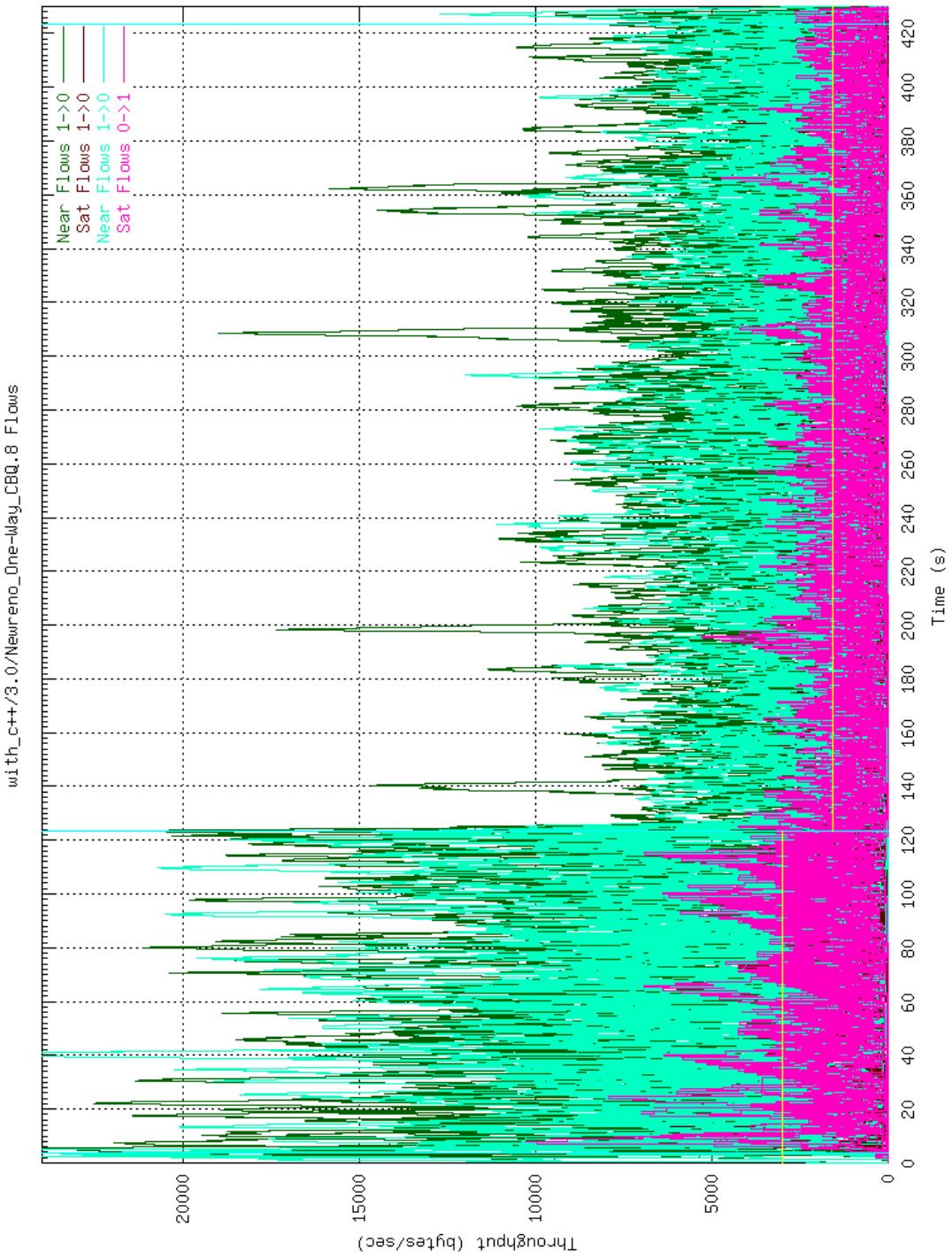

Figure 6.22: New Reno's throughput over time in the near/far test.

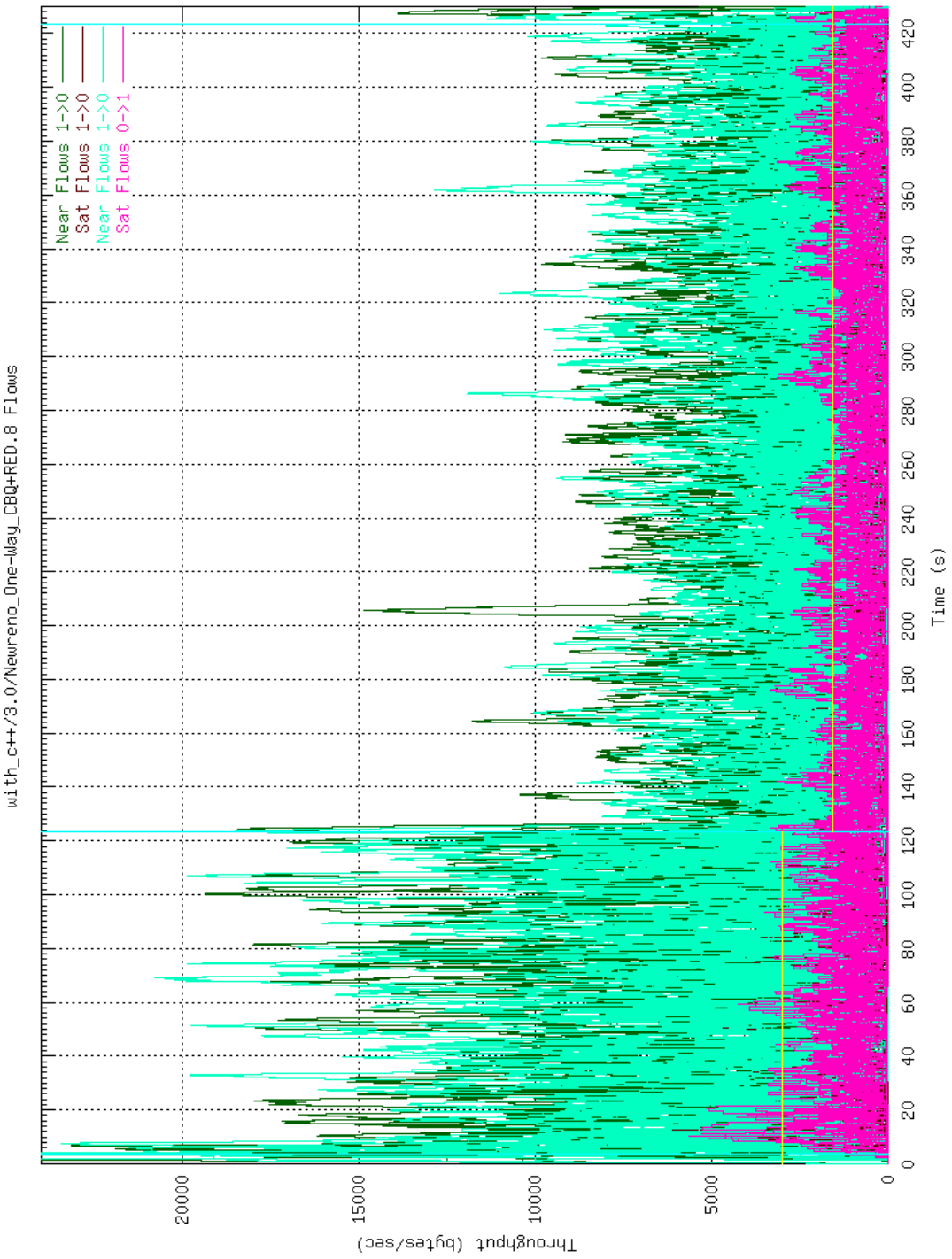

Figure 6.23: New Reno's throughput over time with RED in the near/far test.

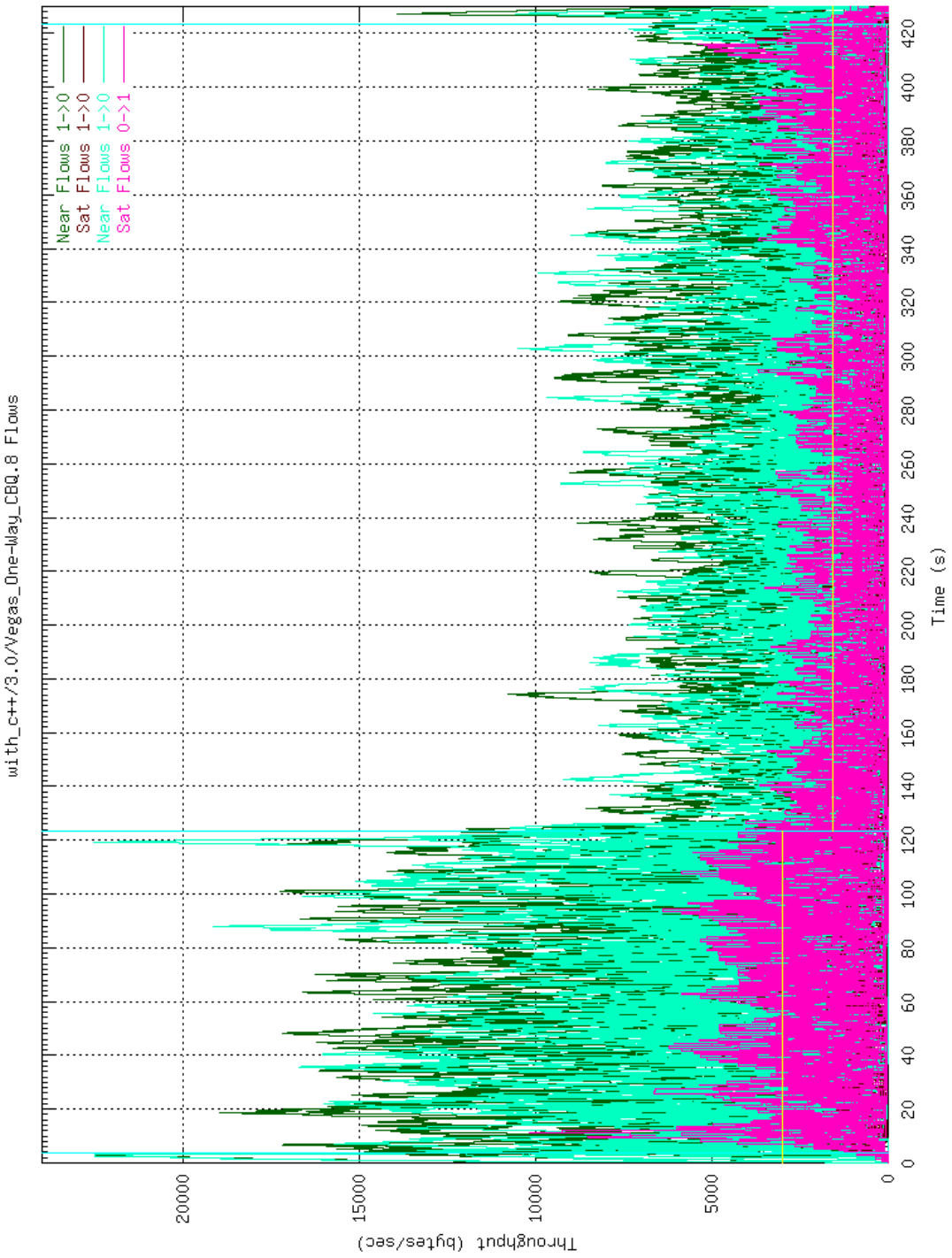

Figure 6.24: Vegas TCP's throughput over time in the near/far test.

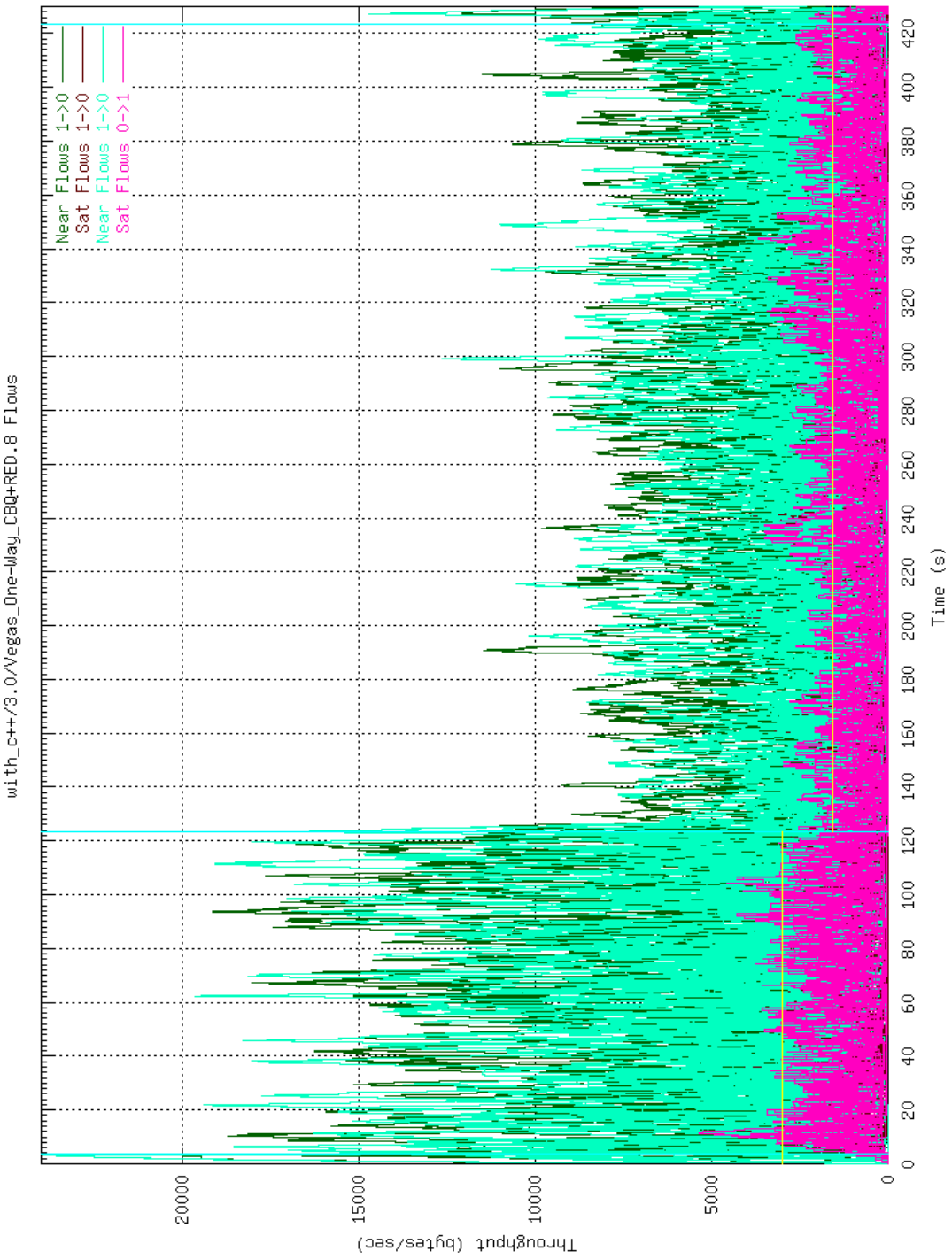

Figure 6.25: Vegas TCP's throughput over time with RED in the near/far test.

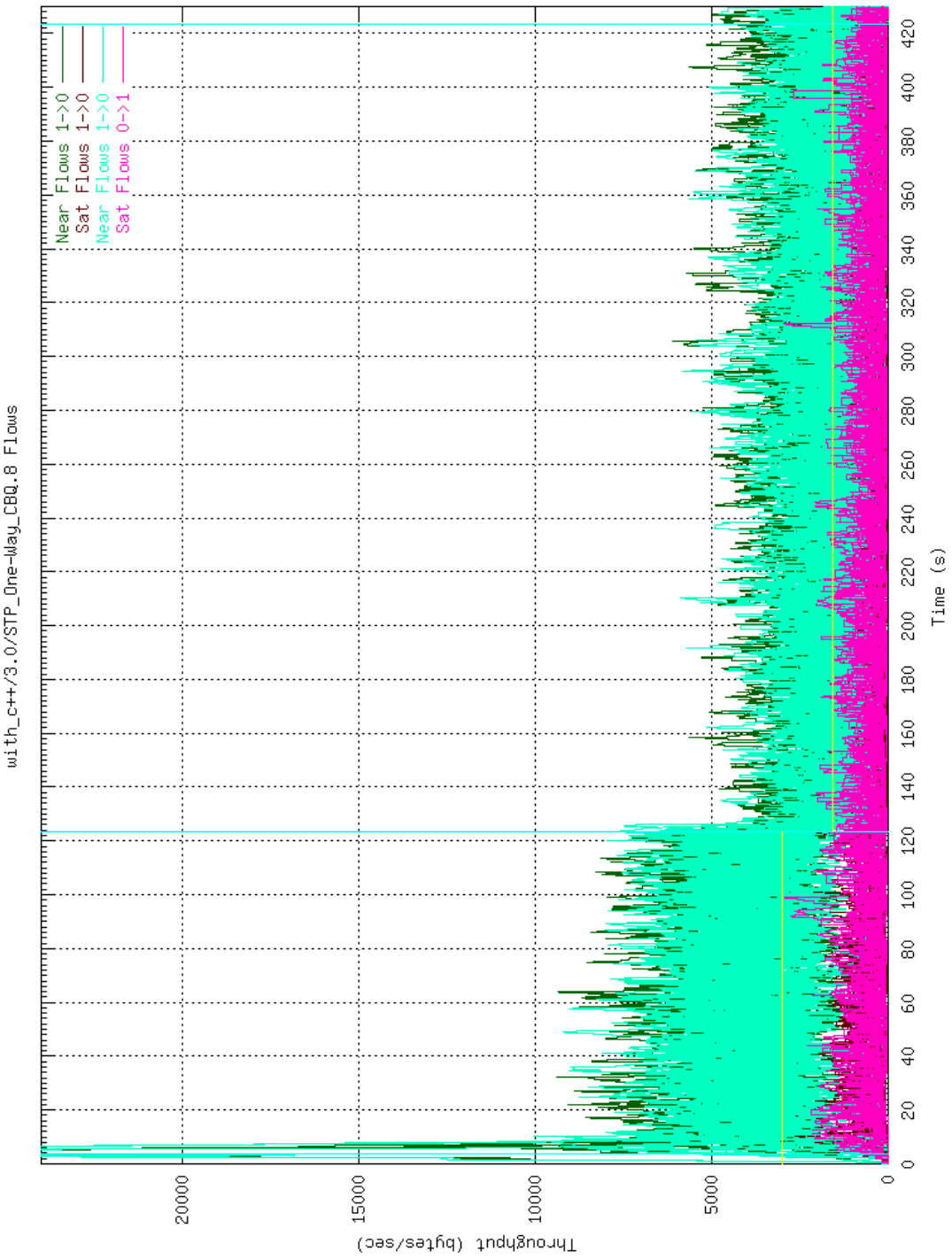

Figure 6.26: STP's throughput over time in the near/far test.

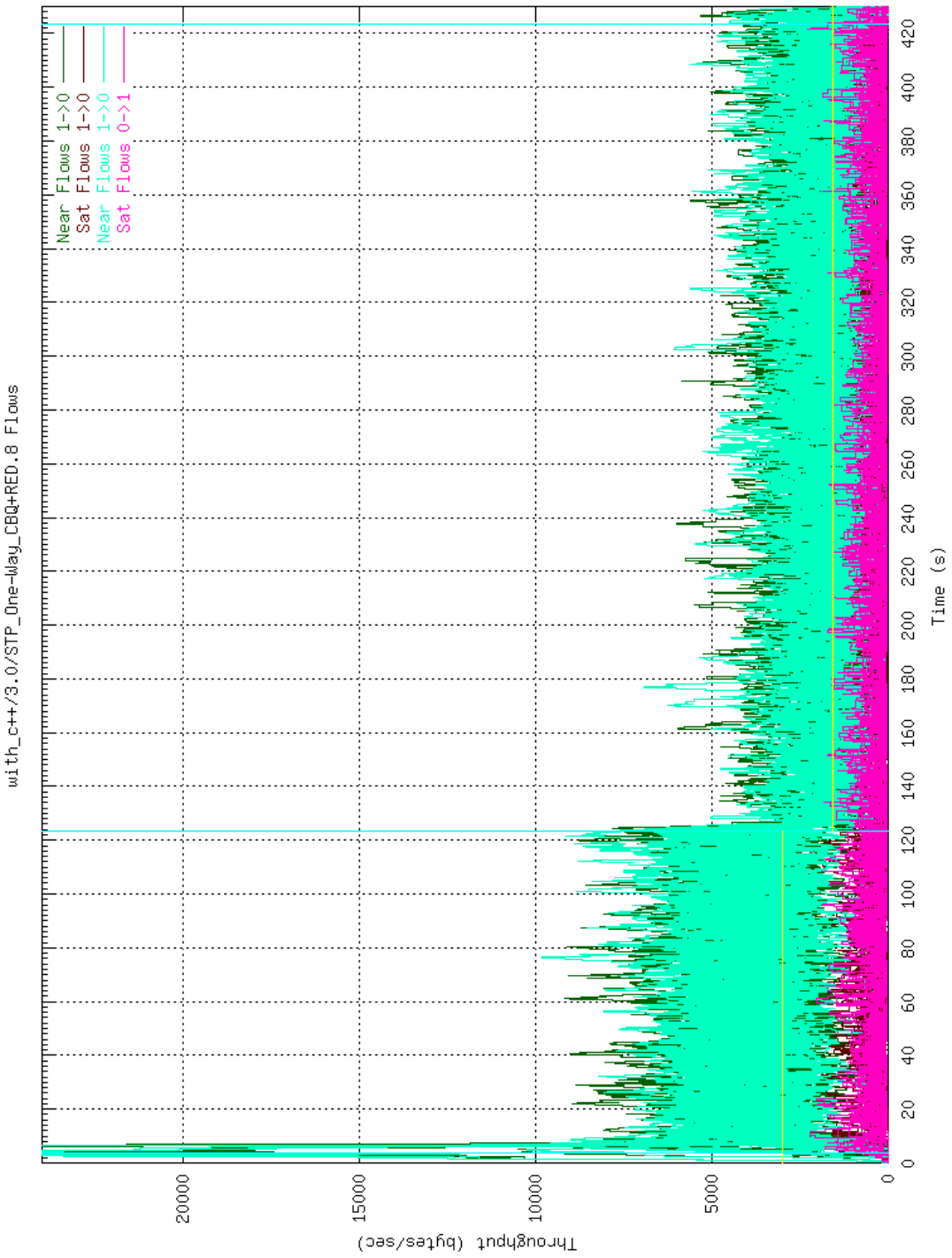

Figure 6.27: STP's throughput over time with RED in the near/far test.

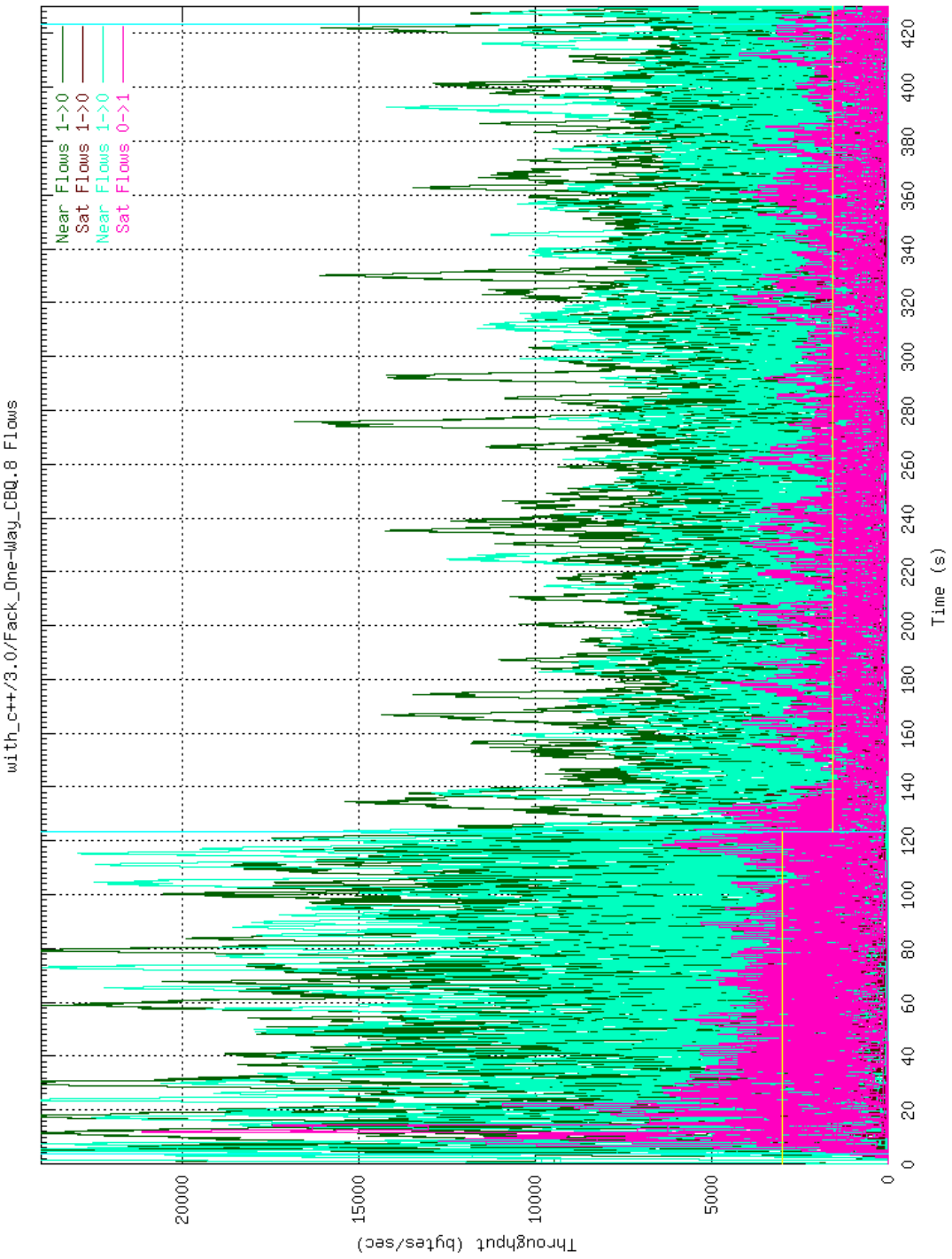

Figure 6.28: FACK TCP's throughput over time in the near/far test.

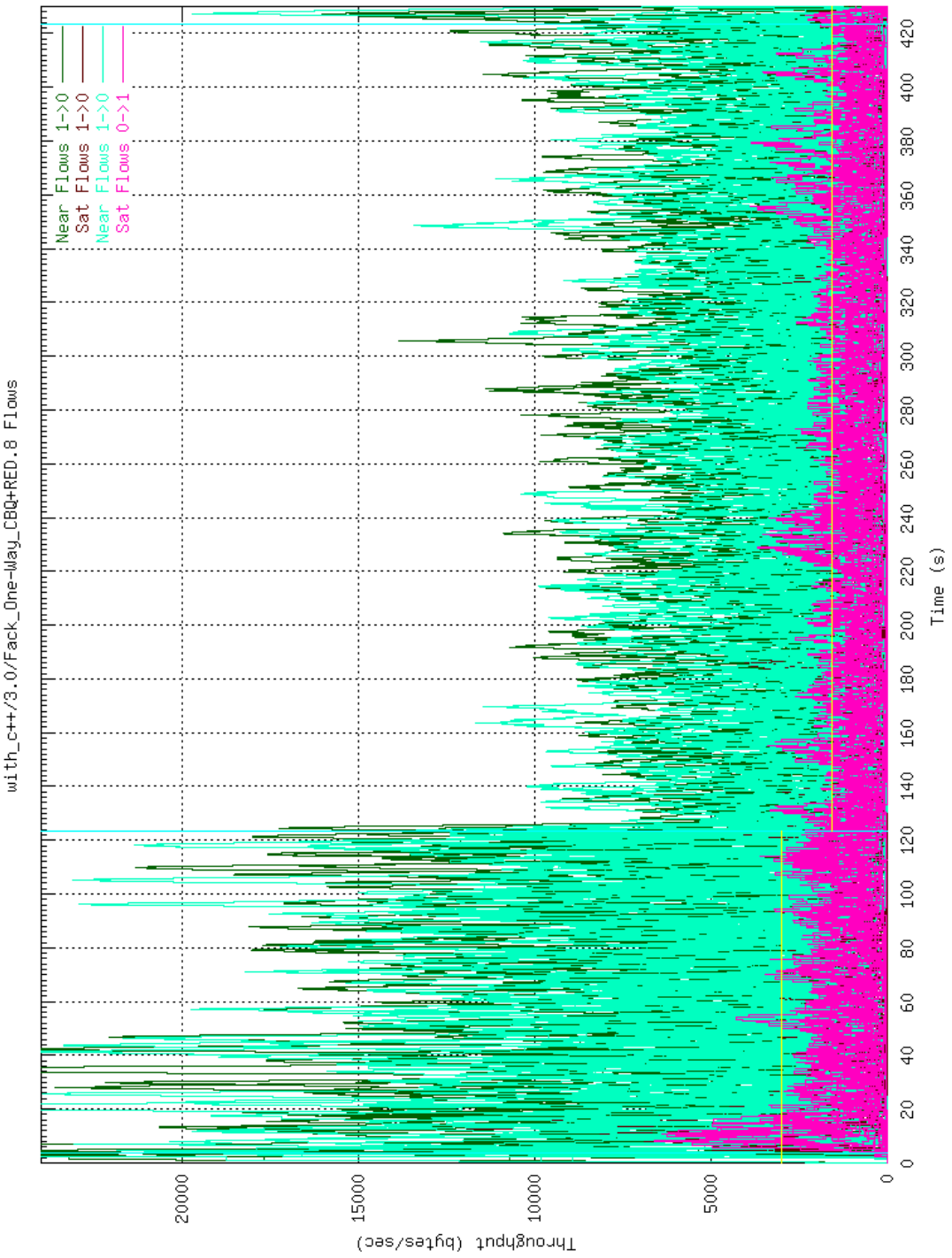

Figure 6.29: FACK TCP's throughput over time with RED in the near/far test.

TCP, and both protocols exhibited a kind of mode locking behavior that evenly scaled back throughput in response to bandwidth changes. However, STP proved unsuitable for window limited self pacing and exhibited oscillatory throughput changes. We concluded that STP's reduced STAT traffic from receiver to sender delayed the flow of information that in TCP would have regulated a sender's outgoing traffic.

When we investigated the effects of localized congestion on bandwidth sharing between sources with short versus long RTT delays, we found a definite throughput disadvantage for flows of any BE protocol with the longer satellite induced RTT. We concluded that the shorter, 6 ms RTT BE connections were able to probe and respond to changes in available bandwidth faster than the BE connections over the satellite link with a 908 ms RTT. Introducing RED at all routers had minimal impact on this performance bias.

Chapter 7

Performance Enhancements

From our experiments, we have observed how STP and the various implementations of TCP behave with bit errors, varying round trip times, and different router queuing disciplines. We learned how to attain a low-latency, guaranteed delivery QoS for CBR telephone traffic using class based queuing and traffic prioritization, and we saw how using RED closes the throughput variation gap between TCP and STP. Most importantly, we observed how to induce self-pacing in TCP through optimized window sizes.

As a result of our investigation, we would like to propose new techniques to optimize bandwidth allocation, notify end nodes of congestion without packet losses, and hasten recovery of lost packets for connections with long RTT. We propose to accomplish these goals without modifying BE protocol mechanisms and therefore would like to suggest three new tools to assist in IP flow management: Route Specified Window TCP (IP-VBR), Managed TCP Acknowledgments PEP (IP-ABR), and Segment Caching PEP.

7.1 Route Specified Window TCP

Fig. 7.1 reemphasizes how New Reno TCP's behavior drastically improves when configured with ideal window limits versus operating with or without RED. This observation from chapter 6 suggests a new TCP QoS level.

In a given operating system's TCP implementation, when a TCP socket is allocated, the OS fills in the socket's window size from a system default. This default window size

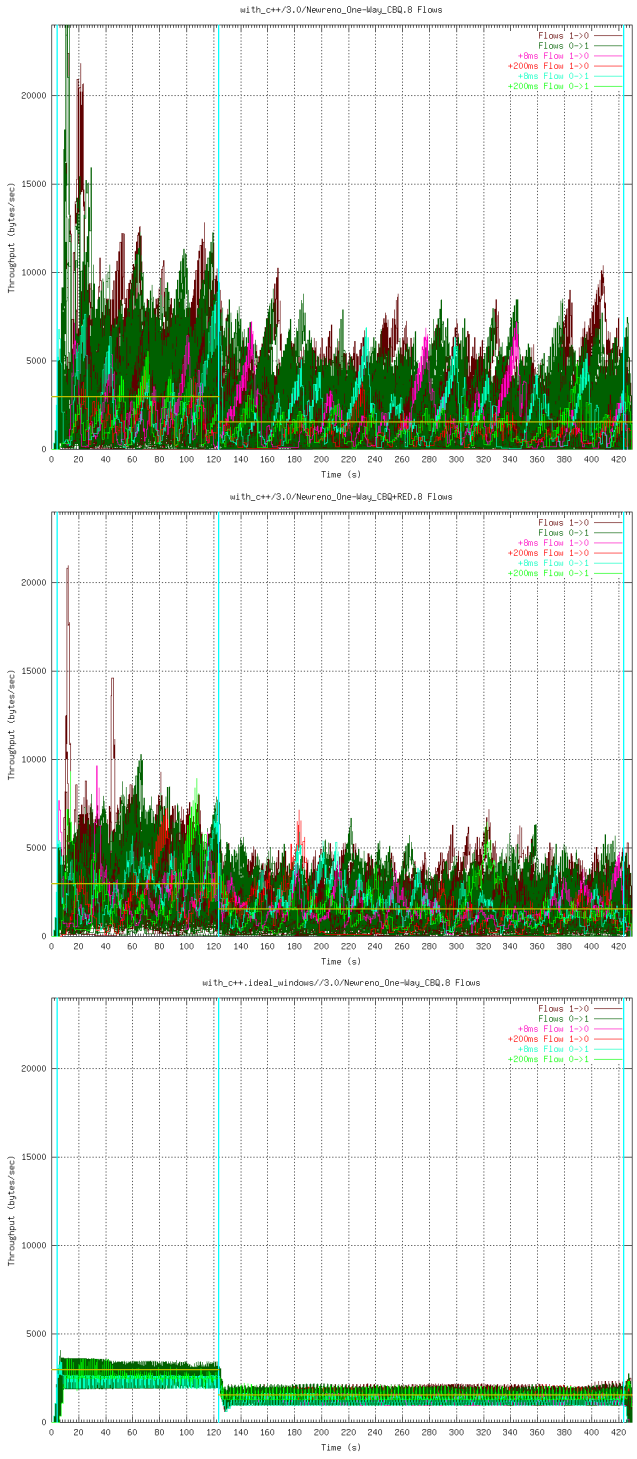

Figure 7.1: Comparing New Reno's behavior when using CBQ, CBQ+RED, and CBQ+Window Limiting (collected from Fig. 6.10, Fig. 6.13, and Fig. 6.18).

is left to be configured by an administrator based on approximations of the local network configuration. However, many networks have multiple gateways and routes to the rest of the Internet, and this single default window size may not provide the flexibility to optimally tune TCP for often-encountered routes and delays.

Here, we propose to set the default window size based on the flow's route so that self-paced behavior is guaranteed. TCP flows from these sources would enjoy nearly constant maximum bandwidth and low jitter owing to low throughput variation. The trade-off is that they will never enjoy bandwidth beyond the set bound imposed by lowered window sizes even when additional bandwidth is available. No modifications to TCP would be required.

7.1.1 Implementation

To implement this technique, we propose to modify the operating system or application code to use a route-dependent entry in the router table for a connection's receive-window size rather than a system global default value. We could also define a new router priority class in between the CBR and BE priorities to separate this traffic from the BE traffic that would otherwise disrupt the window-induced self-pacing.

Modifying the OS permits all existing applications to use this new service without alteration, but modifying the application allows new applications to enjoy this service regardless of the OS in use. However, we will subsequently introduce (when the background has been developed) a means to avoid modification of either the OS or application.

All users of this service would be assigned a priority for class based queuing between that of CBR sources and classic BE sources. Doing so will ensure these sources become self-pacing since their traffic would not be subjected to the congestion, queuing delays and bandwidth variations induced by the behavior of classic, uncontrolled BE sources sharing the same paths.

7.1.2 IP-VBR

The new service class we create using Route Specified Windows (RSW) is very similar to the VBR class of QoS provided by ATM[1], and we now have three IP QoS levels which resemble ATM counterparts: IP-CBR, IP-VBR, and IP-UBR. IP-CBR is our highest priority for Constant Bit Rate users requiring non-TCP, fundamentally unreliable protocols like UDP. Our new RSW, VBR-like class, IP-VBR, provides guaranteed bandwidth and low jitter for compliant hosts without modification of TCP semantics and implementations. Lastly, we rename our Best Effort class to IP-UBR, which provides a low priority, Unspecified Bit Rate (UBR) classification for unconstrained traffic.

However, IP-VBR, as defined, is not without administrative and operational predicaments. Determining the optimal window size per route may be difficult as it depends on how many hosts will share a network link. The number of hosts sharing a link may vary widely such as in ad-hoc networks with roaming users, and implementing IP-VBR would require coordination among all administrators of this service. Also, a security question arises since the bandwidth limits imposed by route defaults are enforced at the end system's operating system level. If the OS is misconfigured or tampered with, it may inject too much traffic and prevent delivery of the QoS implied by this service to other clients.

7.2 Managed TCP Acknowledgments PEP and IP-ABR

In our window limited tests of section 6.3, we basically modeled IP-VBR and found it was an effective solution within the limitations of its restrictions and implementation concerns. Building on the idea of window limited rate control, but removing a few restrictions, we shall permit sources to utilize spare bandwidth and automatically inform them when to decrease throughput. We propose a Performance Enhancing Proxy (PEP), where IP stacks on end systems do not need modification to take advantage of better QoS.

In TCP implementations, receivers advertise their maximum receive window to the sender in acknowledgment (ACK) packets. We propose to enable the network to modify

[1] In ATM, Variable Bit Rate (VBR) is a service agreement where a flow's allocated network bandwidth may vary between an agreed upon maximum down to zero but with lower delay variation than non-cooperating traffic.

the advertised receive window in ACKs in transit back to the sender. With this ability, routers could modify advertised receive windows based on link characteristics and network load and directly influence how fast a TCP sender injects data segments on the network.

7.2.1 IP-ABR

Now that the network can directly optimize flow rates for current conditions, we can elevate this traffic from the best effort base service level of IP-UBR to a new QoS class. We name this class IP-ABR, and describe it is as an Available Bit Rate (ABR) service class where hosts are informed of current network conditions through modification of in-transit ACK packets.

Unlike the means used by RED, managing ACK packets in this fashion does not require packets to be dropped to slow down senders. Our method results in senders responding much more quickly when they are needed to slow down, since they no longer need to wait for round trip timers or other timers. This empowers gateways with tight congestion control for TCP traffic. Even simplified Managed ACK proxies close to the sender and receiver could simply perform router table (with RSW information) lookups to implement our RSW (IP-VBR) scheme without modifying operating systems or applications.

However this approach is not without sensitivities, as it requires that certain precautions be observed to preserve the necessary semantics for correct operation. In the previous development, we assumed that the forward and reverse traffic paths flow through the same router or the same system of coordinated routers such as satellite-based home-web viewing systems using a modem for traffic from the home and a satellite link for traffic to the home. This scheme would not work if a TCP connection was congesting a given Internet router and that connection's reverse traffic (containing ACKs for the forward traffic) was routed through another Internet router since ordinarily routers do not share congestion and bandwidth information. This problem can be avoided by use of route-specific routing for both forward and reverse traffic, or, exchange of information instituted between routers through sideband signaling.

Also, managing ACK packets at intermediate routers is incompatible with encrypted IP traffic such as IPsec. An ACK managing router would need to be trusted with encryption

keys in order to obtain sequence and acknowledgment number information from encrypted TCP headers. While managing ACKs at intermediate routers is infeasible with IPsec, end-node ACK managing can be implemented. However due to RTT delays, end-node ACK managing is less efficient at using variable excess bandwidth.

7.2.2 Verifying the IP-ABR Concept

In section 7.2.1, we introduced a new service class called IP-ABR, and proposed it could optimally set TCP flow throughputs through modification of in-transit ACK packets. To prove the concept of the IP-ABR service class, we first identified possible IP-ABR operating modes in table 7.1, and then selected one of these modes to implement in the Network Simulator.

IP-ABR/EEB	Equi-shared Excess Bandwidth
EEB-MM	Sources request Min & Max bandwidth reservations.
EEB-NM	Sources request reservations with No Min bandwidth.
IP-ABR/SR	Source Responsive bandwidth allocation

Table 7.1: Possible bandwidth allocation modes for an IP-ABR Performance Enhancing Proxy.

In IP-ABR/EEB-MM, sources request bandwidth reservations, and if admitted, all are allocated their requested bandwidth minimum. Excess bandwidth is distributed evenly to all admitted source allocations up to their individual requested maximums. For IP-ABR/EEB-NM, if the minimum requested bandwidth is zero, there is no need for admission checking, and also, a simpler implementation may be possible if the maximum rate is assumed to be the available link bandwidth. Lastly, IP-ABR/SR is similar to the EEB modes except all sources are allocated their requested bandwidth minimum, and any excess available bandwidth is assigned up to each source's requested maximum on the basis of source utilization in a similar fashion to ATM ABR.

For proof-of-concept testing with the Network Simulator, we first implemented the simplified IP-ABR/EEB-NM scheme with no maximum bandwidths. After verifying basic functionality of the proxy, we then executed the step response and "Near/Far" tests of chapter 6 with IP-ABR proxies to answer these questions: Can IP-ABR evenly share available band-

width among TCP sources with large variations of RTT? Also, how would IP-ABR-managed TCP flows react in response to a 50% bandwidth reduction?

Implementing an IP-ABR Proxy

To implement an IP-ABR proxy in ns, the proxy must be able to read and modify in transit TCP acknowledgment and data segments in order to measure RTT and limit advertised receive windows. Unfortunately, ns specifies the advertised receive window at each source instead of inside each TCP ACK packet to simplify the simulator's implementation. When a TCP source receives an ACK, the source uses its local value for the receiver's window instead of any value from the ACK packet.

Fortunately, we found the TCP code in ns closely follows that of BSD 4.3 Reno UNIX. So we reimplemented the missing code fragments of BSD Reno's advertised receive windows and persistence timers in ns's "Full TCP" implementation. As a note, we relied on ns's "Partial TCP" implementations for most of our previous observations in this thesis citing that the results obtained from the partial and full TCP stacks were similar. However, when reimplementing advertised windows and persistence timers, we found that ns's "Full TCP" made fewer approximations and coding short-cuts away from the BSD code, so we decided to only update ns's "Full TCP" to reduce our implementation effort. We ran small tests to verify the functionality of our code additions, but more extensive testing may be required.

To build an IP-ABR proxy in ns, we started by building a simpler proxy that limits the advertised windows in passing ACK packets to a specified constant. We verified the proper operation of this proxy through simulation logs and ns's "Network Animator". After this verification, we proceeded to add the rest of the IP-ABR mechanisms.

For the proxy to automatically determine the window limit, we developed a mechanism to monitor data and acknowledgment packets as they pass through the proxy to measure the RTT and average ratio of data bytes per packet. For every passing data segment, we ignore data that was already acknowledged, and if a duplicate data segment arrives, we set a flag to prevent further updates to our RTT estimate until all outstanding data is acknowledged. Otherwise, we record the sequence number, data length, current time, and update running averages for packet length and data bytes per packet, $l_{\text{avg,packet}}$ and $l_{\text{avg,data}}$.

For each non-duplicate ACK packet passing through the proxy, we first measure the RTT from the proxy to the receiver by subtracting the transmit time of the most recent ACKed data from the current time, and then we update a running average RTT estimate, RTT_{forward}. As a note, the RTT from this proxy to the TCP sender, RTT_{reverse}, is measured in an identical fashion as a separate TCP flow; we model each TCP connection as two separate flows.

Once armed with a TCP flow's RTT and average packet length information, we first find the flow's total RTT (equation 7.1) and then estimate the flow's optimal window, W_{optimal}, according to equation 7.2. Equation 7.2 calculates the total number of bytes (data + header) a flow may have on the network at any time, and then scales that number back using the ratio of data bytes ($l_{\text{avg,data}}$) to total packet bytes ($l_{\text{avg,packet}}$), because a TCP stack's window only counts data bytes and not headers outstanding on the network. After calculating the flow's optimal window, we check the current ACK's window field, W_{current}, and if it is greater than W_{optimal}, we set it to W_{optimal}.

$$RTT_{\text{total}} = RTT_{\text{forward}} + RTT_{\text{reverse}} \tag{7.1}$$

$$W_{\text{optimal}} = \underbrace{BW_{\text{max}} \cdot RTT_{\text{total}}}_{\text{total bytes permitted on network}} \cdot \underbrace{\frac{l_{\text{avg,data}}}{l_{\text{avg,packet}}}}_{\text{deduct header bytes}} \tag{7.2}$$

Fig. 7.2 shows IP-ABR implemented in a separate ns node attached to the gateway with a 0 ms, 1.536 Mbps link. We initially built the IP-ABR proxy inside the gateway without using a separate proxy node, but we encountered difficulty when trying to place our data packet monitor after the gateway's output queue to prevent the queue from affecting our RTT measurements.

Step Response with IP-ABR

To compare our IP-ABR proxy with simple CBQ and CBQ with RED, we re-ran our test from chapter 6 where we monitored how the individual flow throughputs change over time in response to an approximate 50% step reduction in available bandwidth. This time

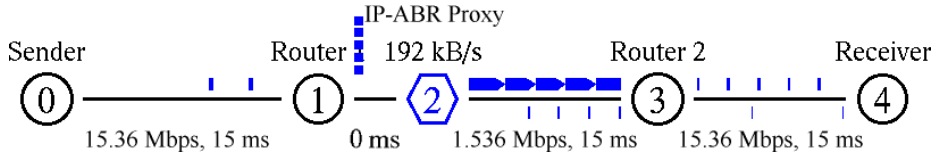

Figure 7.2: An IP-ABR proxy managing the data rate from a sender to a receiver in an example ns configuration.

however, we placed IP-ABR proxies on the satellite link as depicted in Fig. 7.3. The proxies were configured to evenly divide the link's bandwidth among all 64 pairs of best-effort users, and to throttle all flows back when the CBR users start.

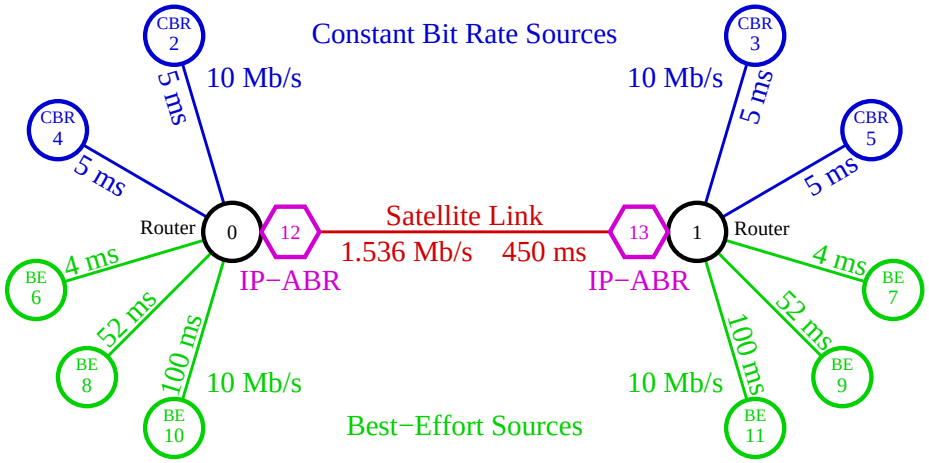

Figure 7.3: Adding IP-ABR proxies to the example ns configuration from section 4.1 involving 3 Best Effort and 2 Constant Bit Rate pairs of sources.

The first two graphs of Fig. 7.4 show the flow throughputs over time when only CBQ and CBQ+RED were used at the routers, and the IP-ABR proxies were disabled. The last graph of Fig. 7.4 shows the throughputs over time when CBQ was used on the routers, and the IP-ABR proxies were enabled. Fig. 7.5 shows the same test results again except it plots the throughput mean and ± one standard deviation across all flows.

In these tests, we found IP-ABR drastically cut the throughput variation across the flows with differing RTT. However, in the flow graph for IP-ABR in Fig. 7.4, there was a single flow during the initial two minutes that received only 500 bytes/sec, and also, several flows

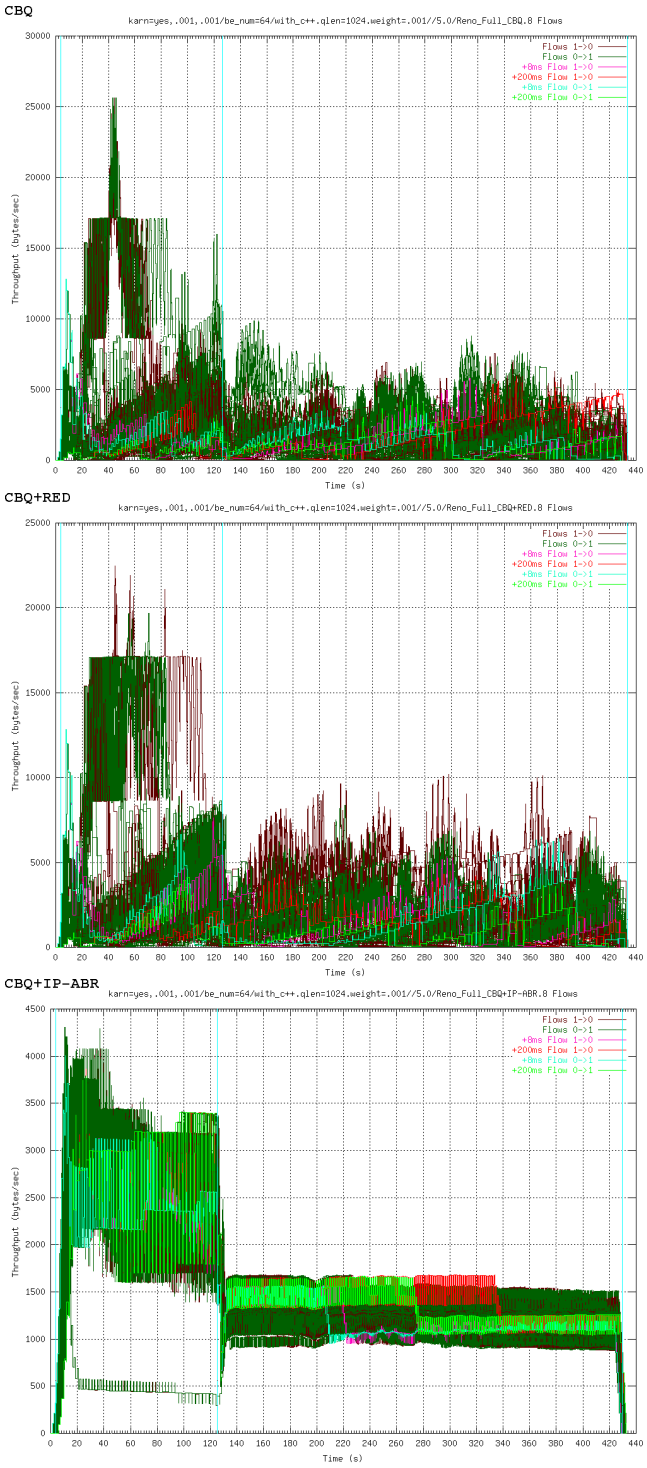

Figure 7.4: Comparing "Full TCP" Reno's throughput over time when using CBQ, CBQ+RED, and CBQ+IP-ABR.

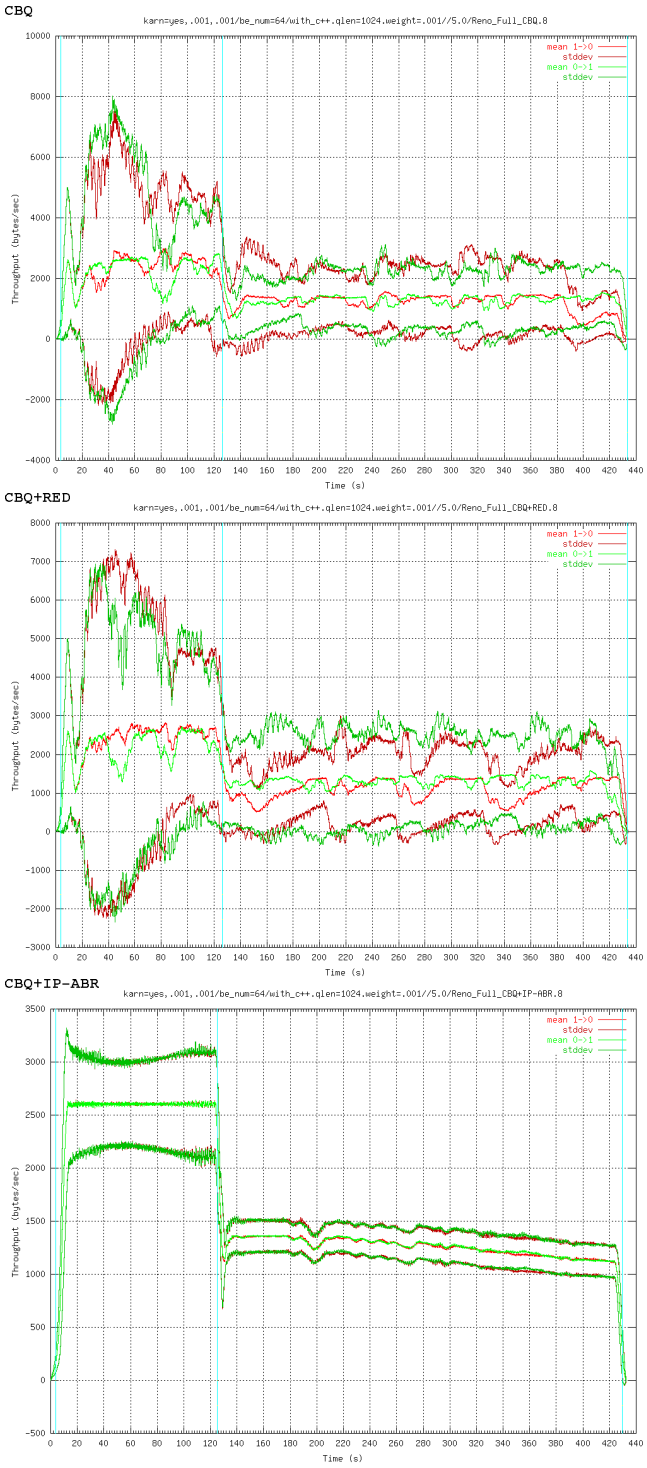

Figure 7.5: Comparing "Full TCP" Reno's throughput mean and variation across differing RTTs when using CBQ, CBQ+RED, and CBQ+IP-ABR.

experience a step reduction in throughput 100 seconds after the CBR flows had started. After further investigation by the student continuing this work as part of a new project, it was found that these anomalies were due to a bug in the test's TCL scripts and an off-by-one programming error in the IP-ABR proxy's ACK tracking tables. Fig. 7.6 and Fig. 7.7 show the results of fixing these errors. In short, we conclude that the IP-ABR service class works well.

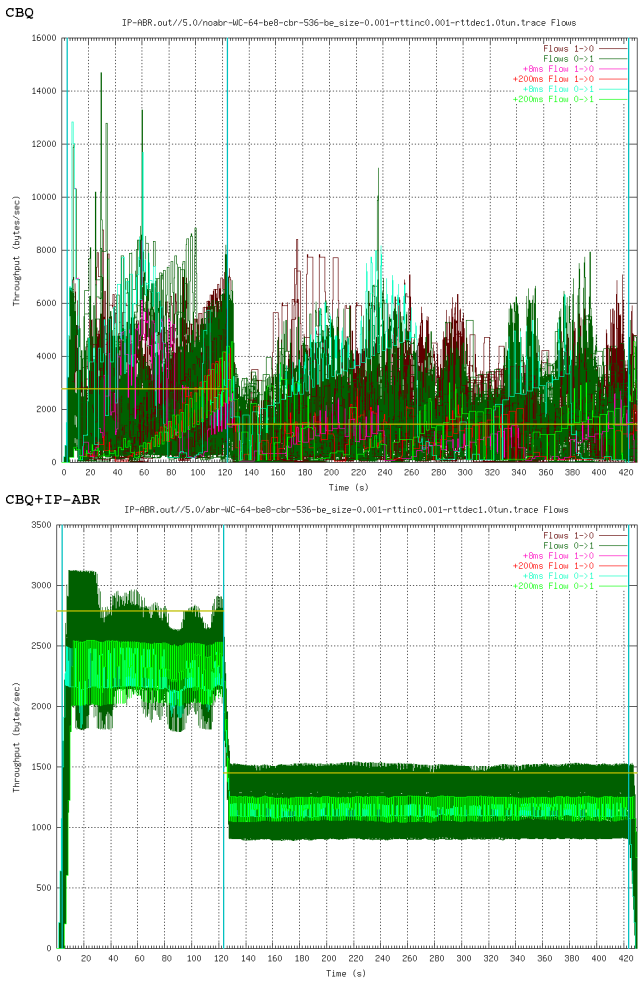

Figure 7.6: Results from the project continuing this thesis work: Comparing "Full TCP" Reno's throughput over time when using CBQ and CBQ+IP-ABR.

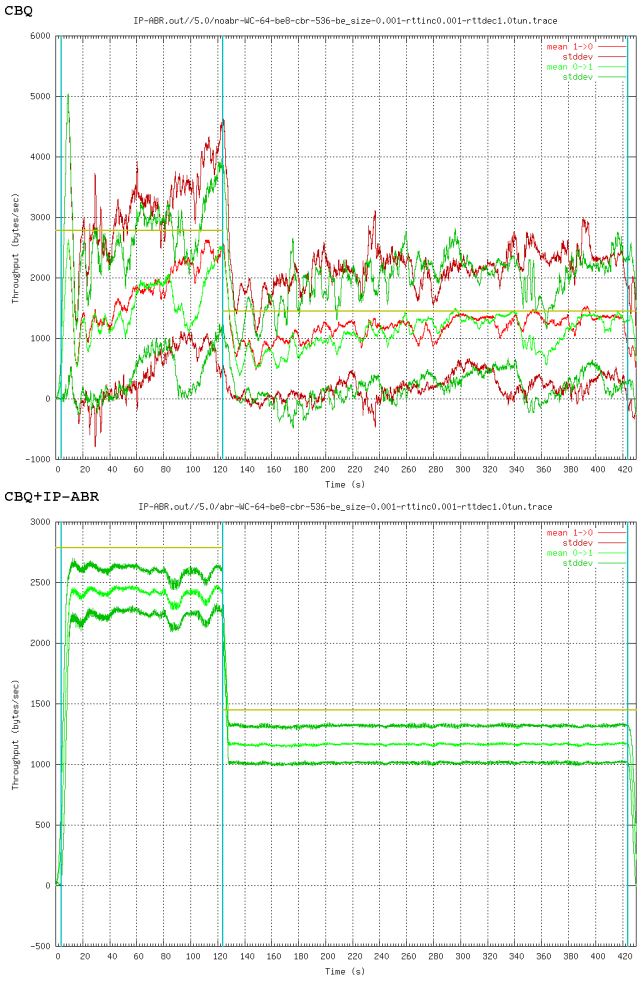

Figure 7.7: Results from the project continuing this thesis work: Comparing "Full TCP" Reno's throughput mean and variation across differing RTTs when using CBQ and CBQ+IP-ABR.

Near/Far Source Mix with IP-ABR

We were also curious to determine if IP-ABR could reduce the throughput difference between sources with drastically different RTTs, so we re-ran our tests from section 6.4. The only change we made to the test was that we added IP-ABR proxies to the shared network links as illustrated in Fig. 7.8.

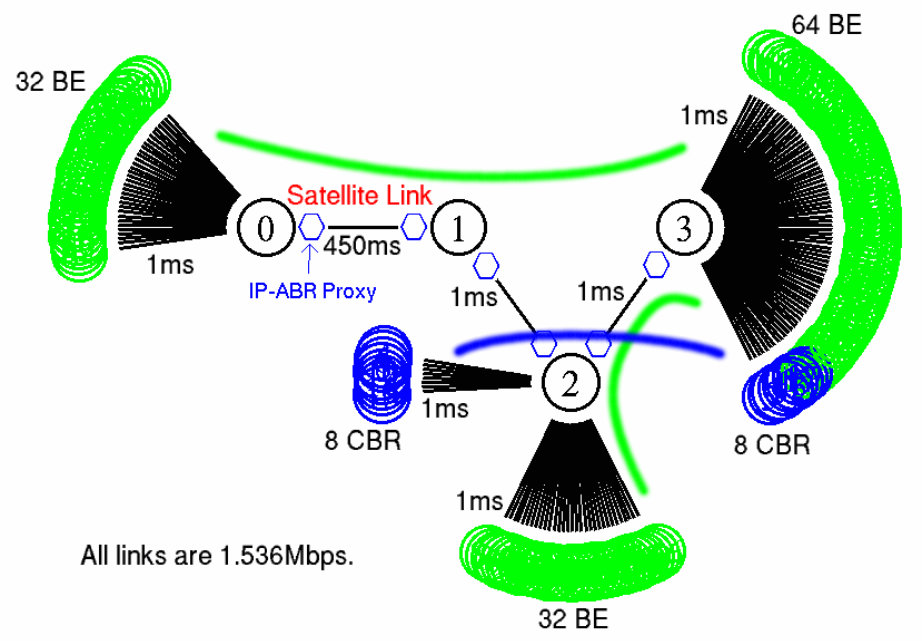

Figure 7.8: Adding IP-ABR proxies to the near/far source mixing network topology.

The first two graphs of Fig. 7.9 show the throughputs of all flows over the duration of the test with the IP-ABR proxies disabled and the routers using CBQ and CBQ+RED. In the test conducted yielding the last graph of Fig. 7.9, the IP-ABR proxies were configured to evenly share the link bandwidth across all flows and to scale all flows back at the start of the CBR traffic.

From these tests, we found that IP-ABR again dramatically stabilized each TCP flow's throughput. However, the near flows with the 6ms RTT still appeared to have a slight throughput advantage compared to the 908ms RTT flows. On inspection, we found that small RTT TCP sources cannot all fit a TCP segment on the link at the same time. This was

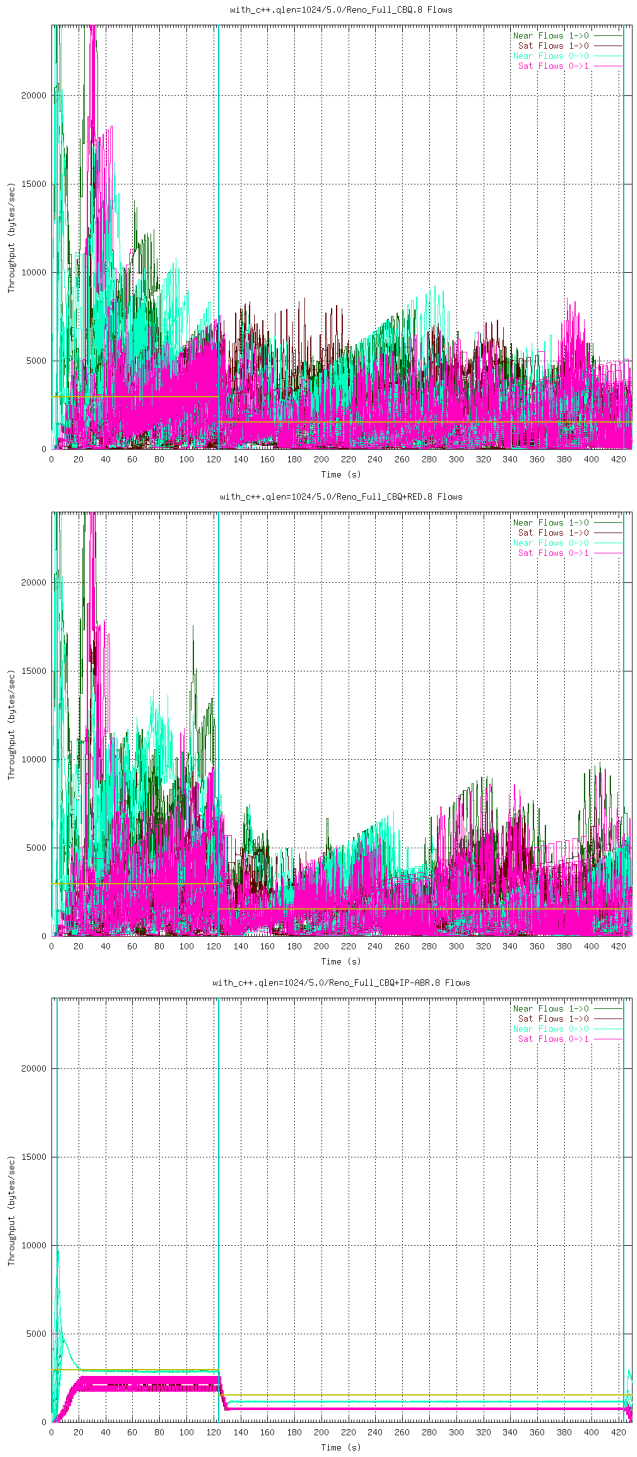

Figure 7.9: Comparing "Full TCP" Reno's throughput over time when using CBQ, CBQ+RED, and CBQ+IP-ABR in the near/far test.

an underlying assumption in our simplified mathematical model which the IP-ABR proxies used when calculating optimal window sizes. IP-ABR will need further modifications if used in network configurations such as this. These modifications are the subject of the project following this one.

7.3 Exploring Fast Packet Recovery

Our previous sections presented our performance optimizations based on tuning a best-effort protocol's window size. We found that if our proxy can rewrite the receiver's advertised window in ACK packets, we can implement low variance flow control such as IP-VBR and IP-ABR. However, there are many applications best served by ordinary Best Effort service not assisted by admissions restricted QoS services such as IP-VBR and IP-ABR. These traffic flows must contend with the effects of congestion, this being the only bandwidth sharing mechanism at work for this traffic component. However, as has been seen, long RTT flows suffer an extraordinary performance penalty compared to low RTT flows sharing the same paths. In this section we will explore means for ameliorating these effects via fast packet recovery at intermediate nodes that can be applied to both ordinary and IPsec based traffic.

The cause of large scale flow variance for Best Effort traffic is the severe flow reduction that is triggered by every Retransmission Time-Out (RTO) event in TCP implementations. After discussing connection splitting, we will present a general approach to enhance best-effort protocol performance which is applicable to any protocol using sequentially numbered packets.

7.3.1 Connection Splitting

Split TCP connections have been suggested as a means for achieving higher TCP throughput in large delay-bandwidth links in RFC 2757[13], RFC 2760[1], RFC 3135[3], and Indirect TCP[2], but several problems can be readily identified.

Using split TCP connections breaks end-to-end semantics. If a fault occurs in the network which isolates any one of the splitting routers, then a connection cannot be re-established

as unrecoverable loss of state has occurred. Specifically, an application might incorrectly assume its data was acknowledged at the receiver, when in fact, its data was only acknowledged at the splitting PEP, and never made it to the destination.

Additionally, IPsec is not compatible with splitting, because TCP header contents (including ACK numbers) are encoded. TCP splitting requires all parties to trust the splitting routers.

High router resource and processing overhead are required to maintain separate buffers, timers, and other resources for every TCP flow and up- and down- stack processing for each. This in turn makes routers highly vulnerable to SYN-flooding attacks since entire flow buffer resources must be allocated for each connection started.

We believe we have identified another effective Performance Enhancing Proxy based strategy that preserves end-to-end semantics, requires no per flow buffering or management, allows mixing of both enhanced protocol aware and standard network equipment, can be applied throughout the network, and is tolerant of network reconfiguration in the face of faults. This PEP requires no modification of the TCP stack implementations at end points to preserve end-to-end connection semantics, and it does not only have to be applied at satellite uplink terminals. In the case of a fault induced network reconfiguration, no state information needs to be transferred between protocol aware network equipment. Let us now explore how to reduce flow throughput variance for the general case of protocols with sequentially numbered data packets.

7.3.2 Segment Caching PEP

TCP and STP sequentially number data segments. We could ameliorate RTO caused flow variation on long-delay links (like satellites) by placing a PEP on the destination side of these links. The router would cache data segments, and when it sees old or duplicate ACKs for a segment in its cache, it may delete the ACK[2] and retransmit the cached segment as illustrated in Fig. 7.10. Another PEP function may detect packets lost upstream of their link from sequence number discontinuity and use out-of-band signaling to request resends of the missing sequences from cooperating upstream PEPs. This strategy of caching and

[2]and/or modify the ACK when using Selective Acknowledgments

resending segments may be used with many protocols using sequence numbers including IPsec as noted in [9].

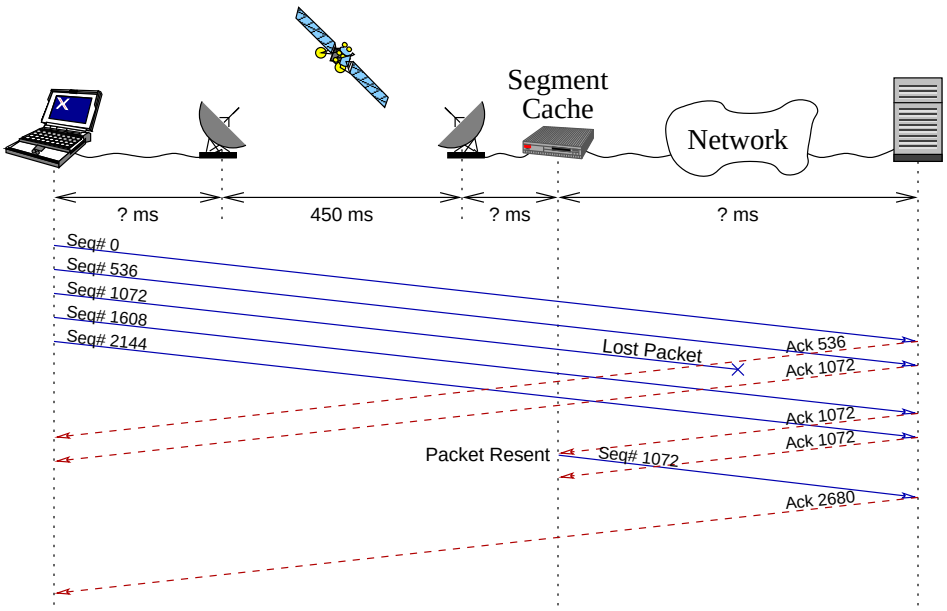

Figure 7.10: A segment cache replays a lost data segment to avoid satellite link delay.

However, similar to Managed Acknowledgments, Segment Caching has its own implementation concerns. Caching routers may need additional memory and processing power to track individual flows. Also, segment caching undermines existing congestion avoidance and notification algorithms like RED. RED relies on notifying senders to slow down by dropping packets, but a segment caching router may insulate TCP and STP end computers from RED's intentional packet losses.

7.4 Summary of Optimizations

In summary, we proposed three new techniques in this chapter for maximizing network efficiency and boosting best-effort performance. The first, and possibly the easiest to implement, involves modifying operating systems or applications to specify an optimized default window size for each route, and then placing traffic from these hosts in an elevated priority service class named IP-VBR to mimic ATM's variable bit rate service.

Our second proposed optimization permits routers to notify TCP end nodes of congestion through scaling back advertised received windows in ACK packets instead of throwing away good data in algorithms like RED. Traffic subjected to ACK management may also be placed into a priority service class, IP-ABR, to reproduce ATM's available bit rate service.

Lastly, we rejected connection splitting as a means to ameliorate RTO caused flow variation on long-delay links in section 7.3.1. We instead proposed that routers cache and retransmit packets in response to information obtained from intercepted acknowledgment information. These segment caches would be coordinated to defend expensive links such as satellites, but may need extra configuration precaution to not unintentionally circumvent lossy congestion avoidance and notification algorithms like RED.

We could now configure a network with Segment Caches at the perimeter, and employ CBQ with prioritized traffic classes on routers. Table 7.2 presents an example CBQ router configuration to provide optimal QoS for different application requirements using our proposed traffic classes.

Service	Description
IP-CBR	highest priority, low delay, guaranteed delivery
IP-VBR	for trusted hosts with ABR-incompatible protocols like IPsec
IP-ABR	for TCP, STP, and if the router is trusted for information, but the host is not trusted to not hurt the network, IPsec
IP-UBR	lowest priority for any other traffic

Table 7.2: IP service classes mimicking ATM from our proposed optimizations.

Chapter 8

Conclusions

In our investigation of the performance of STP and TCP implementations over SATCOM links, we first set out in chapter 4 to measure how CBR load and satellite router queuing discipline affects BE performance and CBR QoS. From this, we found that using CBQ in conjunction with RED provided dynamic bandwidth allocation and QoS for CBR telephone traffic and closed any throughput variance advantage STP has over TCP. We also made our first observations indicating that different implementations had similar throughput performance, and that STP did not live up to the claims of increased throughput over TCP made in STP's founding research.

Next we explored how network link bit errors impact TCP and STP application throughput in chapter 5. Again, we found that STP demonstrated no advantage over TCP except at the edge of usable service breakdown, and even this throughput advantage may only be due to the greedy nature of STP we saw in chapter 4 when CBR traffic was not being protected with CBQ routers. We also noted the lack of advantage displayed by SACK and FACK TCP which were intended to recover from higher packet losses; we reasoned this finding is due to the implementations falling back to New Reno TCP behavior at extreme levels of packet loss.

Then in chapter 6, we developed a new method to measure protocol application throughput over time and used it to evaluate how TCP and STP react to sharp bandwidth changes, to explore BE protocol throttling induced through self-pacing, and to observe how long-

delay, satellite traffic throughput is handicapped by shorter RTT traffic. We found that TCP and STP were in a continual state of global synchronization — a problem which we were able to solve with window limited self-pacing when RED did not meet our expectations.

Using our evaluations from the preceding chapters, we developed three new techniques in chapter 7 to maximize network efficiency and boost best-effort performance. We proposed a new service class mimicking ATM's VBR for traffic from trusted end computers that pick optimal TCP window sizes based on each connection's route. Then we extended this idea to enable routers to dynamically change a connection's window size through modifying ACKs in transit to form a service similar to ATM's ABR. Lastly, we introduced a new approach of caching data segments after we rejected connection splitting as a means to boost connection performance over satellite links. We found segment caching to have many desirable features such as interoperability with IPsec, maintenance of end-to-end connection semantics, and deployability on a wide range of network links besides satellite links.

8.1 Discussing STP

With regard to the Satellite Transport Protocol, STP's founding thesis paper[8] presented a good overview of TCP performance troubles, but it did not address the performance issues directly with the introduction of the STP protocol. Instead it only introduced a new transport protocol with NAKs (instead of ACKs like TCP) which ultimately only reduced reverse bandwidth. STP was not designed particularly for satellite links, but the protocol seems most appropriate for highly asymmetric link bandwidths, such as satellite based home-web viewing systems, and for high loss connections owing to its aggressive bandwidth usage.

Henderson used ns for testing STP throughput but not for simulating connection splitting/terminating as his original split connection scripts (obtained from Henderson) were missing major components. When we reimplemented the missing pieces, the resulting data was still invalid due to ns's fundamental lack of support for blocking receive calls for applications as discussed in section 3.2.

We assert that contrary to the claims in the original STP thesis investigation by Henderson[8],

the Satellite Transport Protocol does not have a larger mean throughput than TCP at low CBR rates. There is little bandwidth saved in reducing ACK packets due to large MTU versus ACK packet size, and also due to the fact that many ACKs are piggy backed which is not exposed by ns's 1-way implementations of STP and TCP. Our simulation results indicate that for high CBR loads, STP's mean throughput exceeds that of TCP on Drop-Tail and RED, because STP's rate control scheme is more greedy than TCP and impairs CBR performance to claim more bandwidth. Henderson's apparent STP throughput improvement in his HTTP loaded ns simulation tests was almost certainly due to STP's greedy behavior stealing bandwidth from the background TCP traffic and not due to any real STP benefit.

It could be argued that our simulations were not identical to those of Henderson as ns had progressed through several revisions since his tests, requiring that we port his code for our tests. However it should be duly noted that our simulation tests with STP duplicated the behaviors reported by Henderson, and our conclusions arise from a careful analysis of the traffic interactions that underly this otherwise unquestioned external behavior.

Our results clarify the situation in such a way as to allow us to reason that STP can only benefit performance in conditions of low error rates, low congestion, when typical data segment packets are close in size to ACK packets, and when TCP-like reliable semantics are required. The conditions of low error rate and low congestion are necessary, or otherwise, large numbers of NAK packets occur. And the last conditions of small typical data segment packets and TCP-like reliable semantics are not typical for web or multimedia traffic, respectively.

Bibliography

[1] Allman, M., Dawkins, S., Glover, D., Griner, J., Tran, D., Henderson, T., Heidemann, J., Touch, J., Kruse, H., Ostermann, S., Scott, K., and Semke, J. RFC 2760: Ongoing tcp research related to satellites, Feb. 2000.

[2] Bakre, A., and Badrinath, B. R. I-TCP: Indirect TCP for Mobile Hosts. *15th International Conference on Distributed Computing Systems* (1995).

[3] Border, J., Kojo, M., Griner, J., Montenegro, G., and Shelby, Z. RFC 3135: Performance enhancing proxies intended to mitigate link-related degradations, June 2001.

[4] Braden, R. T. RFC 1122: Requirements for Internet hosts — communication layers, Oct. 1989.

[5] Brakmo, L., O'Malley, S., and Peterson, L. TCP Vegas: New Techniques for Congestion Detection and Avoidance. In *SIGCOMM* (1994), pp. 24–35.

[6] Clark, D., and Jacobson, V. Flexible and efficient resource management for datagram networks, Apr. 1991.

[7] Floyd, S. Red: Discussions of setting parameters, November 1997.

[8] Henderson, T. *Networking over Next-Generation Satellite Systems.* PhD thesis, University of California at Berkeley, 1999.

[9] Holl, D., and Cyganski, D. Final Report for Raytheon SATCOM Systems/WPI Project: High Demand Network Interface to Military Satellite Services - Performance

Analysis of TCP and STP Implementations Over SATCOM Links and Proposals for New Classes of QoS Services for TCP/IP. unpublished report, Worcester Polytechnic Institute, Sept. 2002.

[10] MacDonald, D., and Barkley, W. *Microsoft Windows 2000 TCP/IP Implementation Details*. Microsoft Corporation, 2000.

[11] Mathis, M., and Mahdavi, J. Forward Acknowledgement: Refining TCP Congestion Control. In *SIGCOMM* (1996), pp. 281–291.

[12] Mathis, M., Mahdavi, J., Floyd, S., and Romanow, A. RFC 2018: TCP selective acknowledgment options, Oct. 1996.

[13] Montenegro, G., Dawkins, S., Kojo, M., Magret, V., and Vaidya, N. RFC 2757: Long thin networks, Jan. 2000.

[14] Postel, J. RFC 768: User datagram protocol, Aug. 1980.

[15] Postel, J. RFC 793: Transmission control protocol, Sept. 1981.

[16] Stallings, W. *High-Speed Networks: TCP/IP and ATM Design Principles*. Prentice-Hall, Inc., 1998.

[17] Stevens, W. RFC 2001: TCP slow start, congestion avoidance, fast retransmit, and fast recovery algorithms, Jan. 1997.

[18] VINT Project, T. The ns Manual. http://www.isi.edu/nsnam/ns/.