Application Level Congestion Control Enhancements for High Bandwidth Delay Product Networks

by

Anupama Sundaresan

B.E. (Electronics and Communication Engineering),
Anna University, India, 1997

Submitted to the Department of Electrical Engineering and Computer Science and the Faculty of the Graduate School of the University of Kansas in partial fulfillment of the requirements for the degree of Master of Science

_____

Professor in Charge

_____

_____

Committee Members

_____

Date Thesis Accepted

Om Sri Sairam

To my Parents

Sundaresan and Kala

Acknowledgments

I would like to express my sincere gratitude to Dr.Joseph B.Evans for serving as my committee chair and for his guidance and advice throughout this research and for making this thesis possible. I would also like to thank Dr.Victor Frost, my advisor for his encouragement during my entire graduate studies. I would also like to express my thanks to Dr.Susan Gauch for serving on my Masters Committee.

First of all I would like to say a very big thank you to Roel Jonkman for his support, encouragement, guidance, advice, motivation and for always being there for me when I needed him. Thanks Roel. I would also like to thank Tom Lehman, Jason Lee and Brian Tierney for all the help. I wouldn't have been here for my Masters, if it hadn't been for the confidence and encouragement of Jayanth and Praveen. So, thanks to the two of you for making me realise that I could do it.

Next, I would like to thank Bhavani and Arioli who have been an infinite source of encouragement and support for me. I guess gratitude is a very insufficient word to express all that I feel for Gowri and Sara for being there for me always. A special thanks to Yogs for being such a great source of encouragement and for all the nice times. I would like to thank Raj, Aarti, Sai and Ramya for all the nice times.

I would also like to thank Anand Iyer, Phongsak, Dhananjay, Deepakys, Pramodh, Deepak, Ram, Ananth, Kamalesh, Pooja, Priyanka Jiang and Yulia for their support during my stay in KU.

Last but definitely not the least, I would like to thank the two people who have showered infinite love and affection on me and who have been an eternal source of encouragement to me. Mom and Dad without you none of this would have been possible. I would also like to thank my sister for all the love and encouragement.A final word of thanks to the God, Almighty for without him none of this would have been possible.

Abstract

The introduction of fiber optics and Gigabit Ethernets is resulting in ever-higher transmission speeds, and these paths are moving out of the domain for which TCP was originally engineered. TCP has some basic performance problems on high Bandwidth Delay Product network enviroments, some of which were solved by the window scaling extensions provided. But, the Slow Start algorithm which TCP employs, forms one major performance bottleneck for short term flows on high latency links because of its inefficient bandwidth utilization. The Congestion Avoidance Algorithm which TCP resorts to in the event of any loss also reduces the throughputs obtained because of its slow increase in the Congestion Window over a period of several RTTs. Due to these problems, the idea of turning off Congestion Control (*NOCC*) in the TCP Stack was explored and this work presents the results obtained by testing *NOCC* on a cross-country WAN network. *NOCC* was tested with a traffic generating tool called NetSpec and the number of packets in flight, throughputs received and reaction to congestion were seen to be significantly better when compared with normal TCP.

NOCC was also tested with a real world application i.e., the Apache Web Server and the improvements in response times are significant. By using *NOCC* an application which uses TCP is not inhibited by the Congestion Window and so can write any amount of data upto the receiver's advertised window onto the network. To give the application more control over what it is writing onto the network, pacing is done at the application. Pacing was implemented in Apache and tested along with *NOCC* and NetSpec with Pacing was also tested with *NOCC*. The results validate that NOCC can be used along with pacing to make use of the abundant bandwidth available in high BDP networks.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Over the years, Transmission Control Protocol (TCP) in the Internet protocol (IP) suite has become the most widely used form of networking between computers. With recent developments in high-speed networking and applications that use TCP, performance issues in TCP are of increasing interest and importance.

The Transmission Control Protocol [32] is the reliable connection-oriented transport protocol that a number of major Internet services use to communicate (e.g., HTTP [1], FTP [34], SMTP [33]). TCP performs well on low latency links but on huge Round Trip Time links, TCP does not make full use of the available bandwidth. A good representative example of such networks is an OC-3c(155Mbps) or OC-12c (622Mbps) Asynchronous Transfer Mode Network or a Gigabit Ethernet Network. Also, TCP as the transport protocol for a protocol like HTTP, which has very short duration flows, yields very poor response time and throughput.

This thesis outlines a mechanism that will help the Transmission Control Protocol (TCP) better utilize the bandwidth provided by huge bandwidth, long delay links. It also presents results to show the advantages and disadvantages of implementing this mechanism.

1.1 TCP overview

TCP provides reliable segment delivery through a positive acknowledgement mechanism. Each data segment transmitted contains a sequence number indicating the position of the data in the transmission. It is a full duplex protocol, meaning that each TCP connection supports a pair of byte streams, one flowing in each direction. It also includes a flow control mechanism for each of the byte streams that allows the receiver to limit how much data the sender can transmit at a given time. TCP also supports a de-multiplexing mechanism that allows multiple application programs on any given host to simultaneously carry on a conversation with their peers. In addition to the above features, TCP also implements a highly tuned congestion control mechanism. The idea of this mechanism is to throttle how fast TCP sends data, not for the sake of keeping the sender from overrunning the receiver, but so as to keep the sender from overloading the network.

TCP is a sliding window protocol. A sliding window protocol allows the sender to transmit a given number of segments before receiving an ACK. When an ACK is received by the sender, the window *'slides'* to allow one more segment to be transmitted. Each TCP segment sent (data segments and ACKs) contains a *"window advertisement"*. The size of the window advertised by the receiver is the upper bound for the sender's sliding window. Generally, standard TCP advertises a window of 65,535 bytes due to the 16 bits allocated for the advertisement in the TCP header. On high BDP links extensions are proposed to this which are discussed later in this thesis.

1.2 Motivation for Congestion Control in TCP

TCP uses a set of congestion control algorithms [3, 15] that further control TCP's sending behavior. These algorithms are important because they ensure that TCP will not transmit data at a rate that is inappropriate for the network resources avail-

able. If TCP's transmission rate is too high, the intermediate routers in the network can be overwhelmed. If segments arrive at an intermediate router faster than the router can forward the segments, the segments will be queued for later processing. If a segment arrives at a router that has no memory to queue the segment, the segment will be discarded. Therefore, it is important for TCP to be able to adapt its sending rate to the network conditions to avoid segment loss.

When too many TCP connections are sending at an inappropriately high rate the network can suffer from *"congestive collapse"* [16]. Congestive collapse is a state when segments are being injected into the network but very little useful work is being accomplished, most of the data segments or their corresponding ACKs are discarded by one of the intermediate routers in the network before reaching their destination. This causes the sender to retransmit the data, further aggravating the problem. Congestive collapse is discussed in more detail in [17]

TCP's congestion control algorithms attempt to prevent congestive collapse by detecting congestion and reducing the transmission rate accordingly. While these algorithms are very important they can also have a negative impact on the performance of TCP over long RTT or high latency links and satellite links [18]. TCP's four congestion control algorithms are slow start, congestion avoidance, fast retransmit and fast recovery [15, 3]. The following is a brief outline of slow start and congestion avoidance. Fast recovery and fast recovery will be explained in later chapters.

1.3 Slow Start and Congestion Avoidance

The slow start and congestion avoidance algorithms [15, 3] allow TCP to increase the data transmission rate without overwhelming the intermediate routers. To accomplish this, TCP senders use a variable called *"congestion window"* (*CWND*). TCP's congestion window is the size of the sliding window used by the sender

and *CWND* cannot exceed the size of the receiver's advertised window. Therefore, TCP cannot inject more than *CWND* segments of unacknowledged data into the network.

The slow start algorithm is used to gradually increase the amount of unacknowledged data TCP injects into the network, by gradually increasing the size of the sliding window. Slow start is used at the beginning of a TCP connection and in certain instances after congestion is detected. The algorithm begins by initializing *CWND* to one segment. For each ACK received, TCP increases the value of *CWND* by one segment. For example, after the first ACK arrives, *CWND* is incremented to two segments and TCP is able to transmit two new data segments. This algorithm provides exponential increase in the size of the sliding window. Slow start continues until either the size of *CWND* reaches the *"slow start threshold"* (*ssthresh*) or when a segment loss is detected. The value of *ssthresh* is initialized to the size of the receiver's advertised window at the beginning of the connection. If TCP's retransmission timer expires for a given segment, TCP retransmits the segment but also uses this as an indication of network congestion. In response to a retransmission timeout, TCP reduces its sending rate by setting *ssthresh* to half of *CWND*'s value and then setting *CWND* to one segment. This triggers the slow start algorithm, which will stop when values of *CWND* meets or exceeds *ssthresh* or another loss is detected. The new value of *ssthresh* places an upper bound on the slow start algorithm of half the sending rate when the loss was detected.

Congestion Avoidance is the phase, which follows slow start. In this phase the value of *CWND* is greater than or equal to *ssthresh*. This algorithm increases *CWND* at a slower rate than during slow start. For each segment ACKed during congestion avoidance, the congestion window is increased by $1/CWND$ (unless this would make the value of *CWND* greater than the receiver's advertised window). This adds roughly one segment to the value of *CWND* every RTT. The congestion avoidance algorithm provides a linear increase in the size of TCP's sliding

window. This mechanism is used to probe the network for additional capacity in a conservative manner.

1.4 ENABLE Overview

Emerging Next Generation Internet (NGI) applications will push the limits of available network bandwidth. There are two critical services required to guarantee maximum efficient use of the network resources. The first is a system for monitoring the performance of each component in the system, enabling detailed performance analysis of the complete end-to-end system. The second is a system for monitoring current network characteristics, and providing this information to network-aware applications, which can effectively adapt to the current network conditions.

These capabilities require a very similar set of services. Both require an adaptive monitoring infrastructure, a monitor data publishing mechanism, and monitor data analysis tools. We propose to develop a *"Grid"* service that will provide both of these capabilities. The overall goal of this project is to address these issues in order to provide manageability, reliability, and adaptability for high performance applications running over wide-area networks.

The next generation of high-speed networks will allow DOE scientists unprecedented levels of collaboration. Large data archives will be easily accessed from anywhere on the network. However, diagnosing performance problems in high-speed wide-area distributed systems is difficult because the components are geographically and administratively dispersed, and problems in one element of the system may manifest itself elsewhere in the network. Problems may be transient, and may be due to activity in the infrastructure. Also, a large volume of monitoring data may be needed for diagnosis and the type of monitoring data and its analysis depends on the nature of the problem, and the necessary monitoring

data may not be available when it is needed.

In addition to the ability to locate performance problems, in order to efficiently use NGI networks this new class of applications will need to be *"network-aware"*. Network-aware applications attempt to adjust their resource demands in response to changes in resource availability. Emerging QoS services will allow the application to participate in resource management, so that network resources are applied in a way that is most effective for the application. Network-aware applications will require a general-purpose service that provides information about the past, current, and future state of all the network links that it wishes to use. This service is called *ENABLE* (Enhancing of Network-aware Applications and BottLeneck Elimination). This service will include monitoring tools, visualization tools, archival tools, problem detection tools, and monitoring data summary and retrieval tools. The monitoring tools will be capable of monitoring the entire end-to-end system, and will include tools for monitoring network components (switches, routers, and links), system components (hosts, disks, etc.), and applications. The results of the monitoring will then be published in directory services via the Lightweight Directory Access Protocol (LDAP) [19], allowing network-aware applications to obtain the information needed to adapt to current conditions.

Presently, the archival tools and the monitoring tools to store per session data in the database are being put together. The work in this thesis will be used by an application when this infrastructure is in place. The application before setting the Congestion Window (*CWND*) to a certain value or before turning off Congestion Control in the kernel checks with the database of monitored data to see if the link to the destination is uncongested and capable of supporting NOCC. The work in this thesis is an effort to validate the turning off of Congestion Control in TCP and analyzing the performance benefits of NOCC over long latency links to see if it can be deployed.

1.5 Motivation for No Congestion Control

TCP has a fundamental performance bottleneck for one transmission regime: paths with high bandwidth and long round-trip delays. The significant parameter is the product of bandwidth (bits per second) and round-trip delay (RTT in seconds); this product is the number of bits it takes to *"fill the pipe"*, i.e., the amount of unacknowledged data that TCP must handle in order to keep the pipeline full. TCP performance problems arise when this product is large, e.g., significantly exceeds 10^5 bits.

There is no one-line answer to the question: "How fast can TCP go?" The issues are reliability and performance, and these depend upon the round-trip delay and the maximum time that segments may be queued in the Internet, as well as upon the transmission speed.

When the RTT of a link is very high, the time TCP spends in the slow start phase is high and so effectively TCP doesn't yield good throughput results in that phase. For a protocol like HTTP which uses TCP as the transport protocol and which has very short duration flows, the response times because of TCP's slow start phase is disastrous over a high RTT link. For HTTP flows, the entire duration of the flow is predominated by TCP's slow start behavior and this has bad effects on the response time of the web server.

To overcome this startup behavior of TCP especially for HTTP flows and other short duration flows and to make full use of the bandwidth available in the DOE-NGI testbed, the idea of turning off Congestion Control in the kernel came up. The application uses a *setsockopt()* interface to turn off the congestion control in the kernel. All the kernel modifications were made to a linux 2.2.13 kernel. Once the congestion control has been turned off in the TCP stack, the transmission control block maintained for each session is unaware of the congestion window (*CWND*). At any point the advertised window advertised by the receiver in ACKs forms the sender's limit on the number of packets it can have outstanding in the

network. In essence the slow start behavior of TCP is overcome by turning off congestion control in the kernel. This thesis explains the implementation of the No Congestion Control (*NOCC*) in Linux. It also describes the testing of TCP with No Congestion Control on a cross-country WAN link and compares the performance statistics obtained with TCP with and without Congestion Control.

TCP with *NOCC* starts off by essentially transmitting an advertised window's worth of packets to the receiver. In cases where the advertised window is high say $\geq$ 65,535 bytes, the probability of an intermediate router being overwhelmed is pretty high. A retransmission event in a short duration flow because of an intermediate router being overrun can be pretty expensive on the performance of TCP and so to avoid this it was decided to experiment with burst traffic or paced traffic over *NOCC*. This thesis also analyses the effect of pacing on *NOCC*. It describes the implementation of pacing in Apache Web Server [42] and describes the performance benefits obtained with pacing over TCP with *NOCC*.

1.6 Organization of this Thesis

This thesis is divided into the following chapters:

Chapter 2 briefly describes related work done in TCP congestion control algorithms to get better performance from TCP on long RTT links.

Chapter 3 provides background information before dealing with the implementation aspects of No Congestion Control in TCP in the Linux kernel. It also briefly describes the HTTP protocol before discussing the implementation of the pacing algorithm in Apache.

Chapter 4 describes the methodology used to run tests and discusses the results obtained. Tests run with NetSpec and with Apache to validate the use of No Congestion Control on high delay links are presented here.

Chapter 5 states the conclusions of the results and future work that could

be done in this area.

Chapter 2

Related Work

Related work can be categorized in three main areas:

- Evolution of TCP and TCP Extensions for high performance networks
- Research on TCP slow start

2.1 Evolution of TCP

TCP [32] is the reliable, connection-oriented, end-to-end error, flow and congestion control protocol in the transport layer of the TCP / IP reference model. It is the most widely used protocol in the Internet, providing reliable transfer of data packets. Its basic implementation is unsuitable for high speed and high delay networks, and therefore modifications and additions were added to enhance the performance of the protocol over such networks [5, 6, 28]

W. Richard Stevens in [3] lists the four basic congestion control algorithms that should be included in any modern implementation of TCP. These algorithms are Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery. The last two algorithms were developed to overcome the shortcomings of earlier implementations, like TCP Tahoe [8]. TCP Tahoe was getting into the slow start phase every time a packet was lost and thus valuable bandwidth was wasted. A mod-

ern TCP implementation that includes the above four algorithms is now known as TCP Reno.

Janey C. Hoe in [4] modified the Reno version of TCP to improve the start-up behavior of the TCP congestion control scheme. These improvements include finding the appropriate initial threshold window *ssthresh* value to minimize the number of packets lost during the start-up period and creating a more aggressive Fast Retransmit algorithm to recover from multiple packet losses without waiting unnecessarily for the retransmission timer to expire. The latter is the congestion algorithm that TCP New Reno is using.

Matthew Mathis, Jamshid Mahdavi, Sally Floyd, and Allyn Romanow in [30] proposed the Selective Acknowledgment (SACK) option to enhance the TCP performance over high speed and high delay networks. With this option, the sender can get enough information from the receiver on the state of the correctly received packets and thus it is able to retransmit in one round trip time multiple lost segments. Simulation based performance results of TCP SACK over low and high delay paths were documented in [2, 20] and showed that TCP SACK can significantly improve the network performance when compared with earlier TCP implementations.

Matthew Mathis and Jamshid Mahdavi in [31] proposed the Forward Acknowledgment (FACK) algorithm to reinforce TCP SACK for higher performance. FACK improves TCP congestion control during recovery.

2.1.1 TCP Extensions for High Performance Networks

Ever since Slow Start and the Congestion Control Algorithms were first proposed, there have been constant modifications to TCP and to the congestion control algorithms to get better performance. A lot of research has been done on improving TCP performance on Satellite links [23, 2, 27].

The original TCP standard limits the TCP receive window to 65535 bytes.

Without the large windows extension proposed to TCP, the maximum throughput of a TCP connection is bounded by the round trip time [32].

$$Throughput_{max} = \frac{Receive\ Buffer\ Size}{Round\ Trip\ Time}$$

Without TCP large windows, then a TCP connection on a typical cross-country WAN link on which the RTT is about 70ms is limited to a throughput of

$$Throughput_{max} = \frac{64Kbytes}{70ms}$$

$$Throughput_{max} = 7.314Mbps$$

Note that this upper bound on TCP throughput is independent of the bandwidth of the channel. As specified by Van Jacobson in [6], large windows (window scaling) can allow TCP to fully utilize higher bandwidth links over long-delay channels.

2.1.1.1 Bandwidth Delay Product

Network performance is measured in two fundamental ways:
Delay in Networks is the two-way latency for information to propagate from the sending node to the receiving node. It is also known as Round Trip Time (RTT).
Bandwidth is the number of bits that can be transmitted in a certain period of time.
BDP is the product of the above two-performance metrics, i.e. the number of bits the network can hold.

As shown in the Figure 2.1, if we imagine a pipe, with its length representing the two way network latency and its width representing the bandwidth of the connection (specified by the line rate of the underlying transport technology), then

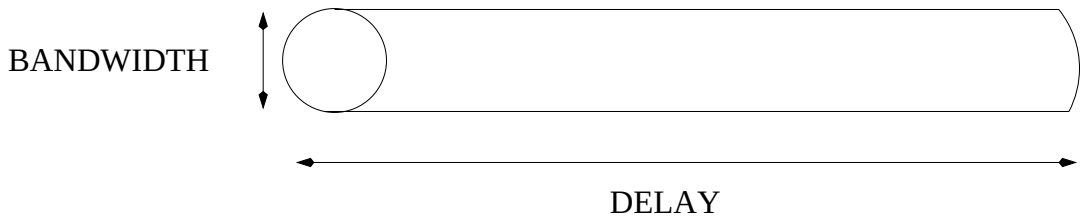

Figure 2.1: Representation of the BDP as given in [37].

the BDP is the volume of the pipe.

In transport / data link layers, the BDP represents the maximum amount of allowed unacknowledged data outstanding at any moment on the network, keeping the link or pipeline full.

TCP performance depends not upon the transfer rate itself, but rather upon the product of the transfer rate and the round-trip delay. This bandwidth*delay product measures the amount of data that would *"fill the pipe"*. It is the buffer space required at sender and receiver to obtain maximum throughput on the TCP connection over the path, i.e., the amount of unacknowledged data that TCP must handle in order to keep the pipeline full. TCP performance problems arise when the bandwidth*delay product is large. We refer to an Internet path operating in this region as a *"long, fat pipe"* and a network containing this path as an *"LFN"*

Van Jacobson, Robert Braden, and David Borman in [5] describe extensions to be included in TCP implementations to make the protocol suitable for high speed and high delay networks. The proposed extensions are:

- Large sliding window sizes for the protocol to be able to accommodate the high BDP value of high speed and high delay networks;
- TCP time-stamps for more precise estimations of the round trip time;
- Protection Against Wrapped Sequences (PAWS) for preventing the TCP sequence number to wrap around after some transfer time depending on the transport network speed.

2.2 Research on TCP Slow Start

TCP Slow Start Phase has been an area of active research. An internet draft by the Internet Engineering Task Force (IETF) TCP_SAT working group [24] states a proposal for an increment of the initial window size (i.e., *"slow-start"* by releasing more than one segment to the network). By increasing the initial window, more packets are released to the network immediately and thus triggering more acknowledgments allowing the window to open more rapidly. This will be also beneficial for World Wide Web (WWW) transfers, which on average are very small, since data will be transferred immediately in certain cases.

Also, [24] suggests a scheme for deactivating delayed acknowledgments during slow start. By doing that and acknowledging every packet received, the time needed to complete slow start will be less than that needed when delayed acknowledgments are enabled in modern TCP implementations, and the window will be increasing more rapidly. Another alternative for trying to increase the window more rapidly and still having delayed acknowledgments during slow start is the use of "byte counting" [24] Using this mechanism, the window increase is based on the number of previously unacknowledged bytes ACKed and not on the number of ACKs received. Mark Allman in [2] carried out experiments with the above three slow start modifications and observed significant performance improvements over regular TCP Reno.

Sally Floyd also proposed [22] an increase in the permitted upper bound for TCP's initial window from one segment to between two and four segments. In most cases, this change results in an upper bound on the initial window of roughly 4K bytes (although given a large segment size, the permitted initial window of two segments could be significantly larger than 4K bytes). A discussion of advantages and disadvantages of such a change, outlining experimental results that indicate the costs and benefits of such a change to TCP can be found in [22].

In just a few years since its inception, the World Wide Web has grown to be

the most dominant application in the Internet and there has been a lot of research in this area to maximize web performance. In large measure, this rapid growth is due to the Web's convenient point-and-click interface and its appealing graphical content. Since Web browsing is an interactive activity, minimizing user-perceived latency is an important goal. However, layering Web data transport on top of the TCP protocol poses several challenges to achieving this goal.

First, the transmission of a Web page from a server to a client involves the transfer of multiple distinct components, each in itself of some value to the user. To minimize user- perceived latency, it is desirable to transfer the components concurrently. TCP provides an ordered byte-stream abstraction with no mechanism to demarcate sub-streams. If a separate TCP connection is used for each component, as with HTTP / 1.0 [12], uncoordinated competition among the connections could exacerbate congestion, packet loss, unfairness, and latency.

Second, Web data transfers happen in relatively short bursts, with intervening idle periods. It is difficult to utilize bandwidth effectively during a short burst because discovering how much bandwidth is available requires time. Latency suffers as a consequence.

To address these problems, a new connection abstraction for HTTP, called persistent- connection HTTP (P-HTTP)[11] was developed by Venkata Padbhanabhan and Jeffrey Mogul. The key ideas are to share a persistent TCP connection for multiple Web page components and to pipeline the transfers of these components to reduce latency. The main drawback of P- HTTP, though, is that the persistent TCP connection imposes a linear ordering on the Web page components, which are inherently independent. The analysis of the interactions between P-HTTP and TCP are given in [14].

This drawback of P-HTTP led to the development of a comprehensive solution, which has two components. The first component, TCP session, de-couples TCP's ordered byte- stream service abstraction from its congestion control and loss

recovery mechanisms.

The second component of the solution, TCP fast start [13], improves bandwidth utilization for short transfers by reusing information about the network conditions cached in the recent past instead of initiating the Slow Start discovery procedure each time. To avoid adverse effects in case the cached information is stale, TCP fast start exploits priority dropping at routers, and augments TCP's loss recovery mechanisms to quickly detect and abort a failed fast start attempt.

TCP is not optimized for multiple request/responses over a single connection. This is the common case with HTTP/1.1 [1]. When a new request/response occurs after the connection has been idle, how should TCP on the server behave? Some TCP implementations force slow-start again (for example 4.4 BSD and Linux 2.x). Other implementations (SunOS) do not even detect this idle time and thus use the old value of the congestion window. The latter approach can overrun queues at intermediate routers, leading to packet loss. Though restarting with slow-start avoids this risk, it means added delay for each time we slow-start and get to steady state. This can degrade the performance of layers that TCP provides service to, a strong example being P-HTTP.

So, to overcome the *"slow start restart"* problem, Vikram Visweswaraiah and John Heidemann suggested Rate Based Pacing (RBP) [10]. It requires the following changes to TCP:

- Idle time detection and indication that RBP needs to be started
- Bandwidth estimation
- Calculation of the window that we expect to send in RBP and the timing between segments in that window
- A mechanism that clocks the segments sent in RBP.

Chapter 3

Background and Implementation

3.1 Transmission Control Protocol (TCP)

TCP [32] is a sophisticated transport protocol that offers connection-oriented and reliable-byte stream service. It is also extremely flexible in that almost any underlying technology can be used to transfer TCP/IP traffic. It is an end-to-end protocol with error, flow, and congestion control functions.

Error control is achieved by having the receiver sending cumulative acknowledgments (ACK) for successful transmissions. By using ACKs to pace the transmission of packets, TCP is said to be self-clocking. In case of a segment loss, the lost segment will not get acknowledged and therefore it will be transmitted again. In cases where the ACK is lost, a retransmission timer (RTO) will expire and the segment will be retransmitted. In order for TCP to avoid duplication of segments, it applies a unique sequence number to every segment released to the network. The retransmission timeout is not a fixed value but it is adaptively calculated according to the network topology using the network round trip times and their variance estimates from the mean. More detailed information about the TCP RTO can be obtained in [35, 38]. It is appropriate to note here that TCP systems use a coarse-grained timer

with a granularity of 500 ms, which is well known as ”tick”. This means that, the sending node has to wait for one ”tick” in most cases to retransmit a segment.

Flow control is achieved by implementing a sliding window scheme and it is used to prevent the sender from overflowing the receiver when there are no available buffers in its system. Having the receiver advertising with every acknowledgment segment a receiving window (RCV_WND) value does this. The sender is then limited to having no more than RCV_WND bytes of unacknowledged data in the network at any time. The receiver selects a RCV_WND to advertise based on the amount of memory allocated to the connection for the purpose of buffering data.

Congestion control is achieved by a variety of algorithms as described in ([3, 16]). The purpose of the congestion algorithms is to reduce the rate by which the sending nodes insert data into the network when congestion is detected or at the beginning of a TCP connection to slowly probe for the network capacity. Since the performance results from our experiments are affected greatly by the congestion algorithms, a description of those (slow start, congestion avoidance, fast retransmit, fast recovery) as well as a description of the large windows extension and delayed acknowledgments follow.

3.1.1 Slow Start and Congestion Avoidance [3]

TCP uses slow start whenever it establishes a new connection, in order to avoid these connections instantaneously contributing to congestion. TCP also uses the slow start technique to recover from congestion. It is an algorithm for controlling the rate at which the sender transmits data into the network. This algorithm leads to an exponential growth of the window size and the time needed to fully utilize the bandwidth (i.e. to reach the receiver’s advertised window) is given by equation

(3.1).

$$Slow\_Start\_Time = RTT * \log_2 W, \quad \text{where:}$$
$$\text{RTT is the round trip time between the two hosts,} \tag{3.1}$$
$$\text{W is the number of segments fit in the receiver window size.}$$

When a lost segment is detected, the *CWND* size is set to one segment, and the slow start algorithm starts over until the sender reaches half of the original *CWND* . From thereafter, TCP enters the congestion avoidance phase and slows down the rate of increment. During this phase, the sender transmits to the network one additional segment for each round trip time, until the receiver's advertised window is reached.

Generally speaking, TCP involves decreasing the congestion window when the level of congestion goes up and increasing the congestion window when the level of congestion goes down. Taken together, the mechanism is commonly called additive increase / multiplicative decrease [37]. In summary:

- When the congestion window is below the slow start threshold (*ssthresh*) threshold, the congestion window grows exponentially.
- When the congestion window is above the threshold, the congestion window grows linearly.
- Whenever there is a timeout, the threshold (*ssthresh*) is set to one half of the current congestion window and the congestion window is then set to one.

3.1.2 Delayed Acknowledgments [21]

Recent TCP implementations, use delayed acknowledgments in order to reduce operating system and processing overheads. Therefore, the receiver is sending cumulative ACKs back to the sender after it receives a predefined number of segments instead of acknowledging every segment. In BSD Unix implementations

the ACK cannot get delayed over 200 ms or over the reception of two segments. In Linux implementations, the delayed ACK timeout period is 500ms and these implementations do not use delayed ACK in the start of a connection. Thus, the time needed to fully fill in the pipe with slow start, i.e. equation (3.1), now becomes equation (3.2) [25].

$$Slow\_Start\_Time = RTT * (1 + \log_{1.5} W) \tag{3.2}$$

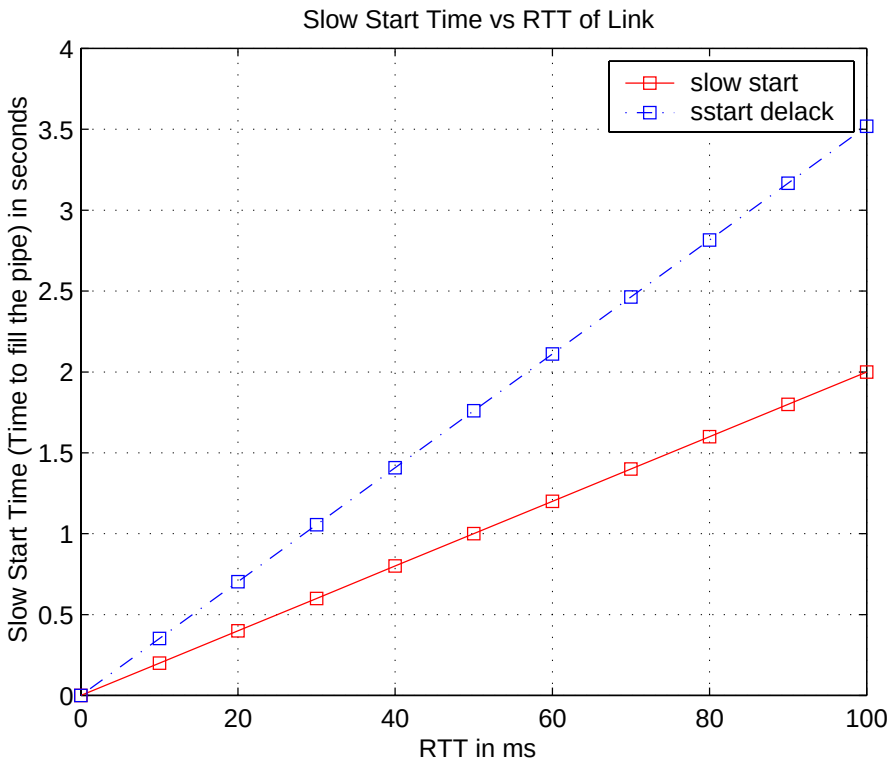

Figure 3.1: Slow Start Time with and without Delayed ACKs.

The Figure 3.1 shows how the slow start time increases for increasing RTTs. It plots normal slow start and slow start with delayed ACKs to show how delayed ACKs increase the slow start time even more than normal slow start.

3.1.3 Fast Retransmit and Fast Recovery [3]

Modern TCP systems generate duplicate acknowledgments for every out-of-order segment received. The TCP fast retransmit algorithm detects a segment loss quickly by counting three duplicate ACKs, and then immediately retransmits the lost segment without waiting for the retransmission timer to expire. The value of *ssthresh* is set to one-half the current *CWND* . The *CWND* now gets the value of *ssthresh* plus three times the segment size. From now on each time another duplicate ACK arrives, the *CWND* increases by one segment size and a new packet is released to the network, if of course allowed by the current new value of *CWND*. When an ACK arrives that acknowledges new data, fast recovery is used, where the *CWND* is set to *ssthresh* and congestion avoidance begins. If for any reason there is a retransmission timeout, the sender will get into slow start.

3.1.4 Large Sliding Windows [5, 6]

The bandwidth-delay product is the amount of unacknowledged data outstanding at any moment on the network, keeping the link or pipeline full, and it corresponds to the minimum buffer size or window size on the receiver host as given by equation (3.3).

$$Window(\text{Buffer size}) = (\text{Round Trip Time}) * (\text{Throughput}) \tag{3.3}$$

The upper limit of the TCP window is determined by the socket buffer space in the source and sink UNIX operating system kernels. For larger RTT connections, and thus larger BDP values, the window needed is bigger. So, for high BDP networks the end hosts must have enough buffers to be able to accommodate the BDP and consequently to achieve high performance.

The initial implementation of the TCP protocol had the capability to provide only 65535 bytes of window sizes, by using 16 bits in the TCP header. According

to equation (3.3), this was inappropriate for high bandwidth, high delay networks, like cross-country WAN links and satellite links. Therefore extensions were added ([5]) to increase the window option to 32 bits in the TCP header. This enhancement is included in modern TCP systems.

3.2 Implementation of No Congestion Control

In this thesis, an attempt was made to make the Linux TCP stack unaware of a state variable called *CWND* . All the modifications were done to a Linux 2.2.13 kernel. To aid the application in turning off congestion control in the kernel, a setsockopt() interface was provided to the application. So, if the application desires to turn off congestion control, it invokes the *setsockopt()* with *TCP_NO_CONGESTION*. This automatically makes the upper bound of the *CWND* equal to the window advertised by the receiver. The application also has the provision of setting the *CWND* to some initial value greater than the default with another setsockopt() invoked through *TCP_SET_CWND*. This provision is made to aid the application in making informed decisions after tracking the network behavior in the past few minutes, seeing the trend in the values of *CWND* and then setting the *CWND* to the optimum value. When the application turns off congestion control in the kernel it sets a flag called *nocc* which is part of the TCP control block.

When the application does a write on a TCP socket, it gets translated into a systemcall called *sys_write()*. This in turn invokes *tcp_do_sendmsg()*, which copies the data to be sent from user space to kernel skbuffs. It allocates space for all the headers and eventually calls *tcp_send_skb()*. This is the main buffer sending routine, which decides whether to queue or transmit now. This is where the function *tcp_snd_test()* is called which checks to see if a transmission can be made now. It checks the number of packets in flight and sees if the *CWND* allows for any more packets to be on the network and if yes, returns a TRUE to *tcp_send_skb()* which in

turn sends a packet down to the lower layers for transmission. This is where the hooks for *NOCC* have been provided. If the flag *nocc* in the TCP control block has been set, *tcp_snd_test()* returns a TRUE irrespective of whether the number of packets in flight have hit the *CWND* limit. The only limitation on the upper limit on the number of packets in flight is the receiver's advertised window. The *tcp_snd_test()* returns a TRUE until that limit has been reached. By doing this we have TCP's flow control mechanism in place but TCP is unaware of the *CWND* parameter, thus disbling congestion control in effect.

In the transmitter, when a packet is received from the receiving end the *tcp_rcv_established()* is called. When the incoming segment is an ACK from the receiver, acknowledging data sent by the transmitter, the transmitter sees if the received ACK opens up the sliding window and in normal TCP, for each ACK received, two more segments can be sent out. Here, *tcp_data_snd_check()* is called which in turn checks if data can be written on the wire. So, the same procedure of checking if the receiver's advertised window has been hit is done in *tcp_data_snd_check()* and the segments are pumped out.

So far the behavior of TCP under normal circumstances has been explored. Now, let us consider what happens when a retransmission timeout occurs because of congestion in the network. Under the normal operation of TCP, it halves the current *CWND* and stores it in *ssthresh* and sets *CWND* to one and starts the slow start procedure until *ssthresh* is reached after which Congestion Avoidance takes over. Since we don't want TCP to do a slow start when a retransmission timeout occurs, modifications had to be made to how TCP handles retransmission timeouts. TCP calls a function called *tcp_xmit_retransmit_queue()* which in turn pumps out the retransmissions. When TCP is about to do a retransmission it again checks if the number of packets in flight has already hit the *CWND* limit because it doesn't want to create further congestion in the network. This check is bypassed if the *nocc* flag is set.

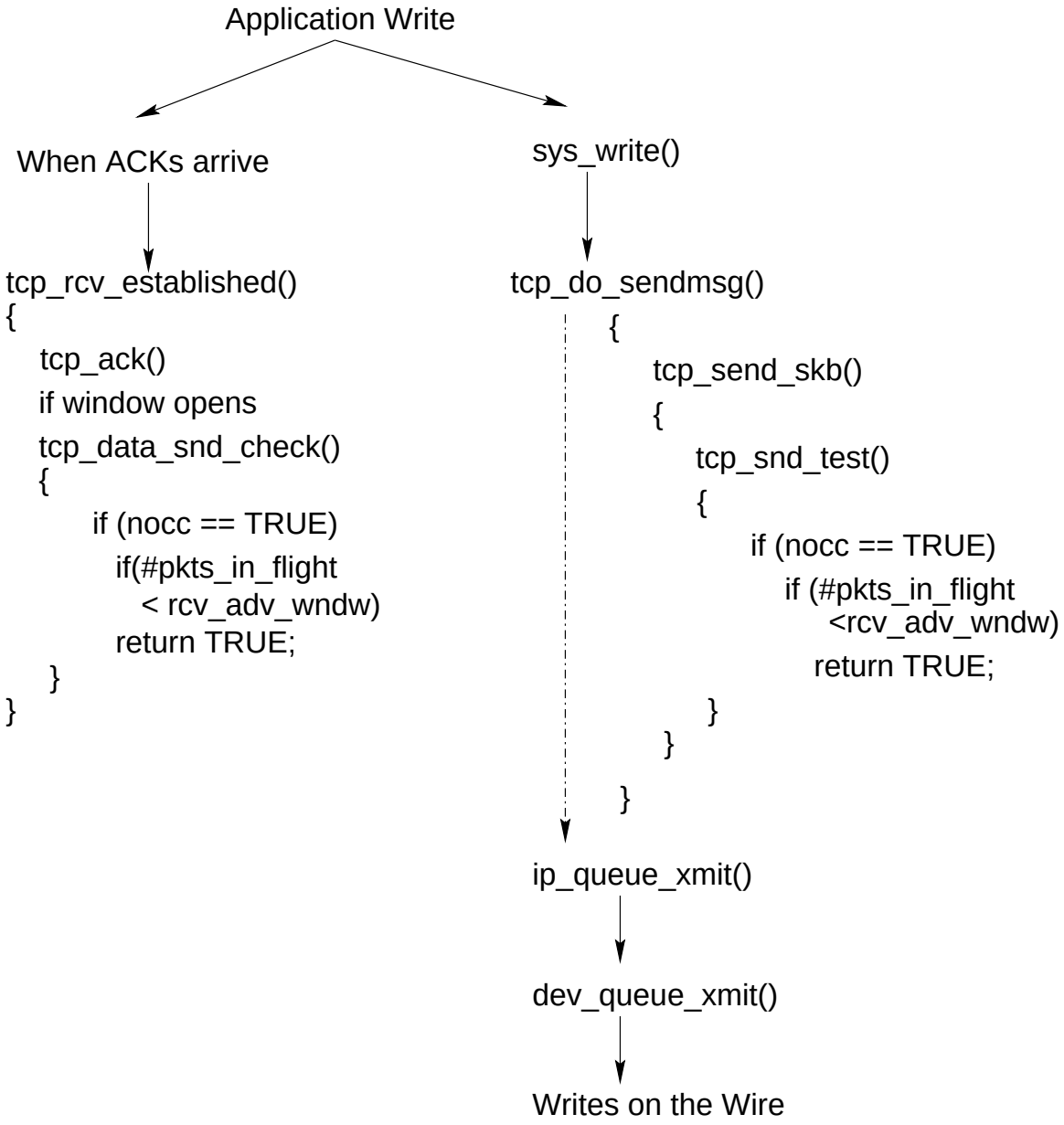

Figure 3.2: Flow of Control through the Protocol Stack on an Application Write.

In the present Linux protocol stack, there is no way to get the number of retransmissions occurring at the TCP layer. Though the */proc* interface has a field which keeps track of the number of retransmissions occurring on a particular connection, it does not grab the correct value since the TCP control block does not maintain correct statistics to count the number of retransmissions. The retransmission variables maintained in the TCP control block are decremented and reset to zero as and when retransmissions occur. So, to count the number of retransmissions and to help the application retrieve the number of retransmissions on a particular socket connection, a setsockopt() interface with *TCP_TOTAL_REXMITS* was provided. Also, a parameter to keep track of the total number of retransmissions on a socket was added. This variable was incremented when a retransmission occurred as signified by the calling of *tcp_retransmit_skb()*. The /proc interface was modified to grab this correct value and print it out in */proc/net/tcp*. This will be used when the entire monitoring infrastructure is in place and when an application wants to store the number of retransmissions that occurred on a particular connection in the database of monitored information.

3.3 HTTP Overview

The transmission of a web page from a server to a client involves the transfer of multiple distinct components, each in itself of some value to the user. To minimize user-perceived latency, it is desirable to transfer the components concurrently. TCP provides an ordered byte-stream abstraction with no mechanism to demarcate sub-streams. If a separate TCP connection is used for each component, as with HTTP/1.0 [12], uncoordinated competition among the connections could exacerbate congestion, packet loss, unfairness, and latency.

Second, web data transfers happen in relatively short bursts, with intervening idle periods. It is difficult to utilize bandwidth effectively during a short burst

because discovering how much bandwidth is available requires time. Latency suffers as a consequence.

To address these problems, a new connection abstraction for HTTP, called persistent-connection HTTP (P-HTTP) [11] was developed. The key ideas are to share a persistent TCP connection for multiple web page components and to pipeline the transfers of these components to reduce latency. The main drawback of P-HTTP, though, is that the persistent TCP connection imposes a linear ordering on the web page components, which are inherently independent.
The two key elements of HTTP performance are latency and throughput:

1. Latency is measured by the RTT and is independent of the object size.

    - Connection latency is the time it takes to establish a connection.
    - Request latency is the time it takes to complete the data transfer once the connection has been established.
    - Network latency is determined by the bandwidth and the physical limitation on how quickly electrons can travel down a wire. The distance between the client and server is again an important factor determining this latency. The speed of the transmission of electrons is a physical limitation, a function of the speed of light and not subject to amendment.
    - End-user latency is the sum of all latencies including connection and request latencies, network latency due to routers, gateways, etc.

2. Throughput is a measure of how long it takes to send data, up to the carrying capacity of the data pipe. Improving throughput is simply a matter of employing faster, higher bandwidth networks and a transport protocol which can transmit to those speeds.

Unfortunately, TCP does not fully utilize the available network bandwidth for the first few round-trips of a connection. This is because modern TCP im-

plementations use slow-start [6] to avoid network congestion. The slow-start approach requires the TCP sender to open its ”congestion window” gradually, doubling the number of packets each round-trip time. TCP does not reach full throughput until the effective window size is at least the product of the round-trip delay and the available network bandwidth. This means that slow-start restricts TCP throughput, which is good for congestion avoidance but bad for short-connection completion latency. A long distance TCP connection may have to transfer tens of thousands of bytes before achieving full bandwidth. So, with the TCP NOCC implementation, the request latency is the one which can be improved considerably.

3.3.1 Description of Apache

The Apache [42] httpd server is a powerful, flexible, HTTP/1.1 compliant web server which implements the latest protocols, including HTTP/1.1 [1] and is highly configurable and extensible with third-party modules. All the changes to Apache were done to the *Apache 1.3.12* and it was run over *Linux 2.2.13*

3.4 Implementation of Pacing in Apache

The Apache Server is configured as a Web Server by using a configuration file called httpd.conf. For the convenience of the system administrator, if he wants the web server to turn off congestion control in the kernel, there is an option that has been provided in the httpd.conf file. Another option has been provided in the configuration file to enable pacing in Apache. Pacing has been implemented in Apache with UNIX timers. Essentially a burst size and burst period are defined and the data retrieved by the web server is chunked out into bursts and written in specific burst periods. A provision has been made to specify the Burst Size and the Burst Period in the httpd.conf file.

An example of the httpd.conf file which shows the options available in A-

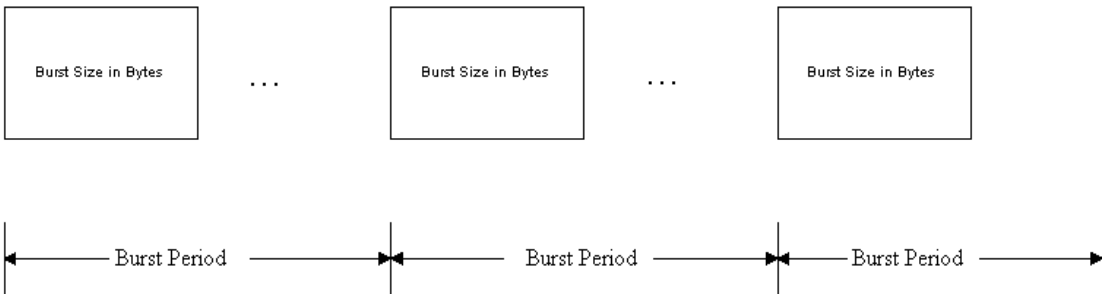

Figure 3.3: The parameters involved in Implementing Pacing in Apache

pache with Pacing.

#Pacing enabled

ExplicitRate On

#NOCC enabled

NoCongestionCntrl On

#Burst Size

ExplicitRateSize 128000

#Burst Period

ExplicitRatePeriod 10000

 # To disable NOCC

NoCongestionCntrl Off

Once the web server has done the directory walk and found the document requested for in the HTTP Get request, it starts a timer for a certain burst period (*explicit_rate_period*). It then starts transmitting the first burst of burstsize (*explicit_rate_size*) bytes. Once it is done with the transmission it waits for the timer to go off and hand it a SIGALRM. When the timer expires, it is programmed to start again for the same burst period. When the SIGALRM is delivered, the thread waiting to do the transmission captures it and the next burst of bytes is written. This procedure is repeated for the size of the document retrieved by the web server. The important thing to be taken care of here is the possibility of a burst period

being too small. Since the timer is being armed automatically when a write call is being performed, the possibility of a write() on write() call occurs. To avoid this we redirect the signal catcher to count the number of failed cycles.

Apache was also modified to accept the burst parameters namely, the burst size and burst period from the HTTP Get request. The benchmarking tool Zeus was modified to accept a few options for the burst size and burst period from the user and send the following modified HTTP Get request to Apache.

Request GET /size100K.txt HTTP/1.1
User-Agent: ApacheBench/1.1
Rate = 64000
Period = 10000
Host: 140.173.170.11
*Accept: */**

Two parameters were added to the Get request

1. Rate: which is the burst size specified in bytes
2. Period: which is the burst period which is specified in milliseconds

Apache was modified to parse the rate and period parameters and set the *explicit_rate_size* and *explicit_rate_period* parameters used in the pacing algorithm. In this way the parameters for the burst algorithm can be set by the application connecting to the web server and so the parameters are specific for a particular connection. In the ENABLE context, the application which wants to download a web page, checks the network status by sending queries to the monitoring database and then decides on the burst algorithm parameters. It then sends the HTTP Get request with those parameters. If the parameters are not specified in the Get request then the default parameters as specified in the httpd.conf file are used for the pacing algorithm.

3.5 NetSpec Overview

NetSpec [7] is a tool that was designed to simplify the process of doing network testing, as opposed to doing point to point testing. NetSpec provides a fairly generic framework that allows a user to control multiple processes across multiple hosts from a central point of control. NetSpec consists of daemons that implement traffic sources/sinks and various passive measurement tools.

NetSpec is a network-level end-to-end performance evaluation tool developed in the University of Kansas, to help in the collection of results delivered by performance experiments on the ACTS Satellite ATM Internetwork (AAI) project. The NetSpec system provides support for large-scale data communication network performance tests with a variety of traffic source types and modes. This software tool provides a simple block structured language for specifying experimental parameters. It also provides support for controlling performance experiments containing an arbitrary number of connections across a LAN or WAN. NetSpec exhibits many features that are not supported by the most often used performance tools like (ttcp, Netperf). Some of the features are parallel and serial multiple connections, a range of emulated traffic types (FTP, HTTP, MPEG, etc.) on the higher levels, three different traffic modes, scalability, and the ability to collect system level information from the communicating systems as well as intermediate network nodes. All the tests can be run on the two most widely used transport protocols namely TCP and UDP.

The basic NetSpec architecture consists of several pieces. The controller is a process that supports the user interface, which is currently a file containing a description of an experiment using a simple block structured language in which the connection is the basic unit for an experiment and via the control daemon controls the daemons implementing the test. For every connection in the experiment, the corresponding test daemons are created. These test daemons concentrate on performing the traffic related tasks (send or receive data transferred across the connec-

tion). Each daemon is responsible for its own report generation after experiment execution is complete, and measurement daemons concentrate on collecting data as accurately as possible, without having to worry about performing traffic functions. The output report is delivered to the controller via the control daemon and this is in turn displayed to the user. The communication between the controller and the daemons is achieved using an ASCII based language, which enhances portability and extensibility.

NetSpec supports three basic traffic modes: full blast mode, burst mode, and queued burst mode. These basic traffic types provide the basis for a variety of system evaluation and debugging approaches.

NetSpec has the potential to emulate [29] FTP, telnet, VBR video traffic (MPEG, video- teleconferencing), CBR voice traffic, and HTTP (World Wide Web traffic) on the application layer. This feature makes NetSpec a unique network performance evaluation tool with the ability to test system performance under traffic conditions that would be experienced in operation.

In this thesis, NetSpec was used as an application, which would turn off congestion control in the TCP stack and transmits traffic in both full blast mode and in burst mode. A comparison of the performance obtained with TCP traffic run with NetSpec with and without Congestion Control was done.

NetSpec was modified to accept a *nocc* setting from the application as part of the script. The user can specify when he wants congestion control turned off in the Linux kernel by modifying the script and that in turn helps NetSpec invoke the setsockopt() with the *TCP_NO_CONGESTION* as the parameter. In this way, we could test the performance of the *NOCC* implementation by using NetSpec as one of the applications.

Chapter 4

Evaluation

4.1 Experimental Environment and Tools Used

The transmitter omega.cairn.net is an Alpha processor with a Linux 2.2.13 kernel with the *NOCC* modifications. Both the transmitter and the receiver have a Gigabit Ethernet Acenic Card[43] capable of delivering over 950 Mbps of TCP-level data. The memory capacity on the transmitter is 1.033GB and on the receiver it is 64MB.

Module	Machine Name	Description
TCP Transmitter	omega.cairn.net	Alpha(DEC XXP1000) with the Linux-2.2.13
TCP Receiver	iss-p4.lbl.gov	Pentium Pro 200MHz
UDP Transmitter	dpss2.cairn.net	Celeron 500Mhz
UDP Receiver	dpsslx03.lbl.gov	Pentium III 450MHz

Table 4.1: Workstations used and their specifications.

Therefore, the memory bandwidth on the receiver and other host considerations like the I/O bus bandwidth (how fast the bus can read/write data from/to the network adapter) and operating system kernel issues [37] limit the maximum achievable TCP throughput to 130Mbps.

NetSpec was used for burst and full traffic generation. The *Apache* Web Server with the pacing module implemented was installed on omega.cairn.net and iss-

p4.lbl.gov was made to issue HTTP GET requests by using a tool called *zeus*.

Module	Version	Description
TCP/IP stack	Linux-2.2.13	With the *NOCC* modifications
NetSpec	NetSpec 3.1	Full Blast and Burst Tests
Web Server	Apache 1.3.12	With pacing module
TCP analyzing Tool	Tcptrace 5.1.0	To analyze tcpdump
Web Server Benchmarking Tools	Zeus and httperf-0.6	Measure performance of Apache

Table 4.2: Tools used and their specifications.

Tcpdump [39] was used to monitor the interface and the TCP flow's packets were captured. The dump output was analyzed later with a tool called *tcptrace* [40]. The output produced by tcptrace was analyzed with a plotting tool called *xplot* [41].

4.2 Experimental Parameters

The Network Connectivity of the testbeds in Lawrence Berkeley Labs (LBL) and Information Sciences Institute, East (ISIe) is as shown in the Figure 4.1.

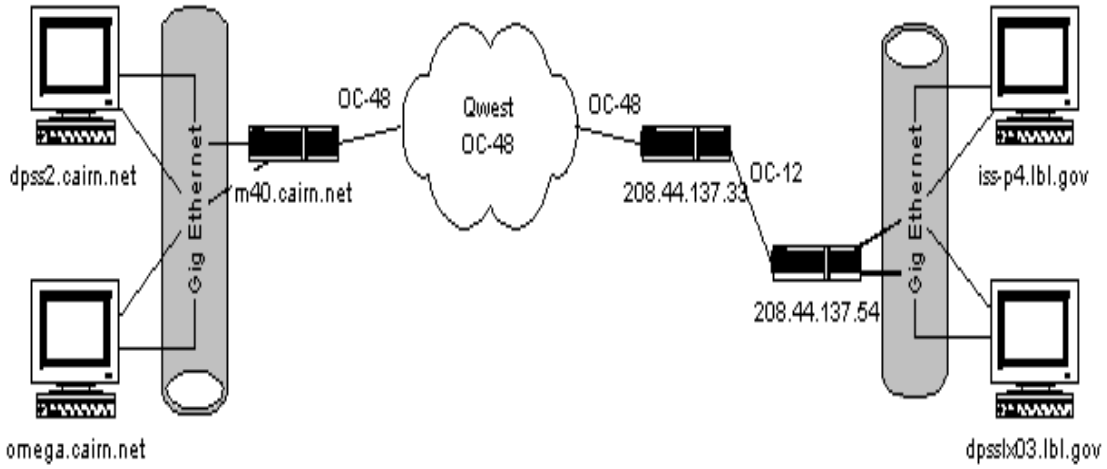

Figure 4.1: Network Topology Diagram

A traceroute from omega.cairn.net to iss-p4.lbl.gov shows the path which the packets take from the source to the destination.

[dartku@omega burst]$ /usr/sbin/traceroute iss-p4.lbl.gov
traceroute to iss-p4.lbl.gov (131.243.2.29), 30 hops max, 46 byte packets
1 m40.cairn.net (140.173.170.1) 0.531 ms 0.726 ms 0.932 ms
2 205.171.40.61 (205.171.40.61) 0.917 ms 0.662 ms 0.938 ms
3 wdc-core-02.inet.qwest.net (205.171.24.85) 0.926 ms 0.889 ms 0.939 ms
4 wdc-core-03.inet.qwest.net (205.171.24.6) 0.926 ms 0.378 ms 0.938 ms
5 hou-core-01.inet.qwest.net (205.171.5.187) 24.379 ms 24.248 ms 24.387 ms
6 hou-core-03.inet.qwest.net (205.171.23.9) 24.378 ms 24.511 ms 24.391 ms
7 lax-core-02.inet.qwest.net (205.171.5.163) 55.647 ms 55.983 ms 55.661 ms
8 lax-brdr-01.inet.qwest.net (205.171.19.42) 55.652 ms 56.354 ms 55.661 ms
9 205.171.40.58 (205.171.40.58) 57.631 ms 57.709 ms 56.641 ms
10 208.44.137.33 (208.44.137.33) 67.376 ms 68.288 ms 67.386 ms
11 208.44.137.54 (208.44.137.54) 66.401 ms 66.841 ms 66.413 ms
12 iss-p4.lbl.gov (131.243.2.29) 69.328 ms 67.406 ms 67.390 ms

In the path from LBL to ISIe, the maximum bandwidth that can be attained is that of an OC-12 link which forms the bottleneck between the two networks. The RTT on this link is ¸ 67ms as seen by the ping shown below.

[dartku@omega burst]$ ping iss-p4.lbl.gov
PING iss-p4.lbl.gov (131.243.2.29) from 140.173.170.11 : 56(84) bytes of data.
64 bytes from 131.243.2.29: icmp_seq=0 ttl=244 time=66.9 ms
64 bytes from 131.243.2.29: icmp_seq=1 ttl=244 time=67.0 ms
64 bytes from 131.243.2.29: icmp_seq=2 ttl=244 time=66.7 ms
64 bytes from 131.243.2.29: icmp_seq=3 ttl=244 time=66.5 ms

So, the BDP is,

$$Bandwidth\ Delay\ Product = 622Mbps * 67ms = 5.2093MB$$

The single most important aspect when dealing with TCP performance is socket buffer settings (SO_RCVBUF and SO_SNDBUF). In particular, it is important to notice that the default maximum settings of 64KB in a standard kernel are far from sufficient for a gigabit network. We have to increase these before trying to set the above mentioned socket buffer sizes or the limits will be truncated silently to the default. For all experiments the socket buffer sizes in the transmitter and the receiver were set to the Bandwidth Delay Product

echo 5209300 > /proc/sys/net/core/rmem_max
echo 5209300 > /proc/sys/net/core/wmem_max

Once the socket buffers have been set to the maximum value, NetSpec uses setsockopts for SO_SNDBUF and SO_RCVBUF to set it to any value within the maximum. NetSpec is the traffic generation tool which generates

- Full Blast Tests
- Burst Tests

In Full blast tests, the transmitter transmits data as fast as possible. It pumps out the specified blocksize worth of bytes and goes on to pump the next block out. This can be used to test the maximum achievable throughput on the WAN.

In burst tests, the burst size and the burst period are specified and the transmitter can be programmed to send data at a particular rate. The Figure 4.2 shows how the transmitter actually pumps out the packets. The burst size and the burst period and the duration of the test are parameters which were changed to observe the behavior of TCP with CC and *NOCC*.

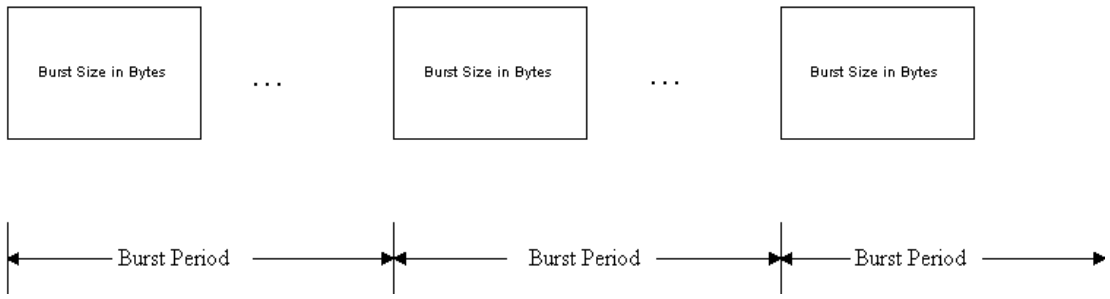

Figure 4.2: A Transmitter pumping out packets in Bursts

4.3 Performance Metrics

Slow start is the main performance bottleneck for TCP on high latency links. The slow start phase occurs in the beginning of a connection and also when there is severe congestion in the network. So, the two important phases to concentrate on are the

- Startup behavior of a TCP connection with Congestion Control (CC) and without Congestion Control (*NOCC*) and the
- Congestion Recovery Behavior with CC and *NOCC*

4.3.1 Number of Outstanding Bytes

The number of outstanding bytes in the network or the number of packets in flight form a direct measure of the Congestion Window. The transmitter of a TCP connection checks to see the *CWND* and when the outstanding bytes on the network is less than *CWND*, it pumps out more packets onto the network. In the slow start phase of TCP with CC, the number of outstanding data on the network increases exponentially as and when the receiver acknowledges data received. In the TCP with *NOCC* case, the transmitter is not limited by the *CWND* and so the number of bytes outstanding in the network increases from zero to the receiver's advertised window . This metric is analyzed in the start up phase of the TCP connection and

also in the Congestion Recovery region of long duration flows.

4.3.2 Received Throughput

In a TCP connection, the received throughput is given by

$$\text{Received Throughput} = \frac{\text{Number of Bytes Received}}{\text{Duration of Transfer}} \tag{4.1}$$

This is a direct measure of the performance the receiver gets from the Network. If there is congestion in the network, this parameter observed at the receiver is reduced and is thus a measure of the congestion in the network to the receiving application. This parameter obtained in the TCP with CC case is compared with that obtained from the TCP with *NOCC* case in both long and short duration flows. In the short duration TCP with CC flow, the entire flow's duration is dominated by TCP's slow start behavior and so the transmitter and receiver never get to utilize the network to its maximum capacity.

4.3.3 Offered Load

When we perform burst tests with NetSpec, the transmitter sends bursts of packets into the network. A flow's duration is divided into many seconds and each second is in turn divided into a certain number of burst periods. In each burst period, a Burst Size worth of bytes is written onto the network. So, effectively the transmitter is made to pump out packets at a certain designed rate and this is called the offered load on the network and it is given by:

$$\text{Number of Burst Cycles/second} = \frac{1 \text{ second}}{\text{Burst Period}} \tag{4.2}$$

$$\text{Offered Load in bps} = \text{Burst Size} * 8 * \text{Number of Burst Cycles/second} \tag{4.3}$$

4.3.4 Response Time

In the Apache Web Server, an important metric to be minimised is the response time. This is the duration of time the client which sends a HTTP Get request to the Web Server spends waiting before it can produce the requested web page to the end user. This metric is analyzed in the TCP with *NOCC* case and is compared with the TCP with CC case, which is what all current implementations of web servers use. In this way we can see if the current implementation would fare better than what is already existing on long latency links.

4.4 Experiment Scenarios

NetSpec was used to create both long and short duration flows. The first scenario to be tested is the startup behavior of TCP with CC and with *NOCC* and this can be tested with the transmitter as omega.cairn.net and the receiver as iss-p4.lbl.gov and with all the intermediate routers. This is a congestion-free test scenario.

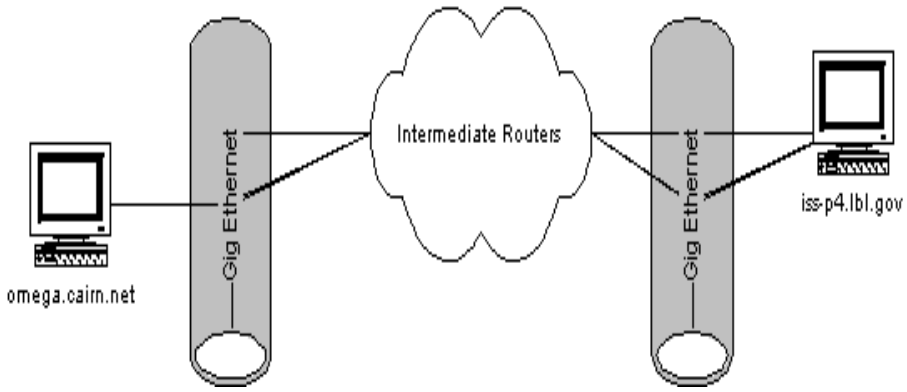

Figure 4.3: Congestion-Free Test Scenario with Two testbeds

The second scenario is to test the Congestion Recovery behavior of TCP with CC

and with *NOCC*. For this we need to have a TCP flow between two machines and also a UDP flow between two other machines to congest the Network.

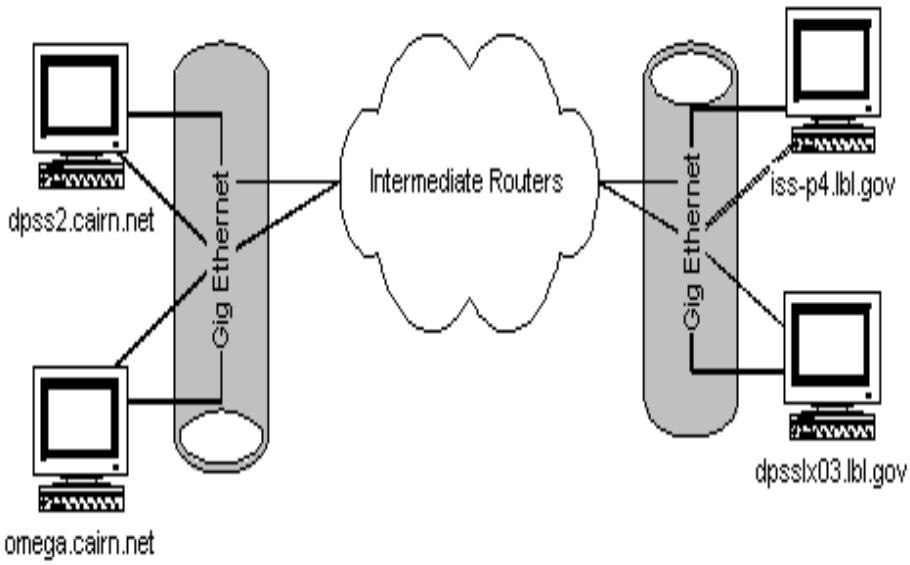

Figure 4.4: Four Testbed Scenario to evaluate Performance during Congestion

The maximum buffer size is set to the Bandwidth Delay Product on all these machines.

4.5 Scenarios and Results with NetSpec

4.5.1 Congestion-free Performance and Results

These tests were run in a congestion-free environment to analyze the startup behavior of TCP with CC and to observe the performance benefits which can be obtained through TCP with *NOCC* due to the lack of a startup phase.

4.5.1.1 Start up behavior of TCP with CC for Full Blast Traffic

In this test case, NetSpec was run in full blast mode. The interface was put in promiscuous mode and all the packets belonging to this flow are captured by tcp-

dump. It was later analyzed with tcptrace. The plot of the *CWND* or the number of bytes outstanding on the network with regard to the time is obtained with xplot.

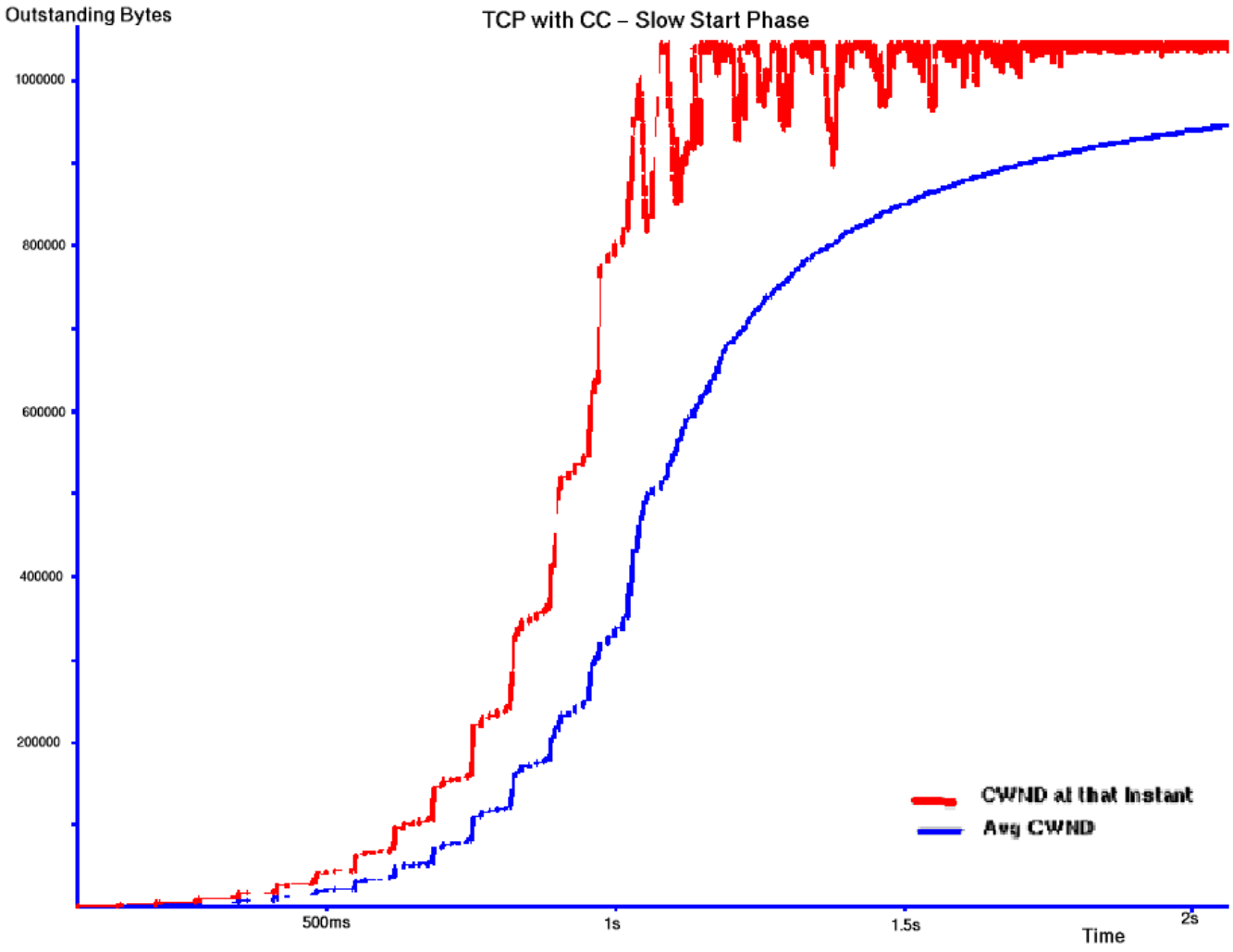

Figure 4.5: *CWND* vs Time in TCP with CC - Slow Start Phase

The first 2 seconds have been magnified here to show how many round trip times it takes for the *CWND* to ramp up to the steady state value in the TCP with CC case. We can see that for the first one second, the *CWND* grows slowly with each RTT and if the duration of a flow is small (of the order of 2-3s) then the flow

never utilizes the available bandwidth from the network.

4.5.1.2 Start up behavior of TCP with *NOCC* with Full Blast Traffic

In this test, we can see with Full Blast NetSpec traffic, that the maximum limit on the congestion window is the receiver's advertised window. The transmitter is able to send an entire *CWND* (receiver's advertised window) worth of packets as soon as the connection is active.

The Figure 4.6 shows how the instantaneous *CWND* shoots up to the maximum as soon as the transmitting application starts writing.

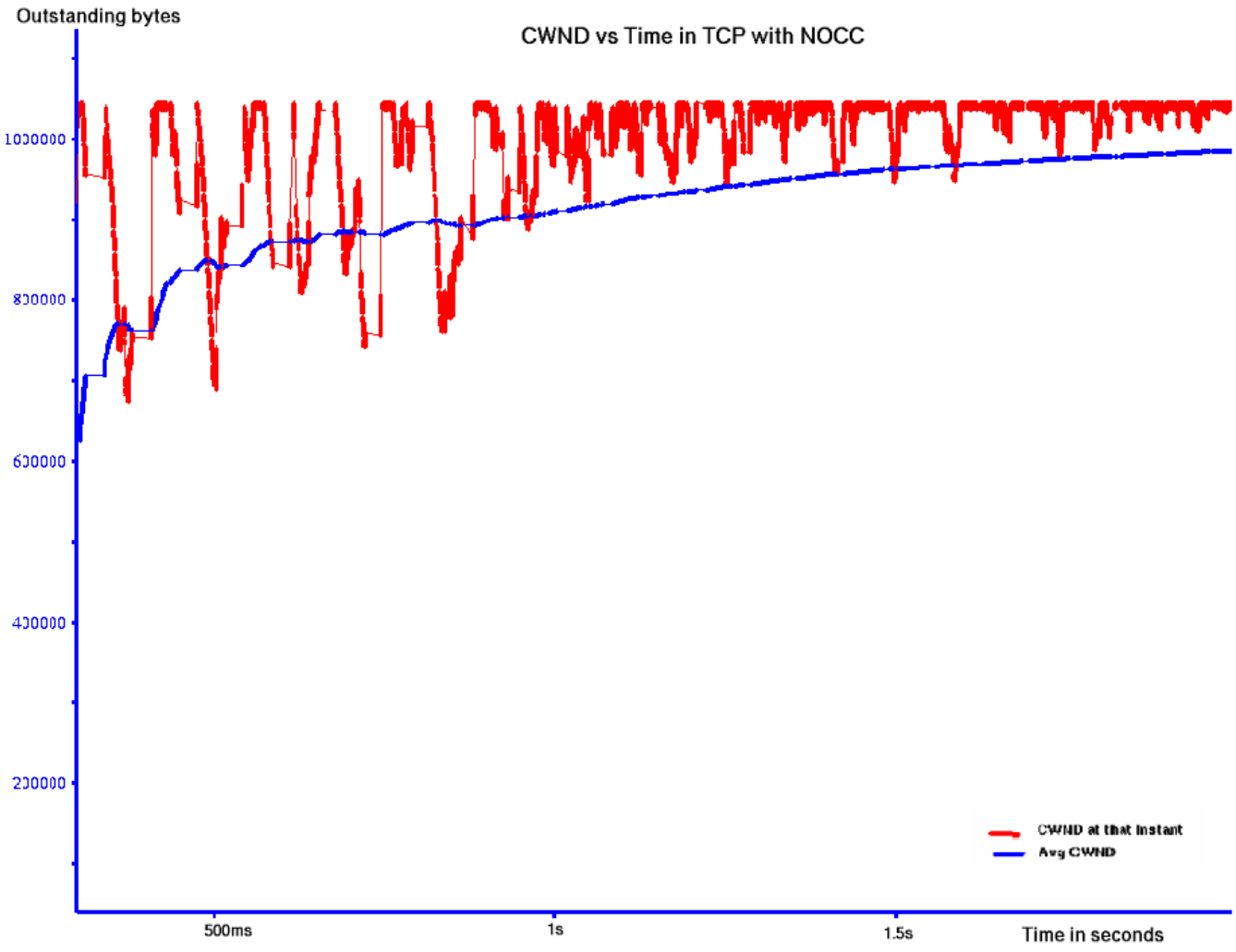

Figure 4.6: *CWND* vs Time in TCP with *NOCC*

4.5.1.3 Long Duration Paced Traffic (Burst Period = 10ms)

This test was conducted with NetSpec running in paced mode. All the tests were conducted with a Burst Period = 10ms. The burst sizes were varied through a range of values and the transmitted and received throughputs were observed.

Burst Sizes = 8KB, 16KB, 32KB, 64KB, 128KB, 512KB

Offered Load = 6.4Mbps, 12.8Mbps, 25.6Mbps, 51.2Mbps, 102.4Mbps, 204.8Mbps

Burst Period = 10ms

Duration = 10s

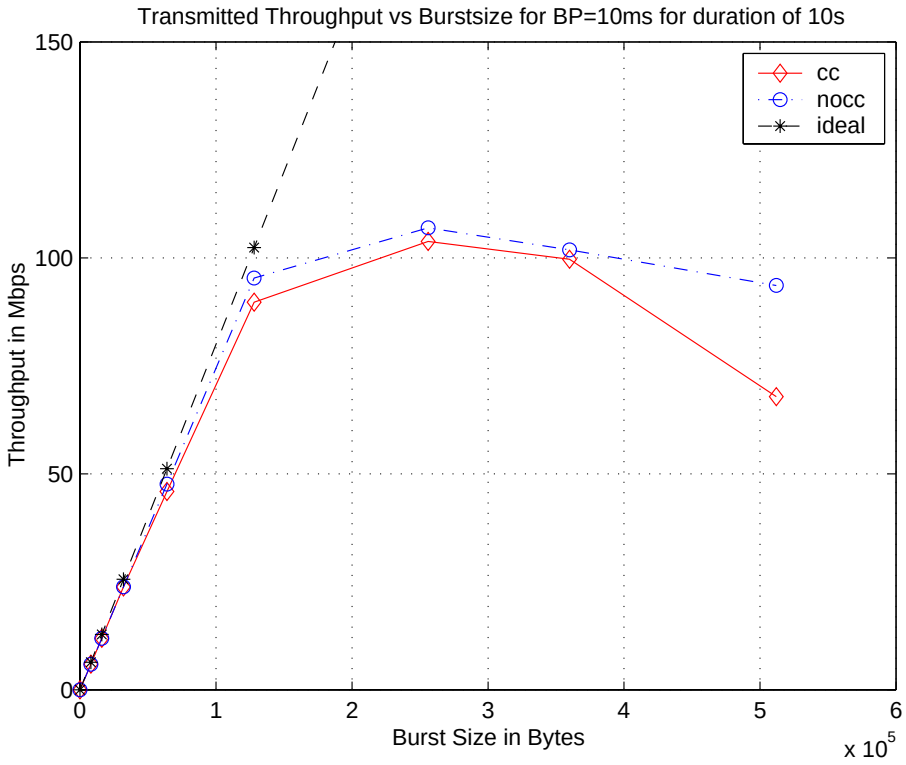

Figure 4.7: Transmitted Throughput for Different Burst Sizes for a Long Duration Flow with BP=10ms, RTT=67ms, BW=622Mbps

For each burst size the ideal or required throughput or the offered load is calculated as given by Equation 4.3 and plotted. The first set of experiments were conducted with TCP with CC and with all the above mentioned burst sizes. The

aim of this experiment is to prove that as long as the burst sizes are small, TCP with CC is able to send a burst worth of packets in a single burst period. As the burst size increases, the number of *failed burst cycles*[*] increase because of the inherent limitation of the *CWND* and so the transmitter is not able to transmit more than the *CWND* worth of bytes and because of this limitation, there is a reduction in the throughput.

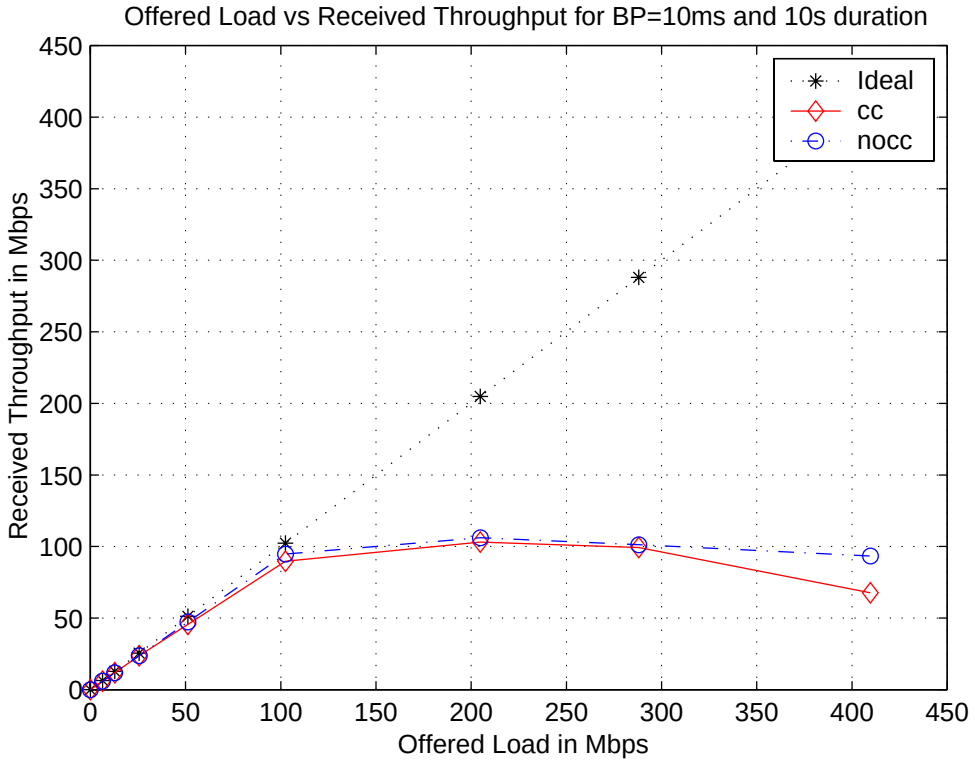

Figure 4.8: Offered Load vs Received Throughput for a Long Duration Flow with BP=10ms, RTT=67ms, BW=622Mbps

The same set of experiments were conducted with TCP with congestion control turned off (*NOCC*) and with the same set of burst sizes. In the case of TCP with *NOCC*, the application is not limited by the *CWND* maintained by TCP and so it is able to handle huge burst sizes with more ease and so the effective throughput

[*] When the current burst period times out and the transmitter hasn't yet transmitted the burst size of bytes in that burst cycle

observed with TCP with *NOCC* is more than TCP with CC. This is also plotted in the same graph.

From the Figure 4.7 it is seen that TCP with CC and *NOCC* perform in a similar way for small burst sizes. As the burst size increases, the throughput from TCP with *NOCC* is better.

For the same set of tests, the received throughput is plotted against the Offered Load and the performance of TCP with *NOCC* is observed to be better than TCP with CC in the high burst size region.

4.5.1.4 Short Duration Paced Traffic (Burst Period = 10ms)

This test is carried out in a similar way as the previous test but for a short duration. The duration is set to 2s.
Burst Sizes = 8KB, 16KB, 32KB, 64KB, 128KB, 512KB
Burst Period = 10ms
Duration = 2s

On a cross-country WAN link which has an RTT of nearly 70ms, the *CWND* takes more than a second to open out and reach a steady state value. In this case, the performance benefits of TCP with *NOCC* is pretty marked because an application which turns off congestion control in the kernel can start off by sending up to the receiver's advertised window worth of packets and this is a significant performance boost in terms of throughput.

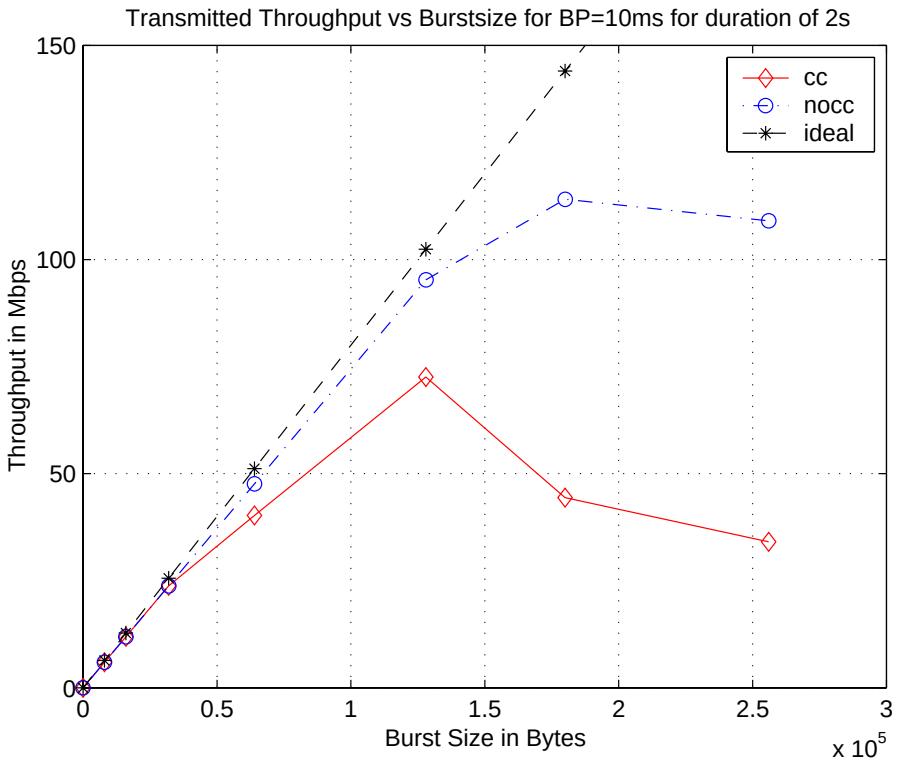

Figure 4.9: Transmitted Throughput for Different Burst Sizes for a Short Duration Flow with BP=10ms, RTT=67ms, BW=622Mbps

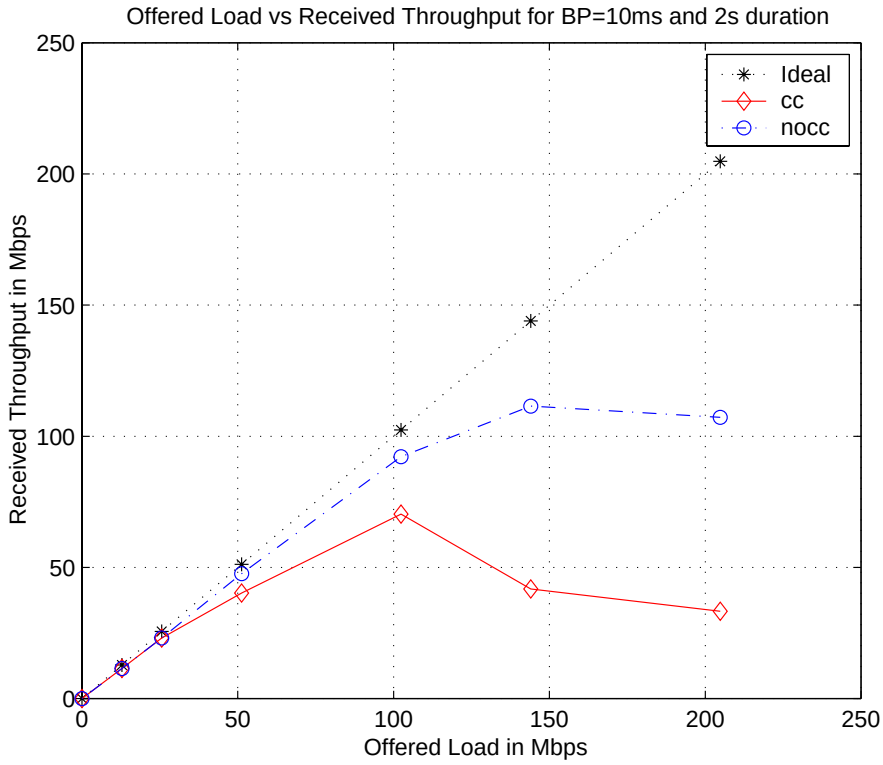

Figure 4.10: Offered Load vs Received Throughput for a Short Duration Flow with BP=10ms, RTT=67ms, BW=622Mbps

So, the same tests as the previous test case are performed for different burst sizes. The ideal throughput, throughput with TCP with CC and throughput with TCP with *NOCC* are plotted and it is seen than for a short duration flow, the performance of TCP with *NOCC* is much better than TCP with CC. The results are significant for TCP with *NOCC* in a short duration flow.

As in the previous test case, the received throughput is plotted against offered load as shown is Figure 4.10 and the performance of *NOCC* is seen to be better than CC. This is more pronounced in the case of short duration flows because in the TCP with CC case, the *CWND* does not even reach the steady state value for the application to achieve good throughputs.

4.5.1.5 Comparison of Throughputs for Short and Long Duration Flows

This is a summary of the previous two test cases. This Figure 4.11 plots the throughputs obtained for short and long duration flows in the CC and *NOCC* case. In the long duration flow, the performance of the TCP with CC flow stabilizes because the *CWND* grows out of the slow start ramping period and reaches a steady state value. Once TCP reaches this steady state, it yields good throughput results.

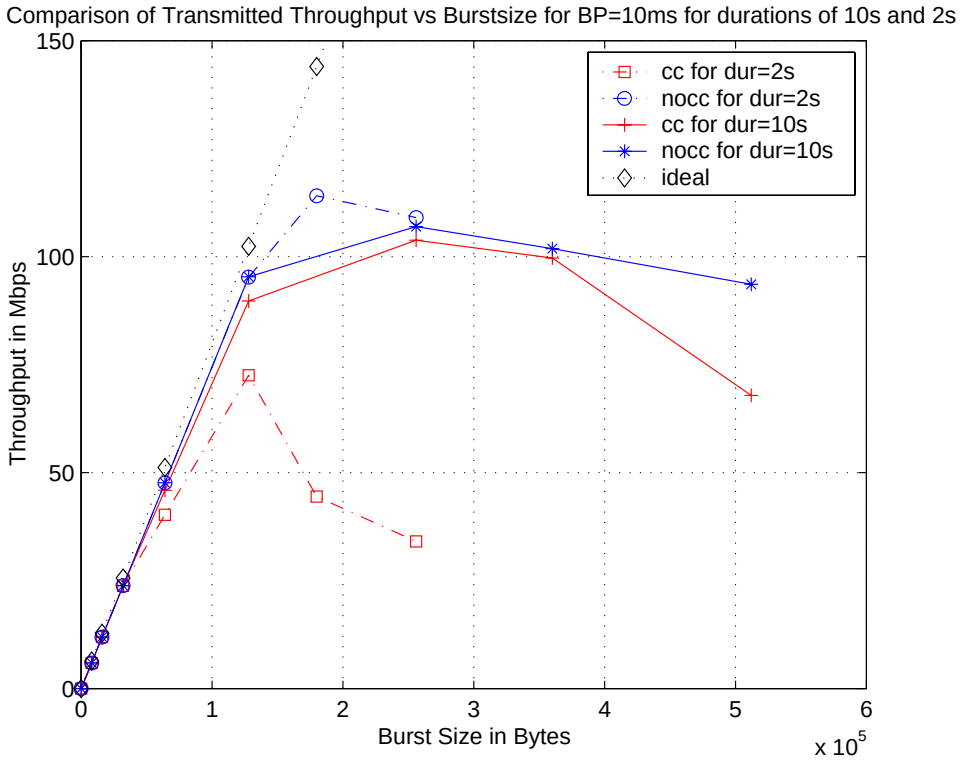

Figure 4.11: Comparison of Transmitted Throughput for Different Burst Sizes for Short and Long Duration Flows with BP=10ms, RTT=67ms, BW=622Mbps

So, if the flow's duration is more, TCP moves out of the slow start phase and makes up for the less throughput yielded in the start of the connection. Though TCP with *NOCC* performs better than CC in the long duration flow especially in the high burst size region, its performance benefits are marked in the short duration flow.

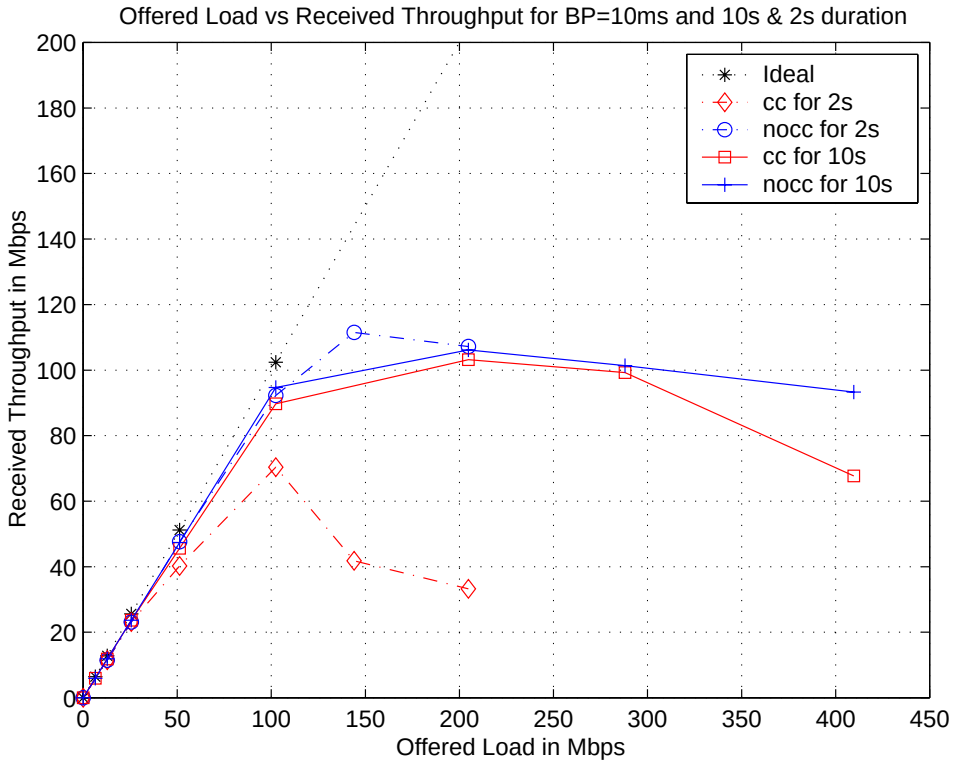

Figure 4.12: Offered Load vs Received Throughput for Long and Short Duration Flows with BP=10ms, RTT=67ms, BW=622Mbps

This is seen clearly in Figure 4.12 where the received throughputs are plotted. The performance of a short duration *NOCC* flow is similar in performance to a long duration *NOCC* flow. But a short duration CC flow performs badly when compared to a long duration CC flow. This is because TCP reaches steady state in a long duration flow and yields good performance benefits. So, the performance of *NOCC* is significantly better than a short duration CC flow. So, in effect over a short duration, *NOCC* can be very effective when compared to CC.

4.5.1.6 Long Duration Paced Traffic (Burst Period = 5ms)

The next set of experiments were conducted with a smaller burst period than the previous test case. In effect we are trying to increase the amount of data sent out per second. Since the BP is less, failed burst cycles start occurring early in the burst size region.

Burst Sizes = 8KB, 16KB, 32KB, 64KB, 128KB

Offered Load = 12.8Mbps, 25.6Mbps, 51.2Mbps, 102.4Mbps, 204.8Mbps

Burst Period = 5ms

Duration = 10s

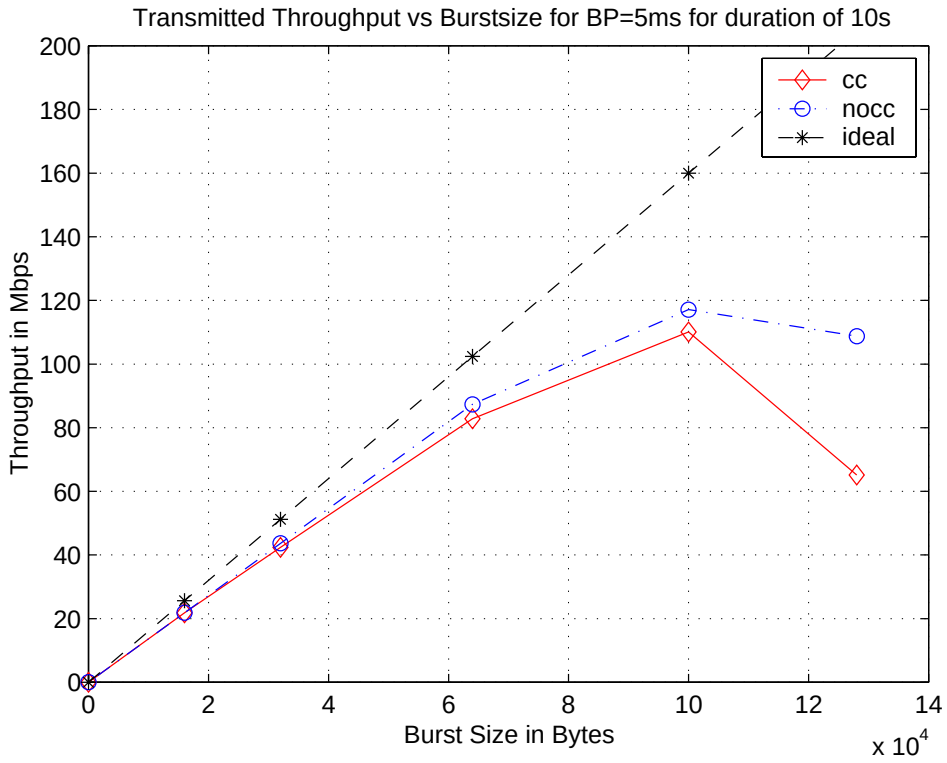

Figure 4.13: Transmitted Throughput for Different Burst Sizes for a Long Duration Flow with BP=5ms, RTT=67ms, BW=622Mbps

This Figure 4.13 plots the transmitted throughputs obtained for a long duration flow with a burst period of 5ms. The test is performed for various burst sizes

and the throughput for TCP with *NOCC* is plotted along with the throughput for TCP with CC. It is seen that TCP with *NOCC* performs better than TCP with CC especially for large burst sizes.

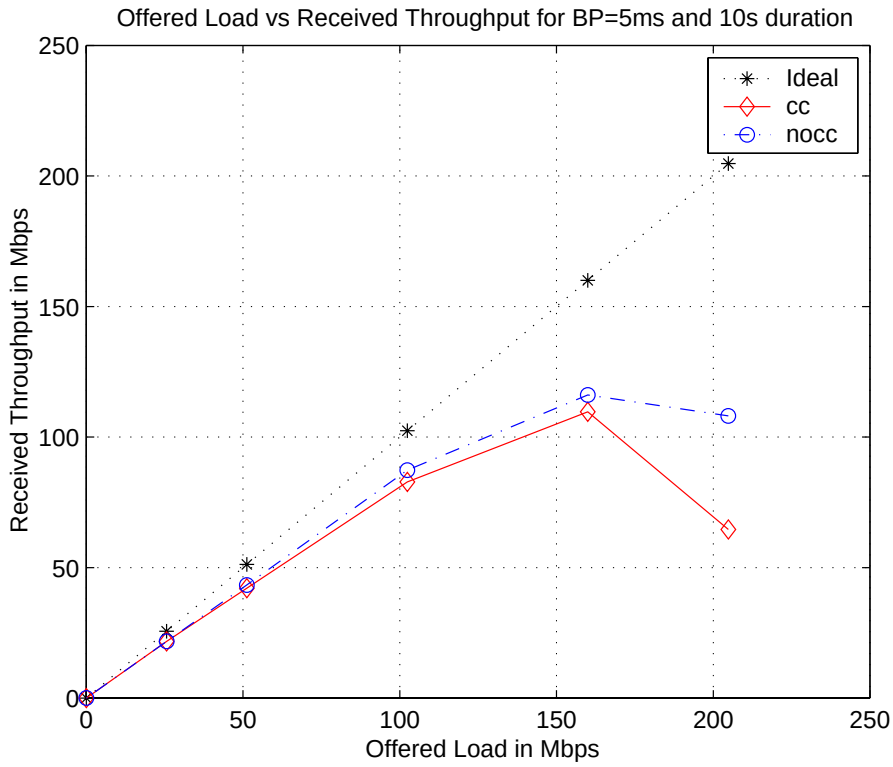

Figure 4.14: Offered Load vs Received Throughput for a Long Duration Flow with BP=5ms, RTT=67ms, BW=622Mbps

Figure 4.14 plots the received throughputs against offered load and the same effect as all the previous test cases is seen here and *NOCC* performs better for large burst sizes.

4.5.1.7 Short Duration Paced Traffic (Burst Period = 5ms)

These experiments were conducted with a burst period = 5ms and a duration of 2s.

Burst Sizes = 8KB, 16KB, 32KB, 64KB, 128KB

Burst Period = 5ms

Duration = 2s

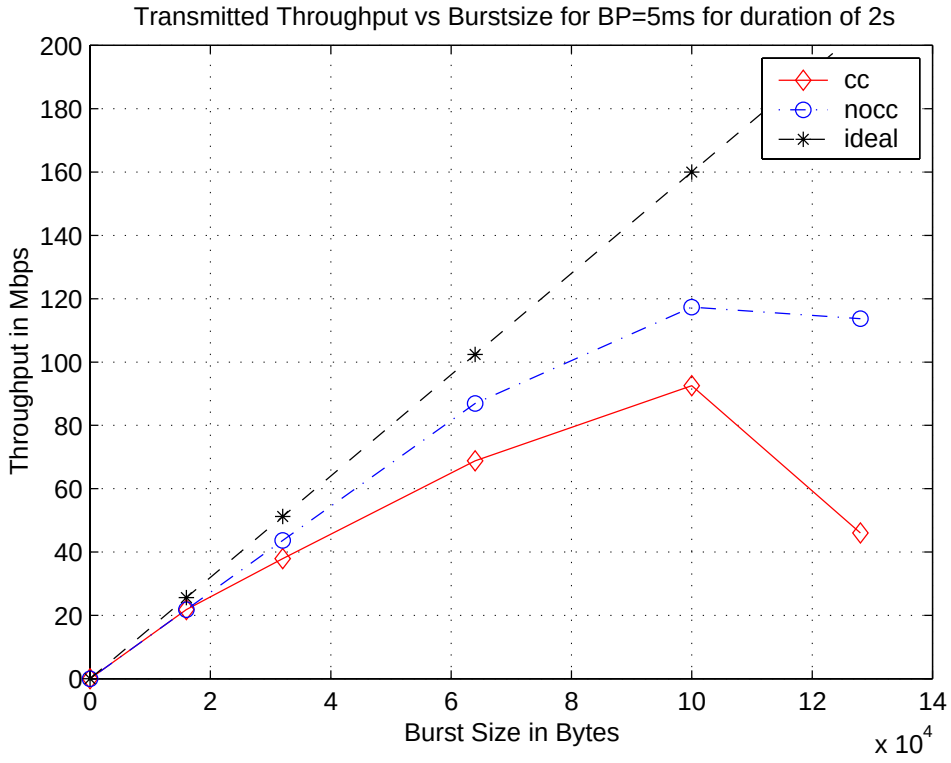

Figure 4.15: Transmitted Throughput for Different Burst Sizes for a Short Duration Flow with BP=5ms, RTT=67ms, BW=622Mbps

The Figure 4.15 shows that the transmitted throughputs for *NOCC* is again better than TCP with CC. The Figure 4.16 shows the received throughputs plotted against offered load. As observed in the previous test cases, the performance of *NOCC* is better than CC on a short duration flow.

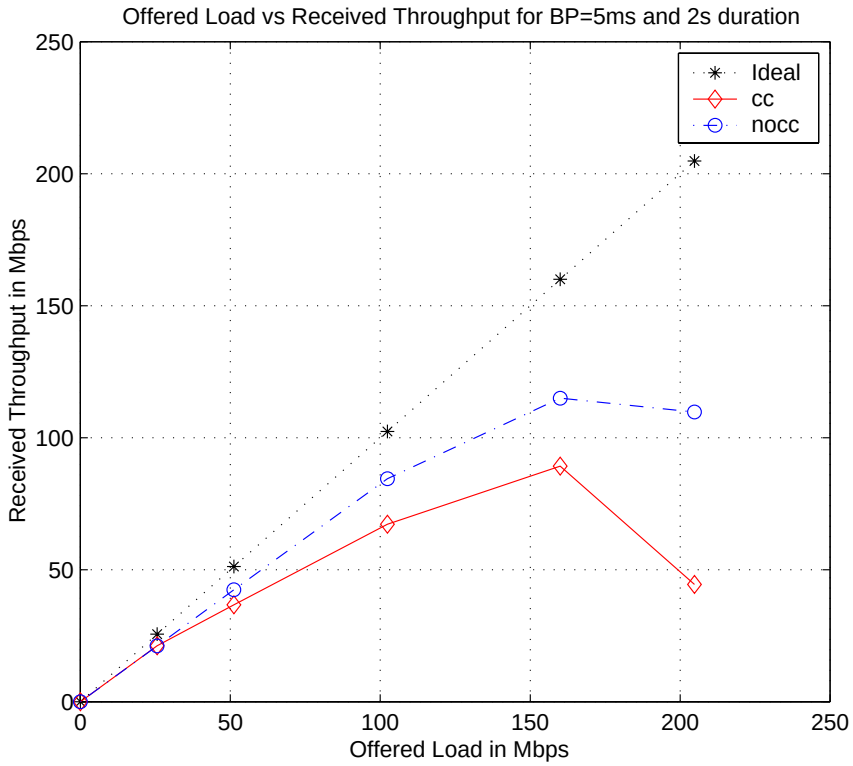

Figure 4.16: Offered Load vs Received Throughput for a Short Duration Flow with BP=5ms, RTT=67ms, BW=622Mbps

4.5.1.8 Comparison of Throughputs for Short and Long Duration Flows

As in the case where the burst period was 10ms, the short duration *NOCC* flow performs much better than TCP with CC flows.

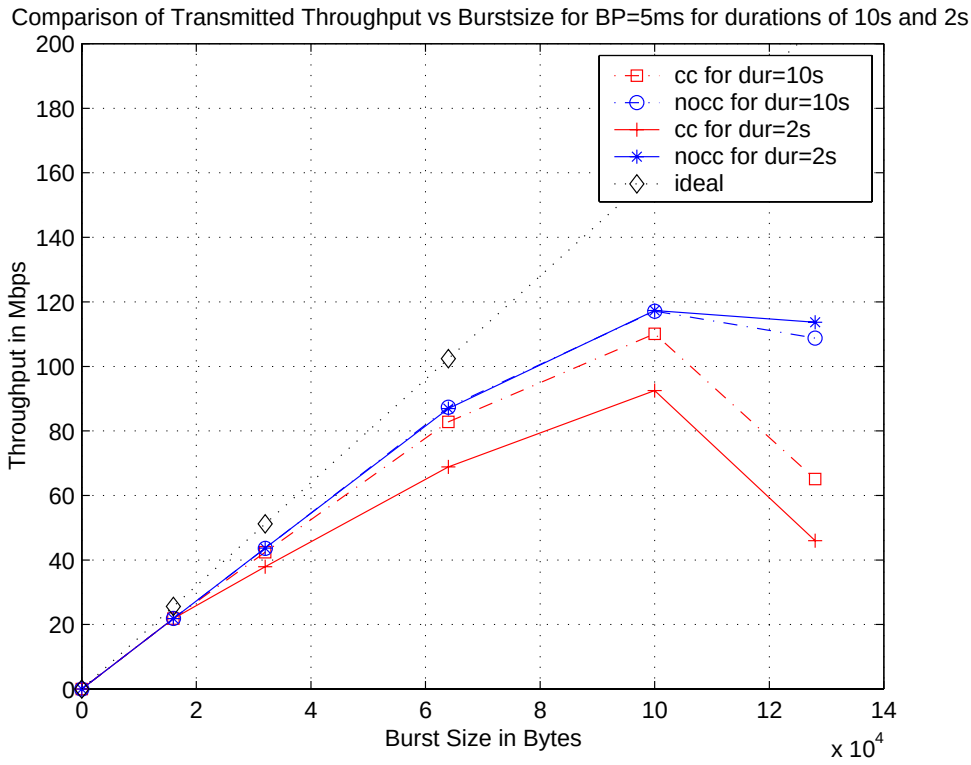

Figure 4.17: Comparison of Transmitted Throughput for Different Burst Sizes for Short and Long Duration Flows with BP=5ms, RTT=67ms, BW=622Mbps

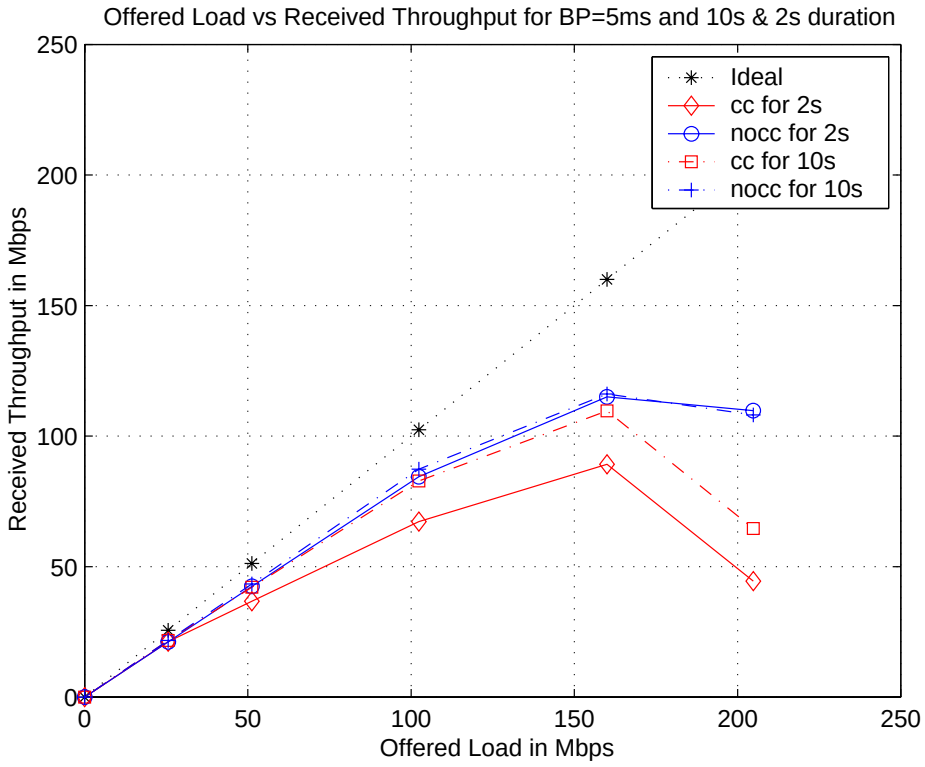

Figure 4.18: Offered Load vs Received Throughput for Short and Long Duration Flows with BP=5ms, RTT=67ms, BW=622Mbps

The above Figure 4.18 plots the received throughputs for long and short duration CC and *NOCC* flows. As before, the performance of *NOCC* short duration flow is significantly better than a short duration CC flow.

4.5.1.9 Conclusions

It is seen that TCP with *NOCC* performs similarly for short and long duration flows. This is as expected because TCP with *NOCC* does not need to reach any steady state to produce good results. It starts off by transmitting upto the receiver's advertised window and so performs well consistently irrespective of short and long duration flows. TCP with CC performs comparable to *NOCC* when the flow's duration is long enough for *CWND* to reach the receiver's advertised window (steady state value). On a short duration flow, TCP with CC does not have

sufficient time to let the *CWND* open up fully and so does not yield good throughputs.

4.5.1.10 Long Duration Instantaneous Transmitted Throughputs

This experiment is run by using NetSpec to generate burst traffic with a burst size of 128Kbytes.

Burst Size = 128KB

Burst Period = 10ms

Duration = 10s

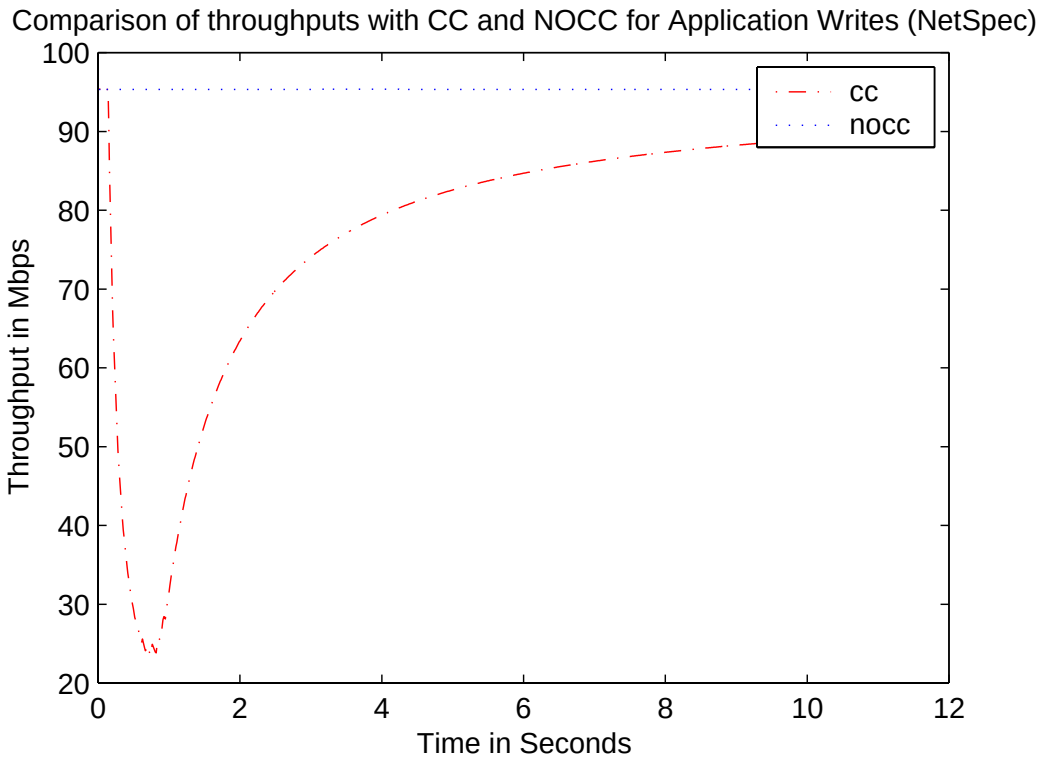

Figure 4.19: Instantaneous Tranmitted Throughputs for Long Duration Flow with BP=10ms and BS=128Kbytes, RTT=67ms, BW=622Mbps

The write durations are instrumented in NetSpec and so, NetSpec stores timestamp information when it does a write. So, in order to avoid unrealistic

spikes in the throughput graph, the throughput was sampled at 50ms durations and the number of bytes transmitted in each burst cycle is plotted against time. It is seen in Figure 4.19 that in the *NOCC* case, NetSpec is able to write the data onto the network without any failed cycles and so the transmitter is able to achieve the expected throughput. A failed burst cycle is one in which the current burst period times out and it hasn't yet transmitted the burst size of bytes in that burst cycle and to prevent a write() on write() we associate the signalhandler with a function that counts the number of failed burst periods.

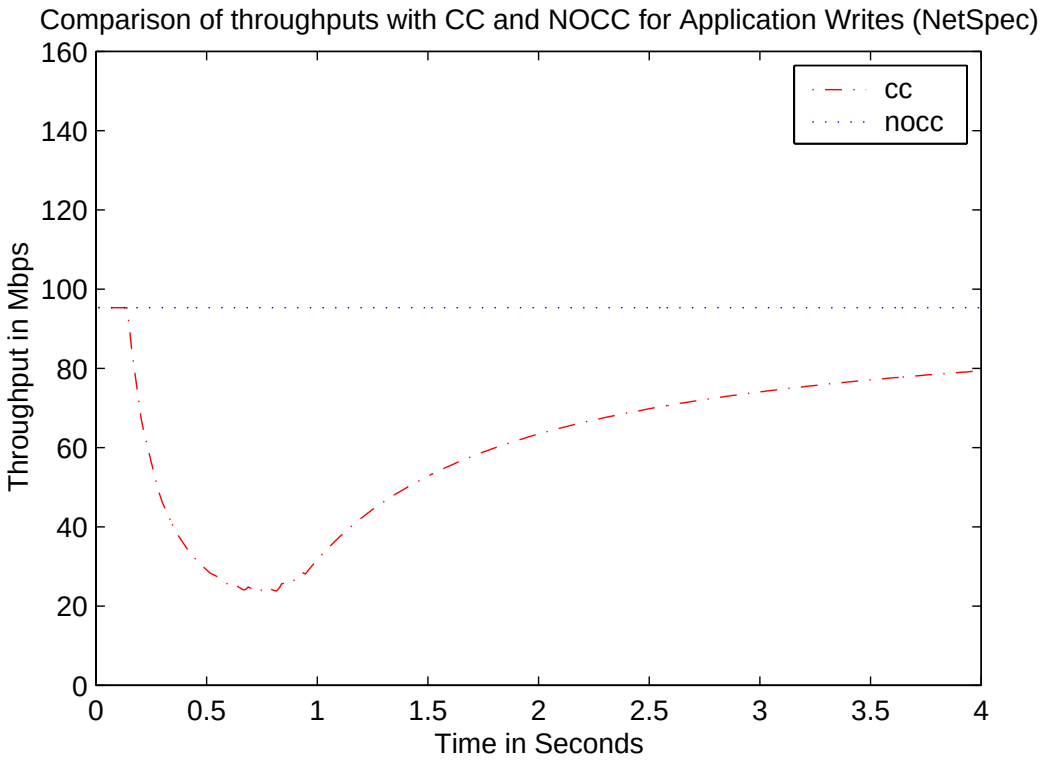

Figure 4.20: Instantaneous Tranmitted Throughputs for Long Duration Flow with BP=10ms and BS=128Kbytes (Magnified)

In the TCP with CC case, initially because of the huge socket buffer allocation, NetSpec's writes succeed and all the data is copied onto the socket buffers but eventually due to the gradual or rather slow increase in the *CWND* (which in turn implies that the number of bytes actually sent out on the network is small),

the socket buffers get filled and so NetSpec's writes fail and this leads to failed burst cycles. So, during the ramp up of the *CWND* in the slow start phase, as seen in Figure 4.20 there is a dip in the throughput and it gradually increases and the throughput steady states.

4.5.1.11 Long Duration Instantaneous Received Throughputs

This experiment is the same as the previous test case but the instantaneous received throughputs are plotted.

Burst Size = 128KB

Burst Period = 10ms

Duration = 10s

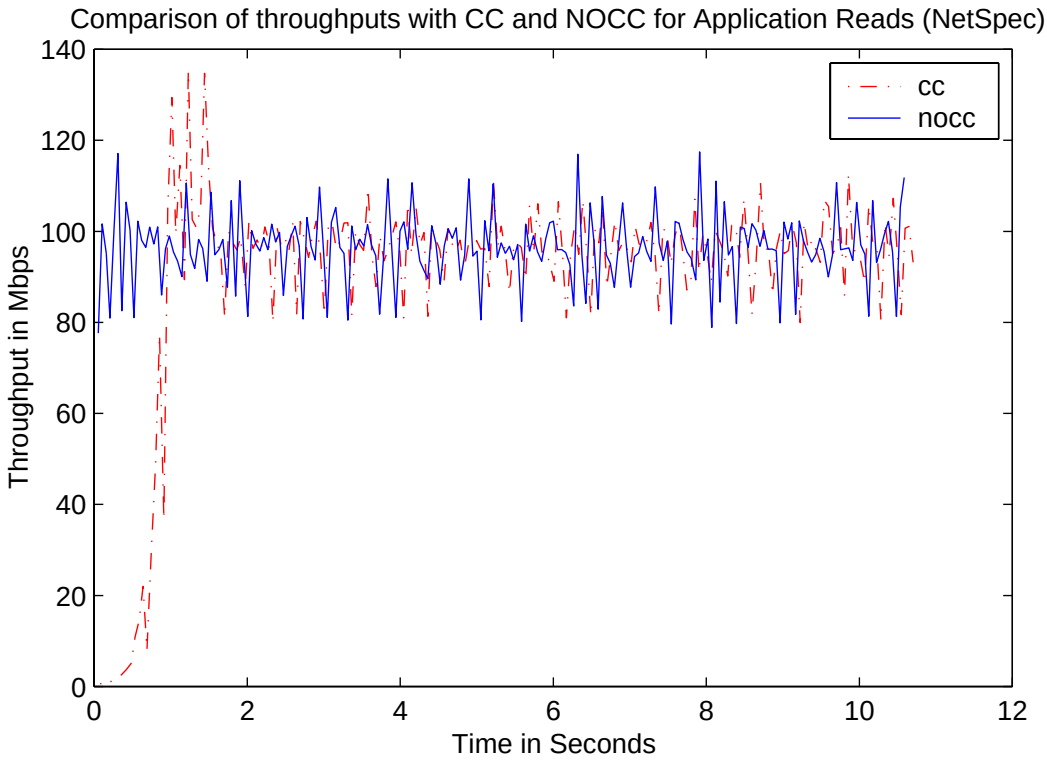

Figure 4.21: Instantaneous Received Throughputs for Long Duration Flow with BP=10ms and BS=128Kbytes, RTT=67ms, BW=622Mbps

The startup behavior in the TCP with CC case is very prominent and it is easily seen that *NOCC* has no such phase and it succeeds in getting over the startup phase as experienced by TCP with CC.

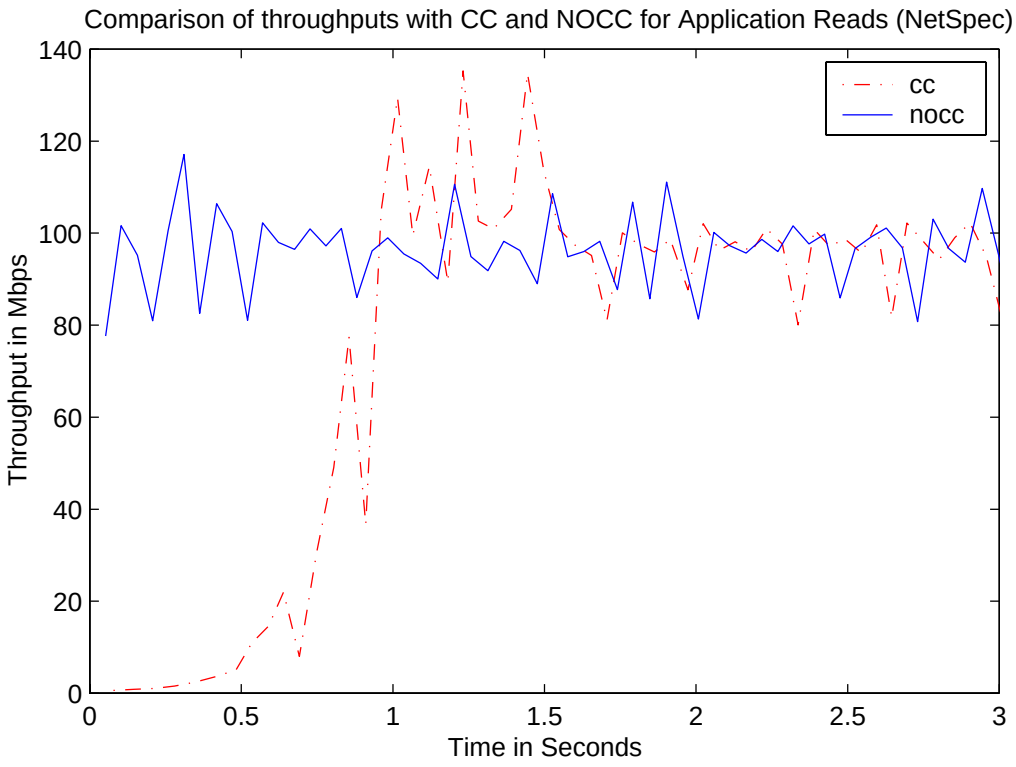

Figure 4.22: Instantaneous Received Throughputs for Long Duration Flow with BP=10ms and BS=128Kbytes (Magnified)

Figure 4.22 is a magnified plot to emphasize the performance reduction due to the startup phase exhibited by TCP with CC and this graph also shows how TCP with *NOCC* overcomes it.

4.5.1.12 Conclusions

Analyzing the transmitted and received throughputs in both CC and *NOCC*, we can clearly see that *NOCC* is able to get over the startup phase problem which we experience with CC. This could go a long way in improving the performance that we can achieve from short duration flows which are dominated by the startup

phase of CC.

4.5.2 Congestion Recovery Behavior

In the case of TCP with CC, the congestion recovery behavior is either the Congestion Avoidance phase or slow start behavior. In normal TCP, as the *CWND* is doubling every RTT in the slow start phase, there might be a packet loss detected in a particular RTT. The lost segment is taken as an indication of congestion in the network. If this lost segment is retransmitted with the Fast Retransmit Algorithm kicking in at the transmitter due to the reception of three duplicate ACKs, then once the loss has been recovered the congestion avoidance phase sets in. In this phase, the *ssthresh* is set to half the value of *CWND* at the time the loss occurred and the *CWND* starts to grow by $1/CWND$ with every ACK it receives. Eventually, this adds roughly one segment to the value of *CWND* every RTT.

If there are multiple losses and the Fast Retransmit Algorithm is unable to recover from the loss, then a retransmission timeout occurs and the lost segment is retransmitted. When a retransmission timeout occurs, TCP does a slow start with the *ssthresh* set to half the value of *CWND* at the time the loss occurred.

An important aspect to be considered on long delay networks is that the transport protocol must handle retransmissions gracefully and not drain the entire pipe as there may be several hundred packets in transit. TCP handles retransmissions better with the Selective Acknowledgement provision.

4.5.2.1 UDP Congestion Scenario with Four Testbeds

To test the congestion recovery behavior of *NOCC* as compared to TCP with CC, the second test scenario as shown in Figure 4.4 is used. Here, four machines were used to run the experiment. A UDP flow was run between two machines (dpss2.cairn.net and dpsslx01.lbl.gov) and the TCP flow was run between omega.cairn.net and iss-p4.lbl.gov. This experiment was conducted to simulate

the event of a single bit error occurring with a ”very” small congestion event. The congestion event is created with a UDP flow. Multiple trials of this experiment was done to find a burst size and burst period which could induce the *”minimum”* congestion event to analyze how TCP with CC reacts adversely in the event of the most minimum congestion. The burst size was varied and the number of bytes in a single burst period was manipulated with another parameter called *repeats*[*] in NetSpec. The NetSpec scripts are produced in the Appendix.

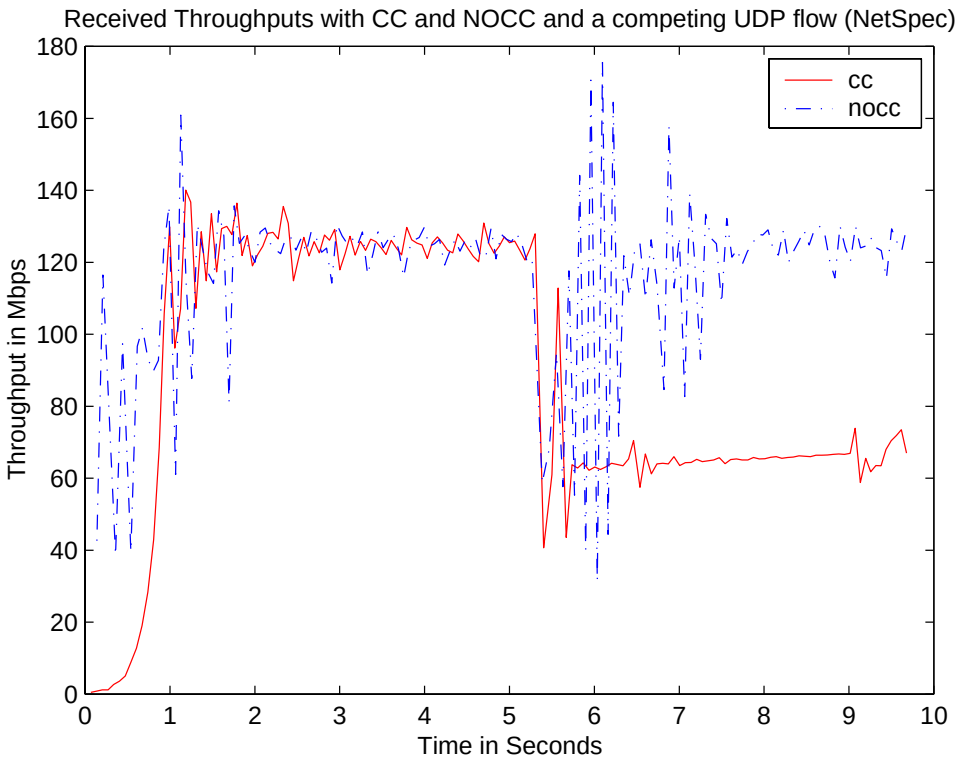

Figure 4.23: Comparison of Received Throughputs for CC and *NOCC* competing with a UDP flow, RTT=67ms, BW=622Mbps

The number of retransmissions occurring in each connection is printed out along with the NetSpec report using the setsockopt() with the parameter TCP_TOTAL_REXMITS. The burst size and the number of repeats to get zero re-

[*]This is a parameter in NetSpec which can be used to send multiple burst sized packets in one burst period.

transmits was found out and the number of repeats is increased slowly to increase the number of bytes sent out in the burst period and the point where it starts producing retransmissions was noted. Thus after multiple trials a single burst of UDP packets was injected to cause a minor congestion event. The experiment was performed with a TCP with CC flow competing with a UDP flow and a TCP with *NOCC* flow competing with the same UDP flow to compare the behavior of the two under the event of congestion.

Figure 4.23 shows the instantaneous received throughputs sampled at 60ms intervals. A congestion event occurs at about 5-6 seconds and we can see that TCP with CC recovers from a loss and halves the *CWND* and does congestion avoidance and the throughput rises slowly after halving. On the other hand we can see that though TCP with *NOCC* is affected it does not do congestion avoidance and so it is able to achieve good throughputs. This shows that even the most minimal congestion event triggers the congestion avoidance phase in TCP with CC. If the loss in a segment occurs because of reasons other than congestion like bit errors caused due to the link like in Satellite links, TCP assumes that it is an indication of congestion and kicks in the congestion avoidance phase, which significantly reduces performance.

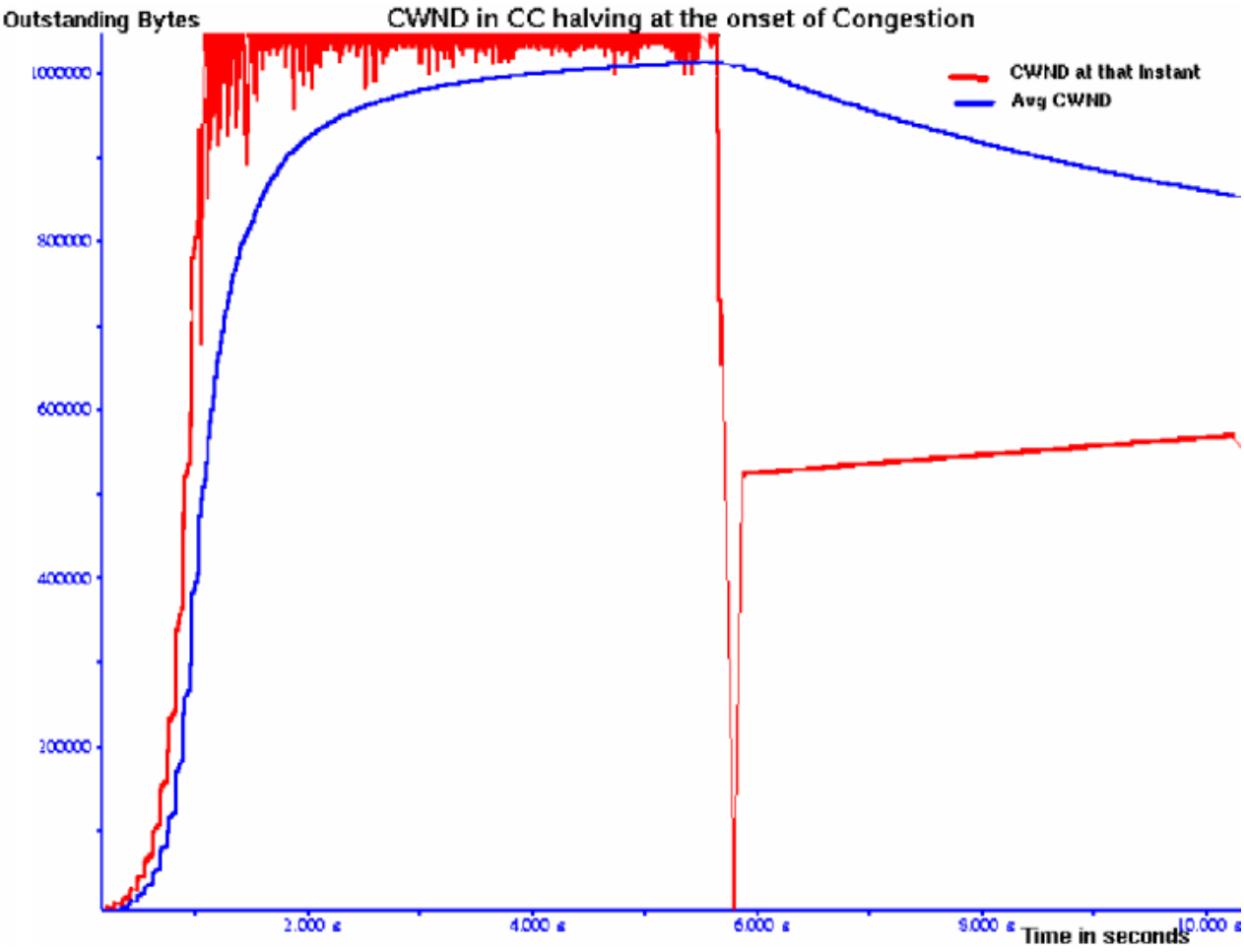

Figure 4.24: *CWND* vs Time in CC case when affected by a UDP flow

Figure 4.24 shows the *CWND* halving at the onset of congestion and from the point of congestion, congestion avoidance takes over with the halved *CWND*.

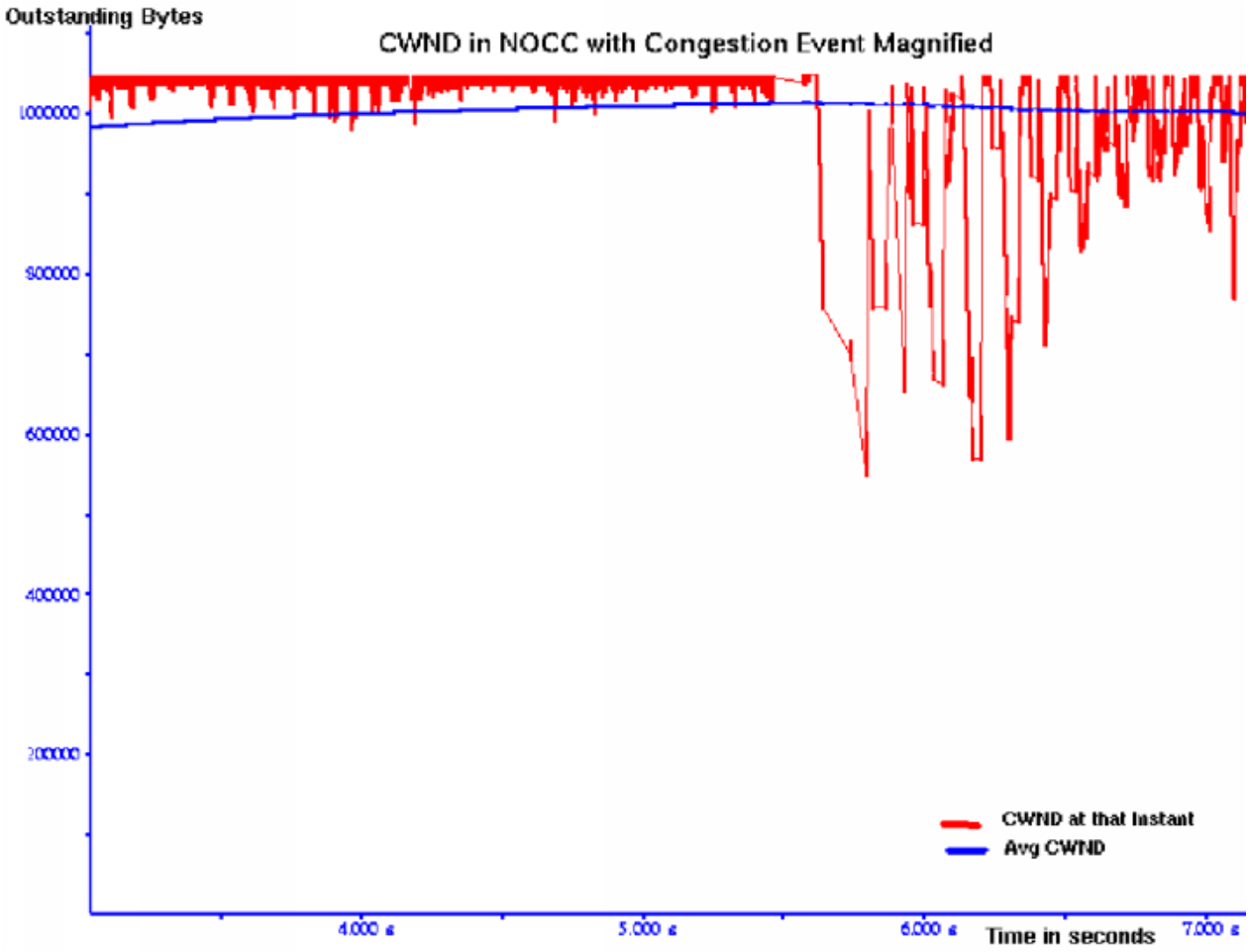

Figure 4.25: *CWND* vs Time in NOCC case when affected by a UDP flow

Figure 4.25 shows TCP with *NOCC* reacting to a 'minor' congestion event simulating a single bit error loss.

4.5.2.2 UDP Congestion Scenario with Periodic Congestion

For this experiment the test scenario as shown in Figure 4.3 is used. A UDP flow was run along with a TCP flow. The UDP flow was made to produce bursts of UDP packets at regular intervals (in our case 3 seconds). The burst size and burst period were experimented with to produce congestion in the other competing TCP flow.

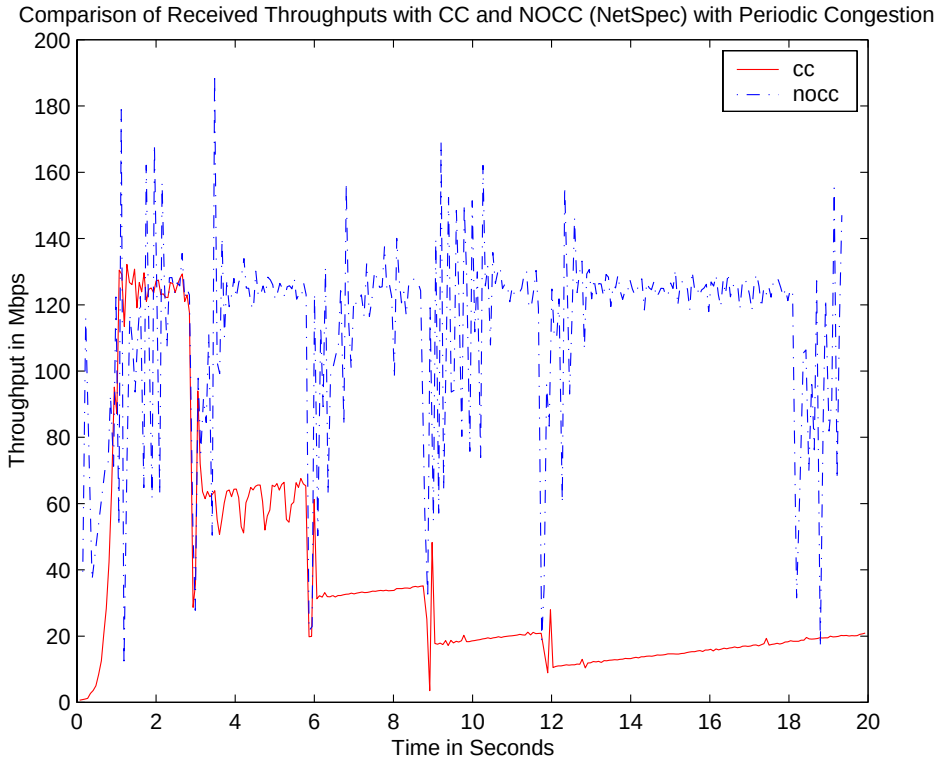

Figure 4.26: Variation of Received Throughputs in the event of Periodic Congestion

The Figure 4.26 shows the received throughputs plotted against time for TCP with CC and *NOCC*. The TCP with CC flow backs off with the first congestion event and does congestion avoidance from the point of loss. When the next congestion event occurs, (the congestion event occurs in multiples of 3 seconds) it again halves and starts doing congestion avoidance. This goes on for the duration of the UDP flow.

We can see *NOCC* behaving in a more steady manner with the throughputs decreasing at the point of congestion but the flow is able to recover quickly from congestion and yield steady throughputs.

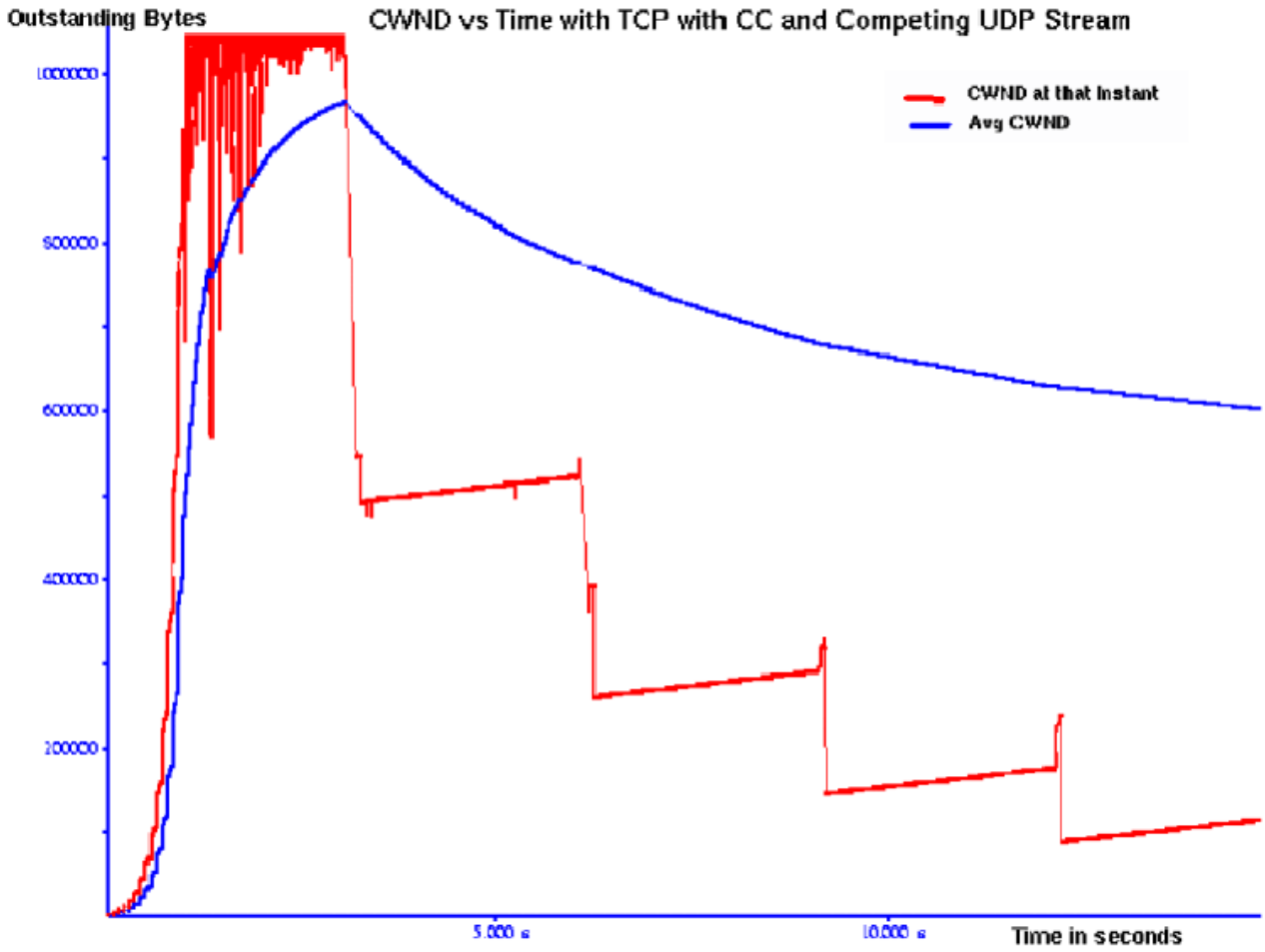

Figure 4.27: *CWND* vs Time in TCP with CC in the event of Periodic Congestion

Figure 4.27 shows how *CWND* gets halved at each point of congestion and how congestion avoidance occurs after each congestion event for TCP with CC.

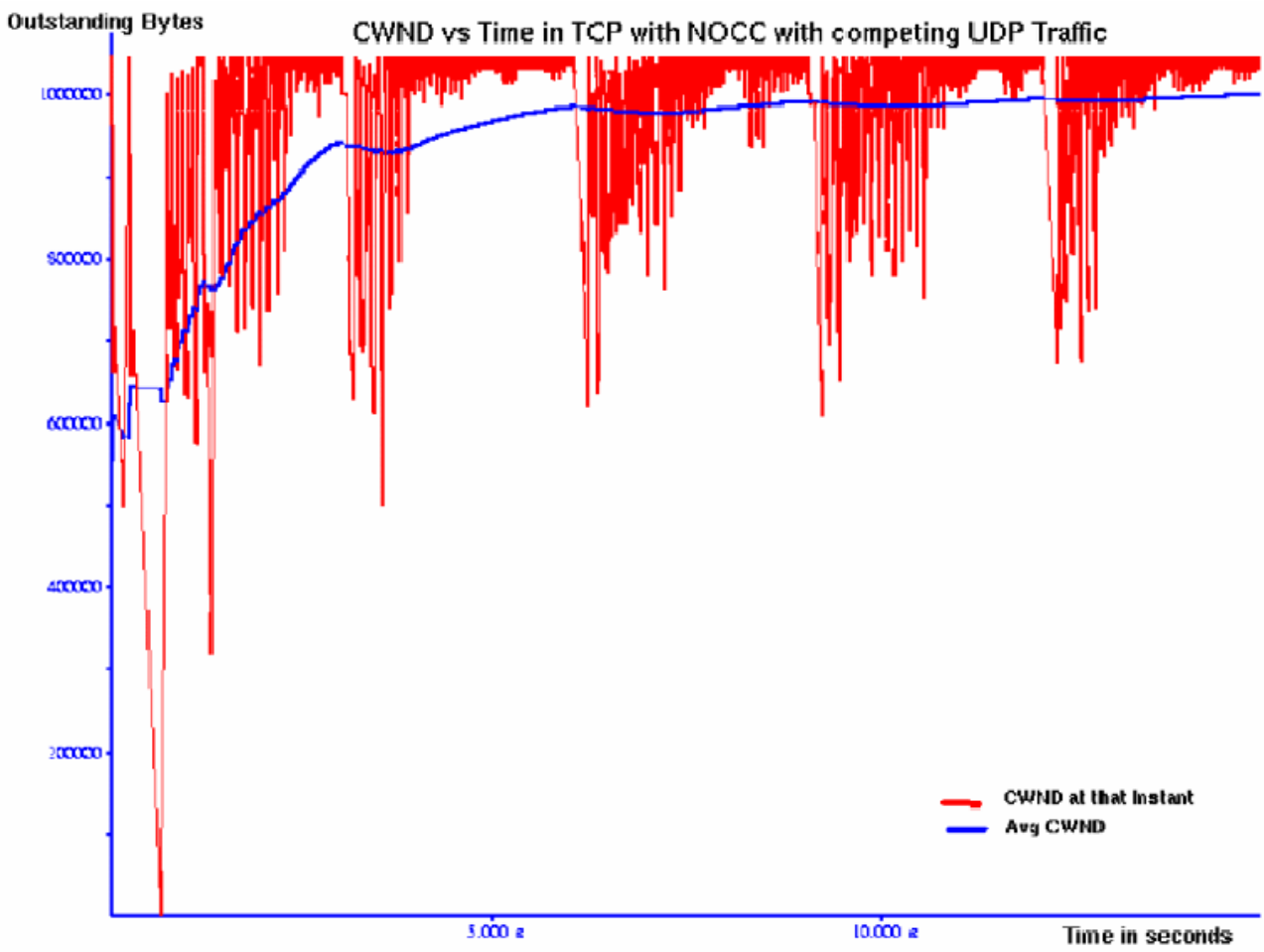

Figure 4.28: *CWND* vs Time in TCP with *NOCC* in the event of Periodic Congestion

Figure 4.28 shows how *NOCC* though affected by congestion maintains a constant Average *CWND* and is thus able to yield significant throughput advantages over TCP with CC.

4.5.2.3 TCP with *NOCC* competing with TCP with CC

This experiment was run with the second test scenario which has 2 machines each having a TCP connection between them as shown in Figure 4.4. So, in effect TCP with CC flow runs between a pair of testbeds and TCP with *NOCC* flow runs between another pair of testbeds.

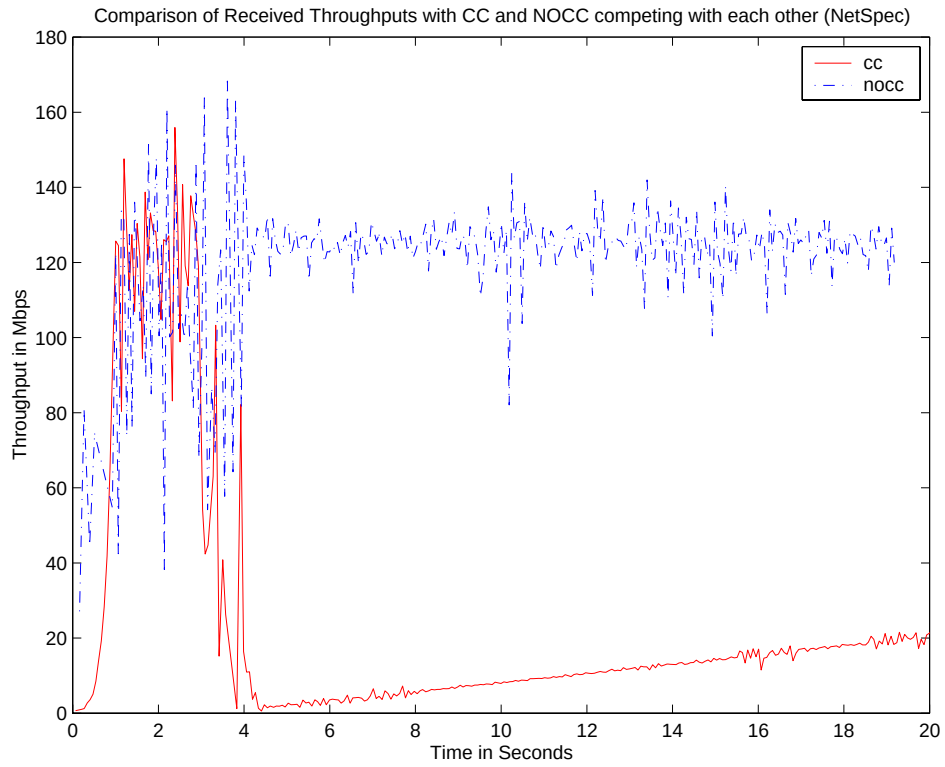

Figure 4.29: Received Throughputs with TCP with CC and *NOCC* competing with each other

We can see that *NOCC* is very aggressive and it penalizes the TCP with CC flow. So, it would not be advisable to deploy CC and *NOCC* together in a network because, TCP with *NOCC* is not tcp-friendly.

4.5.2.4 Conclusions

The above sections have analyzed the behavior of TCP with CC when a minimum congestion event occurred and when multiple congestion events occurred. TCP

with CC does aggressive congestion control by either going into slow start or congestion avoidance and this has dire effects on a high latency link. Therefore, the performance obtained from CC flows is drastically reduced. On the other hand *NOCC* does not employ aggressive congestion control strategies and so is able to achieve good performance benefits. In effect it is clear that TCP with CC is not well suited for huge BDP links, firstly because of the initial startup phase and secondly due to its backing off of the CWND when a congestion event occurs.

4.6 Scenarios and Results with Apache

The Apache Web Server with the pacing module introduced was installed on omega.cairn.net. iss-p4.lbl.gov was made to issue HTTP Get requests with a web server benchmarking tool called Zeus. httperf was also used to carry out some experiments.

The httpd.conf was incorporated with the *NOCC* configuration and the pacing configuration with the burst size and the burst period. Example of a portion of the httpd.conf with the configuration for the *NOCC* and the pacing.

#Pacing enabled
ExplicitRate On
#NOCC enabled
NoCongestionCntrl On
#Burst Size
ExplicitRateSize 128000
#Burst Period
ExplicitRatePeriod 10000

4.6.1 Burst Tests with Multiple Connections

This experiment was conducted by making Zeus request for files of different sizes from the Web Server.

File Sizes = 2KB, 7KB, 10KB, 30KB, 100KB, 422KB.

Burst Sizes = 32KB, 64KB, 128KB.

Burst Period = 10ms and 5ms

The modified Zeus was used to make queries with the burst size and the burst period specified in the HTTP Get request. Multiple requests were sent and for each request a TCP connection was opened. The resulting throughputs were plotted and Figure 4.30 shows the throughputs obtained with *NOCC* with pacing plotted against the normal behavior of web servers which employ HTTP 1.0 over TCP with CC.

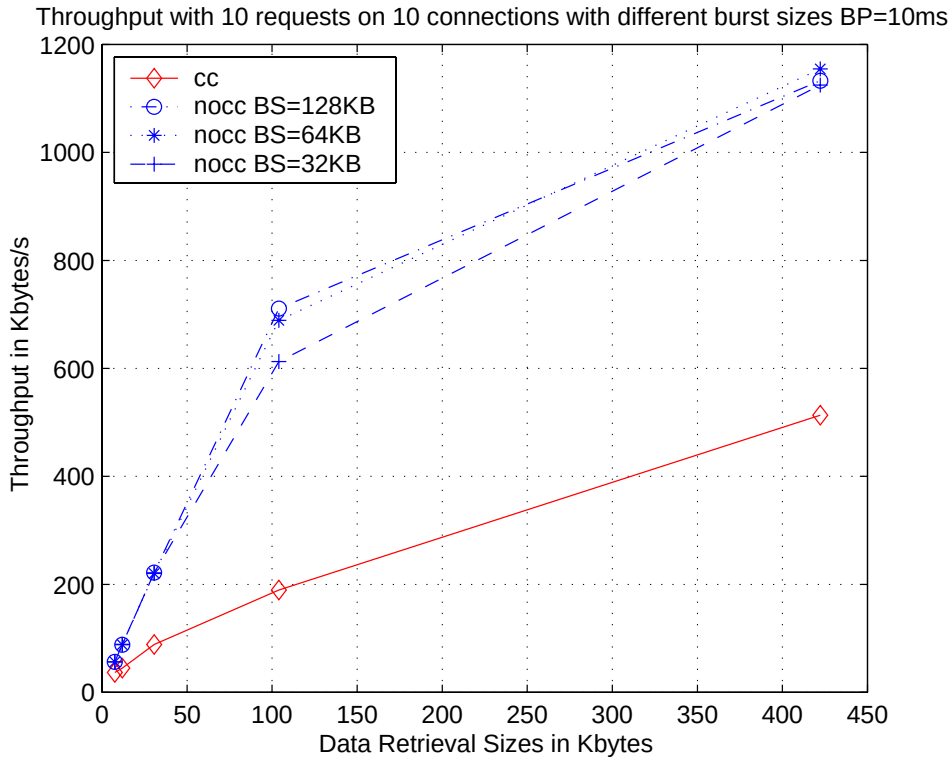

Figure 4.30: Received Throughputs with multiple connections for different Burst Sizes and BP=10ms

The Figure 4.31 shows the received throughputs for a burst period of 5ms. Zeus was made to issue modified HTTP Get requests and multiple requests were sent and the received throughputs were plotted .

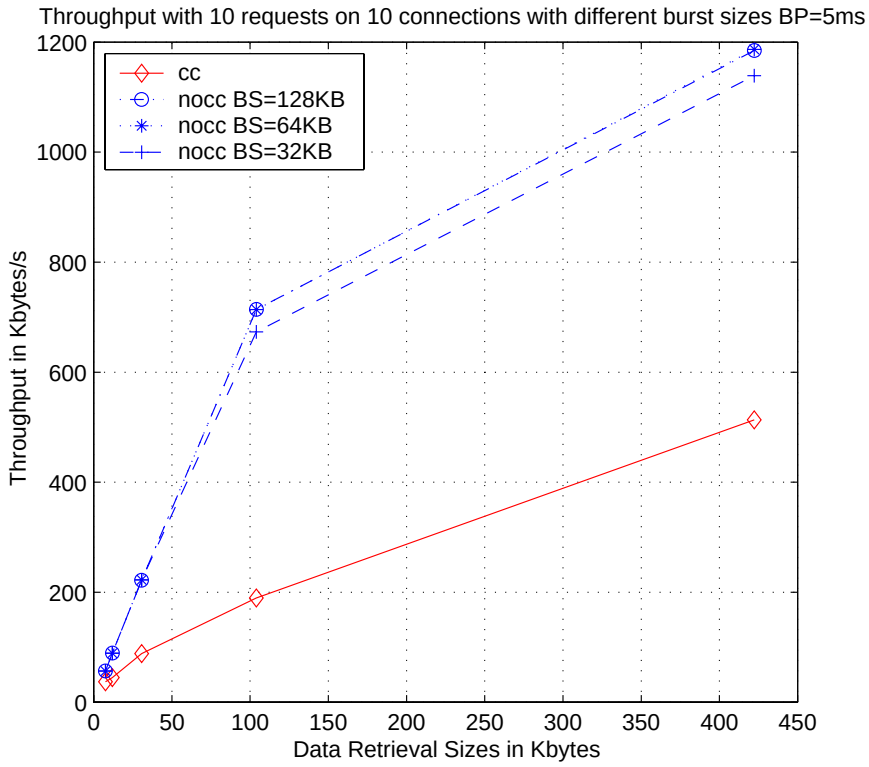

Figure 4.31: Received Throughputs with multiple connections for different Burst Sizes and BP=5ms

The received throughputs obtained in the *NOCC* case with pacing is seen to be significantly better than TCP with CC case.

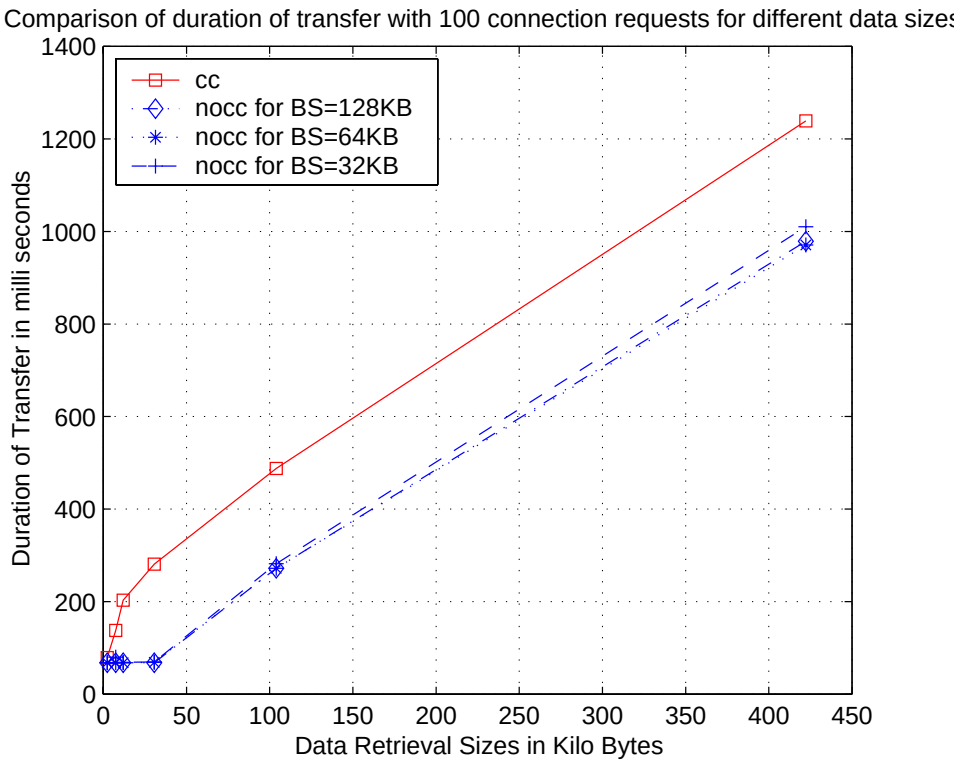

Figure 4.32: Received Throughputs with multiple connections for different Burst Sizes and BP=5ms

Figure 4.32 shows the duration of tranfer for the above conducted tests. There is a significant improvement in the latency since the duration of transfer is considerably reduced in the *NOCC* case.

4.6.2 Burst Tests with Persistent Connection

Zeus was made to retrieve files of varying sizes from the web server.

File Sizes = 2KB, 7KB, 10KB, 30KB, 100KB, 422KB.

Burst Sizes = 32KB, 64KB, 128KB.

Burst Period = 10ms and 5ms

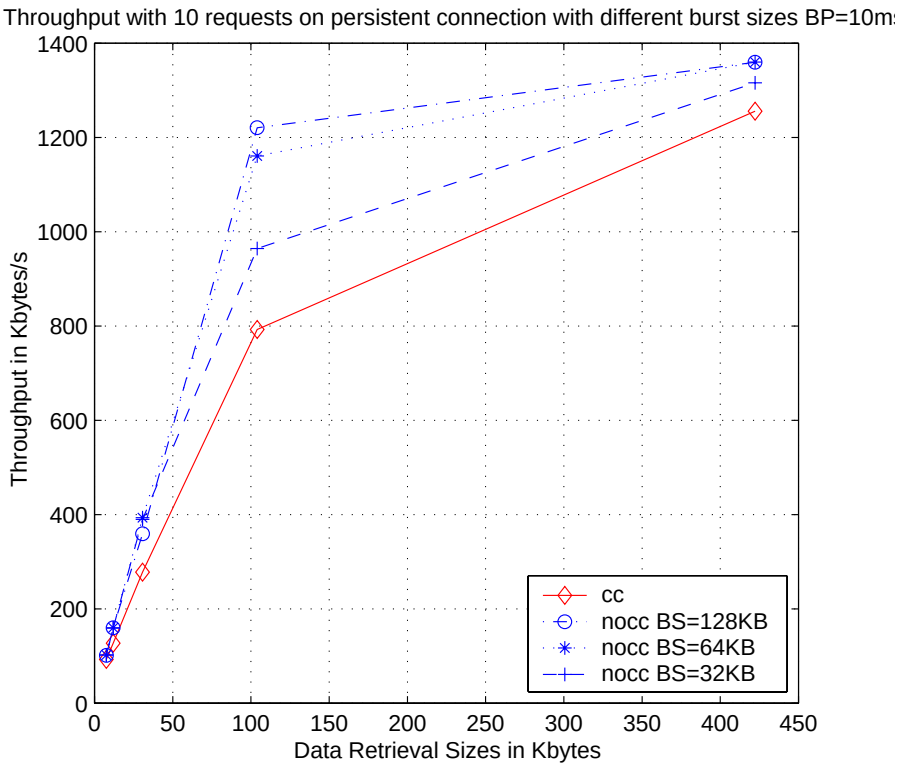

Figure 4.33: Received Throughputs for Persistent connection for different Burst Sizes and BP=10ms

For different burst sizes we observe a difference in throughputs in the higher file size region. This is seen from Figure 4.33.

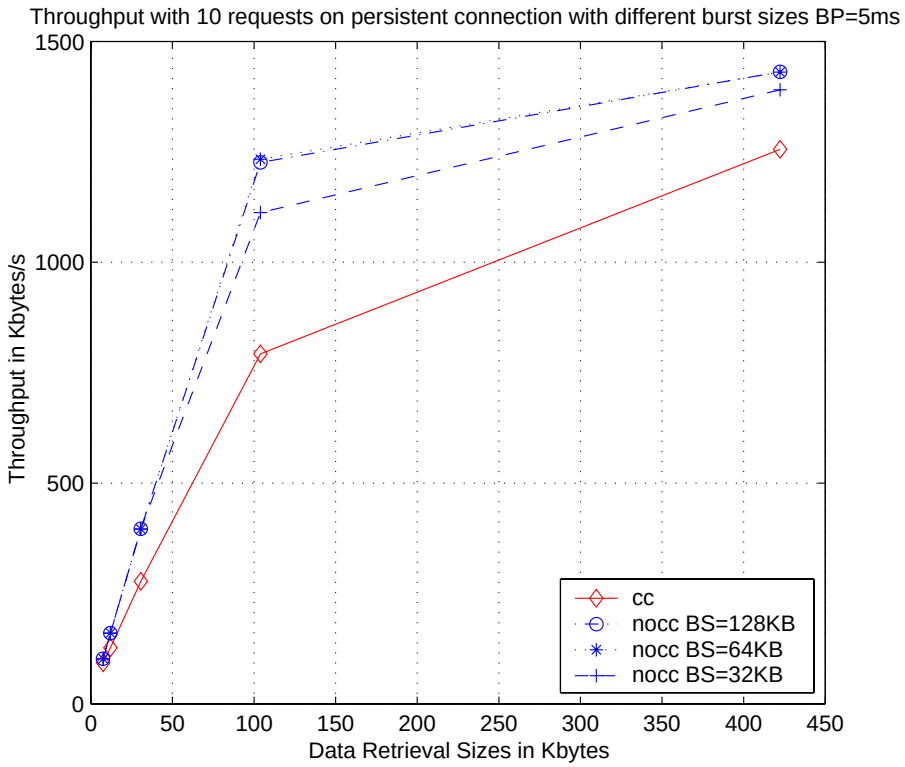

Figure 4.34: Received Throughputs for Persistent connection for different Burst Sizes and BP=5ms

Figure 4.34 shows the received throughputs obtained with a BP=5ms. The received throughputs show a significant improvement in the *NOCC* case.

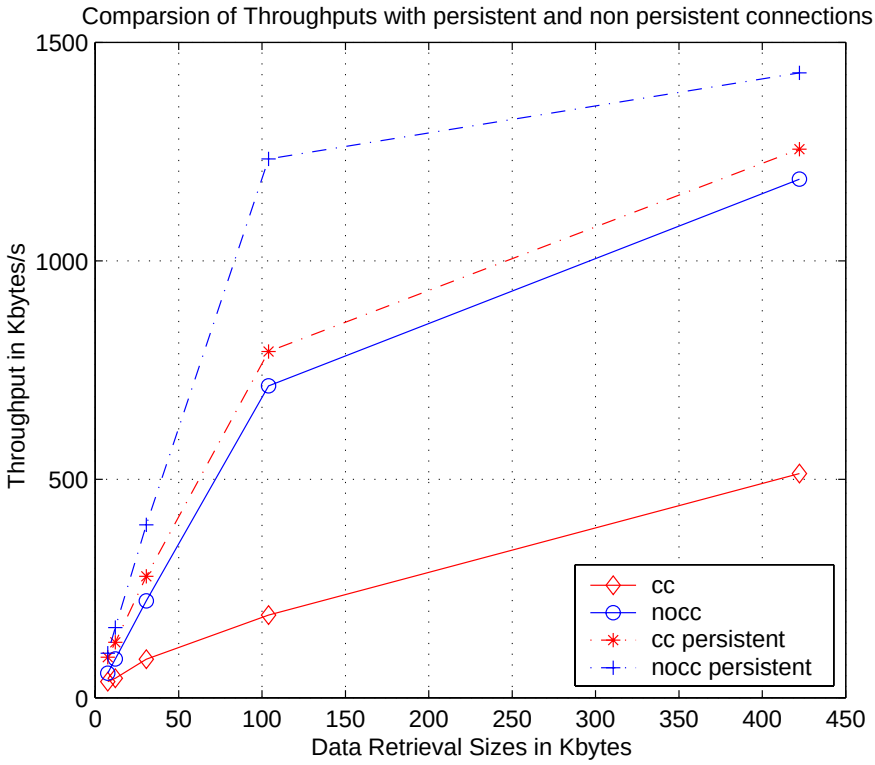

Figure 4.35: Received Throughputs for persistent and non-persistent connection HTTP

The throughputs obtained as seen in Figure 4.35 in both the CC and the *NOCC* case in P-HTTP or HTTP 1.1 is significantly better than those obtained from HTTP 1.0. This difference in the throughputs is because of the absence of connection establishment phase in each request's case. Since it takes a full RTT of about 67ms to establish a connection, it brings a significant performance boost.

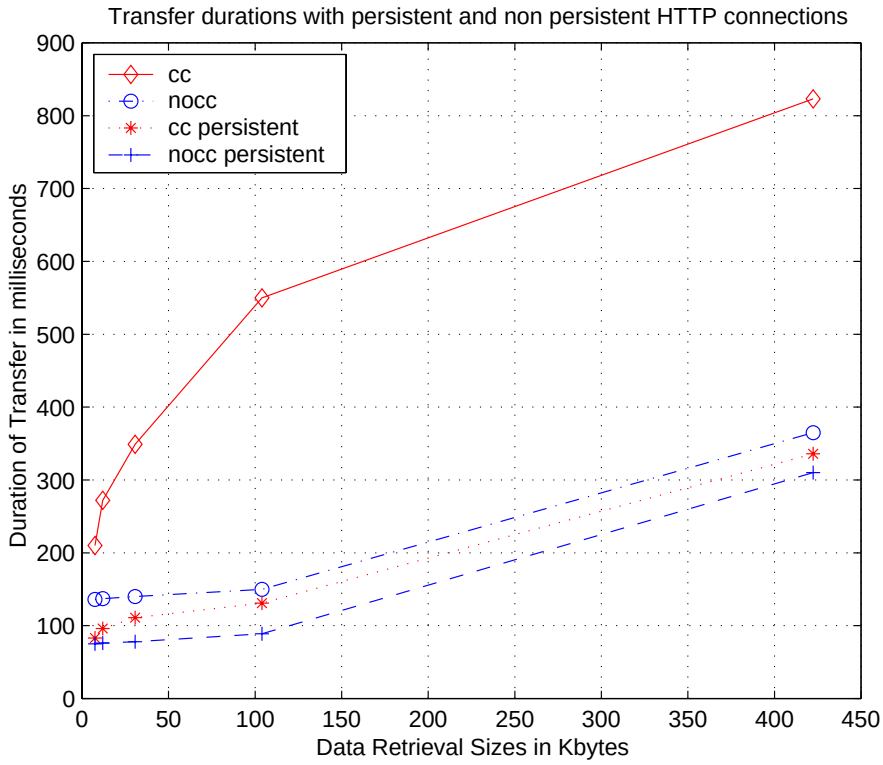

Figure 4.36: Duration of Transfer for persistent and non-persistent connection HTTP

The duration of transfer in *NOCC* is considerably reduced when compared with the TCP with CC case in both the persistent and non-persistent cases. The TCP with CC in P-HTTP case is able to get a significant performance improvement when compared to HTTP 1.0 because of the absence of the overhead of the connection establishment phase for each request. Also since *NOCC* does not have the *CWND* as a limiting factor, it is able to perform well for web data retrievals and so the duration of transfer is further reduced (when compared with CC) in the P-HTTP case.

4.6.3 Burst Tests with a Congestion Event

These tests were conducted with a UDP flow causing congestion in the network. The received throughputs when Zeus issued 50 requests was noted with no com-

peting UDP flows. Then a small burst of UDP packets was injected into the network and the burst size, burst period and the repeats were manipulated to make the smallest UDP burst cause congestion in the network.

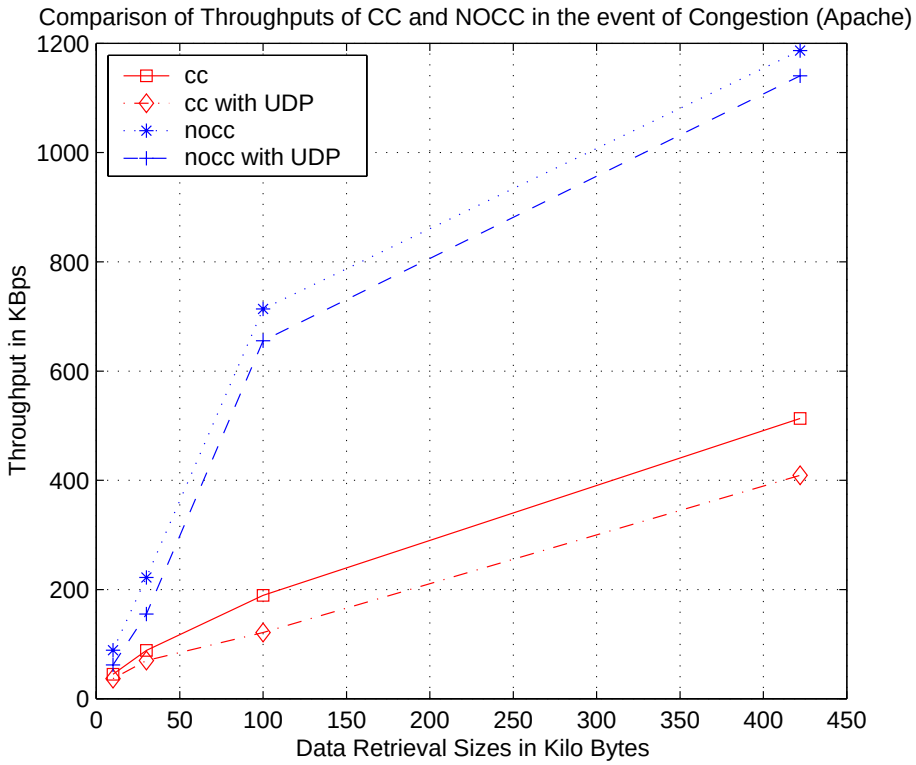

Figure 4.37: Duration of Transfer with Apache for different Burst Sizes and BP=10ms

In effect the performance of web flows in the presence of 'minimum' congestion was observed. Zeus was made to issue HTTP Get requests and received throughputs were noted in that case too. The two throughputs were plotted to see how congestion affects TCP with CC web flows and how it affects TCP with *NOCC* flows.

Figure 4.37 shows how TCP with CC is affected by congestion and how *NOCC* performs better because of its aggressiveness when it comes to the *CWND*.

4.6.4 Conclusions

The user perceived latency is one factor which web users would definitely like reduced. With *NOCC*, we can see that the latency which the user perceives can be reduced significantly in the HTTP 1.0 case. *NOCC* also helps to get better performance from P-HTTP. The difference in performance for different burst sizes is because of the more number of burst cycles required to tranfer the entire data. So, its seen that a 128Kbyte burst performs better than a 32KB burst size. Moreover, *NOCC* in its actual form does not give the user any control over what is being transmitted, but *NOCC* with pacing, gives the user the additional luxury of controlling the burst size and burst period and hence controlling the amount of data actually going out on the network. So, in the ENABLE infrastructure, where the aim is to aid the application in making intelligent QoS decisions and to give full control to the application, NOCC with pacing is an excellent mechanism for the application to control the amount of data sent onto the network.

Chapter 5

Conclusions and Future Work

5.1 Effect of Slow Start and Congestion Avoidance

The Slow Start and Congestion Avoidance algorithms have a negative effect on throughput, especially for long delay links. When one thinks about the topic under consideration, i.e., TCP No Congestion Control, it is but natural to wonder *'why at all did they come up with Congestion Control in the first place?'* Congestion Control Algorithms were devised for low bandwidth links which were prone to frequent congestion and often led to congestion collapse, thereby not utilizing the link bandwidth at all. TCP's Congestion Control algorithms were not devised for high bandwidth, high latency links. Infact it is quite the contrary, slow start causes an incredible startup phase problem which leads to poor utilization of the abundant bandwidth in high bandwidth links.

Congestion Avoidance is too conservative and with a single loss halves the *CWND* and increases it linearly and tries to reach the maximum *CWND* again. If a loss occurs early in slow start, then it takes a very long time to recover using this algorithm. This harms TCP performance on high BDP networks since it takes so much time to ramp up and fill the pipe, hence wasting valuable bandwidth which results in low channel utilization and performance degradation. This algorithm

has more dire effects in high delay connections than on low delay connections.

5.2 Effect of *NOCC* on TCP

Due to the above stated pitfalls in TCP, Congestion Control Algorithms are generally not seen as a useful phenomenon on high BDP links. Thus came the idea of experimenting with turning off Congestion Control in TCP and performing an evaluation of the same on high latency links. From the results, we can see that *NOCC* can be advantageous to short term flows because it is not inhibited by the start up phase problem faced by TCP with CC. This could prove very useful for web transfers which are of short duration and which are traditionally dominated by TCP's slow start phase. This can be deployed on the high performance DOE-NGI networks to have maximum utilization of the available bandwidth.

Also, as we saw, TCP can react adversely to even a single bit error loss. It goes into the Congestion Avoidance phase and causes a reduction in throughput under the surmise that all losses are caused by congestion. This can be detrimental in high BDP links. We saw that *NOCC* deals with this also in a better fashion. Though affected by congestion, it doesn't halve its sending rate and cause a reduction in the performance. It gives a better performance than TCP for random losses incurred.

The results obtained from this research can be summarized as follows:

- Throughputs obtained from *NOCC* are better than TCP with CC for short duration flows. The improvement in performance is because of a lack of a start up phase or building up phase in *NOCC*.
- *NOCC* also gives good response times in web flows when compared with TCP with CC.
- When traffic sources are competing for the same link, throughput drops dra-

matically because TCP retransmissions occur. This is worse in high BDP environments as expected, because of the longer time needed to recover from losses. *NOCC* is more aggressive than CC and so succeeds in achieving better performance.

- In flows where a ”small” congestion event could simulate the occurrence of a single bit error, we could see TCP with CC going into Congestion Avoidance and having reduced performance. On the other hand we saw *NOCC* handling single bit error losses with ease and performing much better than CC in this scenario.

- In short duration flows, the bandwidth utilization is high because the number of packets in flight is more in *NOCC*.

- *NOCC* gives very good performance benefits on HTTP 1.0 over CC. In the case of P-HTTP also, *NOCC* performs better and so validates our solution for short term flows.

- *NOCC* is better suited for high latency, high bandwidth links. It should be deployed with care on low bandwidth links because it can basically lead to a congestion situation.

- *NOCC* has been comprehensively tested on a WAN and it has been tested with a real world application like the Apache Web Server. We have seen that it gives good response time improvements in Apache. The user perceived latency can be considerably reduced by deploying *NOCC*.

5.3 Future Work

One possible extension to this would be to disable just the slow start phase of TCP and let TCP still do the Congestion Avoidance phase because that would help us

to have *NOCC* in a low bandwidth environment too by setting the initial window size to something bigger than what TCP sets it to. When Congestion occurs, Congestion Avoidance could take over and hence prevent *NOCC* from leading to Congestion collapse. This could prove useful because, if we want to get over the slow start phase of TCP but still want TCP's congestion control algorithms to take over when a congestion event occurs, this would help us do just that.

Bibliography

[1] R. Fielding, J. Gettys, J. C. Mogul, H. Frystyk Nielsen, L. Masinter, P. Leach, T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1, June 1999. RFC 2616.

[2] Mark Allman. Improving TCP Performance over Satellite Channels, Master of Science Thesis, Ohio University, 1997.

[3] W.Richard Stevens. TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms, January 1997. RFC 2001.

[4] Janey C. Hoe. Improving the Start-up Behavior of a Congestion Control Scheme for TCP. Proceedings of ACM SIGCOMM '96, August 1996.

[5] Van Jacobson, Robert Braden. TCP Extensions for Long-Delay Paths, October 1988. RFC 1072.

[6] Van Jacobson, Robert Braden, David Borman. TCP Extensions for High Performance, May 1992. RFC 1323.

[7] Roelof Jonkman. Netspec: A Network Performance Evaluation and Experimentation Tool. Master of Science Thesis. Available at: http://www.ittc.ukans.edu/netspec.

[8] T. V. Lakshman, Upamanyu Madhow. The performance of TCP/IP for networks with high bandwidth-delay products and random loss. IEEE/ACM Transactions on Networking, June 1997.

[9] Mark Richard Stemm. An Network Measurement Architecture for Adaptive Applications (SPAND).

[10] Vikram Visweswaraiah and John Heidemann. Rate Based Pacing for TCP

[11] Jeffrey C. Mogul. The case for persistent-connection HTTP. In Proceedings of the SIGCOMM '95, pages 299-313. ACM, August 1995.

[12] T. Berners-Lee, R. Fielding, H. Frystyk. Hypertext Transfer Protocol – HTTP / 1.0, May 1996. RFC 1945.

[13] Venkata N. Padmanabhan. Addressing the Challenges of Web Data Transport, Doctor of Philosophy Dissertation. September 1998.

[14] John Heidemann. Performance interactions between P-HTTP and TCP implementations. ACM Computer Communication Review, 27(2):tba, April 1997.

[15] Jacobson, V., Congestion Avoidance and Control. Proceedings of ACM SIGCOMM '88. August 1988.

[16] Sally Floyd. Congestion Control Principles. Internet-Draft draft-floyd-cong-02.txt, April 2000.

[17] John Nagle. Congestion Control in IP / TCP Internetworks, January 1984, RFC 896.

[18] Hans Kruse. Performance of Common Data Communications Protocols Over Long Delay Links: An Experimental Examination. In 3rd International Conference on telecommunication Systems Modeling and Design, 1995.

[19] Timothy A. Howes. The Lightweight Directory Access Protocol, July 1995.

[20] Kevin Fall, Sally Floyd. Simulation-based Comparisons of Tahoe, Reno and SACK TCP. Computer Communication Review, July 1996.

[21] Robert Braden. Requirements for Internet Hosts - Communication Layers, October 1989. RFC 1122.

[22] Mark Allman, Sally Floyd, C. Partridge. Increasing TCP's Initial Window, September 1998. RFC 2414.

[23] Mark Allman, Chris Hayes, Hans Kruse, Shawn Ostermann. TCP Performance over Satellite Links. Proceedings of the 5th International Conference on Telecommunication Systems, March 1997.

[24] Mark Allman, Dan Glover, Jim Griner, John Heidemann, Keith Scott, Jeffrey Semke, Joe Touch, Diepchi Tran. Ongoing TCP Research Related to Satellites. Internet-Draft draft-ietf-tcpsat-res-issues-03.txt, May 1998.

[25] Craig Patridge, Timothy J. Shepard. TCP / IP Performance over Satellite Links. *IEEE Network Magazine, Volume 11, No. 5,* September / October 1997.

[26] L. S. Brakmo, S. W. O'Malley, L. L. Peterson. TCP Vegas: New Techniques for congestion detection and avoidance. *Proceedings of the ACM SIGCOMM '94*, August 1994.

[27] C.P. Charalambous, G.Y. Lazarou, V.S. Frost, J. Evans, R. Jonkman. Experiments and Simulations of TCP / IP over ATM over a High Data Rate Satellite Channel. *AAI Report, ITTC-FY98-TR-10980-25, August 1997.*

[28] Georgios Y. Lazarou, Victor S. Frost, Joseph B. Evans, Douglas Niehaus. Using Measurements to Validate Simulation Models of TCP / IP over High Speed ATM Wide Area Networks. In Proceedings of the 1996 IEEE International Conference on Communications, June 1996.

[29] Beng Lee. Wide Area ATM Network Experiments using Emulated Traffic Sources, Master of Science Thesis, University of Kansas, 1998.

[30] Matthew Mathis, Jamshid Mahdavi, Sally Floyd, Allyn Romanow. TCP Selective Acknowledgment Options, October 1996. RFC 2018.

[31] Matthew Mathis, Jamshid Mahdavi. Forward Acknowledgment: Refining TCP Congestion Control. Proceedings of ACM SIGCOMM '96, August 1996.

[32] Jon Postel. Transmission Control Protocol, September 1981. RFC 793.

[33] Jon Postel. Simple Mail Transfer Protocol, August 1982. RFC 821.

[34] Jon Postel and Joyce Reynolds. File Transfer Protocol (FTP), October 1985. RFC 959.

[35] Douglas E. Comer. *Internetworking with TCP/IP, Volume I, Principles, Protocols, and Architecture*. Prentice Hall, 3rd edition, 1995.

[36] Andrew S. Tanenbaum. Computer Networks, 3rd edition, Prentice Hall PTR, New Jersey, 1996.

[37] Larry L. Peterson, Bruce S. Davie. Computer Networks: A Systems Approach, Morgan Kaufmann Publishers, Inc., San Francisco, 1996.

[38] W. Richard Stevens. TCP/IP Illustarted, Volume I: The protocols, Addison Wesley, 1994.

[39] Van Jacobson. Packet Sniffing tool. http://www.tcpdump.org

[40] Shawn Ostermann. *tcptrace* - TCP dump file analysis tool. http://jarok.cs.ohiou.edu/software/tcptrace/tcptrace.html

[41] Tim Shepard. *xplot* - A Plotting Tool. ftp://mercury.lcs.mit.edu/pub/shep

[42] The Apache Software Foundation. Apache Web Server. http://www.apache.org

[43] Acenic Gigabit Ethernet Card. Alteon Networks, http://www.sunsetdirect.com/alteon/acenic.html

Appendix A

Appendix

A.1 *tcpdump* Output showing TCP with CC

PF = Number of Packets in Flight

CWND = Congestion Window

16:59:56.681503 iss-p4.lbl.gov omega.cairn.net: 811846348:811846348(0) win 32767 1460,sackOK,timestamp

16:59:56.681503 omega.cairn.net iss-p4.lbl.gov: 1258749515:1258749515(0) ack 811846349 32767

16:59:56.748885 iss-p4.lbl.gov omega.cairn.net: 1:1(0) ack 1 65431

16:59:56.748885 omega.cairn.net iss-p4.lbl.gov: 1:1449(1448)

16:59:56.748885 omega.cairn.net iss-p4.lbl.gov: 1449:2897(1448)

16:59:56.817245 iss-p4.lbl.gov omega.cairn.net: 1:1(0) ack 1449 65431

PF=1, CWND=3

16:59:56.817245 omega.cairn.net iss-p4.lbl.gov: 2897:4345(1448)

16:59:56.817245 omega.cairn.net iss-p4.lbl.gov: 4345:5793(1448)

16:59:56.825057 iss-p4.lbl.gov omega.cairn.net: 1:1(0) ack 2897 65522

PF=2, CWND=4

16:59:56.825057 omega.cairn.net iss-p4.lbl.gov: 5793:7241(1448)

16:59:56.825057 omega.cairn.net iss-p4.lbl.gov: 7241:8689(1448)

16:59:56.884628 iss-p4.lbl.gov omega.cairn.net: 1:1(0) ack 5793 65431

PF=2, CWND=5

16:59:56.884628 omega.cairn.net iss-p4.lbl.gov: 8689:10137(1448)

16:59:56.884628 omega.cairn.net iss-p4.lbl.gov: 10137:11585(1448)

16:59:56.884628 omega.cairn.net iss-p4.lbl.gov: 11585:13033(1448)

16:59:56.892440 iss-p4.lbl.gov omega.cairn.net: 1:1(0) ack 8689 65431

PF=3, CWND=6

16:59:56.892440 omega.cairn.net iss-p4.lbl.gov: 13033:14481(1448)

16:59:56.892440 omega.cairn.net iss-p4.lbl.gov: 14481:15929(1448)

16:59:56.892440 omega.cairn.net iss-p4.lbl.gov: 15929:17377(1448)

16:59:56.952010 iss-p4.lbl.gov omega.cairn.net: 1:1(0) ack 11585 65431

PF=4, CWND=7

16:59:56.952010 omega.cairn.net iss-p4.lbl.gov: 17377:18825(1448)

16:59:56.952010 omega.cairn.net iss-p4.lbl.gov: 18825:20273(1448)

16:59:56.952010 omega.cairn.net iss-p4.lbl.gov: 20273:21721(1448)

16:59:56.959823 iss-p4.lbl.gov omega.cairn.net: 1:1(0) ack 14481 65431

PF=5, CWND=8

16:59:56.959823 omega.cairn.net iss-p4.lbl.gov: 21721:23169(1448)

16:59:56.959823 omega.cairn.net iss-p4.lbl.gov: 23169:24617(1448)

16:59:56.959823 omega.cairn.net iss-p4.lbl.gov: 24617:26065(1448)

16:59:56.959823 iss-p4.lbl.gov omega.cairn.net: 1:1(0) ack 17377 65341

PF=6, CWND=9

16:59:56.959823 omega.cairn.net iss-p4.lbl.gov: 26065:27513(1448)

16:59:56.959823 omega.cairn.net iss-p4.lbl.gov: 27513:28961(1448)

16:59:56.959823 omega.cairn.net iss-p4.lbl.gov: 28961:30409(1448)

16:59:57.020370 iss-p4.lbl.gov omega.cairn.net: 1:1(0) ack 20273 65431

PF=7, CWND=10

16:59:57.020370 omega.cairn.net iss-p4.lbl.gov: 30409:31857(1448)

16:59:57.020370 omega.cairn.net iss-p4.lbl.gov: 31857:33305(1448)

16:59:57.020370 omega.cairn.net iss-p4.lbl.gov: 33305:34753(1448)

16:59:57.027206 iss-p4.lbl.gov omega.cairn.net: 1:1(0) ack 23169 65431

PF=8, CWND=11

16:59:57.027206 omega.cairn.net iss-p4.lbl.gov: 34753:36201(1448)

16:59:57.027206 omega.cairn.net iss-p4.lbl.gov: 36201:37649(1448)

16:59:57.027206 omega.cairn.net iss-p4.lbl.gov: 37649:39097(1448)

16:59:57.027206 iss-p4.lbl.gov omega.cairn.net: 1:1(0) ack 26065 65341

PF=9, CWND=12

16:59:57.027206 omega.cairn.net iss-p4.lbl.gov: 39097:40545(1448)

16:59:57.027206 omega.cairn.net iss-p4.lbl.gov: 40545:41993(1448)

16:59:57.027206 omega.cairn.net iss-p4.lbl.gov: 41993:43441(1448)

A.2 *tcptrace* Output for TCP with CC

The trace is for the test with burst size=128KB, BP=5ms and duration=2s. The initial window is 2 pkts in CC case

---

```
Ostermann's tcptrace -- version 5.2.1 -- Wed Sep 15, 1999

49011 packets seen, 49011 TCP packets traced
elapsed wallclock time: 0:00:00.818008, 59915 pkts/sec analyzed
trace file elapsed time: 0:00:14.297852
TCP connection info:
1 TCP connection traced:
*** 31 packets were too short to process at some point
```

```
   (use -w option to show details)
TCP connection 1:
   host a:       iss-p4.lbl.gov:42015
   host b:       omega.cairn.net:42015
   complete conn: yes
   first packet:  Fri May 19 13:00:47.338729 2000
   last packet:   Fri May 19 13:01:01.636581 2000
   elapsed time:  0:00:14.297852
   total packets: 49011
   filename:     ../../../tcpdump_bin/cc/burst/iss_128K_5ms_2s_1
  a->b:               b->a:
    total packets:        16377          total packets:        32634
    ack pkts sent:        16376          ack pkts sent:        32634
    pure acks sent:       16375          pure acks sent:           1
    unique bytes sent:        0          unique bytes sent:  46779831
    actual data pkts:         0          actual data pkts:     32631
    actual data bytes:        0          actual data bytes:  47248984
    rexmt data pkts:          0          rexmt data pkts:        325
    rexmt data bytes:         0          rexmt data bytes:    469153
    outoforder pkts:          0          outoforder pkts:        112
    pushed data pkts:         0          pushed data pkts:     21650
    SYN/FIN pkts sent:      1/1          SYN/FIN pkts sent:      1/1
    req 1323 ws/ts:         Y/Y          req 1323 ws/ts:         Y/Y
    adv wind scale:           4          adv wind scale:           4
    req sack:                 Y          req sack:                 Y
    sacks sent:               0          sacks sent:               0
    mss requested:         1460 bytes     mss requested:         1460 bytes
    max segm size:            0 bytes     max segm size:         1448 bytes
    min segm size:            0 bytes     min segm size:          744 bytes
    avg segm size:            0 bytes     avg segm size:         1447 bytes
    max win adv:        1048352 bytes      max win adv:        1048352 bytes
    min win adv:          32767 bytes      min win adv:          32767 bytes
    zero win adv:             0 times     zero win adv:             0 times
    avg win adv:        1029639 bytes      avg win adv:        1048320 bytes
```

```
max cwin:              0 bytes    max cwin:          1045456 bytes
min cwin:              0 bytes    min cwin:             1448 bytes
avg cwin:              0 bytes    avg cwin:           503370 bytes
initial window:        0 bytes    initial window:       2896 bytes
initial window:        0 pkts     INITIAL WINDOW:          2 pkts
ttl stream length:     0 bytes    ttl stream length:  51328000 bytes
missed data:           0 bytes    missed data:       4548169 bytes
truncated data:        0 bytes    truncated data:    44703766 bytes
truncated packets:     0 pkts     truncated packets:    32631 pkts
data xmit time:    0.000 secs     data xmit time:       9.229 secs
idletime max:      2911.1 ms      idletime max:         2924.8 ms
throughput:            0 Bps      throughput:        3271808 Bps

RTT samples:           2          RTT samples:          14223
RTT min:           0.0 ms         RTT min:              66.4 ms
RTT max:           0.0 ms         RTT max:              89.8 ms
RTT avg:           0.0 ms         RTT avg:              68.4 ms
RTT stdev:         0.0 ms         RTT stdev:             1.4 ms
post-loss acks:        0          post-loss acks:         525
```

A.3 *tcpdump* Output showing TCP with *NOCC*

The dump output shows that *NOCC* does not wait for an ACK to arrive from the receiver and pumps packets out without being inhibited by the *CWND.*

17:03:53.044784 iss-p4.lbl.gov omega.cairn.net: 1072731545:1072731545(0) win 32767 1460,sackOK,timestamp

17:03:53.044784 omega.cairn.net iss-p4.lbl.gov: 1504904118:1504904118(0) ack 1072731546 32767

17:03:53.112167 iss-p4.lbl.gov omega.cairn.net: 1:1(0) ack 1 65431

17:03:53.112167 omega.cairn.net iss-p4.lbl.gov: 1:1449(1448) ack 1 65522

17:03:53.112167 omega.cairn.net iss-p4.lbl.gov: 1449:2897(1448)

17:03:53.112167 omega.cairn.net iss-p4.lbl.gov: 2897:4345(1448)

17:03:53.112167 omega.cairn.net iss-p4.lbl.gov: 4345:5793(1448)

17:03:53.112167 omega.cairn.net iss-p4.lbl.gov: 5793:7241(1448)

17:03:53.112167 omega.cairn.net iss-p4.lbl.gov: 7241:8689(1448)

17:03:53.112167 omega.cairn.net iss-p4.lbl.gov: 8689:10137(1448)

17:03:53.112167 omega.cairn.net iss-p4.lbl.gov: 10137:11585(1448)

17:03:53.112167 omega.cairn.net iss-p4.lbl.gov: 11585:13033(1448)

17:03:53.112167 omega.cairn.net iss-p4.lbl.gov: 13033:14481(1448)

17:03:53.112167 omega.cairn.net iss-p4.lbl.gov: 14481:15929(1448)

17:03:53.112167 omega.cairn.net iss-p4.lbl.gov: 15929:17377(1448)

17:03:53.112167 omega.cairn.net iss-p4.lbl.gov: 17377:18825(1448)

17:03:53.112167 omega.cairn.net iss-p4.lbl.gov: 18825:20273(1448)

17:03:53.112167 omega.cairn.net iss-p4.lbl.gov: 20273:21721(1448)

17:03:53.112167 omega.cairn.net iss-p4.lbl.gov: 21721:23169(1448)

17:03:53.112167 omega.cairn.net iss-p4.lbl.gov: 23169:24617(1448)

17:03:53.112167 omega.cairn.net iss-p4.lbl.gov: 24617:26065(1448)

17:03:53.112167 omega.cairn.net iss-p4.lbl.gov: 26065:27513(1448)

17:03:53.112167 omega.cairn.net iss-p4.lbl.gov: 27513:28961(1448)

17:03:53.112167 omega.cairn.net iss-p4.lbl.gov: 28961:30409(1448)

17:03:53.112167 omega.cairn.net iss-p4.lbl.gov: 30409:31857(1448)

17:03:53.113143 omega.cairn.net iss-p4.lbl.gov: 31857:33305(1448)

17:03:53.113143 omega.cairn.net iss-p4.lbl.gov: 33305:34753(1448)

17:03:53.113143 omega.cairn.net iss-p4.lbl.gov: 34753:36201(1448)

17:03:53.113143 omega.cairn.net iss-p4.lbl.gov: 36201:37649(1448)

17:03:53.113143 omega.cairn.net iss-p4.lbl.gov: 37649:39097(1448)

17:03:53.113143 omega.cairn.net iss-p4.lbl.gov: 39097:40545(1448)

17:03:53.113143 omega.cairn.net iss-p4.lbl.gov: 40545:41993(1448)

17:03:53.113143 omega.cairn.net iss-p4.lbl.gov: 41993:43441(1448)

A.4 *tcptrace* Output for TCP with *NOCC*

The trace is for the test with burst size=128KB, BP=5ms and duration=2s. The initial window is 327 pkts in *NOCC* case

---

```
Ostermann's tcptrace -- version 5.2.1 -- Wed Sep 15, 1999

40662 packets seen, 40662 TCP packets traced
elapsed wallclock time: 0:00:01.216116, 33435 pkts/sec analyzed
trace file elapsed time: 0:00:08.983399
TCP connection info:
1 TCP connection traced:
TCP connection 1:
    host a:       iss-p4.lbl.gov:42015                                  10
    host b:       omega.cairn.net:42015
    complete conn: yes
    first packet:  Fri May 19 12:59:23.706893 2000
    last packet:   Fri May 19 12:59:32.690292 2000
    elapsed time:  0:00:08.983399
    total packets: 40662
    filename:     ../../../tcpdump_bin/nocc/burst/iss_128K_5ms_2s_1
   a->b:                b->a:
     total packets:        13328          total packets:        27334
     ack pkts sent:        13327          ack pkts sent:        27334        20
     pure acks sent:       13326          pure acks sent:           1
     unique bytes sent:        0          unique bytes sent:  39574320
     actual data pkts:         0          actual data pkts:     27331
     actual data bytes:        0          actual data bytes:  39574320
     rexmt data pkts:          0          rexmt data pkts:          0
     rexmt data bytes:         0          rexmt data bytes:         0
     outoforder pkts:          0          outoforder pkts:          0
```

```
pushed data pkts:         0          pushed data pkts:       488
SYN/FIN pkts sent:      1/1          SYN/FIN pkts sent:      1/1
req 1323 ws/ts:         Y/Y          req 1323 ws/ts:         Y/Y
adv wind scale:           4          adv wind scale:           4
req sack:                 Y          req sack:                 Y
sacks sent:               0          sacks sent:               0
mss requested:         1460 bytes    mss requested:         1460 bytes
max segm size:            0 bytes    max segm size:         1448 bytes
min segm size:            0 bytes    min segm size:          480 bytes
avg segm size:            0 bytes    avg segm size:         1447 bytes
max win adv:        1048352 bytes    max win adv:        1048352 bytes
min win adv:          32767 bytes    min win adv:          32767 bytes
zero win adv:             0 times    zero win adv:             0 times
avg win adv:        1007124 bytes    avg win adv:        1048314 bytes
max cwin:                 0 bytes    max cwin:           1045720 bytes
min cwin:                 0 bytes    min cwin:               480 bytes
avg cwin:                 0 bytes    avg cwin:            997157 bytes
initial window:           0 bytes    initial window:      473496 bytes
initial window:           0 pkts     INITIAL WINDOW:         327 pkts
ttl stream length:        0 bytes    ttl stream length:  51328000 bytes
missed data:              0 bytes    missed data:        11753680 bytes
truncated data:           0 bytes    truncated data:     37442502 bytes
truncated packets:        0 pkts     truncated packets:     27331 pkts
data xmit time:       0.000 secs     data xmit time:        3.737 secs
idletime max:        3090.8 ms       idletime max:         3103.5 ms
throughput:               0 Bps      throughput:         4405272 Bps
```

A.5 NetSpec Scripts

NetSpec full blast script

```
cluster {

  test omega.cairn.net{

    type = full (blocksize=32000, duration=10, stamps=50000);
    protocol = tcp (window=1048576);
    own = omega.cairn.net:42015;
    peer = iss-p4.lbl.gov:42015;

  }

  test iss-p4.lbl.gov{

    type = sink (blocksize=32000, duration=10, stamps=100000);
    protocol = tcp (window=1048576);
    own = iss-p4.lbl.gov:42015;
    peer = omega.cairn.net:42015;

  }

}
```

NetSpec burst test script for *NOCC*

```
cluster {

  test omega.cairn.net{

    type = burst (blocksize=128000, period=10000, duration=10, stamps=500
    protocol = tcp (window=1048576, nocc=1);
    own = omega.cairn.net:42015;
    peer = iss-p4.lbl.gov:42015;

  }

  test iss-p4.lbl.gov{

    type = sink (blocksize=128000, duration=10, stamps=100000);
    protocol = tcp (window=1048576);
    own = iss-p4.lbl.gov:42015;
    peer = omega.cairn.net:42015;

  }

}
```

NetSpec UDP script

```
cluster {

  test dpss2.cairn.net{

    type = burst (blocksize=32000, duration=3, repeats=50000, pe-
riod=3000000, stamps=50000);
    protocol = udp;
    own = dpss2.cairn.net:42060;
    peer = dpsslx03.lbl.gov:42060;

  }

  test dpsslx03.lbl.gov:42030{

    type = sink (blocksize=32000, duration=3, stamps=100000);
    protocol = udp;
    own = dpsslx03.lbl.gov:42060;
    peer = dpss2.cairn.net:42060;

  }

}
```