EXPLORING DELAY-BASED HIGH-SPEED TCP PROTOCOLS UNDER DIFFERENT ROUTER BUFFER REGIMES

Michael K. Muraya

Contents

I. INTRODUCTION

Throughout history, effective communication has been the lynchpin of human evolution. The ability to send and receive data effectively and in a timely manner has greatly affected our ability to learn, grow and adapt to our environment. With the advent of the internet in the late 20th century, communication has become the driving force behind economic and socio-political advances through the wealth of information that can now be quickly and easily disseminated.

The glut of innovations that have been developed to improve the standards of data transfer have led to a great difficulty in finding the best way to set up the internet's infrastructure. This is compounded by the highly modularized structure of the internet, thanks to the adoption of the **Open Systems Interconnections Model**, the multi-layered internet specification that, in part, inspired the currently used 5-layer **Internet Protocol Suite**. By modularizing the different tasks involved in internet communication and standardizing their interfaces, it has become much easier for any one of the modules to be improved upon and have new implementations developed. The problem, therefore, is how to select the best implementations of each module to take full advantage of the infrastructure offered.

This paper aims to look at the innovations from two different layers or modules of the internet[1]; the **transport layer**, which determines how messages sent over the internet are packaged and sent, and the **internet layer**, which moves the packaged messages across network boundaries. While the development cycles of these layers are not entirely disparate acts, they certainly do not occur in tandem. Many a time, solutions to problems in one layer are developed, tested and implemented fully without touching the other. By comparing the performance of delay-based protocols (transport layer) under different router buffer size regimes (internet layer), this paper seeks to develop a clear understanding of the intertwined nature of these largely independent modules.

1.1 Structure of This Paper

This paper begins with a review of TCP by exploring the evolution of its congestion control functionality, whose different stages of development are the defining characteristics of each variety of TCP. Section III examines high-speed protocols, including the weaknesses of TCP that led to their development as well as the different ways they take advantage of the resources provided by the underlying network. Section IV examines different router buffer sizes and the reasoning behind them.

[1] Here, the Internet Protocol Suite standard is used to define the layers, not the OSI standard.

Section V explains the motivation for the experiments and related work, while Section VI explains the experimental setup used to test the performance of delay-based high-speed protocols in networks with different router buffer sizes.

1.2 Contribution of this Paper

This paper seeks to shed light on the behavior of delay-based high-speed TCP protocols under different buffer regimes, by using simulations in NS-2[2]. Smaller router buffers have proven advantageous for the core of the internet, leading to higher utilization and reduced power consumption. Intrinsically, a delay-based protocol's view of the network is dependent upon the size characteristics of the router; most of them have parameter values set to check the current buffer occupancy, for instance. Others rely on the delay caused by queue size to determine their growth functions. Understanding how changing the sizes of routers affects the efficacy of delay-based protocols furthers our understanding of the correlation between how congestion is detected (i.e. detecting imminent congestion as opposed to palpable congestion detected by packet loss) and the amount of buffer space available in the queue.

[2] S. McCanne and S. Floyd. ns Network Simulator. http://www.isi.edu/nsnam/ns/.

II. THE TRANSPORT CONTROL PROTOCOL

The workhorses of the internet are the **transport protocols**. These represent the contractual obligations of the communicating parties and stipulate the conventions followed to accept, format and package data for transmission and reassembly, as well as any other services the parties may require, such as error checking and flow control. Transport protocols provide a logical communication channel between applications, hiding the complexity of the undercarriage of the network from the applications using it. The main transport protocols in use today are the **User Datagram Protocol (UDP)** and the **Transmission Control Protocol (TCP)**.

UDP was designed to allow data transmission between hosts with little protocol overhead and formatting [rfc 768]. It provides a transaction-oriented data channel with a minimum of service guarantees - it simply sends the packets without any notion of reliability and provides only simple error checking as a protocol service.

TCP, on the other hand, provides a reliable data transfer channel between applications. In addition to the packaging and error checking provided by UDP, TCP checks for packet loss, duplication and reordering, retransmitting packets as needed. It maintains and preserves communication sessions and allows the receiver to throttle the sender and control the rate at which it receives data from the network. TCP also attempts to balance out the maximum bandwidth requirement of the applications while avoiding overloading the network and causing excessive packet loss; this functionality is referred to as **congestion avoidance**.

2.1. A History of TCP the Rise of Congestion Control

2.1.1 Fundamentals of Sliding Window-Based Protocols

In order to cater to the reliability and flow control requirements, TCP is implemented as a *sliding-window-based* (SW) protocol. An SW protocol consists of four main parts: a sender, a receiver, the faulty transmission channel and the faulty acknowledgement channel. It is assumed that a faulty channel has the following characteristics [snep 95]:

1. messages may not arrive in the order they were sent
2. any message sent along the channel can be lost or duplicated
3. error detection is perfect; thus while any message may contain errors or be garbled, the error can and will be detected
4. the channel is capable of submitting data, i.e. despite being faulty, only a finite number of messages are lost, duplicated or garbled

Given a message, the sender breaks it up into a Protocol Data Unit (in TCP, this is the *packet*), and the sender and receiver exchange entire packets. The sender must timestamp each packet with an incremental *sequence number*. TCP, being a bytestream-oriented protocol, makes the sequence number of each packet the byte-number of the first byte within the packet. This sequence number allows the receiver to know whether a packet has been lost, re-ordered or received with errors. The receiver acknowledges receipt of the last valid packet by sending back an *acknowledgement number*, which, essentially, is the sequence number of the next packet the receiver expects.

Should any packet not be acknowledged within some agreed-upon period, the sender is required to retransmit the first unacknowledged packet, and every unacknowledged one after that, till all of them are acknowledged, before it starts sending previously unsent data.. In order to keep track of the packets sent and acknowledged, both the sender and receiver must maintain some overhead about the status of the packets, i.e.: [stenning 76]

1 The sender must keep track of all the packets that have been sent and are yet to be acknowledged. This value is known as the *sender's window size*. In order to ensure that packets can be re-transmitted if need be, the sender must buffer all the sent-yet-unacknowledged packets until they are acknowledged.
    a. The variable "*lowest_unacked"* refers to the sequence number of the lowest numbered packet still awaiting acknowledgement – since sequence numbers are incrementing, this would represent the packet whose acknowledgement the sender expects to receive soonest. This value is considered the right edge of the window. Once this is acknowledged, the sender removes it from the buffer.
    b. "*highest_sent"*, on the other hand, refers to the most recently sent and unacknowledged packet. This is considered the left edge of the window.

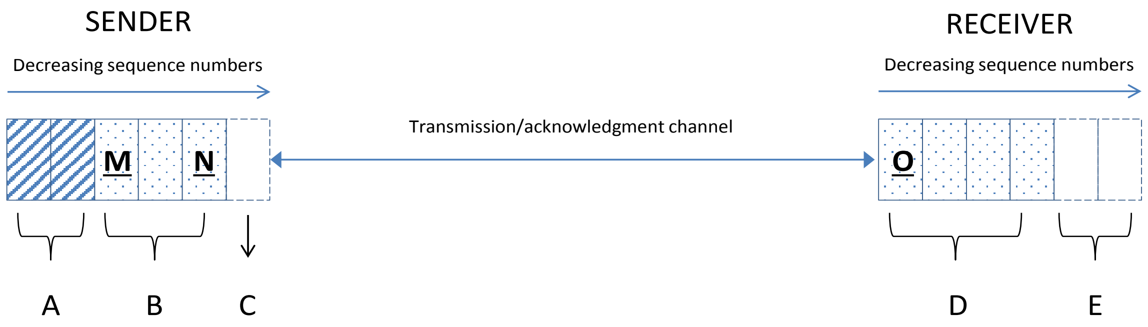
The number of packets in the window should not exceed the sender's window size.

2 The receiver must keep track of the packets received and (possibly) buffer those received out of order. Given the *receiver's window*, then:
    a. "*next_required*" value represents the packet that the receiver expects to receive next, given all those with sequence numbers less than *next_required* have been successfully received.
    b. The size of the receiver's buffer is the "*receive_window"*, and represents the highest number of packets the receiver can accept at any one time.

A graphical representation of the windows is as shown below.

The sender can only send more data if:
1 An acknowledgement has been received, and no timeout has occurred waiting for it.
2 The sender has more data to send.

The actions taken by the sender and receiver if an acknowledgement is missed or data is available to send are specific to the implementation of the protocols themselves. A sender may


SENDER
Decreasing sequence numbers
M
N
A
B
C
Transmission/acknowledgment channel
RECEIVER
Decreasing sequence numbers
O
D
E

A: packets just received from application
B: transmitted but not acknowledged (sender's window)
C: transmitted and acknowledged: removed from buffer
M: sequence number = highest_sent
N: sequence number = lowest_unacked

D: packets just received from sender (receiver's window)
E: received and delivered to application: removed from buffer
O: acknowledgement number = next_required

Figure 1: Sender and receiver state in sliding window protocols

choose to maintain the same sending frequency once an acknowledgement is received, safe in the knowledge that the sending rate chosen works well enough. Alternatively, the sender may choose to try a larger sending rate, and test if the network can handle it, in order to facilitate faster sending of data. Similarly, if either the acknowledgement or the packet is lost, then the sender has to decide whether to maintain the same sending rate, and risk further loss, or to reduce the sending rate to a more agreeable value before sending again.

Thus the amount of data sent per unit time is therefore determined by the sender's send window size and the time it takes to receive an acknowledgement from the receiver. This time, known as the *round trip time* ($\boldsymbol{RTT}$) is a factor of all the time components that affect how fast a packet moves from the sender to the receiver, and how fast an acknowledgement (enclosed as a packet) moves from the receiver to the sender. These time components include:

1 Time spent encoding the packet onto the channel.
2 Time spent propagating the packet through the channel.
3 Time spent waiting for resources to be available to a packet, e.g. if the channel is unavailable.
4 Time spent decoding the packet from the channel to the end point, or any intermediate nodes.

TCP's adherance to the principles of a sliding window protocol occur as follows:

1 Each sender has a send window $swin$, and each receiver a receive window $rwin$. The amount of data that can be sent before an acknowledgement is received is given by:

$$win = \min(rwin, swin)$$

For analysis purposes, we assume $swin$ is the limiting factor.

2 The transmission/acknowledgement channel is a duplex channel, i.e. data can simultaneously be sent and received across the same channel.
3 Both the sender and receiver keep track of the variables described above, i.e. highest_sent and lowest_unacked for the sender, and next_required by the receiver
4 The channel, as we mentioned before, is susceptible to packet loss, reordering packets and encoding errors. All these cause retransmissions to occur. The probability of a packet to be lost, damaged or re-ordered is quantified as the *loss probability* (p) of the channel.
5 The sender cannot send data faster than the channel's propagation capacity. This capacity, known as the *bandwidth*, and notated as C, is measured in bits per unit time, and represents the maximum throughput a sender/receiver pair can achieve.
6 TCP operates a positive-acknowledgement/retransmit-on-timeout system. Acknowlegements are used as indicators that all packets with a sequence number lower than the acknowledgement number have been successfully received. Once an acknowledgement is received, the sender's window size is increased, allowinf more data to be sent per unit time. If a timeout occurs, the sender's window is halved.

Given a sender receiver pair, let the sender's window be of average size $\overline{W}$, in bytes. If the sender and receiver maintain an uninterrupted flow of packets and acknowledgements, the channel and all its components will be operating at maximum capacity. This means that the queues within the channel (if any) and the channel itself will always be occupied by packets; at no point will the channel be idle. This means that the channel will be at full utilization. The amount of time it takes for a sender to receive an acknowledgement of a previously sent packet (assuming a perfect channel) is RTT. If a channel has a capacity of C bytes per second, then assuming the channel's RTT is 1 second, there can, at maximum utilization, only be C bytes within the channel. Similarly, if RTT = 2, then the channel can hold:

$$2\ seconds * C \text{ bytes/second } = 2C \text{ bytes}$$

at maximum utilization. This value, which is the product of the link's capacity C and the RTT of the link, represents the maximum amount of data that can be in the link at any one time, and is known as the *bandwidth delay product* (BDP).

The maximum number of bytes allowed in the link at any instant is therefore given by the BDP, while the sender's throughput T (number of bytes received by receiver per unit time), taking loss p into account, is given by:

$$bytes\ sent = \overline{W}$$

$$time\ it\ takes\ to\ send\ those\ bytes = RTT$$

$$bytes\ received = (1-p) * \overline{W} \ \leq \ \overline{W}$$

$$\therefore Throughput\ (T) \leq\ (\bar{W}\ /\ RTT)\ bytes\ per\ second$$

If BDP represents the maximum number of bytes that can be available in the link, then it follows that the sender's throughput T is capped by the channels' BDP. Thus, given a link capacity C, an average window size $\bar{W}$ and a round trip time RTT, then:

$$BDP = C * RTT \Rightarrow C = {}^{BDP}/_{RTT}$$

and

$$T \leq {}^{\bar{W}}/_{RTT}$$

To achieve the maximum transfer rate, then

$$T \leq C$$

$$\Rightarrow {}^{\bar{W}}/_{RTT} \leq {}^{BDP}/_{RTT}$$

$$\Rightarrow\ \bar{W} \leq BDP$$

Thus the sender will only achieve maximum throughput when the sender's window size is equal to the bandwidth delay product.

2.1.2 Congestion Avoidance & the Evolution of TCP

2.1.2 (a). TCP 4.2 BSD

Various implementations of TCP were defined in [Cerf 74], [rfc 761] and [IEN 124], among others. Nine of these specifications were brought together in [rfc 793], and the result was the 4.2BSD implementation of TCP. Keeping in mind that TCP was meant to be a reliable transport protocol, [rfc793] lists the characteristics desired as:

Basic Data Transfer - TCP was to be able to transfer a continuous stream of octets (again, the *elements* from [Cerf 74]) via a duplex channel. This continuous stream was to be packaged as individually-addressed and routed segments.

Reliability - TCP was to recover from lost, reordered, duplicated or corrupted data, by including a system through which data sent could be sequenced and acknowledged; sequence and ack numbers server this purpose.

Flow Control - The receiver has the ability to throttle the sender by letting them know how much data they can receive at once, via the receive window. The sender is required to reduce the number of packets they send at once to this value.

Multiplexing - TCP provides a set of ports within each host to allow multiple parties within the host to communicate simultaneously. The host-address/process-port tuple identifies a socket on the host, and a pair of sockets identifies a connection.

Connectivity - TCP has to maintain some overhead information for each flow. This includes the receive window value, the port number, etc.

Precedence and security - TCP allows users and applications to indicate the precedence and security of their communications by use of flags/control data within the header of a packet.

These requirements and mechanisms were sufficient to support the communication needs that TCP met at the time. Networks ran on ARPANET IMP packet-switching nodes [rfc 896] with uniform bandwidth and sending capacity that helped avoid congestion. Also, as there weren't many hosts, available bandwidth was usually over-provisioned.

Congestion Collapse

As networks grew and became more heterogeneous, a potential problem that TCP wasn't capable of handling was described: "congestion collapse" [rfc 896]. This was the result of one (or both) of the following scenarios:

1. The **small-packet problem** was caused by TCP's sending algorithm, which normally sent data as soon as some was available to send, and the receiver's window allowed it. Thus, in systems where data generation was slow, TCP would send single-byte packets with 40 bytes of TCP overhead. While lightly-congested networks could easily handle this, congested networks were susceptible to sudden, heavy increases in traffic, which would lead to packets being dropped when buffers are filled. This would lead senders to retransmit the 41-byte packets, lowering network throughput.
2. As the number of nodes increased, the amount of traffic in the network increased as well. As this amount approached critical mass, the buffer size increases, lengthening the time each packet spends in the buffer and, consequently, the RTTs of the flows as well[3]. If a packet is dropped and the sender doesn't get confirmation of a packet being received, TCP was designed to retransmit packets several times at increasing time intervals until some upper limit is reached. These retransmissions would further congest the network, increasing the RTT and leading to a cycle of timeouts, retransmissions and increased congestion. If the upper retransmission timeout limit is reached, TCP may terminate the connection.

While the small-packet problem was successfully solved through buffering at the sender (via **Nagle's Algorithm**), the congestion was yet to be resolved. Once the network described in (2) above experiences packet drops after the buffers fill up, the RTT of the packets reaches its maximum. Eventually, one of the copies of the packet will get to the sender, and one of the ACKs confirming receipt will make it back. However, since the senders are repeatedly backing

[3] See section 2.1.1 for more on the relationship between RTT and network load

off their retransmissions, and sending copies of packets into an already heavily congested network, the throughput of the network is reduced to almost nil, which Nagle was able to confirm experimentally over a private network.

2.1.2 (b) TCP Tahoe

In 1986, the congestion collapse occurred on a production network, when the throughput from Lawrence Livermore Labs to UC Berkeley (400 yards and two Internet Message Processor hops away) dropped from 32Kbps to 40bps. The growth of the internet (in terms of number of hosts) had not been matched by an increase in sending capacity of the links, and as such, more and more hosts had to share a dwindling resource (bandwidth). A suggested solution was to design an algorithm to make sure that the sender does not push too much data. Thus, while flow control was the receiver's way of throttling the sender and letting them know how much data to send, congestion avoidance and control was to be the sender's self-throttling mechanism that would respond to cues to congestion that would be supplied by the network [jacobson 88].

Self-Clocking Principle

Per [jacobson 88], TCP operates on the principle of "conservation of packets" i.e. for a connection running at available network capacity, a new packet is only injected into the network after a packet is taken out. This is done by having the receiver send an acknowledgment number back to the sender, in affirmation of having received all packets prior to the one whose sequence number matches the acknowledgement number sent. Should a packet arrive at the receiver in an unexpected order, the receiver once again sends an affirmation of the last correctly received packet; if lost, no affirmation is sent back. This helps maintain a system in equilibrium, as illustrated in the image below[4]:

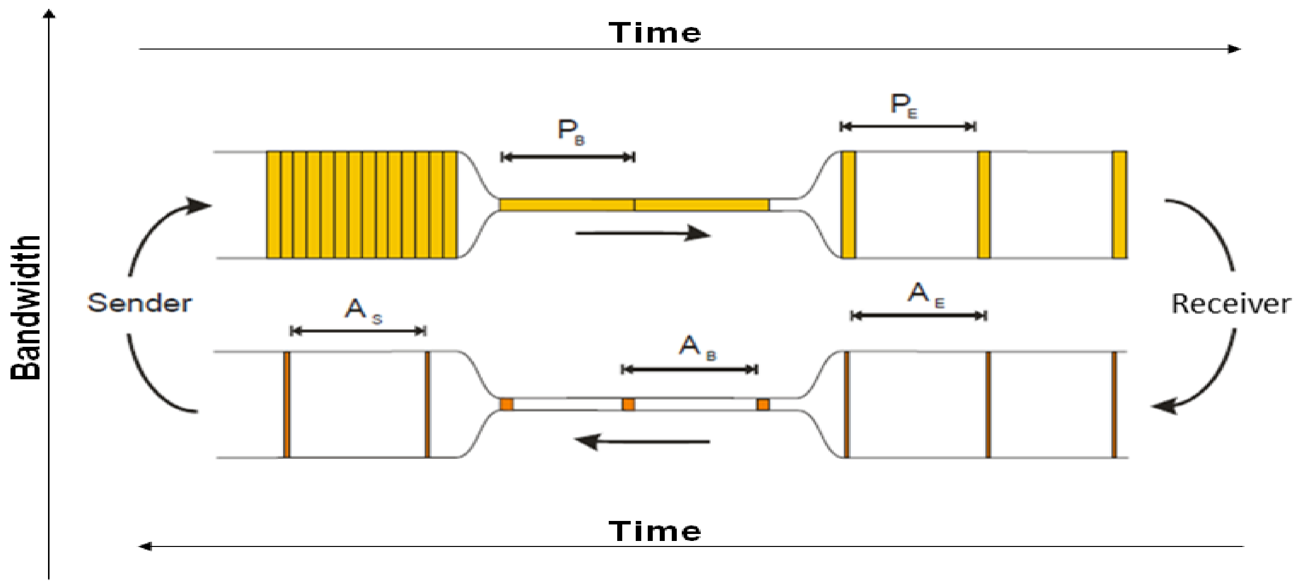


Figure 2: TCP sender and receiver in equilibrium

[4] Image courtesy of V. Jacobson, *Congestion Avoidance and Control*. ACM/SIGCOMM 1988 Computer Communication Review.

In the diagram, the sender (left, top and bottom) and receiver (right) are connected via a network. The vertical axis represents the bandwidth of the connection; the narrow middle portion, therefore, represents a bottleneck link. The yellow blocks represent the data packets sent over time; the narrower the bandwidth, the longer the time it takes to transmit the packets. As the packets transition from the slower link to the faster one, the transmission interval remains the same; thus the period PB is the same as the inter-packet interval PE. Assuming the receiver's processing time is constant for all packets, the ACK inter-packet period AE will match the packet period PE; this period is maintained throughout the system, such that PB = PE = AE = AB = AS. For each ACK received, the sender would then transmit another packet. Hence, after the initial burst from the sender, the sender's packet sending rate is clocked by the rate of receipt of ACKS, which, in turn, is controlled by the bottleneck link's capacity. This, conceptually, creates a system in equilibrium.

As described in [jacobson 88], in order to maintain the equilibrium of this system, three factors had to be taken into consideration:

(1). The system would have to get to equilibrium from a state of non-equilibrium
(2). The system would have to conserve that state of equilibrium
(3). The system would have to adapt to the changing environment to maintain equilibrium, or attain it again if lost.

(1). The system would have to get to equilibrium from a state of non-equilibrium

To avoid congestion collapse, each sender maintains a congestion window variable ($cwnd$) that limits the number of unacknowledged packets in transit. From an initial low value (e.g. 1 MSS), TCP Tahoe uses slow start to increase $cwnd$ by 1 MSS for each acknowledged packet - effectively doubling $cwnd$ in each RTT. The amount of packets sent is, therefore, the lesser of the receive or congestion windows. Slow start continues until $cwnd$ reaches a preset slow-start threshold ($ssthresh$) value, or congestion occurs. As the initial $ssthresh$ value is usually set to a very high value, congestion is more likely to be the terminating factor for the first slow-start stage. Therefore, given a maximum segment size MSS bytes, receive window of size $rwnd$ and congestion window of size cwnd, TCP Tahoe's algorithm for achieving initial equilibrium is as follows:

```
$cwnd = 1 * MSS$
        FOR EACH $received\ ACK$
        DO
        $cwnd = cwnd + MSS$
                $packets\_sent = min(cwnd, rwnd)$;
        WHILE ( $[cwnd < ssthresh]\ \&\&\ \sim [Congestion]$ );
```

(2). The system would have to conserve that state of equilibrium

Tahoe's congestion control algorithm infers congestion through packet loss. As the system moves towards equilibrium, an important issue arises; how is loss determined? Since a packet loss would represent the tipping point from equilibrium to chaos, it is important that the sender be able to accurately determine that a packet has been lost. The sender waits to receive an ACK from the receiver before sending another packet into the network; if no ACK is received, is it merely delayed, or was the packet not received? How long must a sender wait before determining that a missing ACK means the corresponding packet is lost? If the sender waits too long, the transmission rate tumbles; too short, and the retransmitted packet disturbs the equilibrium (and lowers the throughput as well) by pumping spurious packets into the network.

When a packet is sent, TCP starts a timer. This keeps track of how much time has passed before a packet's ACK is received, i.e. the round-trip time. Being a packet-switched network, the route a packet and its ACK take cannot be predetermined; nor can the state of congestion of the network prior to sending the packet. Thus, the sender's expectation of the amount of time it takes to receive an ACK is an estimate that takes into consideration the RTT values of previous packets, and the variance observed so far. This exponential weighted moving average is known as the *Retransmission Time-Out* (RTO) value, and is calculated as follows: let $mRTT$ be the last observed RTT value. The system calculates a *SmoothedRTT* $(sRTT)$ value that weights the $mRTT$ with all the previously observed RTT values, as follows [jacobson 88] [rfc 2581]:

- Before the initial packet is sent, the RTO value is set to 3 seconds.

- After the first packet is successfully sent, $sRTT$ is set to the measured RTT and the deviation to half the measured RTT, i.e.:

$$sRTT = mRTT$$
$$Deviation = (mRTT) / 2$$

- Thus the initial RTO value is given by:

$$RTO = sRTT + (k * Deviation)$$

where k is a safety margin factor; empirical evidence has determined that 4 is a suitable value.

- Subsequent packets have their $sRTT$, $Deviation$ and RTO values updated as follows:

$$sRTT = (\alpha * mRTT) + (1 - \alpha) * mRTT$$

where $0 < \alpha < 1$. Using this new $sRTT$ value, the system updates the observed deviation as follows:

$$Deviation = (\alpha * Deviation) + (1 - \alpha) * |mRTT - sRTT|$$

- From these two figures, the *Retransmission Timeout* (RTO) is calculated by adding the safety margin to the updated $SRTT$ value, i.e:

$$RTO = sRTT + (k * Deviation)$$

From experience, α is set to 0.125. An observation was made that during periods of heavy network load, calculating deviation would become extremely costly. As such, an easier estimation of average deviation is used, i.e.

$$Deviation = |mRTT - sRTT|$$

(3). The system would have to adapt to the changing environment to maintain equilibrium, or attain it again if lost.

Other than RTO, the sender has one other way to detect congestion; receiving multiple ACKs for a packet that has only be transmitted once. The packet sent to the receiver includes a sequence number - representing the unique (and sequential) bytestream number of the first byte in the packet - as well as the size of the payload in bytes. From this, the sender can calculate the bytestream number of the first byte in the next expected packet. This value is encoded into the ACK as the acknowledgement number. If a receiver gets a packet whose sequence number does not match the last acknowledgement number it sent, it resends an ACK for the last packet received correctly. Thus a duplicate ACK represents a packet received in the incorrect order, and serves to notify the sender that a gap exists in the packet transmission. The sender assumes three such dupAcks to be an indication that the packet was lost, and retransmits the packet. This is known as ***fast retransmit***.

Once a packet is retransmitted, the sender has to re-attain the equilibrium lost. TCP Tahoe does this by resetting the value of $ssthresh$ to half of the $cwnd$ value when loss occurred, and resetting $cwnd$ back to its initial value ($1MSS$). After this, the sender returns to the Slow Start stage in an attempt to regain equilibrium.

It may be (and it usually is) that after the initial packet loss, any other attempt to ramp up the transmission rate may lead to $cwnd$ reaching the $ssthresh$ value before congestion is detected again. Once this happens, the sender transitions from slow start into **Congestion Avoidance**. Given that the last congestion event occurred at ($cwnd = ssthresh * 2$), continuing to double $cwnd$ once $ssthresh$ is attained may lead to congestion again. As such, TCP slows down its $cwnd$ growth algorithm, from increasing it by 1 MSS for each ACK to increasing it by 1 MSS each RTT, or rather, by ($1/cwnd$) for each ACK. Tahoe therefore introduced **additive increase (AI)** to TCP's congestion algorithm. Additive increase represents the system's attempts to adapt to the network conditions and regain equilibrium; the sender continues to probe for the optimal sending rate until congestion occurs again.

2.1.2 (c) TCP Reno

While Tahoe, by nature, assumes the worst (heavy congestion) whenever loss is detected, Reno attempts to use a binary signal to determine the ***severity*** of congestion observed. Reno [rfc 2001] differentiates between congestion detected via packet loss and that detected via triple dupAcks. If a sender receives dupAcks, the indication is that the network is still delivering packets to the receiver, though one may be lost or reordered and congestion isn't nearly as severe as that detected via timeout. As such, Reno's response to packet loss depends on the perceived severity of congestion in the network.

If congestion is detected via timeout of a lost packet, Reno behaves in the same way as Tahoe does - retransmit the packet, set $ssthresh$ to half $cwnd$, set $cwnd$ to initial MSS value and proceed with slow start. When a packet loss is detected through triple dupAcks, however, the sender responds through the **Fast Recovery** [jacobson 90] mechanism. Fast Recovery alters the congestion avoidance algorithm to incorporate multiplicative decrease of $cwnd$, as suggested by [chiu 89].

1 When the third dupAck is received, $ssthresh$ is set to half $cwnd$ value at congestion, not to the initial MSS as Tahoe does.
2 Sender enters fast retransmit stage, and retransmits the lost segment without waiting for a timeout.
3 $cwnd$ is set to $ssthresh + (3 * MSS)$. The "3" represents the number of packets that have left the network and have been cached by the receiver (i.e. the three extra packets that caused the triple dupAck). Thus $cwnd$ is reset to about half the size it was before a loss. This means there are double the value of $cwnd$ packets in transit, and the sender must then wait till $cwnd$ dupAcks have been received before it can include new packets in its window.
4 Each time a dupAck is received, increment $cwnd$ by segment size, to account for the packet that has left the network
5 When an ack for new data arrives, reset $cwnd$ to the value of $ssthresh$ in (1) above.

TCP Reno's Fast Recovery algorithm is optimized for cases where a single packet is lost from a window of data [fall 96]. It improves upon Tahoe's ability to re-attain equilibrium after a loss by skipping slow start after a packet is retransmitted due to triple dupAcks, but, like Tahoe, suffers slow recovery when multiple packets are lost in a single window.

2.1.2 (d) TCP NewReno

While Reno improved on Tahoe's handling of congestion severity, it, like Tahoe, was incapable of dealing well with multiple packet losses within a single window. This is because data transmission over the internet tends to occur in bursts; as such, it's likely that multiple packets would be lost within a single window. Both Reno and Tahoe, via their Fast Retransmit mechanism, assume only a single packet has been lost. When faced with bursty losses, Tahoe causes multiple timeouts and refrains from growing the window beyond the initial MSS value for long periods of time due to a loss of ack clocking. Reno, on the other hand, invokes Fast Retransmit and Recovery several times upon encountering multiple losses, leading to a greatly reduced $cwnd$ and throughput. Ack starvation may also occur from the receipt of multiple rounds of dupACKs, as the ACKs encounter wildly fluctuating network conditions and lose their timing capabilities, resulting in bursts of data packets [clark 91], [hoe 95] . This may cause the sender to stall and timeout into slow start.

NewReno was designed with these weaknesses in mind. By modifying the Fast Recovery and Fast Retransmit algorithms on the sender's side, Reno's problems are easily corrected. This is achieved by having Reno send a *partial ack* during Fast Recovery, acknowledging some of the packets that were missing at the start of Fast Recovery. It therefore will not exit Fast Recovery until all the packets that were outstanding at the time it entered Fast Recovery have been retransmitted (rather than going into Fast Recovery for each packet lost). Thus, while in Fast Recovery, two kinds of ACKS may be received:

1 A full ACK, acknowledging all segments that were outstanding at the time of Fast Recovery. Once this is received, the Fast Recovery plan proceeds in the same manner as Tahoe - it exits Fast Recovery, reduces $cwnd$ to $ssthresh$ and continues Congestion Avoidance.
2 A partial ACK. While NewReno was already in Fast Recovery for another packet, a partial ACK would mean that the segment whose sequence number matches the ACK number has been lost. As such, the Fast Recovery congestion window (which is given by $ssthresh + (3 * \mathrm{MSS})$) is reset to $ssthresh$, and NewReno attempts to recover from this new loss without involving timeouts.

NewReno can therefore recover from congestion without waiting for a retransmission timeout, especially when loss is caused by multiple packets in a window getting lost.

2.1.2 (e) SACK TCP

SACK TCP is an extension of Reno which extends the partial ACKS concept, but retains the Slow-Start, Congestion Avoidance and Fast Retransmit portions. In it, TCP headers include a **Selective Acknowledgement** field that affirms the successful receipt of a non-contiguous block of packets. During Fast Recovery, TCP maintains a *pipe* variable that represents the estimated number of packets that are still outstanding and only retransmits a packet or sends a new one when pipe value is less than $cwnd$. Each time it receives an ACK or retransmits a packet, pipe is incremented by 1, and when a dupAck is received, pipe is decremented by 1. Whenever pipe falls below $cwnd$, the sender either retransmits any missed/lost packets or sends new data. This helps decouple the decision of *when* to send from *which* packet to send. [fall 96].

III. HIGH SPEED TRANSPORT PROTOCOLS

3.1. Problems with TCP

As the link capacities of networks increased, the following issues were detected in TCP.

Poor Scalability - Due to TCP's conservative and linear increase function, it takes a long time for TCP to take full advantage of the network's capabilities. High link capacities imply that the links have a high BDP as well. From § 2.1.1:

$$T \leq \bar{\bar{W}}/_{RTT} \leq C$$

$$\Rightarrow T \leq \bar{W} \leq C * RTT$$

This implies that for TCP to take full advantage of the link's capacity, the average window size must be as close to the BDP as possible. Scaling to such lengths requires not only a much less conservative increase function, but also a bit error rate of at most $2 * 10^{-14}$, which is far lower than can currently be sustained [rfc 3649].

Poor Convergence - TCP's initial convergence and fairness tests were modeled upon the assumption of infinitely long flows that would, eventually, exhibit long-term fairness. Recent studies [leith 07], [jin 04] have shown that TCP takes far longer to converge to fairness than most flows last; a majority of flows, in fact, never leave the slow start stage [duki 06]. This means that if a new flow starts after an old one is already established, the second flow takes a long time to converge to a steady state and fair share of network bandwidth.

RTT Unfairness - TCP's congestion control policy responds to network conditions based on RTT-clocked feedback. TCP's throughput equation shows that the throughput of a flow is inversely proportion to the RTT, i.e:

$$T \leq \bar{W}/_{RTT}$$

Therefore, keeping loss constant, for two flows i and j, the ratio of their throughput is given by:

$$FR_{ij} = \frac{T_i}{T_j} = \frac{RTT_j}{RTT_i}$$

where FR_{ij} represents the *fairness ratio* of the two flows. Thus flows with shorter RTTs probe for (and acquire) higher bandwidth share faster than those with higher RTTs [rfc 3649]. This means that TCP allows flows with short RTTs to acquire a greater share of the throughput than those with longer RTTs. This unfairness has been shown to increase in the prescence of synchronization [lakshman 97].

Attempts to relieve this unfairness have concentrated mostly on the congestion avoidance algorithm, since this is deemed to be the main portion of a steady, long-lived flow. However, analytical study has shown that the Slow Start algorithm, network characteristics (e.g. persistent queuing on bottleneck links, and the independence of *ssthresh* from the RTTs of the flows) also lead to this unfairness [gavaletz 10].

Buffer Overload - TCP's congestion avoidance policy fills up the buffers and only detects congestion when a packet is dropped. This has a number of disadvantages, viz:

1 A drop-based congestion avoidance policy favors large buffers – at first glance, increasing buffer size appears to be a solution to the dropped-packets problem. These large buffers, however, are costly, especially with the requirements of high RTT and high bandwidth links.
2 The coarse granularity (up to 500 ms) of system clocks used to calculate timeouts leads to an improbably large delay if loss is detected via congestion. Experimental evidence has shown that loss may not detected and a packet retransmitted for up to 1100 ms after a segment was originally sent, rather than the 300 ms it should take on a transcontinental link [brakmo 95].
3 The large queuing delays experienced on links with large buffers are a hindrance to interactive and real-time applications.

These problems led to a need to develop alternative high-speed protocols whose growth function wasn't as conservative as TCP's. This led to the development of **high speed TCP protocols**.

3.2. Loss-Based High Speed Protocols

These are high-speed protocols whose congestion-avoidance algorithms aggressively increase the size of their congestion window to improve their throughput and efficiently use the high-speed link. They mirror TCP in their behavior, in that they rely purely on packet loss and triple dupAcks as an indication of congestion. However, in an attempt to deal with the issues described in § 3.1 above, their congestion-avoidance algorithms adopt much less conservative increase functions. Examples of these include:

1 **STCP -** Scalable-TCP [kel 03] uses fixed increase and decrease parameters to make the recovery time after a packet loss independent of window size. It uses growth functions similar to TCP, but increases the window by a much larger additive increase and reduces it by a much smaller multiplicative factor. When the window size is below a pre-defined threshold, it uses TCP's congestion control algorithm in order to maintain TCP fairness.
2 **HSTCP** - HighSpeed TCP [rfc 3649] uses an AIMD algorithm that increases and decreases the congestion window depending on its current value. The increase and

decrease parameters, therefore, are a function of the currently observed *cwnd*. To successfully take advantage of high BDP links yet ensure TCP fairness, HSTCP behaves more aggressively than TCP in low loss-rate environments by slowing down its growth rate as link capacity increases (using smaller increase parameters) and becomes quite passive in high loss rate environments. Much like STCP, HSTCP uses TCP's congestion control algorithm below a threshold value. It uses lookup tables for determining the values of the increase and decrease parameters.

3 **CUBIC -** CUBIC [ha 08] employs a binary search algorithm to probe the network for an optimal bandwidth. Using a cubic function (rather than TCP's linear window growth), it aggressively increases the window size when the network is far from saturation, and reduces its rate of growth as it approaches saturation. The growth function parameters are based on the length of the congestion epoch (i.e. time between consecutive congestion events), making it independent of RTT and promoting RTT fairness.

4 **HTCP -** H-TCP [leith 04], like CUBIC above, increases its congestion window based on the time between successive congestion events. It also uses the ratio of the minimum-observed RTT to the maximum-observed RTT, in an attempt to promote RTT fairness. The AIMD decrease factor is adapted to respond to estimated queue provisioning on a link, in order to improve link utilization.

Loss-based protocols efficiently scale to take up the bulk of the link's available throughput. However, loss-based protocols exhibit RTT and TCP unfairness [tan 06]. For normal TCP flows, bias against flows with long RTTs can be alleviated by adding randomized background traffic. However, even in the presence of background traffic, high-speed flows remain susceptible to global synchronization. This greatly increases RTT unfairness; longer flows are penalized and their throughput reduced when buffers are full of packets from shorter flows. TCP unfairness is caused by the aggressive increase function needed to scale efficiently.

3.2. Delay-Based High Speed Protocols

By default, loss-based congestion control protocols fill up router buffers and induce loss before detecting congestion. The resulting large network delays are one of the reasons that lead to a preference among network administrators to over-provision backbone links [diot 02] and not utilize the network to full capacity. This has led to a need to develop protocols that would maintain low buffer occupancy while maintaining a high level of utilization.

In [chiu 89], the relationship between network load and throughput is as described in the graphs below. The authors identify a region known as the "knee" of the throughput curve, beyond which increasing the network load would lead to minimal throughput gains but significant latency due to queuing. This is shown in the figure below[5]:

[5] Image courtesy of D. Chiu, R. Jain. Analysis of increase and decrease algorithms for congestion avoidance in computer networks. *Computer Networks and ISDN systems 17: 1–14.*

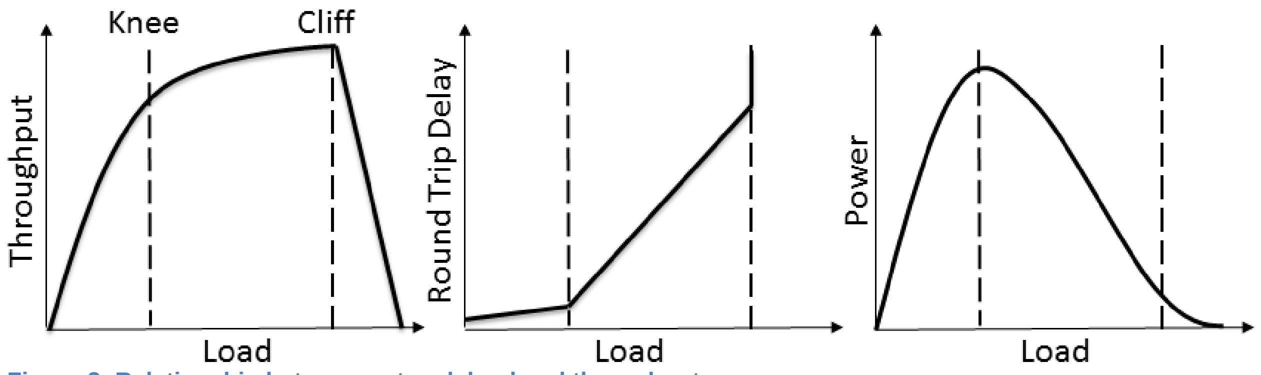


Figure 3: Relationship between network load and throughput.

From this observation, a congestion avoidance protocol that would attempt to maintain throughput, power and RTT at the knee of the curve, irrespective of the buffer size, was suggested. A potential solution was to use delay, not as an indicator of packet loss, but of the latency caused by queuing caused by congestion. Delay-based high speed protocols attempt to fill this need. Examples of such protocols are TCP-Vegas, Compound TCP, FAST TCP and RAPID.

3.2.1 TCP Vegas

TCP Vegas[6] represents an attempt to achieve better throughput and reduce losses by improving TCP Reno's congestion avoidance algorithm on the sending side. It is, strictly speaking, **not** a high-speed protocol, but rather, an early attempt (one of the first) to improve TCP Reno by making it sensitive to delay as well. Vegas aims to achieve this in three ways, viz:

1) An Improved Retransmission Mechanism.

TCP Reno detects congestion through retransmission timeouts and triple duplicate ACKs. As mentioned earlier, however, the coarse granularity of system clocks makes retransmission timeouts an expensive option for networks, leading to devastatingly low throughput. Triple dupAcks were meant to relieve this problem; however, they don't entirely eliminate the reliance on timeouts.

The goal of TCP Vegas is not to eliminate timeouts (as calculated by Reno) entirely, but rather, to detect packet loss even in situations where a dupAck is unlikely to be received, without relying entirely on Reno's timeout mechanism. Vegas proposes using the clock to record timestamps when a segment is transmitted and when its ACK is received. This information is used to get an accurate estimate of the RTT of a flow, and is used to determine when to retransmit a packet, as follows;

(a) When a dupAck is received, Vegas compares the (current) RTT of the dupack-leading packet (i.e. the received packet that caused the dupAck) and the previously recorded RTT (i.e. that of the packet acknowledged by the dupAck, presumed lost or re-ordered). If the difference between the current RTT and the recorded RTT exceeds a timeout value, the packet

[6] Symbolic notation adopted from original publication [brakmo 95]

is retransmitted without waiting for a set of triple dupAcks. This alteration of the fast restransmit algorithm was inspired by the observation that if congestion is high, it's unlikely that enough packets will make it to the receiver to trigger triple dupAcks, and a timeout is bound to occur. Not waiting for the dupAcks reduces the need for a timeout mechanism.

(b) When the first or second non-dupAck after retransmission is received, Vegas checks to see if the RTT is greater than the timeout value. If it is, the packet is retransmitted. This helps Vegas deal with multiple packets within a window getting lost, a situation that Reno (and Tahoe) deal with rather poorly.

An illustration of this is shown below[7]:

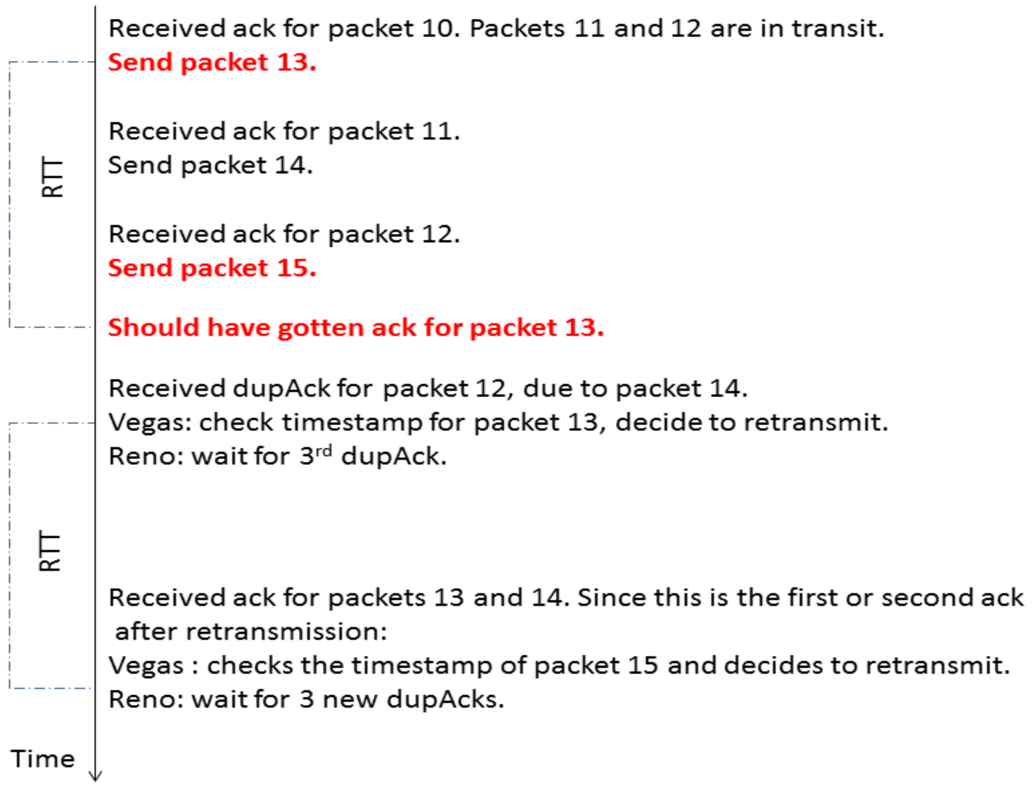

Received ack for packet 10. Packets 11 and 12 are in transit.
Send packet 13.
Received ack for packet 11.
Send packet 14.
Received ack for packet 12.
Send packet 15.
Should have gotten ack for packet 13.
Received dupAck for packet 12, due to packet 14.
Vegas: check timestamp for packet 13, decide to retransmit.
Reno: wait for 3rd dupAck.
Received ack for packets 13 and 14. Since this is the first or second ack after retransmission:
Vegas : checks the timestamp of packet 15 and decides to retransmit.
Reno: wait for 3 new dupAcks.
RTT
RTT
Time

Figure 4: Sample execution of TCP Vegas congestion control algorithm.

2) A Delay-based Congestion Avoidance Mechanism

Vegas proposes being a proactive protocol, rather than being reactive, i.e. to detect and respond to congestion at its initial stages, rather than at its loss-inducing zenith. Vegas' approah to congestion avoidance is as follows:

[7] Image courtesy of L. Brakmo, L. Peterson, S. O'Malley. TCP Vegas: New Techniques for Congestion Detection and Avoidance. ACM SIGCOMM Computer Communication Review Oct. 1994

First, Vegas keeps track of the lowest-observed RTT. This minimum is normally the RTT of the first (few) packets, and is recorded as the baseRTT. If the connection is not congested, then the expected throughput (ExpectedTput) can be calculated as follows:

$$ExpectedTput = cwnd \,/\, BaseRTT$$

Since Vegas keeps track of the RTT of the last acknowledged segment, $sRTT$, then the actual throughput (*ActualTput*) can be calculated as well:

$$ActualTput = cwnd \,/\, s\,RTT$$

From these two values, Vegas calculates the difference, diff, as follows:

$$diff = ExpectedTput - ActualTput$$

Intuitively, Vegas attempts to set the throughput within a region that achieves as high utilization as possible, yet maintains low queuing delay. It therefore sets two thresholds representing the minimum and maximum values of this region. However, a more accurate representation is the number of packets each of these throughputs maintains in the buffer. These values, α and β (usually 2 and 4 resp.), determine the linear increase or decrease of the congestion window in each RTT as follows:

$$cwnd = \{cwnd + \phi\}, diff > \alpha, \alpha > 0$$

or

$$cwnd = \{cwnd - \theta\}, diff < \beta, \beta > 0$$

where ϕ and θ represent the linear increase and decrease factors, respectively.

3) An Improved Slow-Start Mechanism

On long-lived flows, TCP's initial slow-start foray, by design, almost always ends through a loss. This is because the initial $ssthresh$ value is set to an impossibly high throughput. Vegas tries to avoid that loss, and yet maintain the exponential growth of $cwnd$ by incorporating the delay-based structure of congestion avoidance into slow-start. The modified version exponentially increases $\boldsymbol{cwnd}$ every other RTT, if the actual rate is equal to the expected rate. If it falls below the expected rate, Vegas switches to the linear increase/decrease of congestion avoidance.

3.2.2. Compound TCP

Compound TCP[8] [tan 06] is designed with the competitive weakness of other delay-based protocols in mind. By combining standard TCP's loss-based congestion control with a delay-based component, CTCP aims to achieve the quick scaling of high-speed loss-based protocols

[8] Symbolic notation adopted from original publication [tan 06]

with the RTT and TCP fairness of delay-based protocols. At the same time, it attempts to maintain the competitive edge required in network environments dominated by loss-based protocols.

CTCP adds a delay-window variable, $\boldsymbol{dwnd}$, to the conventional TCP structure, to control the delay-based component of CTCP. Hence the sending window $s\boldsymbol{wnd}$ is incremented by both the regular increment in TCP and the delay-based component's increase. Thus;

$$swnd = min\,(cwnd + dwnd, rwnd)$$

where $rwnd$ is the receiver's advertised window.

Upon receiving an ACK, the conventional congestion window is increased as follows:

$$cwnd = cwnd + 1/(cwnd + dwnd)$$

The delay-based component keeps track of the minimum RTT encountered through the $\boldsymbol{baseRTT}$ variable, and uses it to find out how many packets are in the queue. Using a weighted measure of current rtt, $sRTT$, CTCP calculates the number of queued packets $diff$ as:

$$ExpectedTput = swnd\ /\ baseRTT$$

$$ActualTput = swnd\ /\ sRTT$$

$$diff = (ExpectedTput - ActualTput) * baseRTT$$

CTCP detects congestion if diff is larger than a preset threshold value γ, set to 30 packets as a tradeoff between TCP fairness and avoiding false early congestion detection. So the overall window $swnd$ is increased binomially as follows:

$$swnd\,(t + 1) = swnd\,(t) + \alpha * swnd(t)^k$$

where when no congestion is detected, and reduced multiplicatively, i.e.

$$swnd\,(t + 1) = swnd\,(t) * (1 - \beta)$$

when congestion is detected, with and α, β and k empirically tuned to provide desirable scalability, smoothness and responsiveness, and can be altered at will.

While the loss-based component increases and decreases the window in line with standard TCP conventions, the delay-based component adds the aggressive yet RTT-sensitive portion of the high-speed delay-based protocol. Its window component dwnd is changed as follows:

$$dwnd\,(t + 1) = dwnd(t) + \left(\alpha * win(t)^k - 1\right)^+, \quad \boldsymbol{diff} < \boldsymbol{\gamma}$$

$$dwnd\,(t+1) = \left(dwnd(\mathrm{t}) - \zeta * diff\right)^{+}, \qquad \boldsymbol{diff \geq \gamma}$$

$$dwnd\,(t+1) = \left(swnd(t) * (1-\beta) - cwnd\,/\,2\right)^{+}, \qquad \boldsymbol{diff \geq \gamma}$$

where $(.)^{+}$ = max (., 0), and ζ is the delay-based window's decrease parameter.

Thus, when no congestion is detected and no packet is lost, CTCP's sending window quickly probes network bandwidth. Since all CTCP flows attempt to maintain the same number $\gamma$of packets in the queue, CTCP is conceptually RTT fair. CTCP's delay-based component kicks in only in the congestion avoidance phase, and since dwnd is never less than 0, CTCP is lower bound by TCP's performance metrics, successfully solving the lack of competitiveness suffered by other loss-based protocols such as TCP-Vegas.

3.2.3. FAST TCP

Being an packet-switched protocol, TCP semantics operate on datagram granularity. In designing FAST TCP[9] [jin 04] shows that the characteristics of a single flow - and thus, of the aggregate traffic throughout the network - are determined by the packet-level decisions made at the end points. The conservative linear increase at the packet level means the flow has to maintain a very low loss rate in order to successfully scale. Similarly, $cwnd$ oscillations due to packet loss cause the flow to be unstable.

Motivated by these issues, FAST TCP seeks to improve on TCP's performance by separating congestion control into four functionally independent modules. These are:

1) Data Control

This component determines which packet to send. It selects the packet to send from three different categories, viz:

(a) new packets
(b) packets deemed lost
(c) transmitted packets yet to be acknowledged.

The data control portion has to make sure that recovery from a loss is fast enough to avoid disastrous throughput losses, yet controlled enough to avoid compounding the congestion.

2) Burstiness Control

This component determines when to transmit packets. It smoothes down the transmission of packets into a fluid-like stream. This makes it easier to estimate available bandwidth, as it keeps

[9] Symbolic notation adopted from original publication [jin 04]

the sender from having stale information gathered from intermittent transmissions. In large BDP environments, burstiness may lead to a large number of packets being introduced to the network at once. This would lead to long queues and an unstable, oscillating flow.

FAST combines two modules to control the burstiness of a flow, i.e:

- a **burstiness reduction** algorithm, a self-clocking mechanism at low BDP values that decides how many packets to transmit when an ACK acknowledges a large number of packets. This avoids sudden increases to $cwnd$ that may suddenly lead to congestion.
- a **window pacing** mechanism gently increases the congestion window during the sender's idle periods. This increment is determined by the window control component, taking the RTT values so far observed to estimate the amount of growth the window would experience over the idle period. This reduces the flow's oscillation to manageable levels.

3) Estimation

This provides estimations on how of various input parameters to the other decision-making components. The estimation component, upon receiving an ACK for packet from a flow i, records the k^{th}RTT sample $sRTT$ of the acked packet. It then updates a smoothed queuing delay value $\underline{T}(k + 1)$ as follows:

$$\underline{T_i}(k + 1) = (1 - \eta(t_k))\,\underline{T_i}(k) + \eta(t_k)\underline{T_i}(k)$$

where

$\eta(t) = min\,\{3/cwnd_i(t), 1/4\}$
t_k = time k^{th} RTT sample $sRTT$ of the acked packet was taken, and
$cwnd(t)$ is the congestion window at time t.

Taking $baseRTT$ to be the lowest RTT observed so far, the average queuing delay $\hat{q}_i(k)$ is estimated as follows:

$$\hat{q}_i(k) = \underline{T_i}(k) - baseRTT$$

If every packet is acked, the estimation component provides an entire window's worth of RTT samples, giving a more fine-grained measure of congestion than packet loss. If delayed acks are used, the number of values may fall by up to a half of the window size, but the multi-bit output would still provide more information than the binary signal in packet loss.

4) Window Control

This controls the congestion window $cwnd$'s size, based on the information provided by the estimation module. FAST responds to both packet loss and queuing delay, and unlike TCP, attempts to use the same window control function irrespective of sender state. Under normal network conditions, FAST updates its congestion window as follows:

$$cwnd = min\{2 * cwnd, (1-\gamma)cwnd + (\frac{baseRTT}{sRTT} * cwnd + \alpha(cwnd, \hat{q}_i(k))\}$$

where $\gamma \epsilon (0, 1]$, $baseRTT$ is the minimum observed RTT so far, $sRTT$ is the last observed RTT, and $\hat{q}_i(k)$is the average queueing delay. The constant α, whose value is empirically determined, represents the number of packets each flow attempts to maintain in the network buffers at equilibrium, and the window control component carries out the update of $cwnd$ every other RTT. Thus, while TCP Vegas [brakmo 95] increases or decreases $cwnd$ by 1 packet per RTT irrespective of how far the queue size is from the protocol's target, the magnitude of FAST's $cwnd$ changes depending on how far the queue size is from the intended target α.

When a loss occurs, FAST halves the congestion window.

3.2.4. TCP-Illinois

TCP Illinois incorporates both delay and loss into its congestion control algorithm under the following assumptions:

1) delay remains a useful signal, since congestion and packet loss have a direct bearing on the delay encountered
2) delay remains inaccurate, especially in networks prone to cross-traffic. Thus the correlation between delay and loss is weak.

In an attempt to improve the fairness, scalability and stability of TCP, TCP-Illinois relies on packet loss as the primary congestion determinant, affecting the *direction* of window size change, and delay as a secondary determinant, controlling the *pace* of window size change. This allows it to achieve a concave window curve; Illinois is therefore also known as a *Concave-AIMD* protocol.

Unlike Compound, which uses *both* packet loss and delay as primary determinants (i.e. to set the size of the window), TCP-Illinois[10] uses the round-trip delay d_a only to alter the increase and decrease functions, and not to determine whether to increase or decrease. This is in order to avoid the inaccuracies encountered when measuring delay, caused by noisy links [liu 06], leading to a more robust behavior.

TCP Illinois works by setting its window increase function α large when the network is far from congestion and small when congestion is imminent. Similarly, its window decrease function β is large when congestion is imminent and small otherwise. This means that in its initial stages, the throughput increase is high, and tapers off as the link fills up; similarly, if a loss occurs in the initial stages, the decrease is larger than if the loss occurred when the network was congested. This growth function leads to Illinois having a Cubic-like concave window curve.

Given that the imminence of congestion is detected through changes in delay, the increase and decrease functions can be written as:

[10] Symbolic notation used here is derived from the original publication [liu 06]

$$\alpha = f_1(d_a) = \begin{cases} \alpha_{max} \; if \; d_a \le d_l \\ \dfrac{k_1}{k_2 + d_a} otherwise \end{cases}$$

$$\beta = f_2(d_a) = f(x) = \begin{cases} \beta_{min}, & if \; d_a \le d_2 \\ \kappa_3 + \kappa_4 d_a, & if \; d_2 < d_a < d_3 \\ \beta_{max}, & otherwise \end{cases}$$

where κ_1 - κ_4 are protocol parameters, d_a is the average queuing delay, d_l and d_m are the low and high queuing delay thresholds, while d_2 and d_3 are the queue sizes at β_{max} and β_{min} respectively.

The TCP-Illinois protocol behaves as follows:

3) All features of TCP NewReno are maintained, including fast recovery and fast retransmission. Only the AIMD semantics are changed.
4) In congestion avoidance, the sender determines the average RTT T_a by measuring the RTT of each ack and dividing it over the last W acks, where W is the current window size. From this, it computes the following values:
   i. T_{min} and T_{max}, the maximum average RTT computed yet
   ii. $d_m = T_{max} - T_{min}$, which gives the maximum queuing delay
   iii. $d_a = T_a - T_{min}$, which gives the current average queuing delay

5) The values of $0 < \alpha_{min} \le 1 \le \alpha_{max}, 0 \le \beta_{min} \le \beta_{max} \le 1/2, W_{thresh} > 0$ and $\kappa_i, (i = 1 \; to \; 4)$ are computed and set.
6) The values and $\kappa_i, (i = 1 \; to \; 4)$ are updated each time T_{min} and T_{max} are updated (this could happen every RTT). The α and β values are updated once each RTT. This constitutes the main use of delay in the protocol.
7) If the window size W is greater than the threshold W_{thresh}, Illinois reverts to TCP behavior; $\alpha = 1$ and $\beta = {}^{1}/_{2}$. Thus:
   i. $W = W + {}^{\alpha}/_{W}$ for each ACK received, and
   ii. $W = W - \beta * W$, if in the last RTT, a packet loss was detected.

8) After a timeout, the sender enters fast recovery and retransmit ($W_{thresh} = W/2$, $\alpha = 1$ and $\beta = 1/2$). The values of α and β remain unchanged until one RTT after slow start ends.

IV. ROUTER BUFFER DESIGN

4.1. Introduction

Internet router buffers play an important role in molding network traffic. The number of packets that can be accommodated during congestion determines whether losses occur or not, affects the robustness of the underlying transport protocols, sets the magnitude of latency experienced and the utilization of the egress and all other antecedent links. As such, it is important that router buffer sizes are set to an optimal amount. Two factors must be taken into account when determining the size of a router buffer:

1 The router buffer must be able to accommodate the bursty nature of network traffic. While attempts have been made to pace TCP traffic to reduce its bursty nature [cai 09] [towsely 06] [agar 00] [coulter 00], a variety of factors may still lead to having too many packets arrive at a network link; user behavior [hc 06], introduction of new, heavy TCP or congestion-unresponsive flows into the network, etc. Router buffers must be big enough to accommodate these packets and avoid packet loss.

2 The router buffer must be big enough to ensure the egress link remains at acceptably high levels of utilization. TCP is the workhorse of internet traffic, and being an AIMD protocol, congestion windows are halved when loss occurs. This temporarily halts transmission while the unacknowledged packets are transmitted and the lost packet(s) re-sent. If the queues are too small, the buffers are bound to drain and leave the egress link idle.

4.2. Router Buffer Size Models

4.2.1. Bandwidth Delay Product Model

The BDP rule of router buffer sizing dictates that given a bottleneck link has a capacity C and the effective two-way propagation delay RTT, then the buffer size $B = \kappa(RTT * C)$, where κ is a measure of proportion..

A single long-lived TCP flow exhibits sawtooth behavior as shown in the image below. Once the flow is in Congestion Avoidance, the window size rises slowly till its zenith, W_{max}, where loss occurs. At this point, the window size is halved and the flow resumes the additive increase.

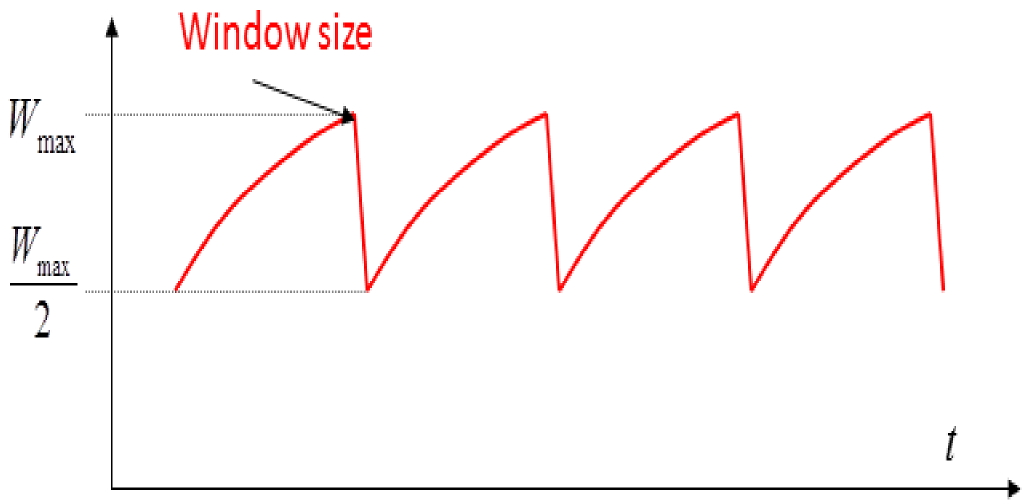


Figure 5: Sawtooth behaviour of single TCP flow.

As explained in [app 04], assuming a buffer size of B packets and a bottleneck link rate C, when a loss occurs and the congestion window is halved, the number of outstanding packets in the network is twice the window size after congestion. Thus, for the sender to start transmitting packets again, $W_{max}\ /\ 2$packets must be acknowledged. The amount of time it takes to transmit these $W_{max}\ /\ 2$ packets over a link of capacity $C$and get their acks back is:

$$time\ =\ (W_{max}\ /\ 2)\ /\ C$$

Meanwhile, the buffer is full, and continues to transmit packets over the bottleneck link. The amount of time it takes to drain the buffer of size B of all the packets within is:

$$time\ =\ B\ /\ C$$

To maintain high link utilization, the next packet from the sender has to arrive at the router just as the last buffered packet is transmitted. Thus the time it takes to transmit the $(W_{max}\ /\ 2)$ packets and receive their acks should be upper bound by the time it takes to drain the buffer, i.e.:

$$(W_{max}\ /\ 2C)\ \leq\ (B\ /\ C)$$

which implies

$$(W_{max}\ /\ C)\ \leq\ B$$

Meanwhile, given the path has a round trip time value RTT, then the sending rate at the router can be derived as:

$$C\ =\ (W_{max}\ /\ 2)\ /\ RTT$$

which implies

$$B \leq (W_{max} / 2) = C * RTT$$

This shows that the best buffer size for one long-lived TCP flow is equal to the bandwidth delay product. This rule, suggested in the infancy of TCP's congestion avoidance [jacobson 90], was proven through simulations showing that full utilization could be achieved for a small number of flows (1 – 8) if the buffer size was equal to the BDP of the link [song 94]. With such a small amount of flows, synchronization causes the flows to (almost) simultaneously experience loss, leading to the same sawtooth-pattern aggregate flow that a single TCP flow exhibits.

The BDP model is meant to maintain high utilization in the egress link. It was developed after the realization that a poorly buffered bottleneck link is susceptible to bi-stable behavior - alternating periods of high utilization and low utilization - as well as leading to premature loss before full link utilization.

4.2.2. Stanford Model, aka The Small Buffers Rule

The Small Buffers rule is also meant to measure the effectiveness of a buffer's size by measuring the utilization of the bottleneck link. It takes advantage of the fact that when a gateway has a large number of flows, the flows are unlikely to be synchronized. As such, the out-of-phase flows have their sawtooth patterns cancel each other out according to the central limit theorem, leading to the aggregate sawtooth having a much smaller variation in overall flow throughput [enach 05].

The Small Buffer rule calls for buffer sizes to be reduced by a factor of $\sqrt{N}$, where N is the number of long-lived flows sharing the link [app 04]. This reduces buffer size by up to two orders of magnitude. Experimental evidence [beh 08] shows that for a throughput T and a sending rate C, when the buffer size B falls below $T * \frac{C}{\sqrt{N}}$ hardly any utilization is lost.

4.2.3. UMass Model, aka The Tiny Buffers Rule

The Tiny Buffers rule was motivated by research aimed at making all-optical routers a possibility [tow 06]. It has been shown that gateways with very small buffers are feasible if the small resultant sacrifice in utilization is an acceptable tradeoff [rai 05]. Alternatively, TCP can be modified to pace out the transmission rate to reduce the occurrence of burstiness [aggar 00], or decoupling congestion control and reliable data transmission while maintaining the bottleneck link at a steady state of congestion [tows 066]. In networks where the access links are slower than the core links, the natural smoothing of the packet arrival rate into the core routers means that buffers with 10-20 packets are enough to maintain utilization at around 90% [mckeo 06]. With the over-provisioning of core network links, this represents a small sacrifice to make for much smaller and cheaper buffer space.

4.2.4. Other Router Buffer Models

Other buffer router schemes include the Drop-Based Buffers rule (aka Ga. Tech model), created for access links [dham 05], which attempts to set router buffers to the minimum size required to achieve utilization, loss and delay thresholds. Attempts to reconcile the varying router buffer sizing theories have called for buffer size to be set to a value proportional to the number of TCP connections at the router [jef 05], or the number of incoming links [gor 05]. Other models, spurred by the fact that previously mentioned buffer sizing rules relied on some estimation of the number of long-lived flows N or the average RTT of the flows, have attempted to define adaptive buffer sizing models which do try to bind N or RTT but rather let the router adapt its buffer size to reflect the volatility of the networking environment [xan 10], [stan 06], [log 08].

V. MOTIVATION

The overarching motivation for the development of new protocols and router buffer designs is the desire to reduce the amount of time it takes for useful data transmission. As mentioned before, both of these are almost always develop independently. The predominant use of and reliance on loss-based protocols means that router buffers are designed primarily to suit protocols that are meant to drop packets when congestion occurs, and not respond to the congestion imminent from increasing delay. While a number of studies have been done on the effectiveness of various protocol and router-buffer design combinations, there exists a dearth of useful research on the confluence of delay-based protocols and router buffer sizes.
Inspired by the desire to develop all-optical networks, router-buffer sizes have trended toward smaller sizes in the last few decades [enac 05], with a few exceptions, such as [dham 05]. These buffers attempt to maintain high throughput while keeping queuing delay at a minimum, by making sure the number of packets queued remains low. While this is important for loss-based protocols, it is even doubly so for delay-based protocols, which either have the queue size as a distinct control variable (as Vegas and Fast) or rely on values derived from properties of queue size as control variables (as Compound and Illinois do). The confluence of present-day evaluations of router buffer sizes and delay-based protocols either evaluates a few protocols over a single router buffer size [lieth 08], or a slew of router buffer sizes over with one delay-based protocol. Lacking is a combination of both – an evaluation of the major delay-based protocols, over a significant range of router buffer sizes, investigating the effects under different sets of background traffic, RTT and bandwidth combinations.

The eventual development of all-optical networks, as mentioned before, would suggest that queue sizes fall to a few dozen packets at most. It appears likely that if tiny buffers are eventually accepted as the *modus operandi*, the performance of delay-based congestion avoidance algorithms would be severely hampered, since the queue sizes may fall well below their queue-size control variables. Understanding the behavior of these protocols under small router buffer sizes is an important step towards creating delay-based protocols that would stand the test of future all-optical network switches.

5.1. Related Works

A number of studies have been done to test the efficacy of high-speed protocols under different sets of conditions. In [leith 07], the authors perform a series of benchmark tests to compare the performance of several high-speed protocols under different simple network environments. By comparing their performance with TCP, they test RTT fairness, TCP friendliness, responsiveness to network changes and scalability of various high-speed protocols. In it, they find that most high-speed protocols exhibit significant RTT unfairness and poor responsiveness to rapidly-changing network environments. In [leith 08], the authors extend their previous work to present an in-depth study of Compound TCP and TCP Illinois, two protocols that use both delay and loss to detect congestion. Using an experimental lab testbed and implementations in Vista and Linux respectively, they observe that both Compound and Illinois exhibit poor scaling behavior in high BDP environments, but have tolerable TCP fairness. Both, however, appear to suffer throughput loss in the presence of reverse traffic, possibly due to ack compression or queuing on the reverse path.

Through ns-2 simulations, [has 08] and [has 09] study the effects of router buffer sizing on loss synchronization in loss-based high-speed flows, and presenting results with and without RED-enabled routers. Flows generally become synchronized when they suffer drops at roughly the same time. This tends to significantly reduce the utilization of egress links, as well as increasing the RTT unfairness experienced by long-RTT flows. Here, HighSpeed was found to exhibit the highest level of loss synchronization across most router buffer sizes. As shown in [bohacek 05], while randomized background flows may prevent phase effects in regular TCP flows, this doesn't seem to be the case for high-speed flows. Instead, the results in [has 09] were confirmed, that using random queue drop algorithms (such as RED) better helps mitigate synchronization in high speed TCP flows. Both studies, however, explore a small set of delay-based protocols and use a limited set of network conditions. In [bul 03], measurements are taken on production networks, comparing the performance of TCP NewReno with loss-based high-speed protocols. Using a range of RTTs and throughput links, the authors compare the throughput, stability and intra-protocol fairness of various high-speed protocols. In their experiments, BIC-TCP appears to exhibit the best overall performance, with Scalable being too aggressive on short links and HighSpeed exhibiting odd fairness behavior.

VI. EXPERIMENTAL METHODOLOGY

The simulations are run over NS-2 using a dumbbell topology. Two routers are connected to each other using a bottleneck link and to three nodes apiece. The three nodes are used to generate the following flows, in order of initiation:

1 A single long-lived TCP NewReno flow (henceforth called the *pre* flow), from the node *Pr.* This flow starts at the beginning of the simulation.
2 A single *test flow, Hs*. This is the flow that represents the protocol being tested. The protocols under observation are Compound, Illinois and FAST. These are compared to test scenarios involving a NewReno TCP test-flow, which is used as a base calibration for each scenario. This flow begins at the 50s point of the simulation.
3 A second single long-lived TCP NewReno flow (henceforth called the *post* flow) from the node *Po*. This flow begins at the 100s point of the simulation.

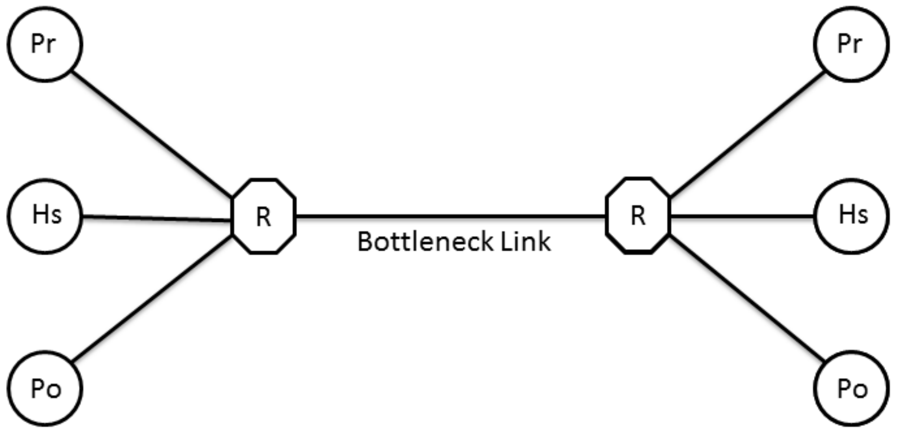


Figure 6: Experimental Setup.

The end nodes are connected to the routers via 1Gbps links, and the direction of test is for flows from router A to router B in the figure above. The propagation time from the node to the closest router is randomized between 3ms and 11 ms, to avoid phase effects. Each of the three flow categories is allowed to stabilize before the next flow starts. The first and third long-lived flows are used to test for convergence time and TCP friendliness of the test flow. Each simulation is run for 700 seconds, and data is collected and analyzed from the time the test flow starts, giving 600 seconds worth of simulation data. The metrics used to analyze the performance of the test flows are throughput and link utilization, burstiness, and convergence time of the high-speed flows, as compared to a NewReno TCP flow under the same conditions.

The values of the network conditions used are:

- RTT: 50ms, 100ms.
- Bottleneck bandwidth: 500Mbps, 1000Mbps

- Buffer model:
  - BDP – the largest bandwidth delay product was used (correlating to the 1000Mbps/100ms environments), i.e. 8,300 packets.
  - Stanford – assuming ~ 2500 long-lived flows, the Stanford router buffer size was 1/50th of the BDP router, i.e. 166 packets
  - UMass – the router buffer size was 20 packets.

- Protocol: Compound, Fast, Illinois.

Each {RTT, bandwidth, buffer model, protocol} tuple represents a single test scenario.

VII. RESULTS

7.1. Calibration: Individual Flow Behavior

Before the experiments were carried out, the simulations were run without the pre and post NewReno flows, in order to observe the behavior of the individual flows. For each {buffer, bandwidth, rtt}, combination, the graphs below show the throughput of each flow in isolation.

7.1.1 BDP Router Buffer Size

Under the 8300-packet router buffers, Compound and Illinois are quickly able to scale and thoroughly utilize the link. Fast, while exceedingly stable, remains hampered by its buffer occupancy parameter α, and attains far lower throughput. As the rtt increases in the 500Mbps environment, NewReno, as expected, scales much slower than Compound and Illinois (figs. 7 and 8)

Under larger bottleneck bandwidth simulations, Illinois and Compound still appear to rapidly scale and take up as much bandwidth as possible. NewReno follows suit, while Fast is again held back severely (figs 9 and 10).

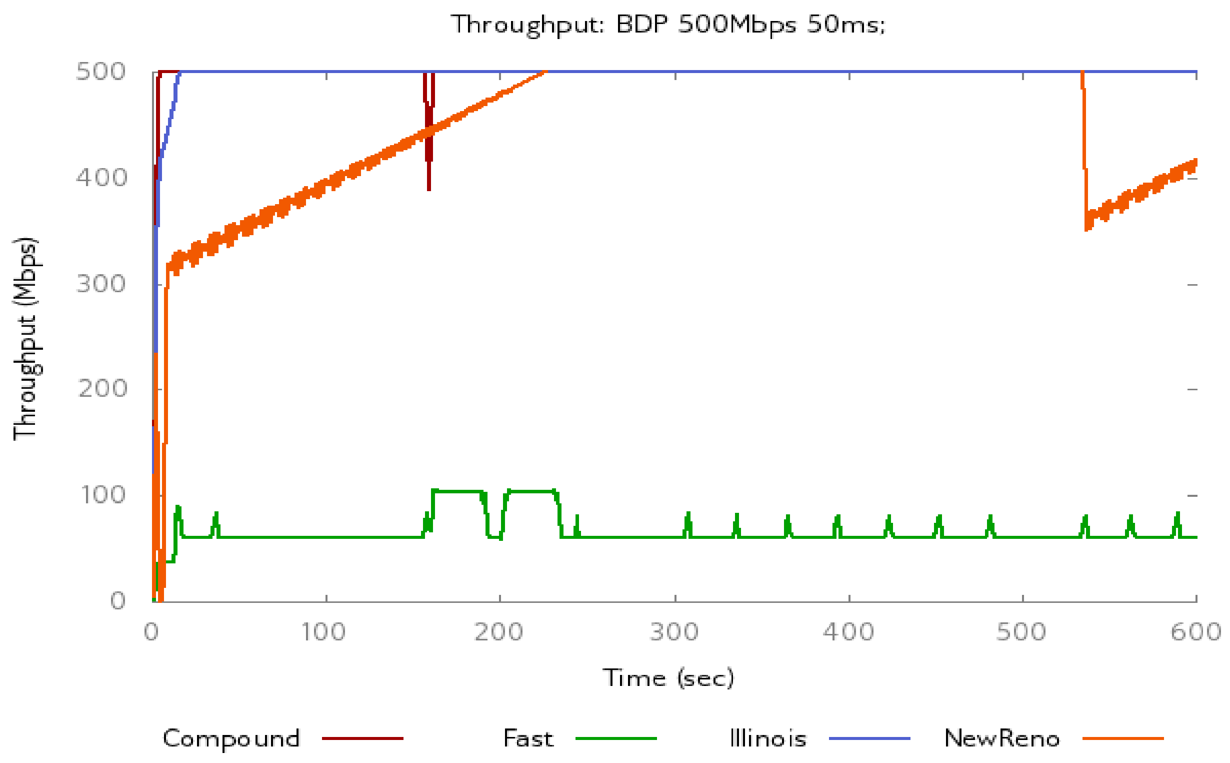


Figure 7: Throughputs: 500Mbps/50ms/8,300 packets

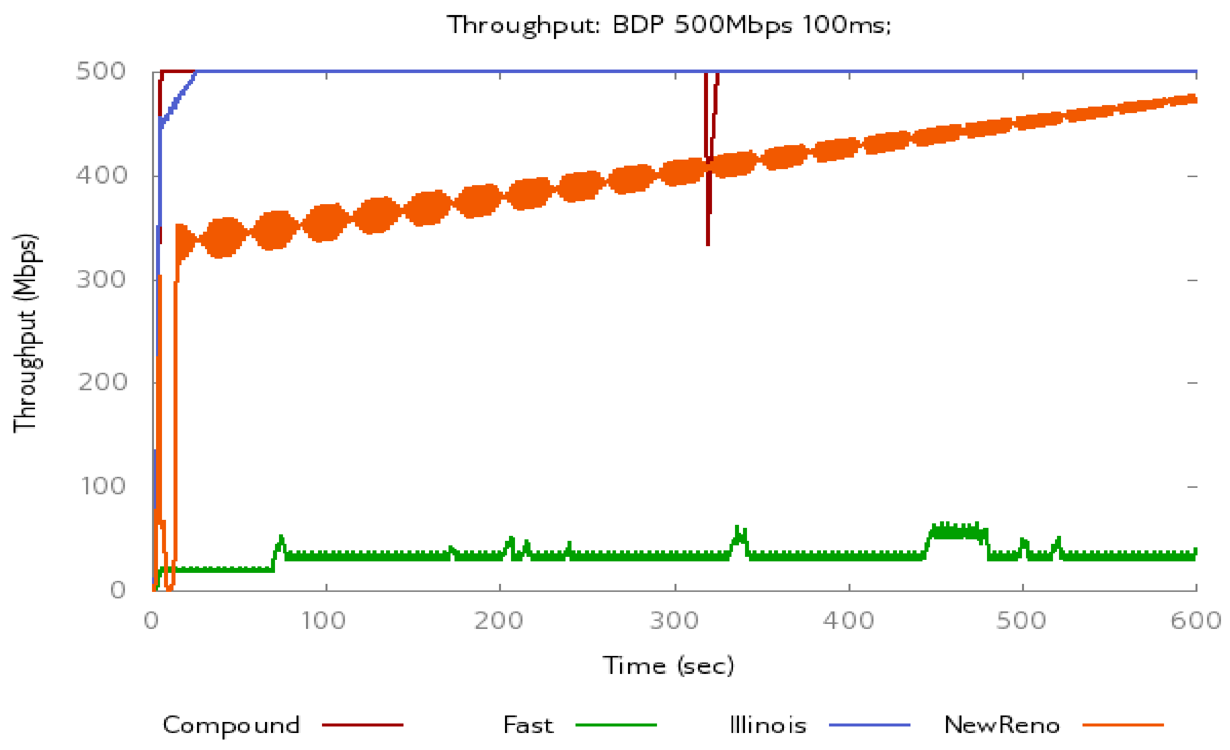


Figure 8: Throughputs: 500Mbps/100ms/8,300 packets

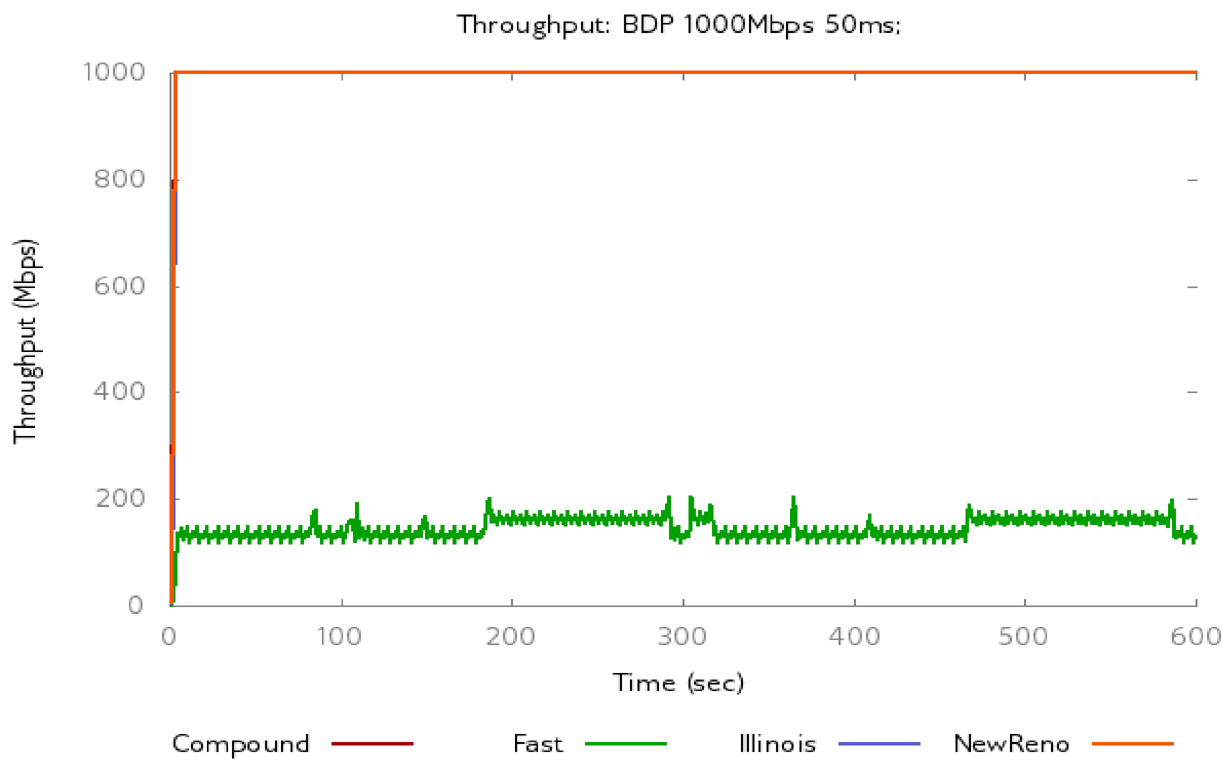


Figure 9: Throughputs: 1000Mbps/50ms/8,300 packets

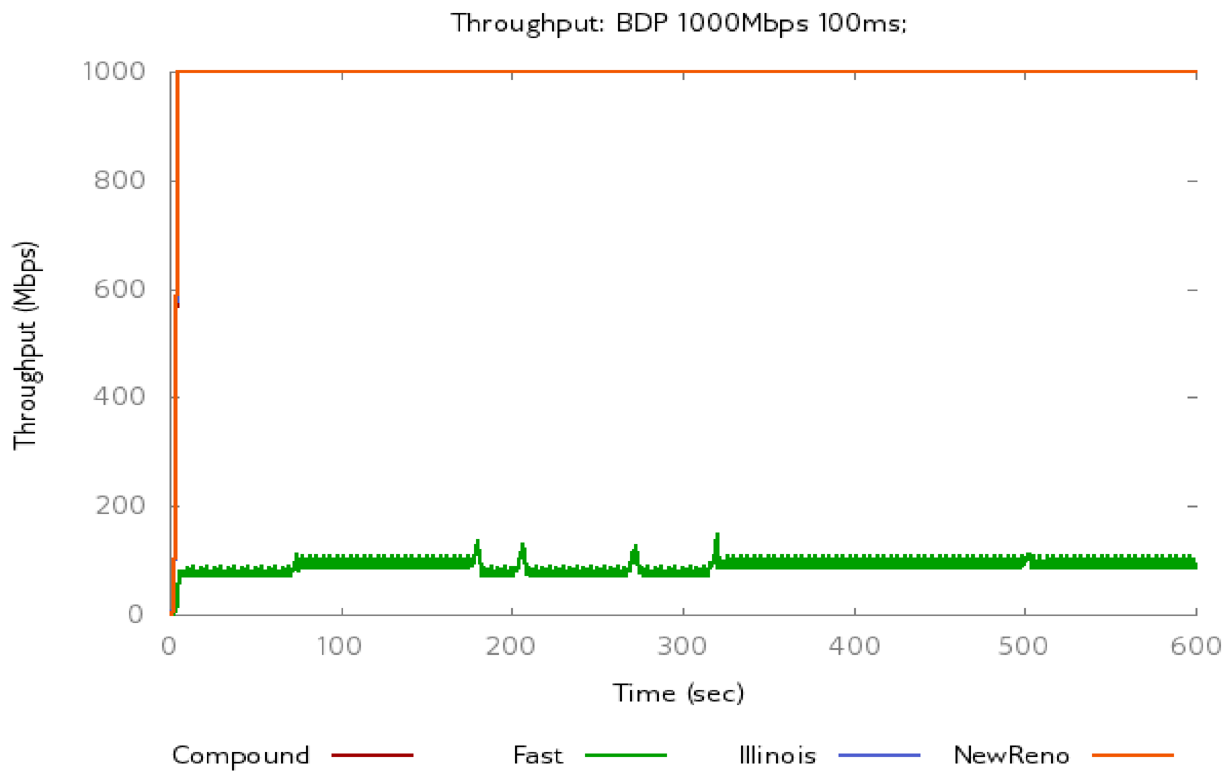


Figure 10: Throughputs: 1000Mbps/100ms/8,300 packets

7.1.2 Stanford Router Buffer Size

Under the 166-packet Stanford buffers, Compound displays a lot of burstiness in the lower bottleneck bandwidth environments. Illinois steadily increases its throughput with little variation. NewReno scales at a much slower pace, especially under longer RTTs, while Fast remains steady in its low throughput (figs 11 and 12)

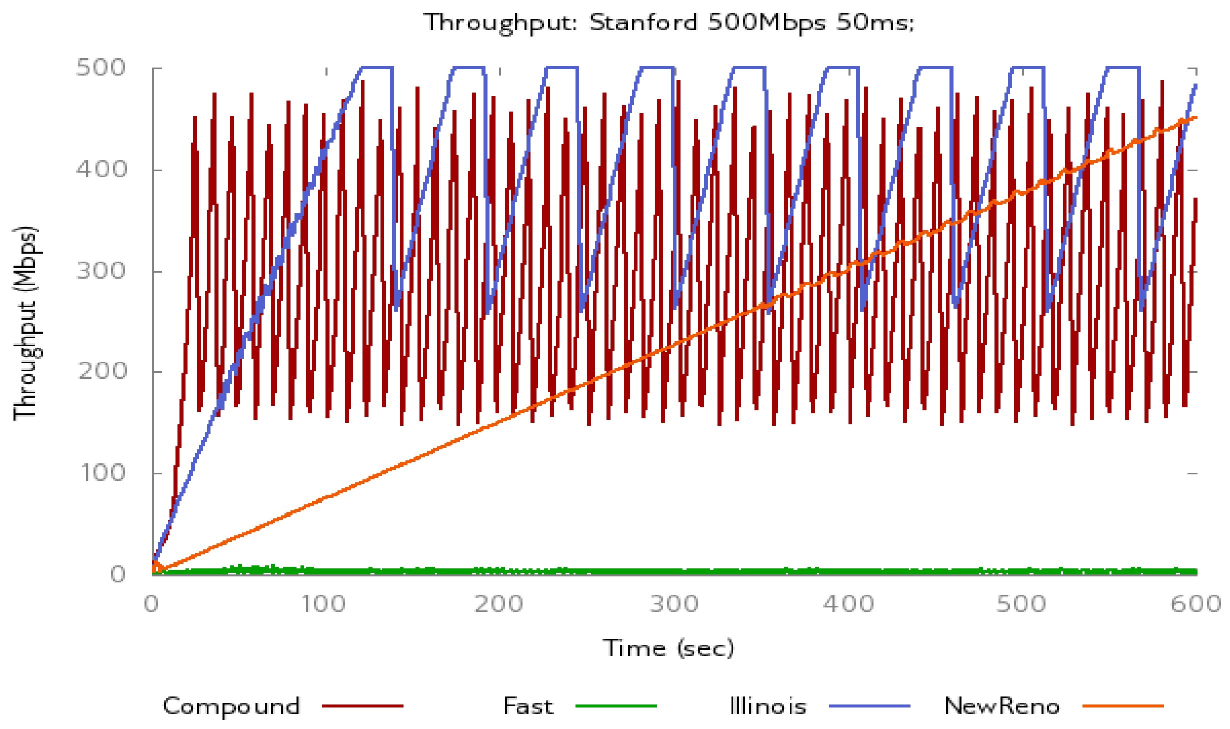


Figure 11: Throughputs: 500Mbps/50ms/166 packets

Under larger bottleneck bandwidth regimes, the protocols (sans Fast) are quickly able to scale and take up as much of the bandwidth as is available (figs 13 and 14), since little queuing is experienced.

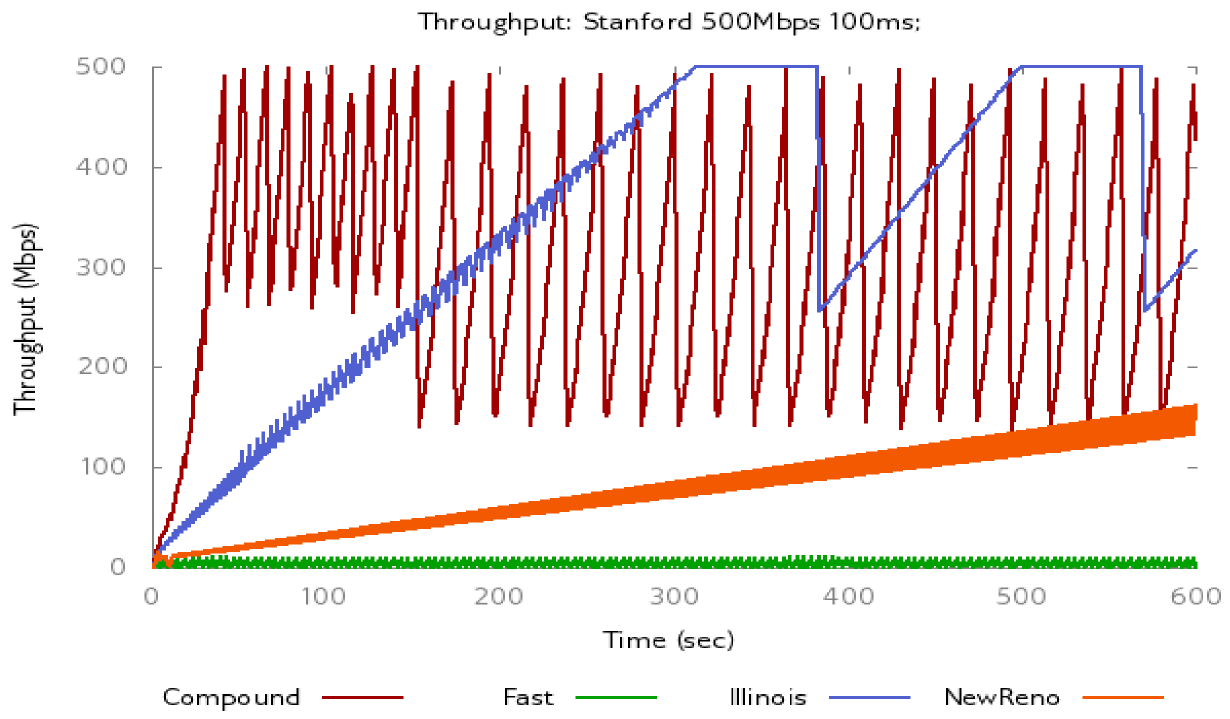


Figure 12: Throughputs: 500Mbps/100ms/166 packets

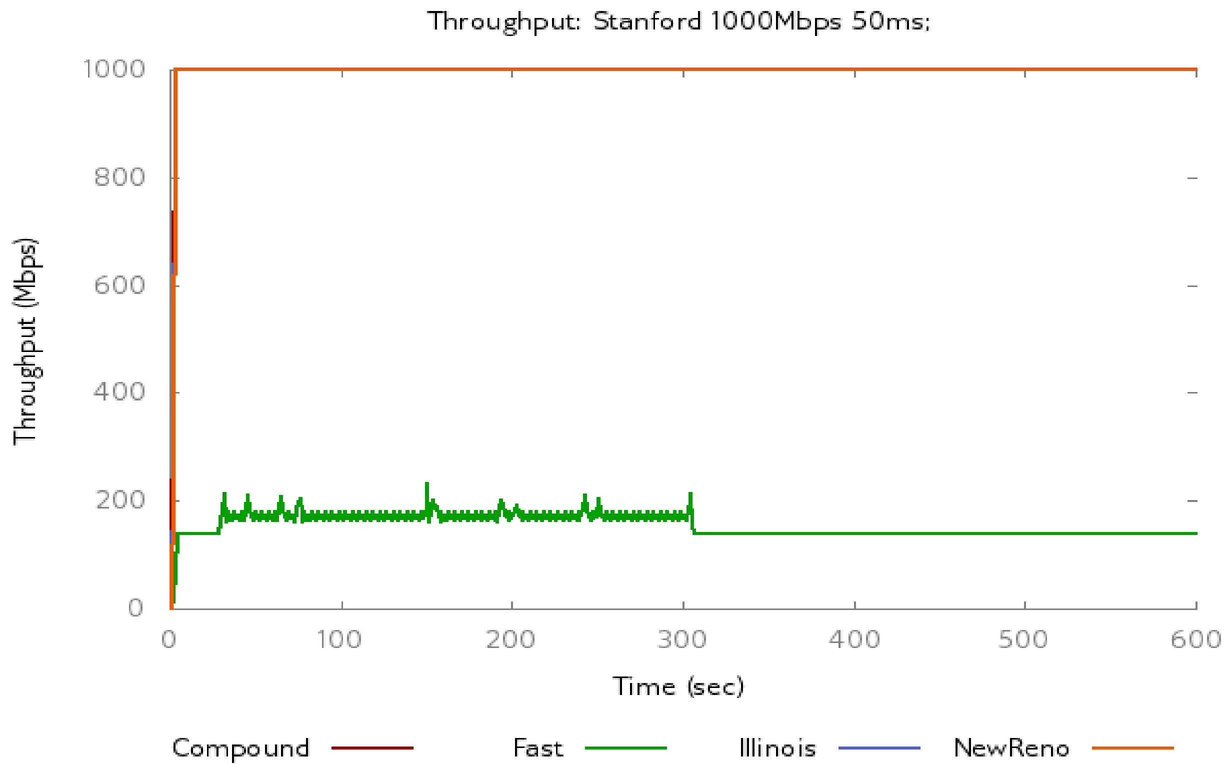


Figure 13: Throughputs: 1000Mbps/50ms/166 packets

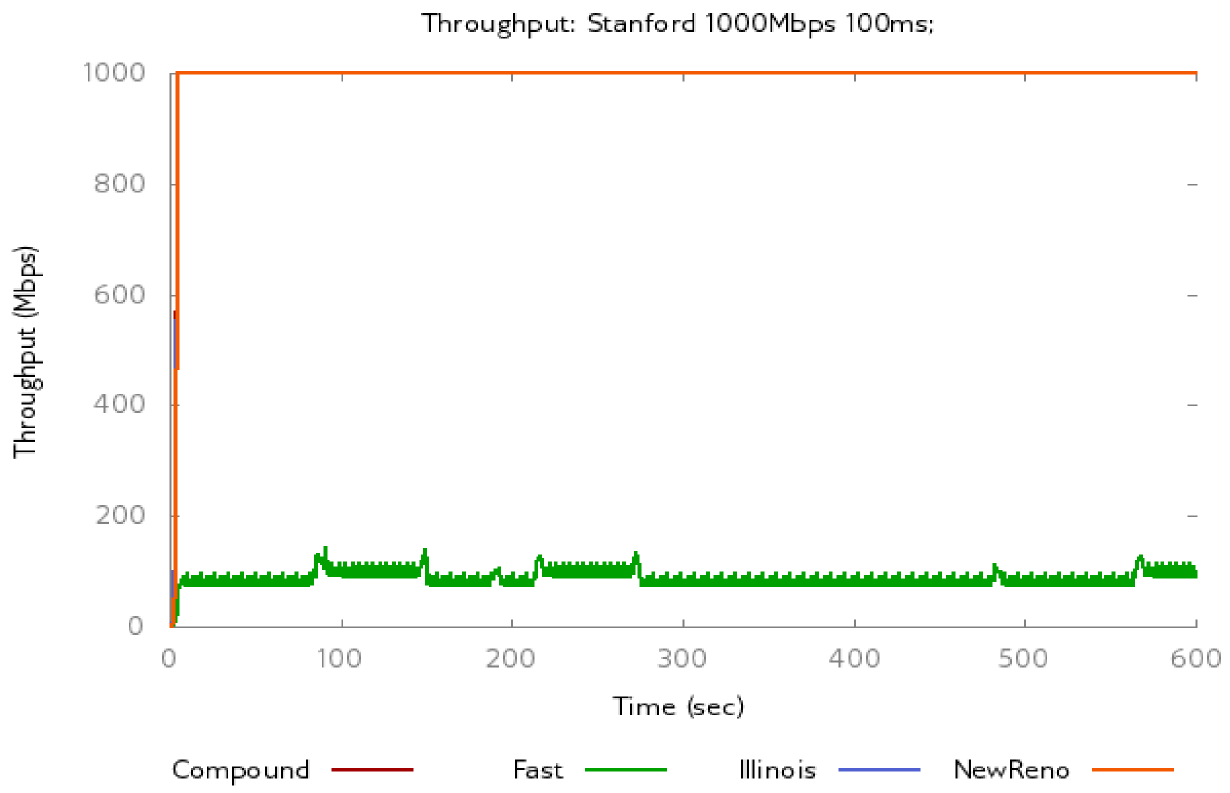


Figure 14: Throughputs: 1000Mbps/100ms/166 packets

7.1.3 UMass Buffer Size

Under the 20-packet UMass buffers, Compound's throughput appears to suffer severely under low bottleneck bandwidth environments. Illinois, on the other hand, steadily builds up its throughput. In comparison, NewReno also appears to scale, but at a much slower pace than Illinois (fig 15). When the rtt is high, the slower response time dulls the growth rates of Illinois and NewReno. Compound's throughput remains low as well, and the burstinesss displayed, while less than in Stanford, is still the greatest of the three high-speed protocols. Fast, held back by the router occupancy limit, is unable to achieve much throughput (fig 16).

When the bottleneck bandwidth is increased, Compound and Illinois are again able to quickly scale and utilize all of the link, despite the small router buffer size. Fast's throughput appears to be marginally higher, but it remains unable to scale effectively (fig 17 and 18).

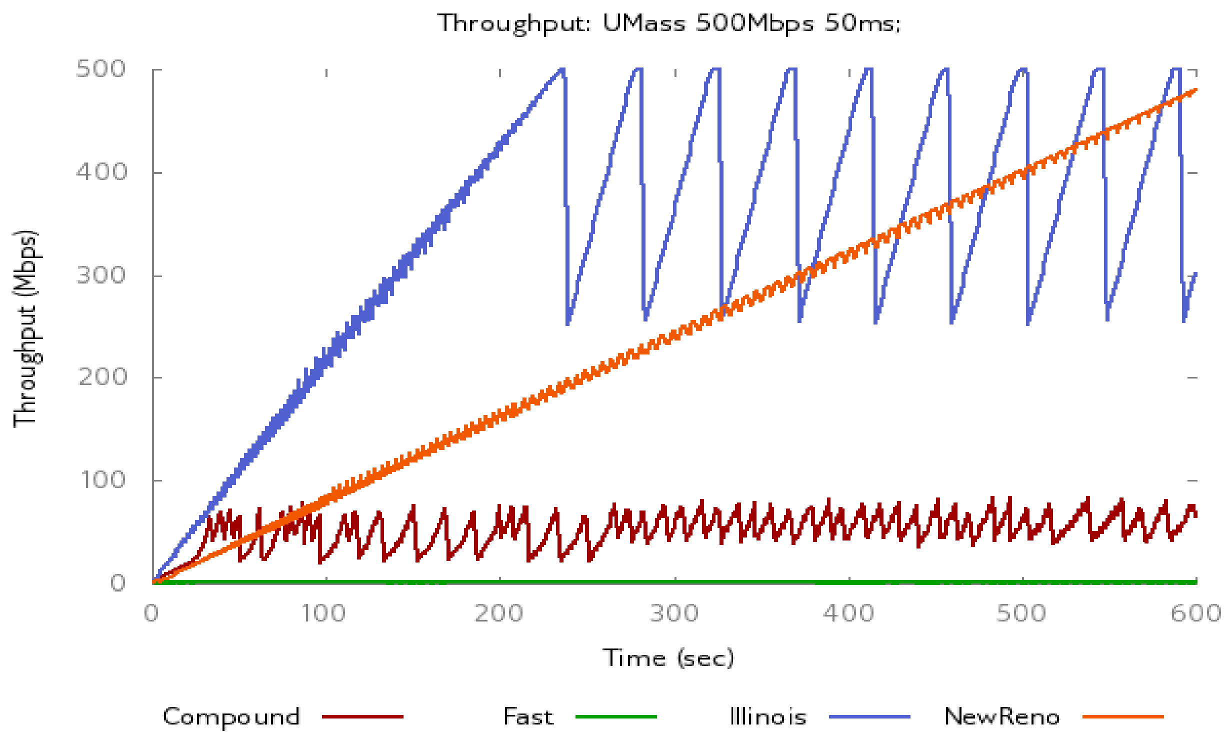


Figure 15: Throughputs: 500Mbps/50ms/20 packets

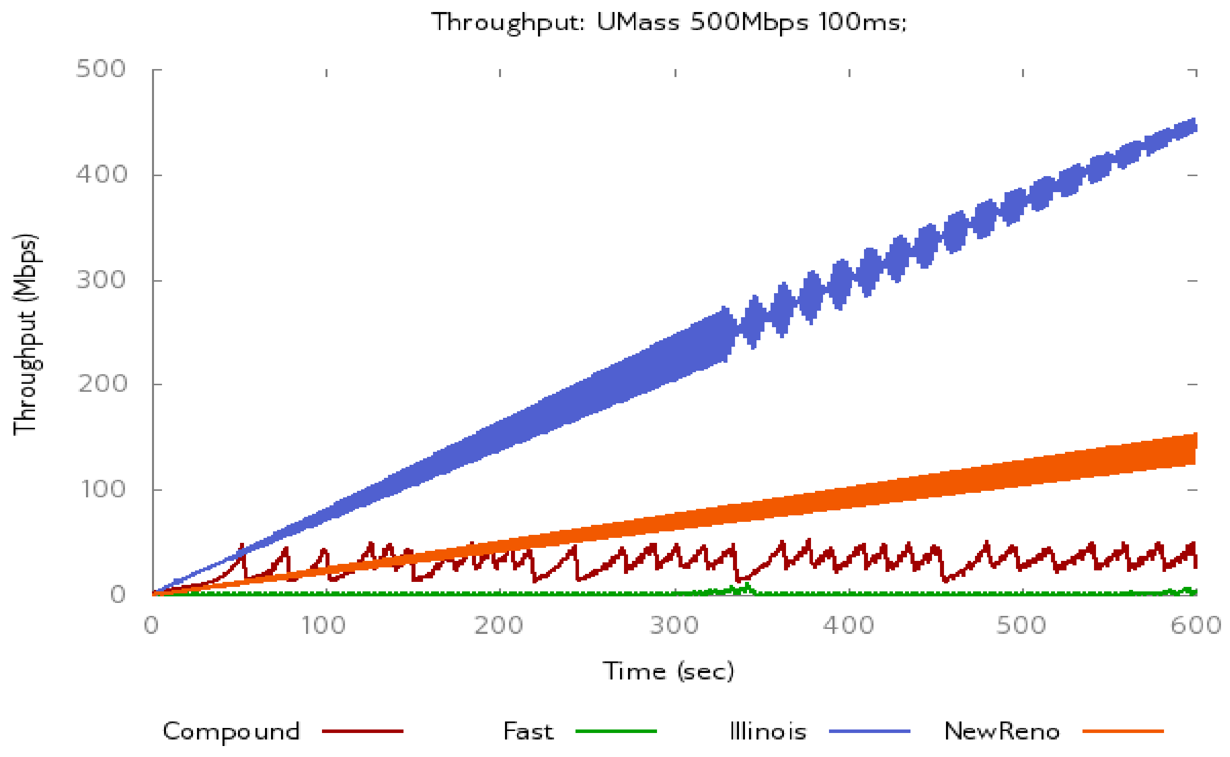


Figure 16: Throughputs: 500Mbps/100ms/20 packets

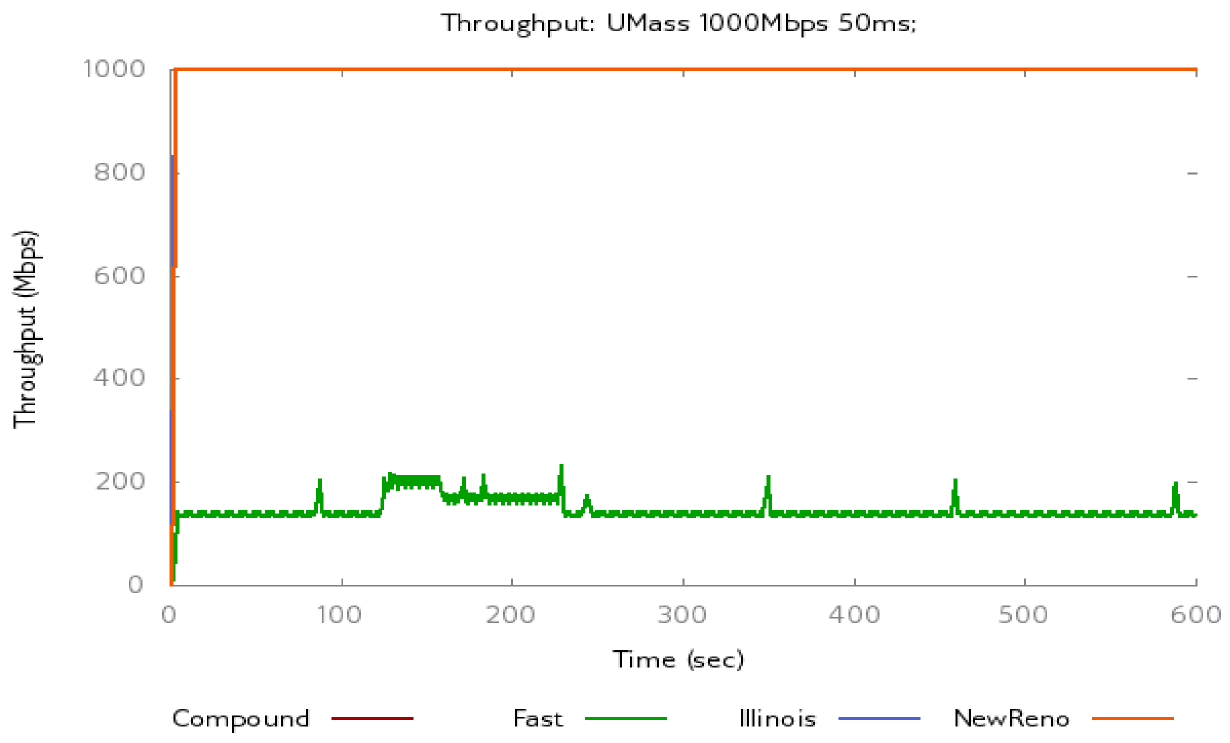


Figure 17: Throughputs: 1000Mbps/50ms/20 packets

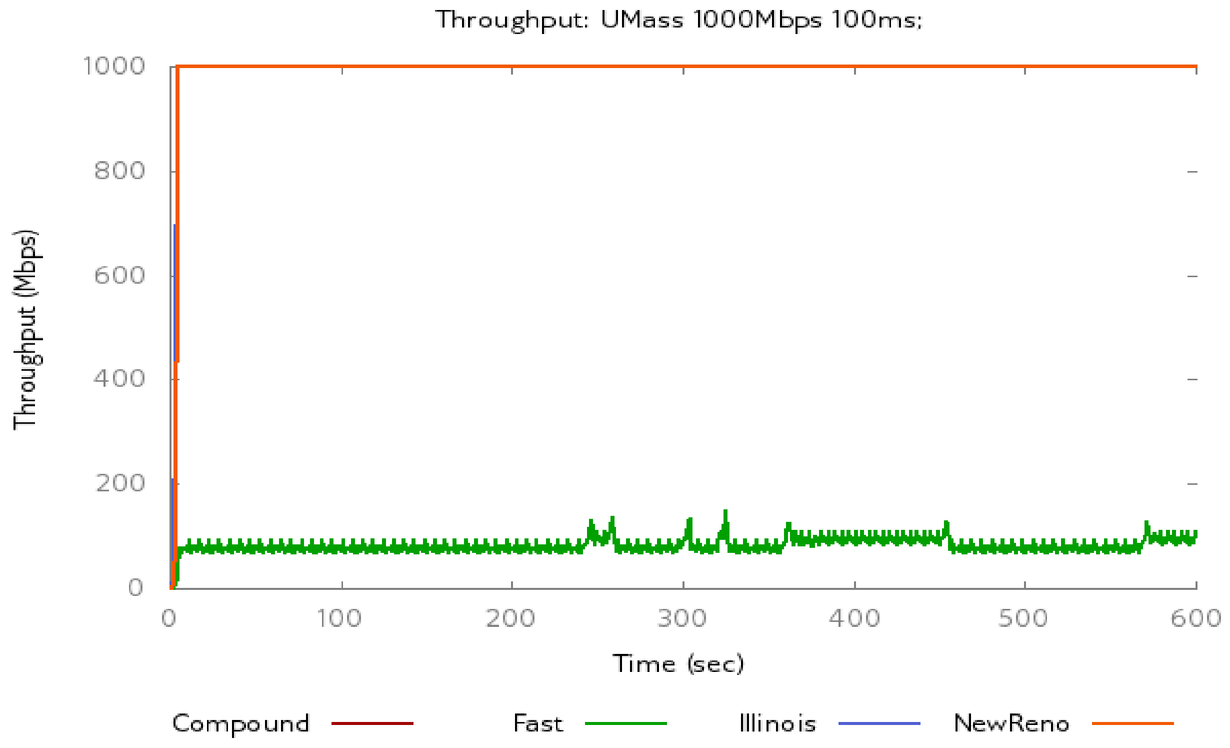


Figure 18: Throughputs: 1000Mbps/100ms/20 packets

7.2 Throughput

For each set of network conditions, the throughput was compared to that of a NewReno flow under the same conditions. For each { buffer, bandwidth, rtt } combination, the throughput of the test flow was extracted. These throughputs (one each - i.e. Compound, Fast, Illinois and TCP, for each combination) were plotted together for easier comparative analysis.

7.2.1 BDP Router Buffer Size

Under the BDP router buffer size of 8,300 packets, the high-speed protocols appear to lose ground as the NewReno flows increase the router buffer size. The graph below shows the throughputs of all four protocols in a 1000Mbps, 50ms rtt link.

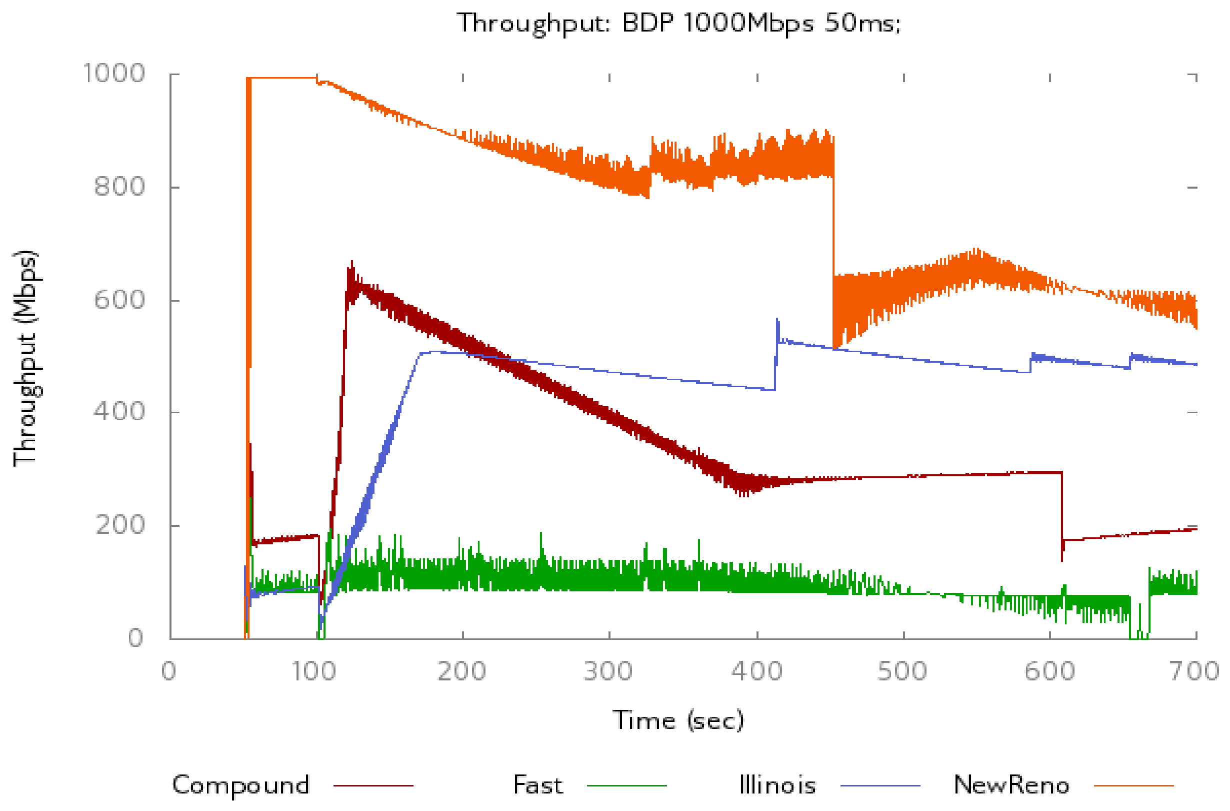


Figure 19: Comparative throughputs under 1000Mbps/50ms/8300 packets buffer size

After an initial quick burst, all the high-speed protocols greatly reduce their growth as TCP fills up the buffer. Fast, while largely stable, appears to continuously underperform; it does well to maintain the set number of packets as determined by its α and β parameters[11] [tan 07], but fails to take advantage of any extra router buffer size that there may be. TCP Illinois, by using the

[11] Parameters for FAST were derived from [tan 07] and from the creators of Fast's ns-2 module, the CUBIN Lab at the University of Melbourne [http://www.cubinlab.ee.unimelb.edu.au/ns2fasttcp/]

delay only as a secondary signal, is able to achieve much higher throughput than Compound, which, by using delay as a primary signal, reduces its window size when higher delay is encountered.

This behavior appears to hold under smaller links as well, as shown in figure 8 below:

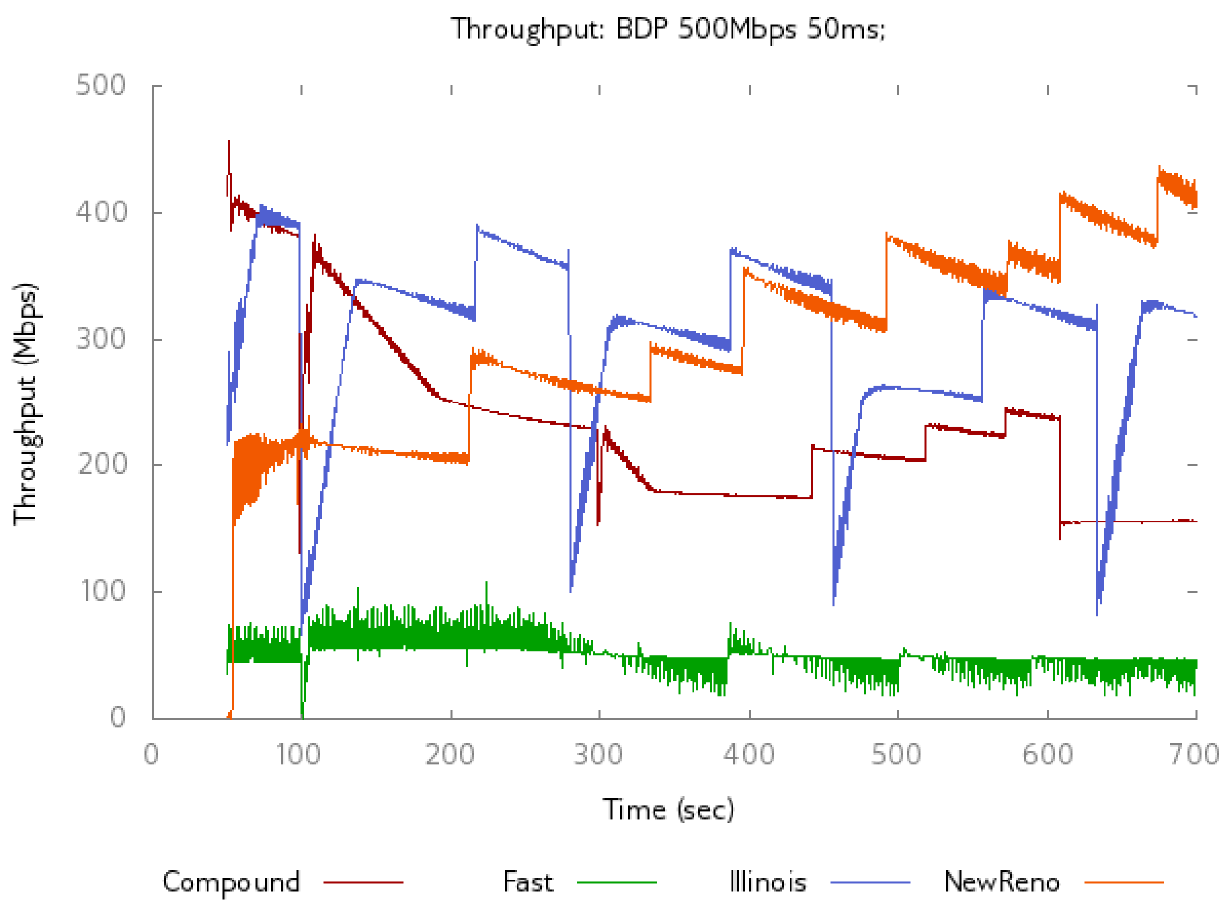


Figure 20: Comparative throughputs under 500Mbps/50ms/8300 packets buffer size

Once again, Fast remains stable, but underperforms. Compound, after an initial burst, reduces its window size when delay is encountered. Illinois attempts to maintain the concave window curve as long as possible, displaying behavior reminiscent of CUBIC.

Under larger RTT environments, Illinois again appears to outperform Compound and Fast, as shown in both figures 9 and 10. In figure 9, NewReno's inability to quickly scale under larger BDP networks is apparent; Illinois, however, quickly takes up more of the network bandwidth.

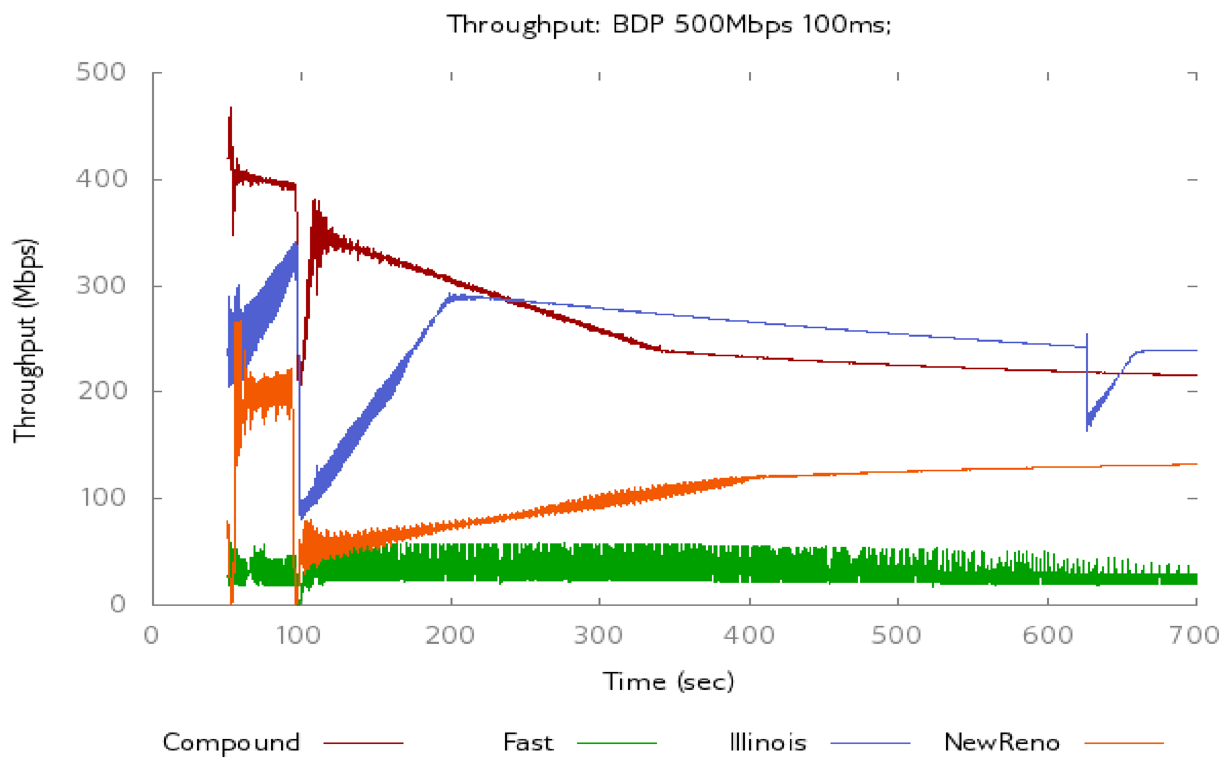


Figure 21: Comparative throughputs under 500Mbps/100ms/8300 packets buffer size

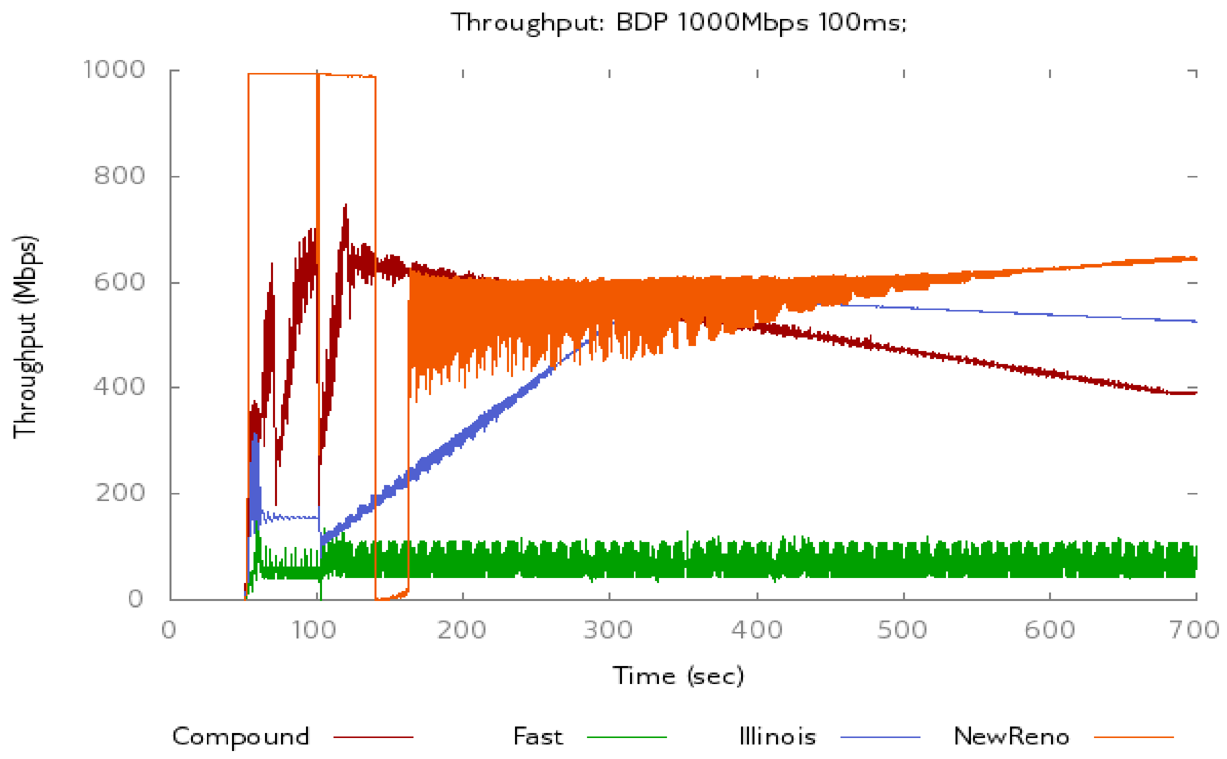


Figure 22: Comparative throughputs under 1000Mbps/100ms/8300 packets buffer size

7.2.2. Stanford Router Buffer Size

The Stanford router buffer size used in the simulation was significantly smaller { $\left({}^{1}/_{25}\right)th$ } than the BDP router buffer. Under smaller router buffer sizes, the high-speed protocols appear to display much more burstiness. This might be due to the fact that the TCP flows may experience more loss-and-recovery cycles under smaller router buffers, thus drastically changing the proportion of the router available at much shorter time periods than under BDP. This would lead to the delays experienced by the delay-based protocols to alter rapidly within a short time period.

As figure 23 below shows, in a small bandwidth-delay environment, Compound is quickly able to scale and take up as much bandwidth as possible. However, it fails to maintain any steady throughput. Fast fails to adapt to any increase in bandwidth and remains stable, yet underperforming, while Illinois behaves in much the same way as TCP would.

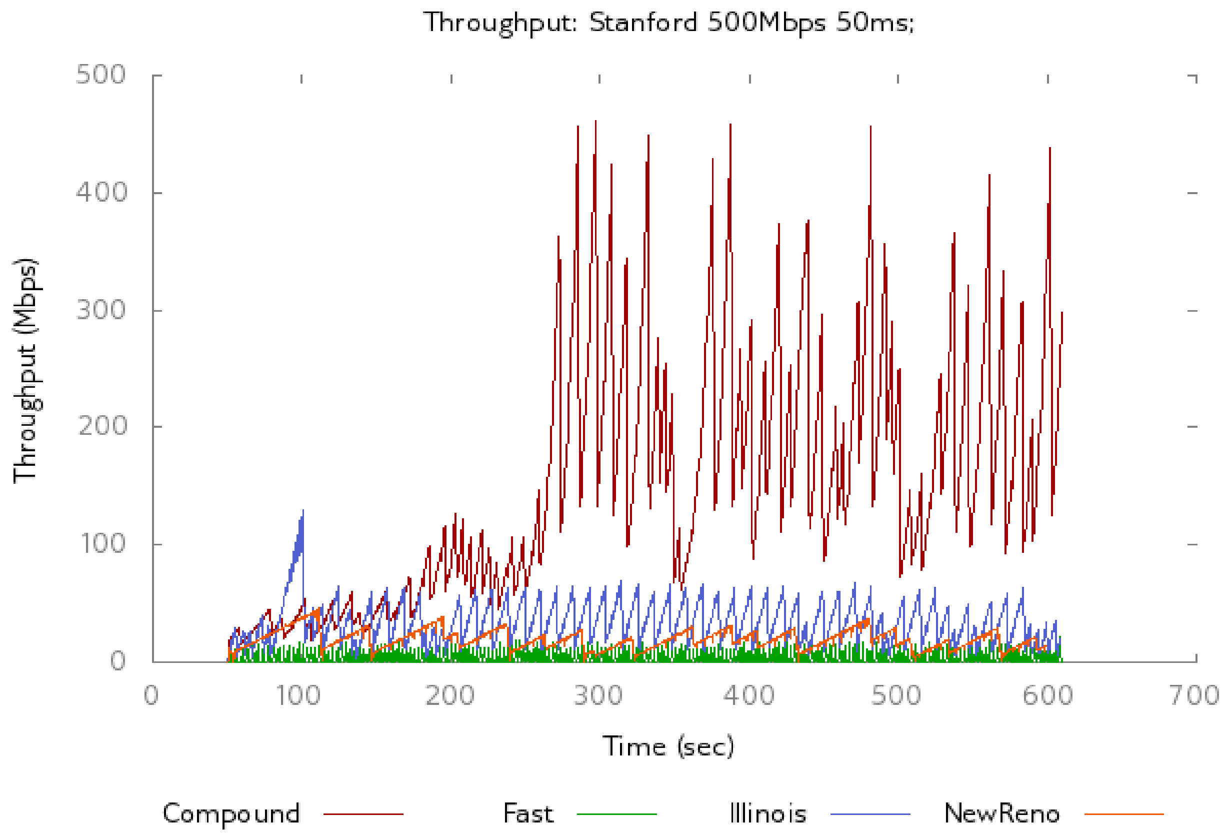


Figure 23: Comparative throughputs under 500Mbps/50ms/166 packets buffer size

In larger bandwidth-delay environments, Compound continues to display significant burstiness. Illinois, while still bursty, appears geared towards attaining a concave throughput function, as shown in figures 24, 25 and 26 below. In figure 24, Fast still maintains its low throughput and is clearly outdone by the rest.

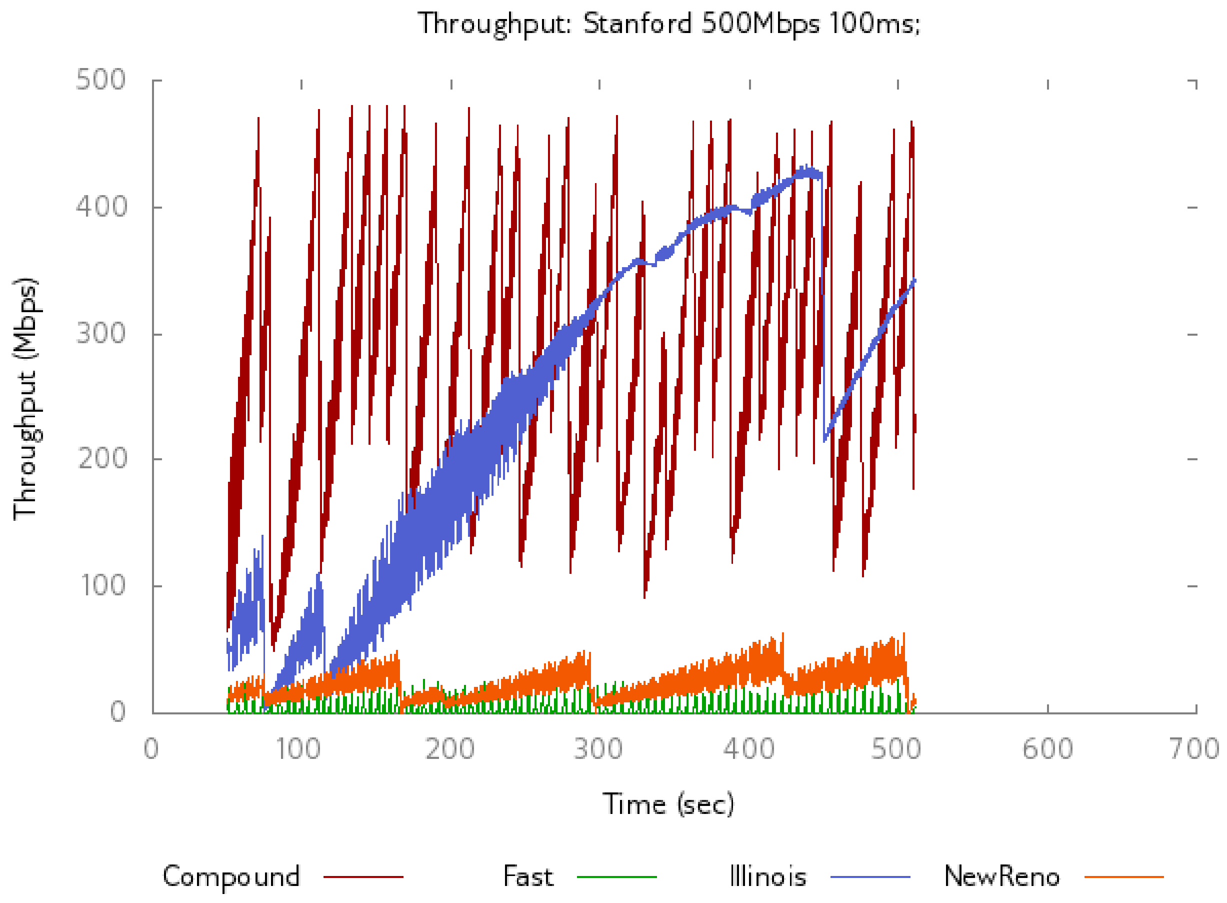


Figure 24: Comparative throughputs under 500Mbps/100ms/166 packets buffer size

In figures 24 and 25, Fast's higher α value (which is dependent on the bandwidth) helps it achieve slightly higher throughput, and the protocol displays more discernible burstiness as it responds to changing delays. It is, however, still woefully outdone.

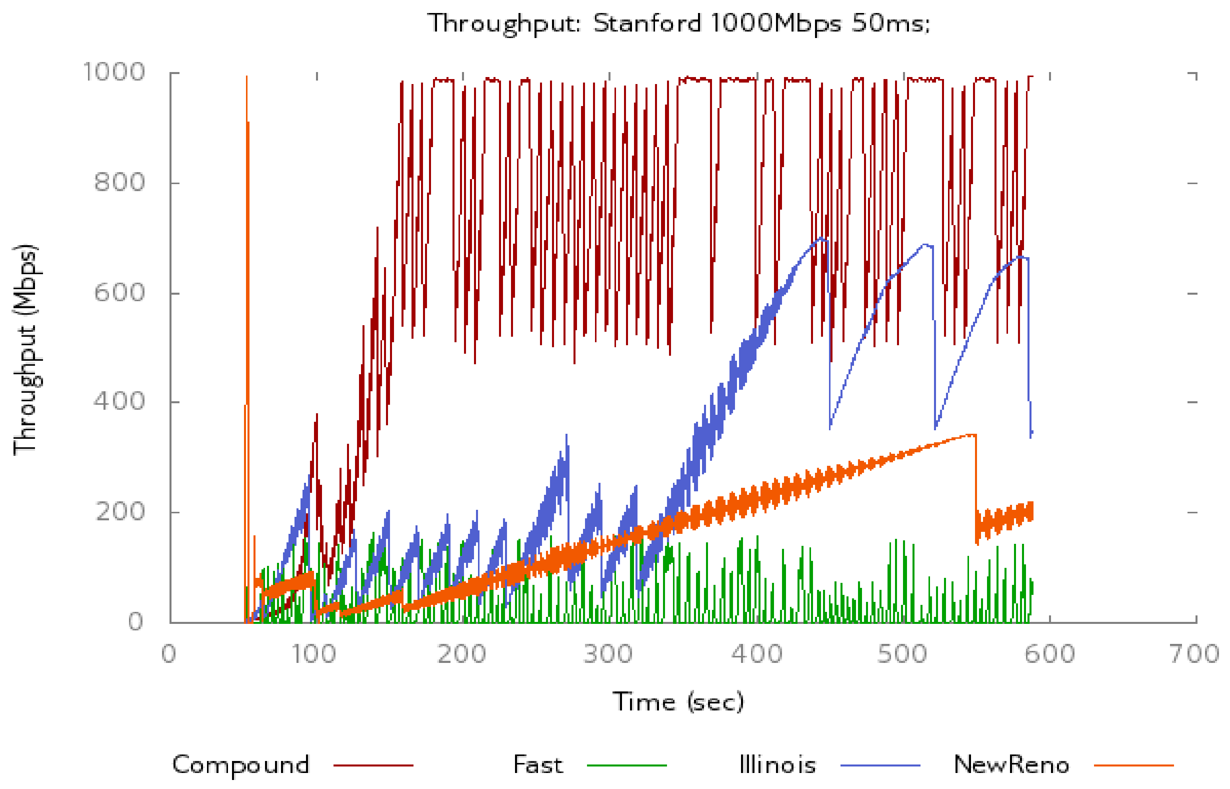


Figure 25: Comparative throughputs under 1000Mbps/50ms/166 packets buffer size

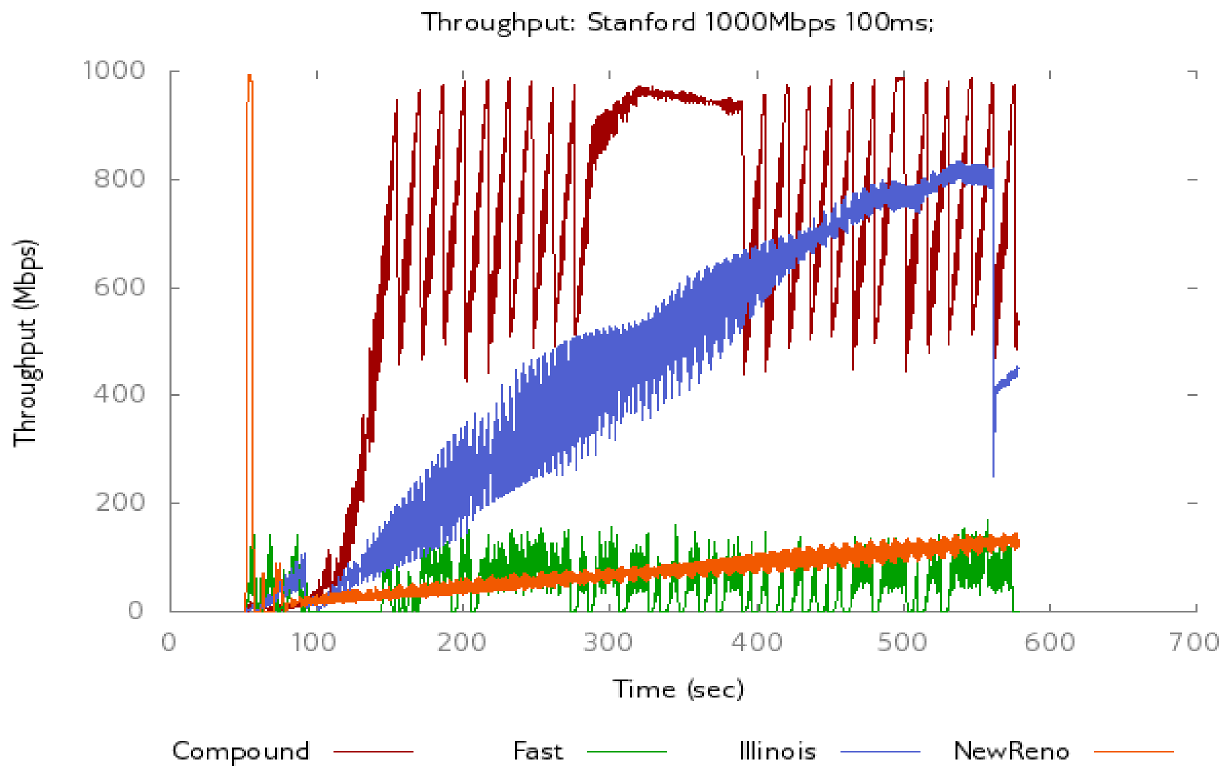


Figure 26: Comparative throughputs under 1000Mbps/100ms/166 packets buffer size

7.2.3. UMass Router Buffer Size

Under the UMass router buffer size specifications, a router buffer size of 20 packets was used. As earlier mentioned, the recent shift towards smaller router buffer sizes would adversely affect the efficacy of delay-based protocols, as both router buffer occupancy and delay experienced are deciding factors on how the protocols behave, and are affected by the size of the buffer.

With this in mind, the protocols were observed to greatly underperform at all environments using the UMass router buffer size. As most high-speed protocols are meant to behave like TCP under heavy loss, the protocols closely mirror TCP behavior and offer no added advantage over TCP in small router buffer environments. Under the 500Mbps environments, neither protocol shows any discernible difference, as figures 27 and 28 below show:

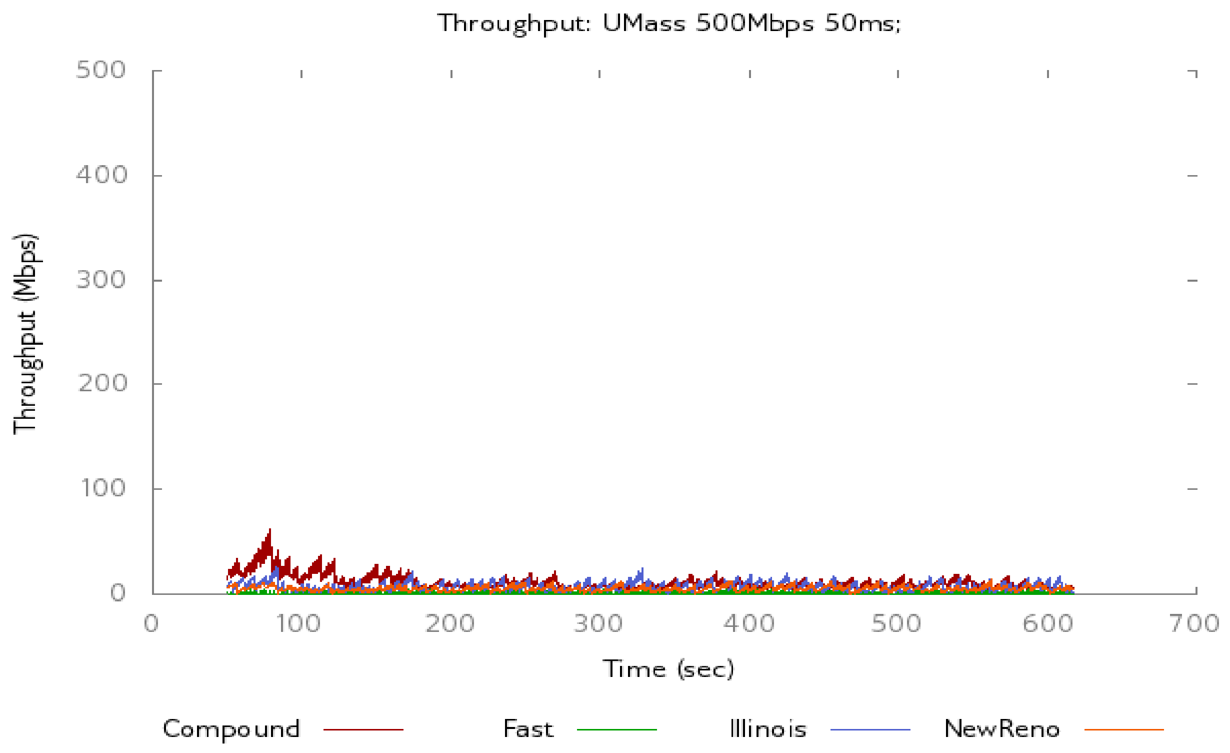


Figure 27: Comparative throughputs under 500Mbps/50ms/20 packets buffer size

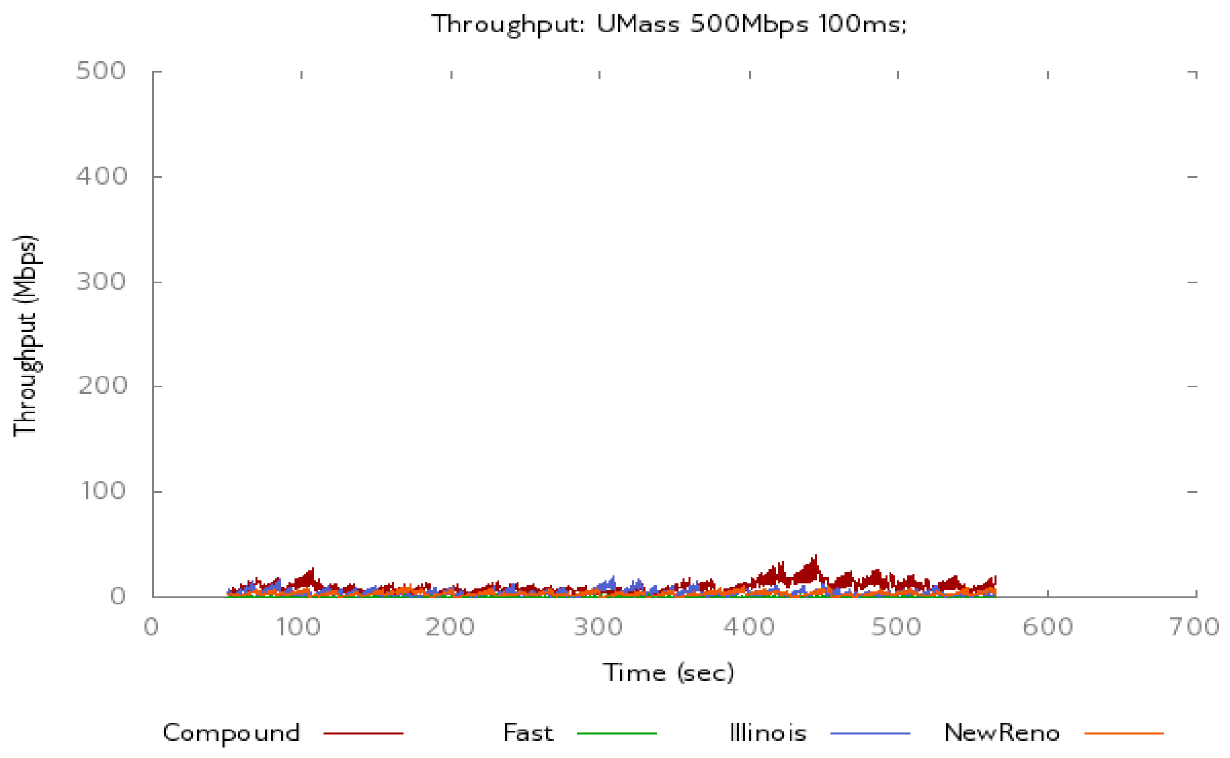


Figure 28: Comparative throughputs under 500Mbps/100ms/20 packets buffer size

Under larger bandwidth delay environments, Fast appears to show a surprising surge in throughput and burstiness, while the rest remain largely unable to handle the small router buffer sizes. While Compound is tightly coupled to router dynamics, Illinois isn't, yet it too is unable to gain much traction in trying to achieve greater throughput.

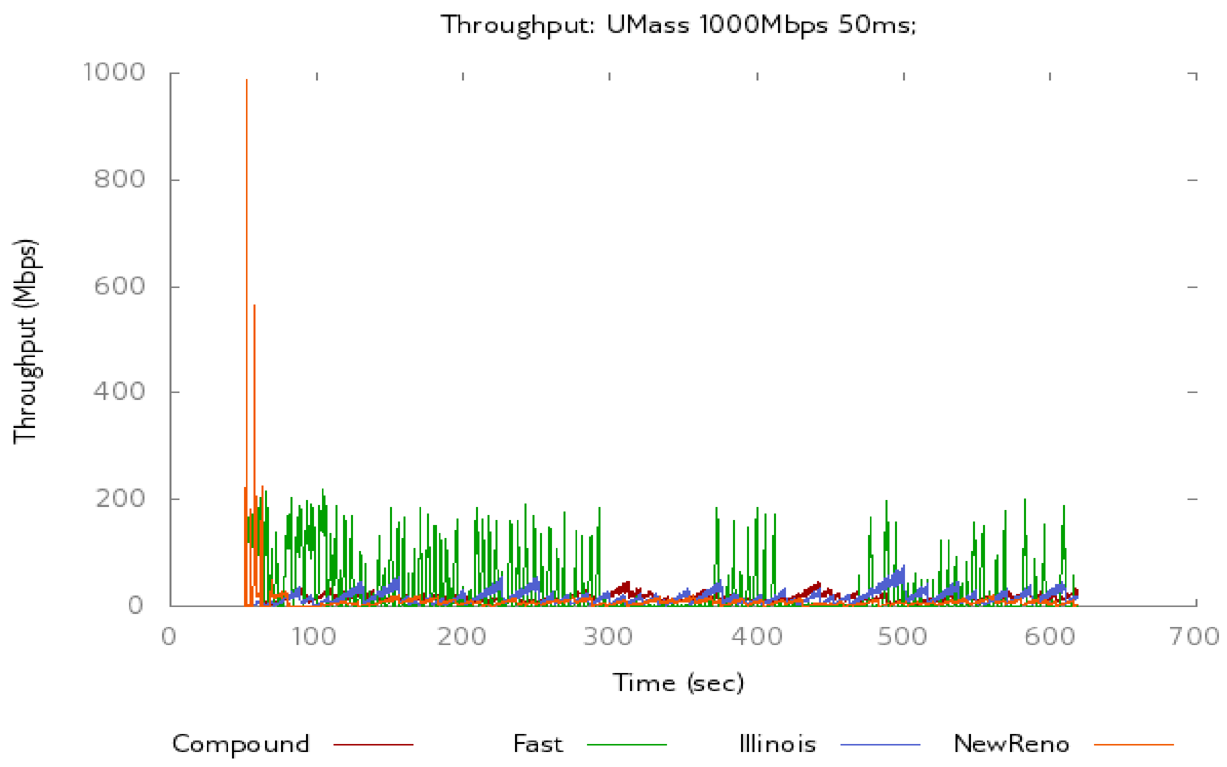


Figure 29: Comparative throughputs under 1000Mbps/50ms/20 packets buffer size

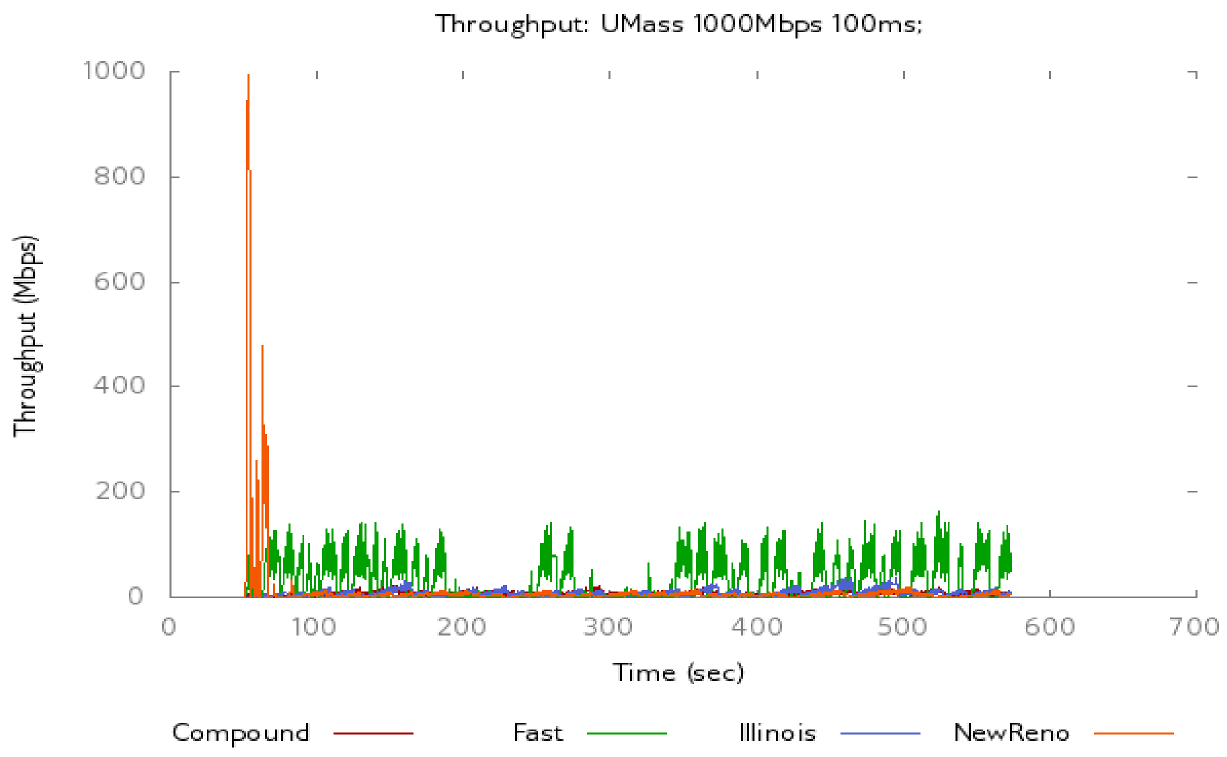


Figure 30: Comparative throughputs under 1000Mbps/100ms/20 packets buffer size

7.3 Burstiness & Stability

TCP and its various variants periodically send packets in bursts, especially in environments with large buffers or large available bandwidths [shak 05], where it has been shown to be a significant problem for non-high-speed TCP. While TCP might fail to react to bursts that do not lead to losses, delay-based protocols are bound to react to the resultant burstiness-induced delays. This may then affect other non-delay-based flows, leading to reduced throughput, unfair sharing and synchronized flows, compounding the problem even further.

To measure the burstiness of the protocols tested, the coefficient of variation (CoV) of the throughput was measured as modeled in [ha 06] and [goh 07]. An arithmetic mean and standard deviation of the throughput of each flow are taken at 5-second intervals, and used to generate the ratio of the standard deviation to the mean; this gives a normalized value of the distribution/spread of the values from the mean. The CoV has a value in the bounded range [0, 1], with magnitude correlating to the flow's burstiness.

7.3.1. BDP Router Buffer Size

Under the BDP router buffer size (8,300 packets), the amount of burstiness in the Compound and Illinois flows is fairly low, largely the same as NewReno. Fast appears to display slightly larger amounts of burstiness, which would be detrimental to an almost-full router buffer.

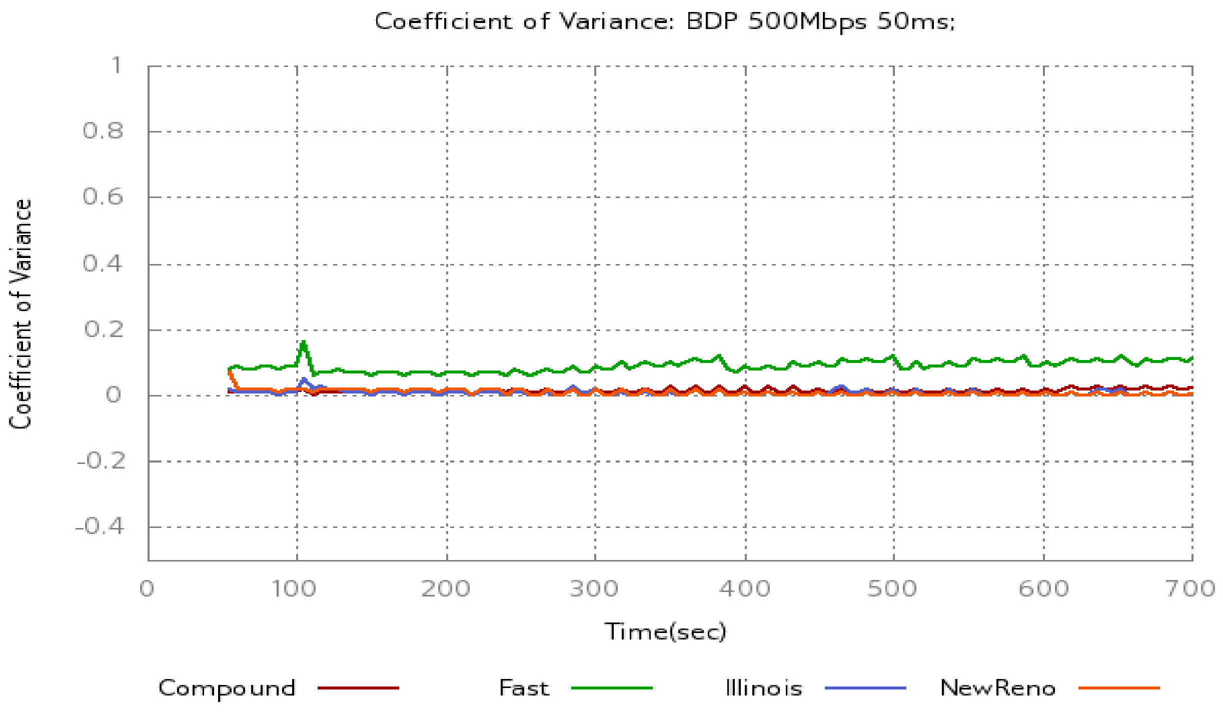


Figure 31: CoV at 500Mbps/50ms/8300 packets

While the trend appears to hold at higher bandwidth delay environments, increased RTT, rather than increased throughput, appear to cause greater burstiness in Fast and NewReno. Figures 32 and 33 display this behavior.

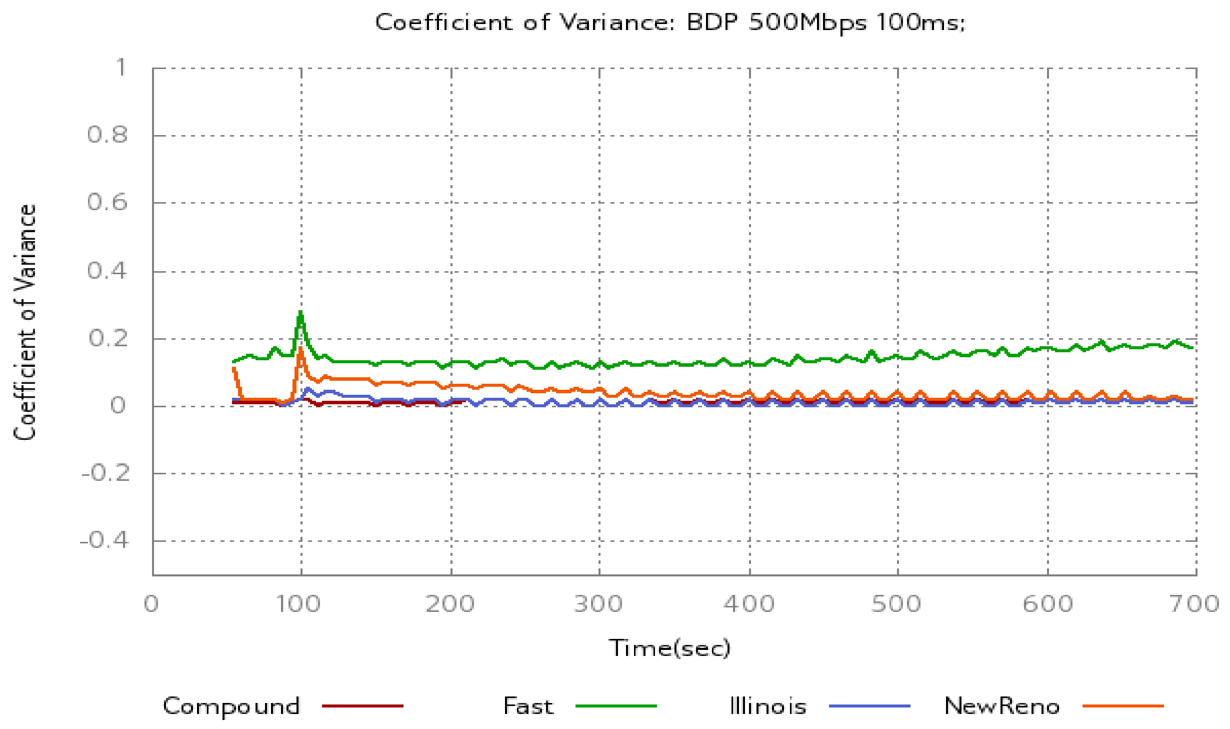


Figure 32: CoV at 500Mbps/100ms

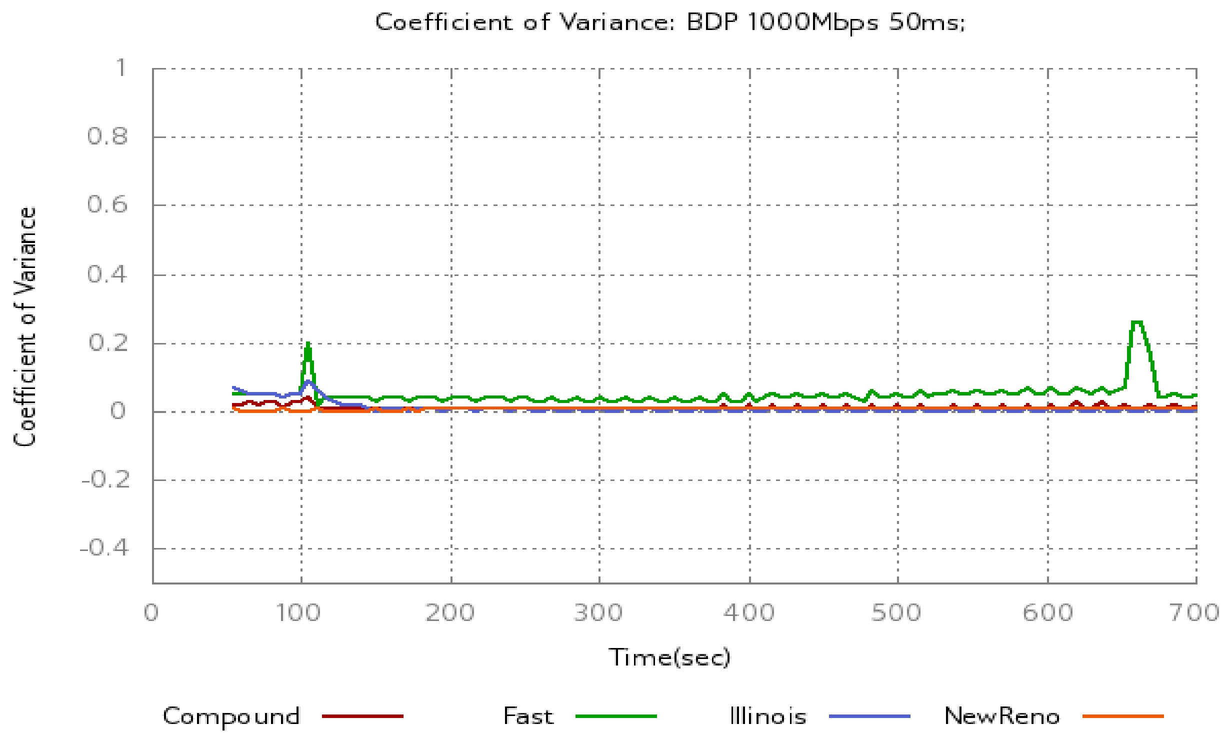


Figure 33: CoV at 1000Mbps/50ms

This behavior holds for all three high-speed protocols at 100Mbps/100ms as well.

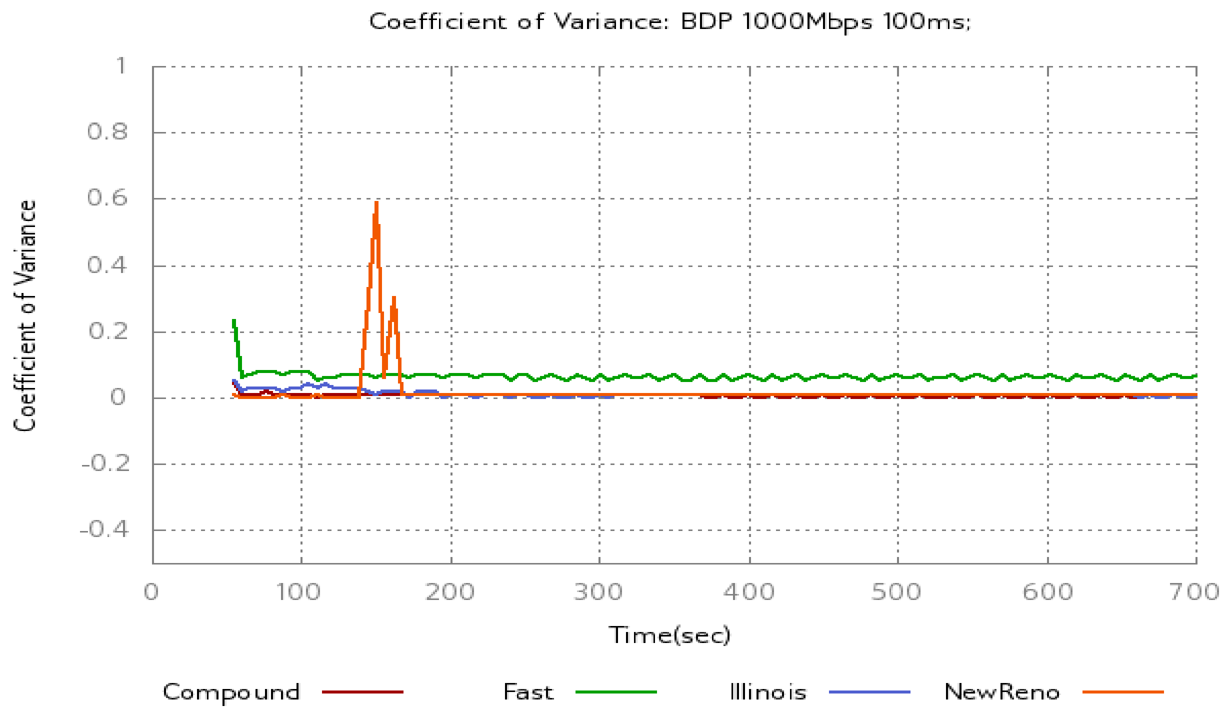


Figure 34: CoV at 1000Mbps/100ms

While Fast displays the highest amount of burstiness, the periodicity (i.e. the magnitude) of its bursts remains largely the same. Thus, while the throughput graphs in 7.2 above show the throughputs changing over time, the CoV shows that the change happens slowly and over a long duration, keeping the protocols fairly stable in high bandwidth delay environments. This stability is one of the main selling points of delay-based protocols; by adapting to the delays within the network, they can easily avoid the sudden and catastrophic throughput variations caused by loss-induced congestion window reductions.

7.3.2. Stanford Router Buffer Size

Under the Stanford router buffer size (166 packets), the stability of the protocols begins to suffer (figs. 35 & 36).

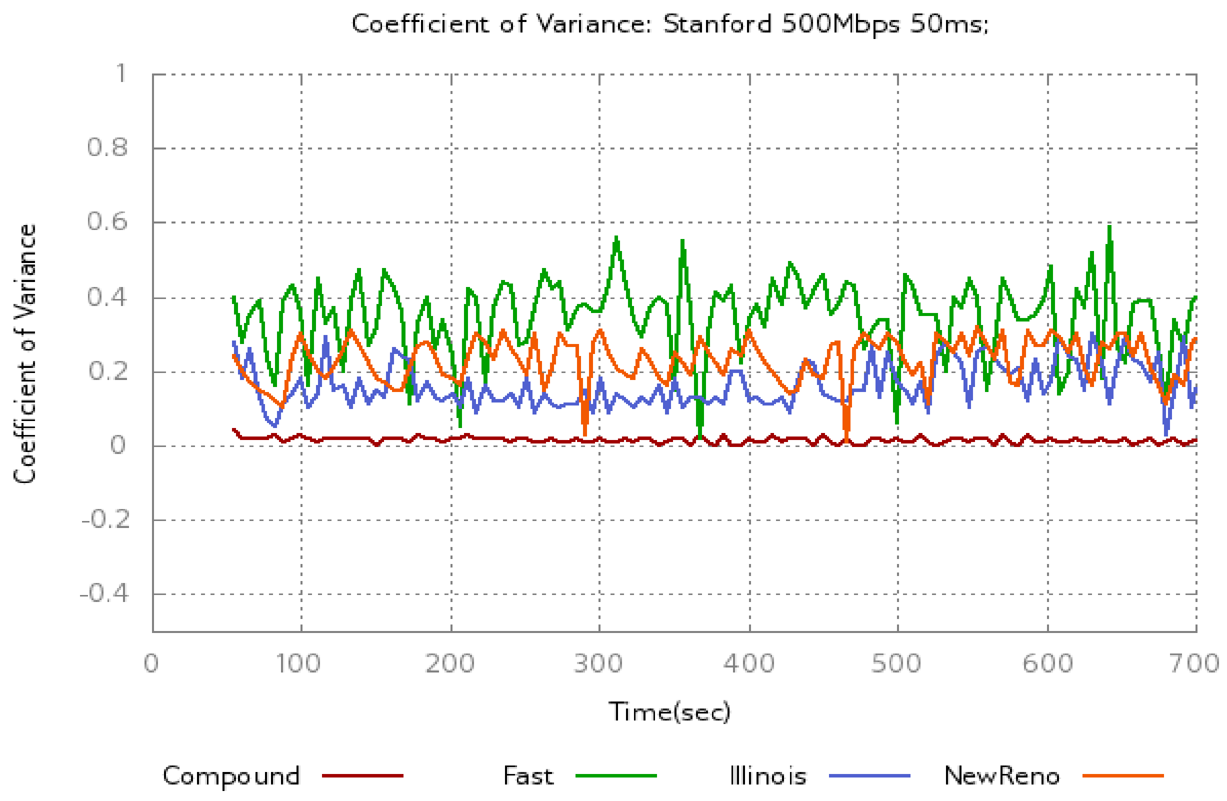


Figure 35: CoV at 500Mbps/50ms

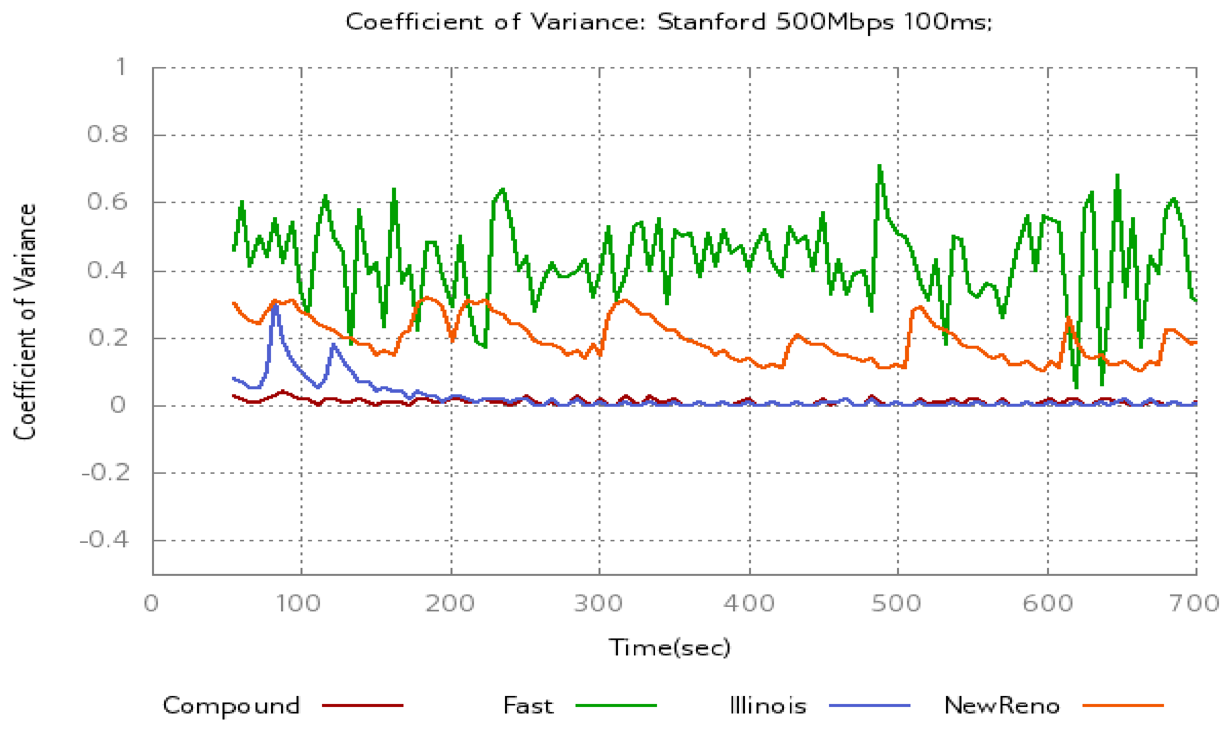


Figure 36: CoV at 500Mbps/100ms

The CoV of Illinois drops considerably more than that of Compound when the RTT is relatively short. This may be due to Illinois' propensity to use delay only as a secondary congestion control factor, i.e. using it only to determine how much to reduce the window by, rather than whether or not to reduce the window, like Compound does. By not reducing the window size when delay increases, it is likely to have to reduce the window by a much greater proportion when a loss occurs. Thus, Compound's growth function more aggressively adapts to the changing network environments than Illinois does. Fast, on the other hand, displays far worse burstiness than the other high-speed protocols, and loses the periodicity displayed under larger buffer environments. In longer RTT environments, Compound and Illinois appear to stabilize, as shown in figure 37.

Under larger bandwidth environments, Fast continues to display stubbornly bursty behavior. Compound and Illinois appear better suited to the larger bandwidths and perform better than Fast, once the flows settle.

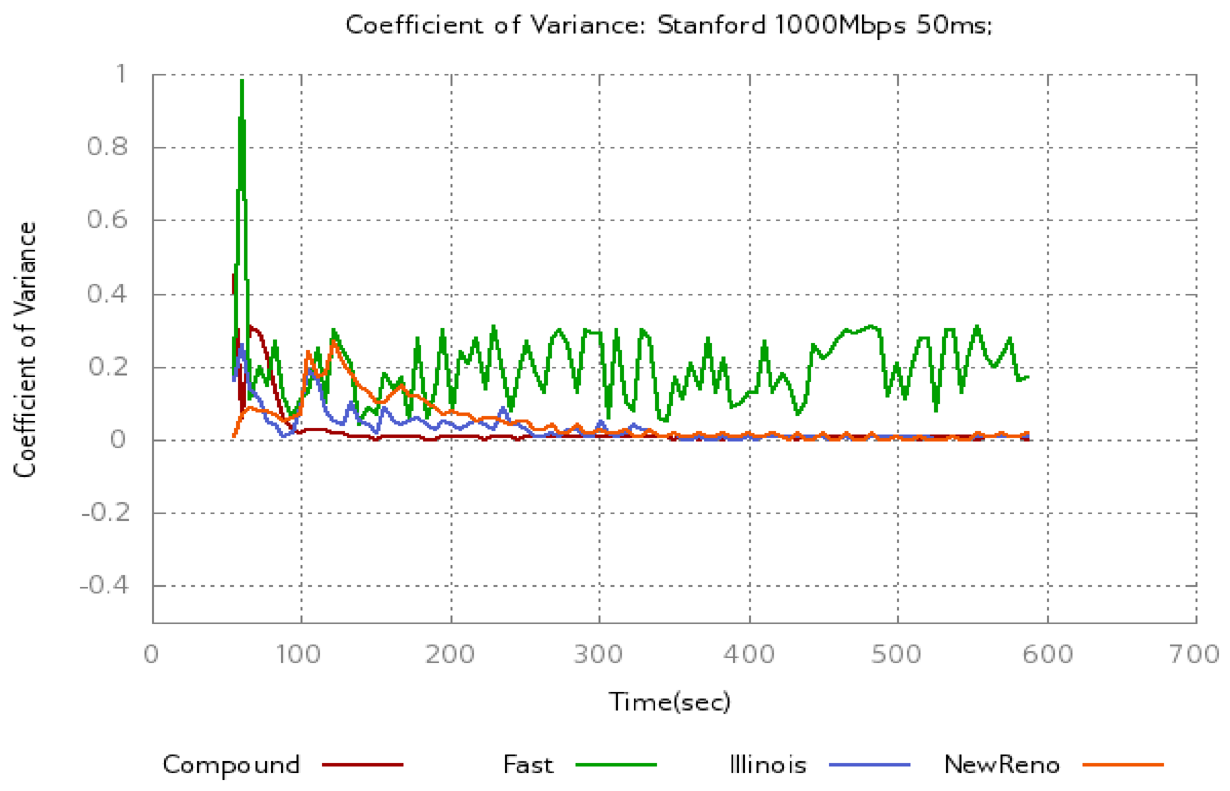


Figure 37: CoV at 1000Mbps/50ms

Under 1000Mbps/100ms, the bursty throughput displayed by Fast in figure 14 is easily confirmed in figure 38 below:

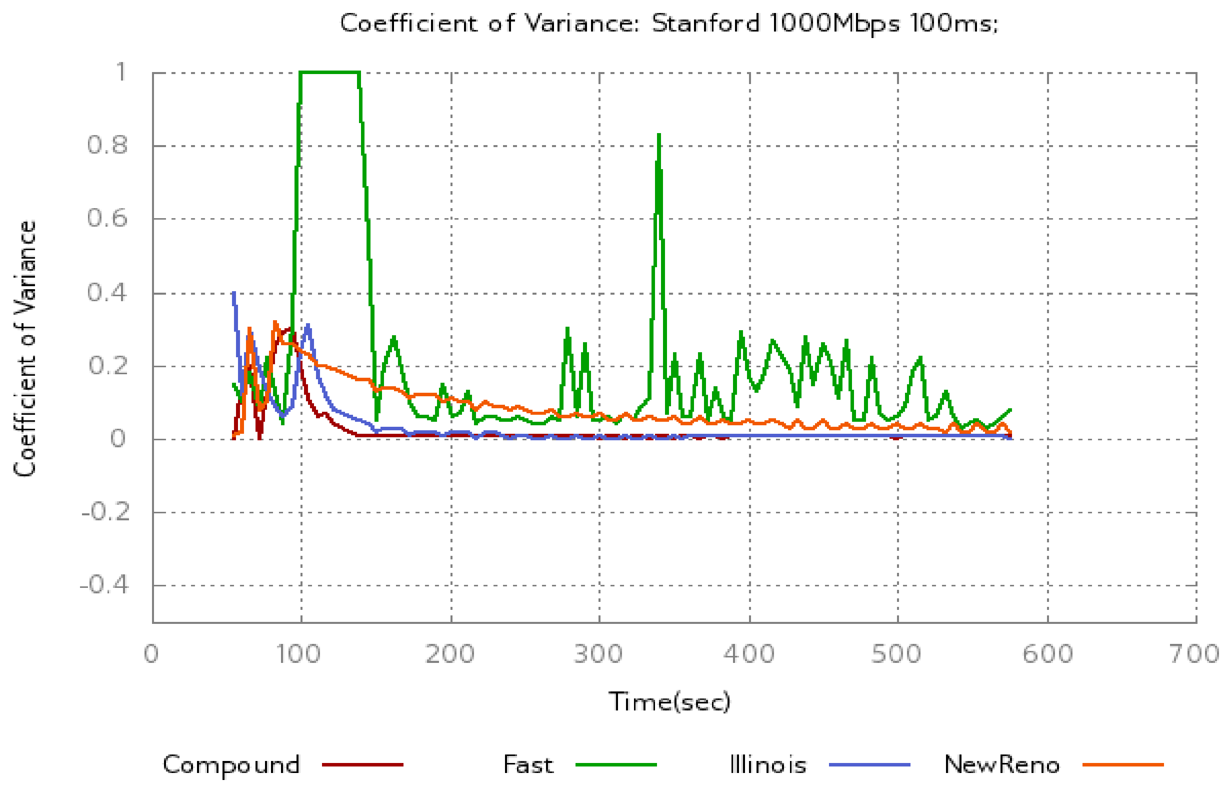


Figure 38: CoV at 1000Mbps/100ms

When the flows begin, the rush of packets during slow-start and the subsequent first few losses causes the throughput to vary wildly, which leads to the initial variation in the CoV. However, as the flows settle into congestion avoidance, the flows generally stabilize, leading to less burstiness. Compound is again shown to stabilize much faster than Illinois, as it responds faster to the burstiness-induced delay by adaptively reducing its window size and therefore not suffering large variation in throughput when a loss occurs. NewReno follow suit, while Fast suffers extremely bursty behavior.

7.3.3 UMass Router Buffer Size

Under the much smaller UMass router buffer size (20 packets), the delay-based flows are likely to suffer a much greater number of losses, causing them to behave much like NewReno. Barring a few disastrous losses and extraordinarily large bursts, Compound and Illinois are virtually indistinguishable from NewReno under smaller RTT regimes (figs. 39 and 40)

While Compound and Illinois appear to behave no worse than NewReno, Fast displays the greatest amount of burstiness so far. It continues its trend of erratic behavior even in environments with larger round trip times (figs. 41 and 42). Illinois, having a more aggressive growth function than Compound, begins to display far worse burstiness as well.

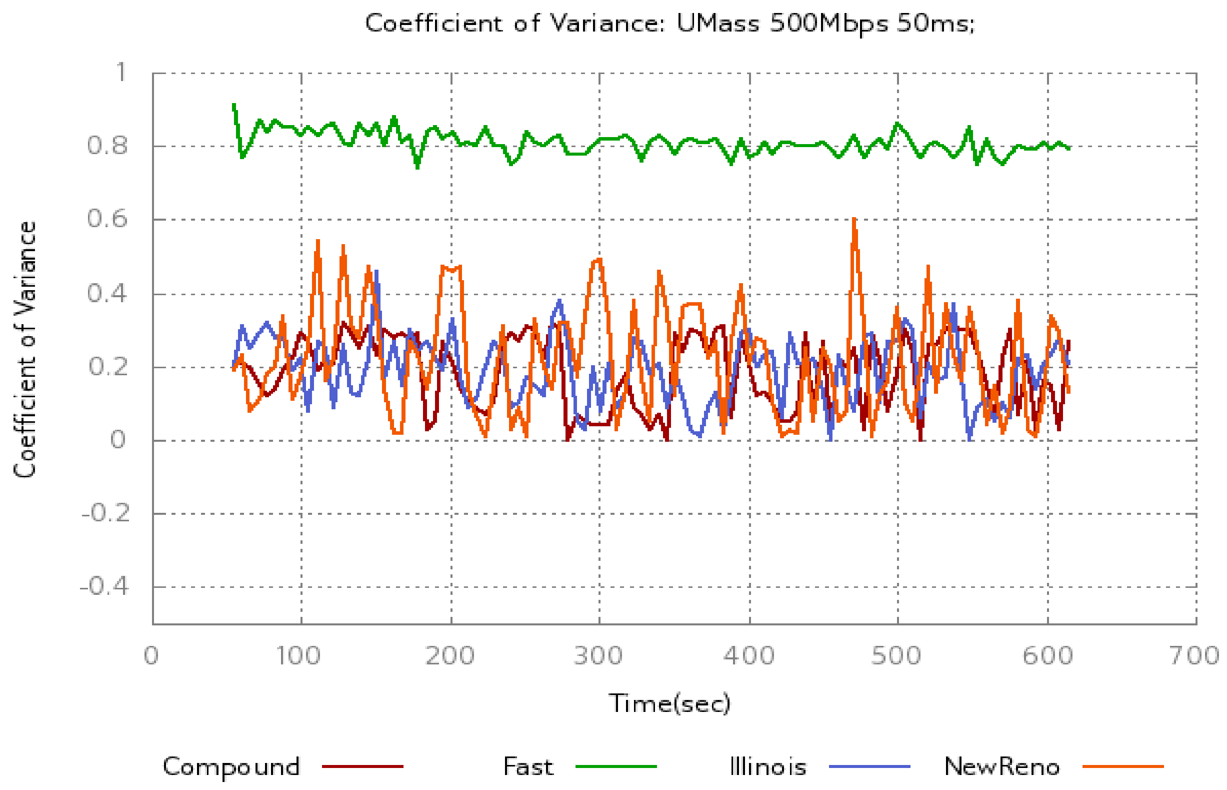


Figure 39: CoV at 500Mbps/50ms

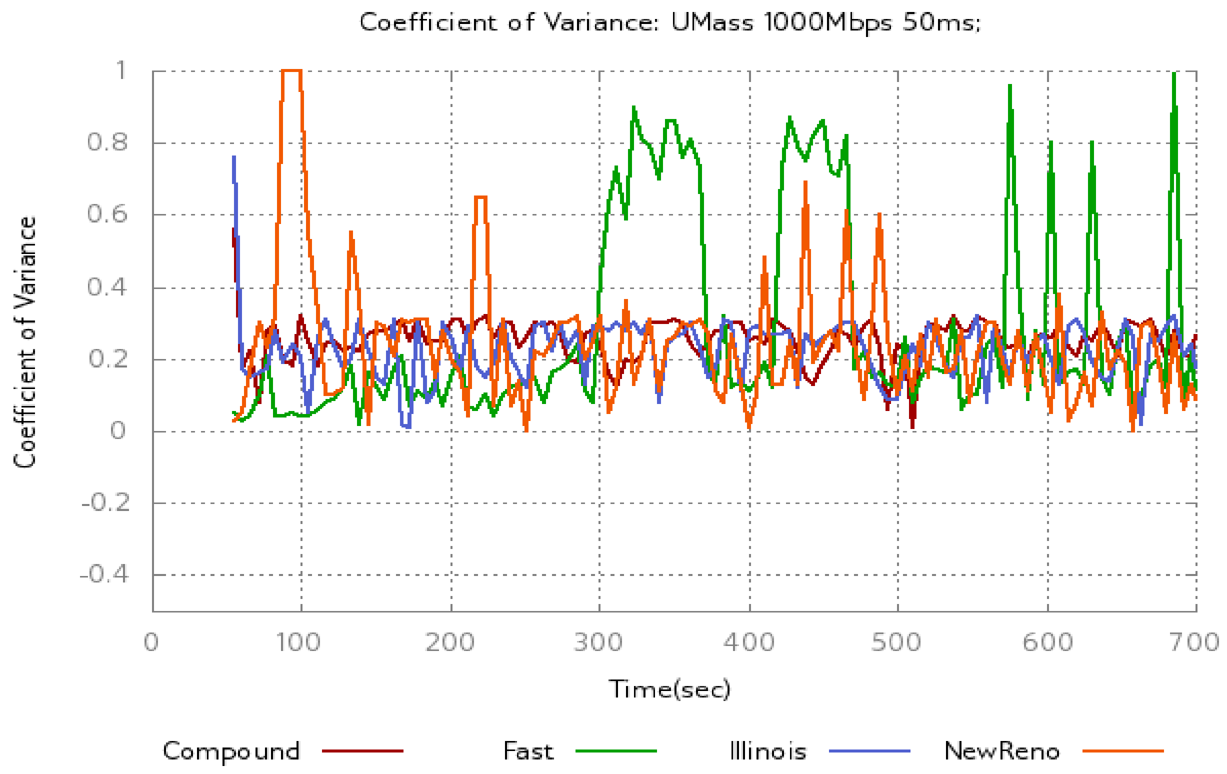


Figure 40: CoV at 1000Mbps/50ms

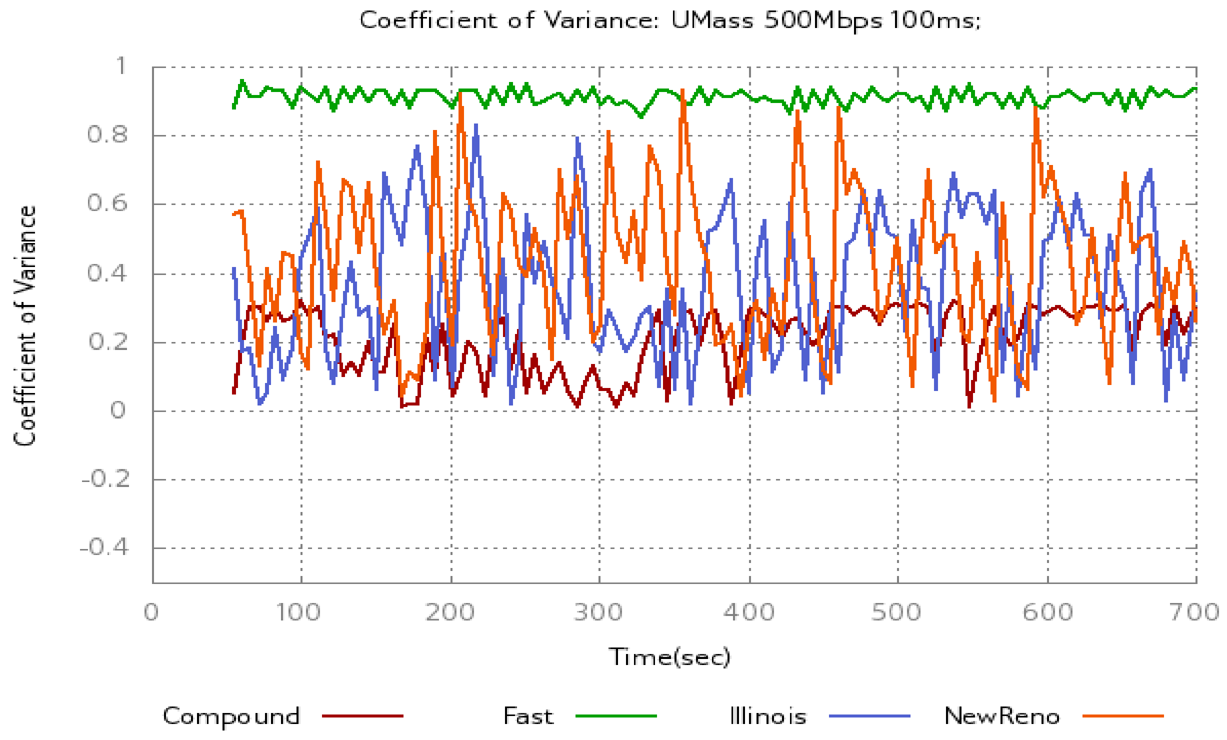


Figure 41: CoV at 500Mbps/100ms

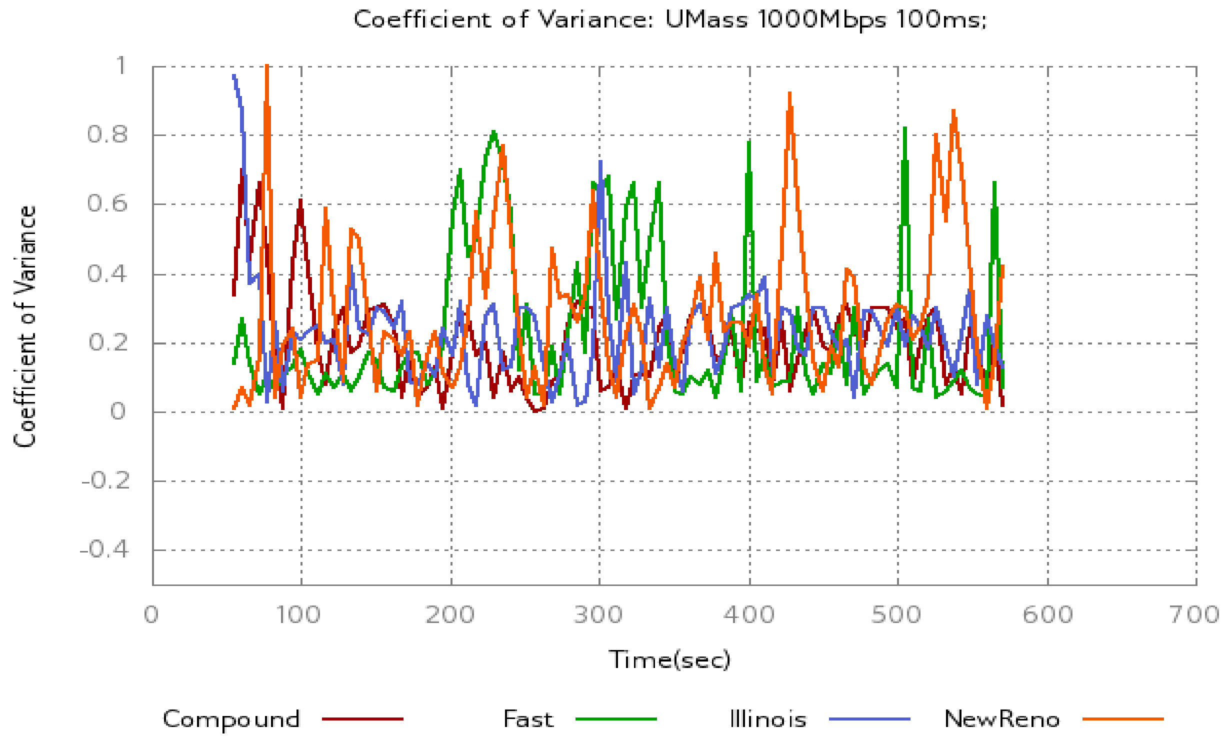


Figure 42: CoV at 1000Mbps/100ms

7.4 TCP Fairness & Convergence Time

Here, the ability of the protocols to coexist with NewReno TCP flows is observed. This is done by:

- comparing the throughput achieved by the high-speed flows to that achieved by a NewReno flow in a similar environment, and
- comparing the throughputs, at 1 second intervals, of the pre and post TCP flows with and without a high-speed flow, and determining convergence time (as derived from [ha 06], as the amount of time it takes a flow's throughput to reach 80% of a pre-existing flow's throughput).

7.4.1 Calibration – TCP NewReno

As the convergence of the pre and post flows was to be compared to their behavior without a high-speed flow, simulations were carried out to show how the pre and post flows would behave with a NewReno test flow, as an example of their base (i.e. expected) behavior. The results of these were as follows:

7.4.1.1 BDP Buffer Size

Under the BDP buffer size, the NewReno flows appear to experience a loss whenever a new flow's influx of packets is injected into the network. While it always happens that at least two flows converge, which two seems to rely purely on which flow recovers first from the loss, rather than which flow started first. Under the 500Mbps, 50ms environment, despite having no advantage at all, the test NewReno flow appears to take perform far better than the converging pre and post NewReno flows, as shown in figure 25.

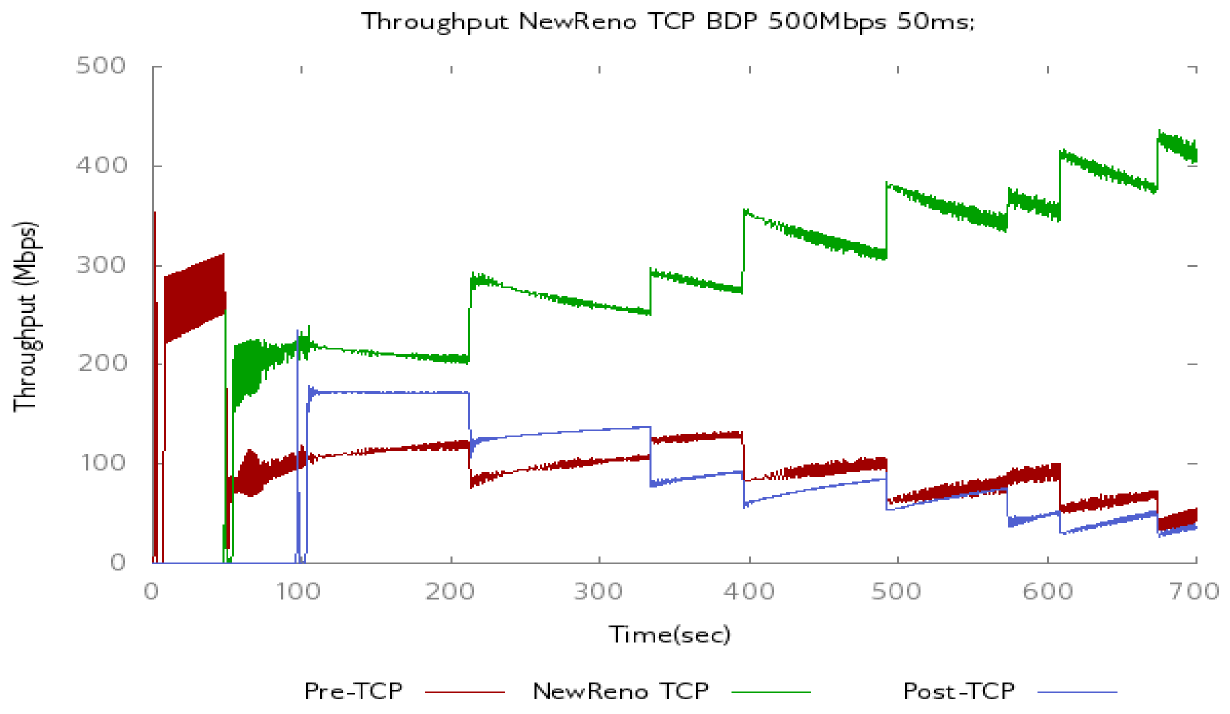


Figure 43: Convergence & TCP fairness of NewReno, 500Mbps/50ms

This behavior, of a random flow appearing to quickly build up while the other two converge, is also seen under 500Mbps and 100ms, where the post flow outperforms the converging pre and test flows.

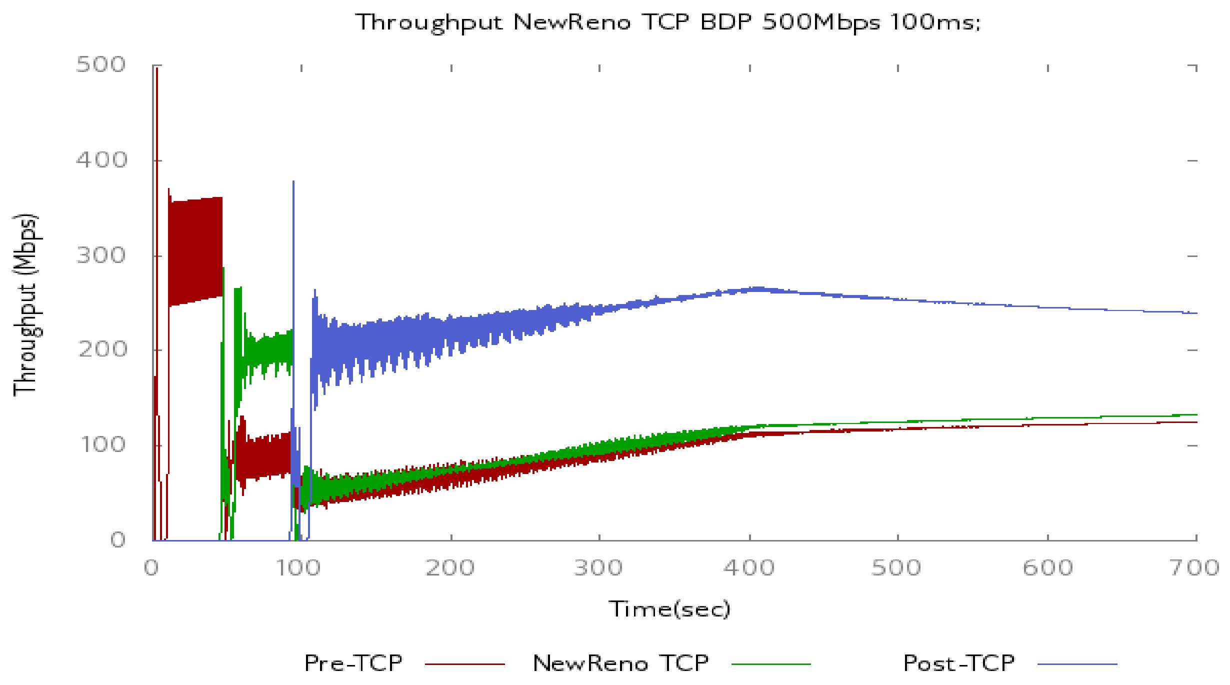


Figure 44: Convergence & TCP fairness of NewReno, 500Mbps/100ms

Under larger bottleneck bandwidth environments, the test NewReno flow, which recovers faster, appears to gain a far greater advantage. The pre and post NewReno flows appear to converge, barring a packet loss or two. Under the 1000Mbps, 50ms environment (figure 27), the throughputs appear to suggest that eventual convergence of all three flows may be possible, as they appear to approach a common point.

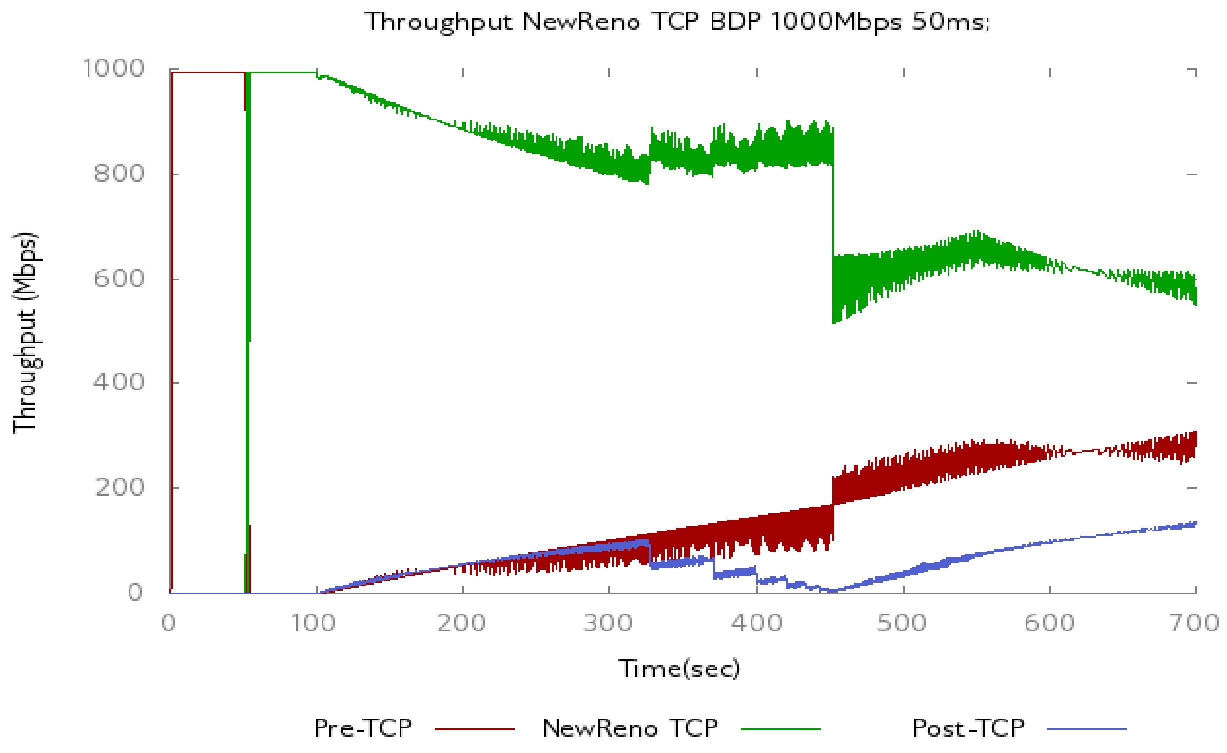


Figure 45: Convergence & TCP fairness of NewReno, 1000Mbps/50ms

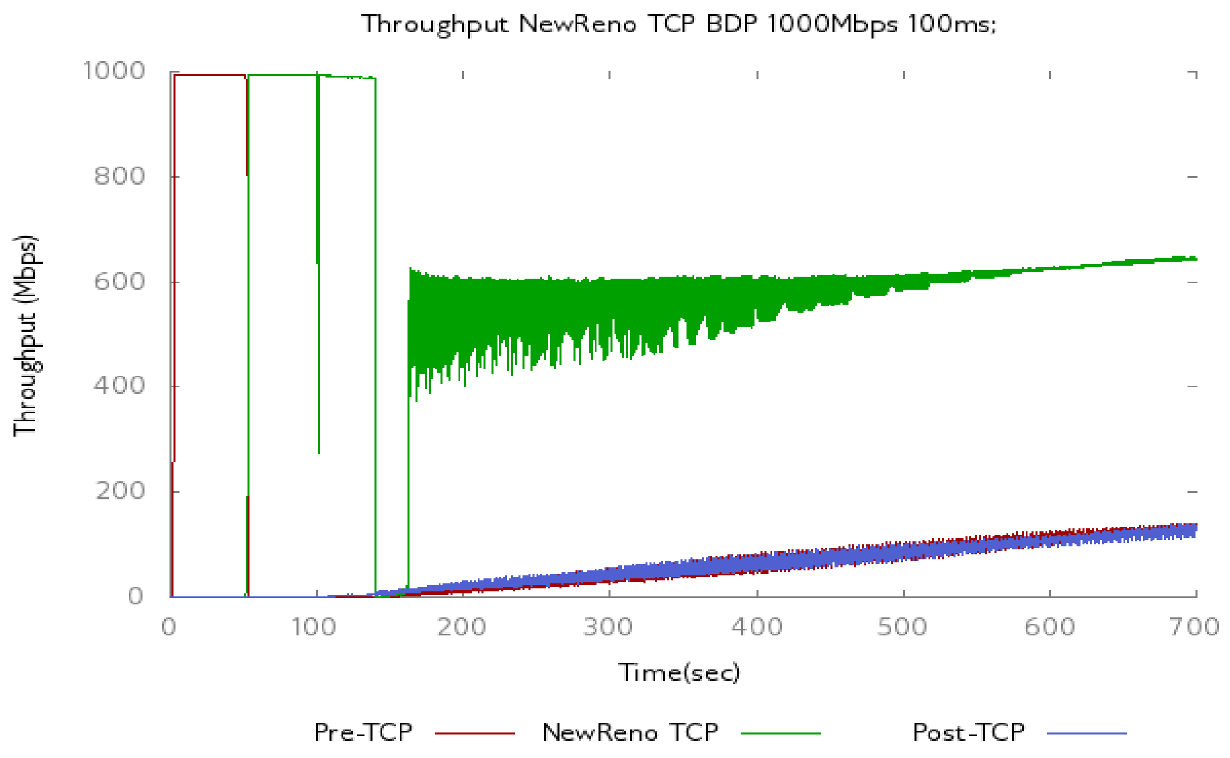


Figure 46: Convergence & TCP fairness of NewReno, 1000Mbps/100ms

7.4.1.2. Stanford Buffer Size

Under the Stanford buffer size, in environments with the lower bottleneck bandwidth (500Mbps), all the NewReno flows appear to perform rather poorly, as shown in figure 29 and 30 below:

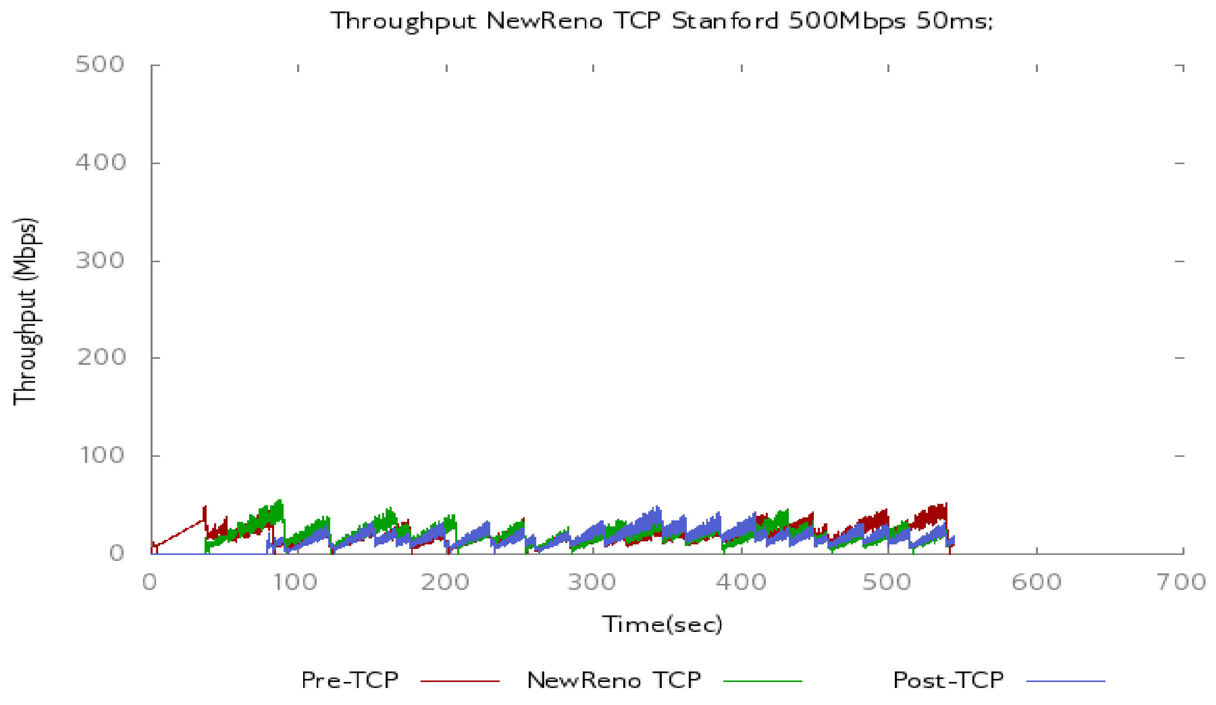


Figure 47: Convergence & TCP fairness of NewReno, 500Mbps/50ms

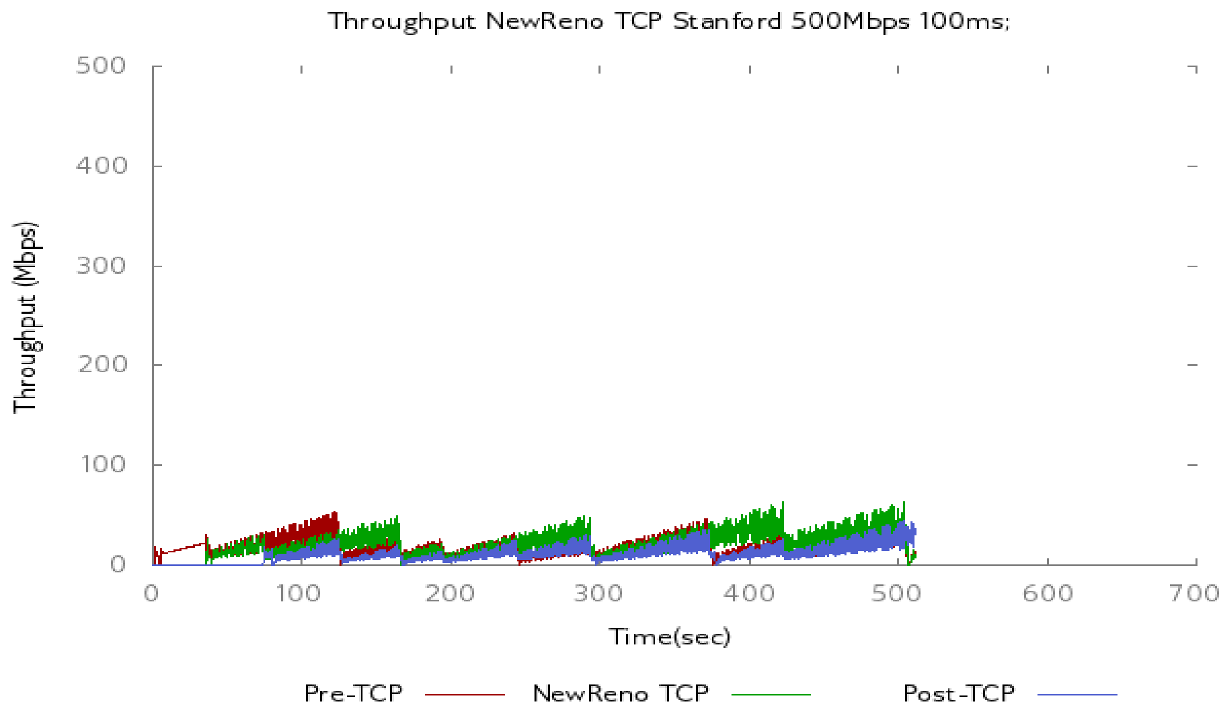


Figure 48: Convergence & TCP fairness of NewReno, 500Mbps/100ms

As the bottleneck link size increases to 1000Mbps, the NewReno perform far better. Under the 100Mbps/50ms regime, while the pre NewReno flow recovers from loss much faster, the short RTT allows the test and post NewReno flows to quickly recover and converge (see figure 31).

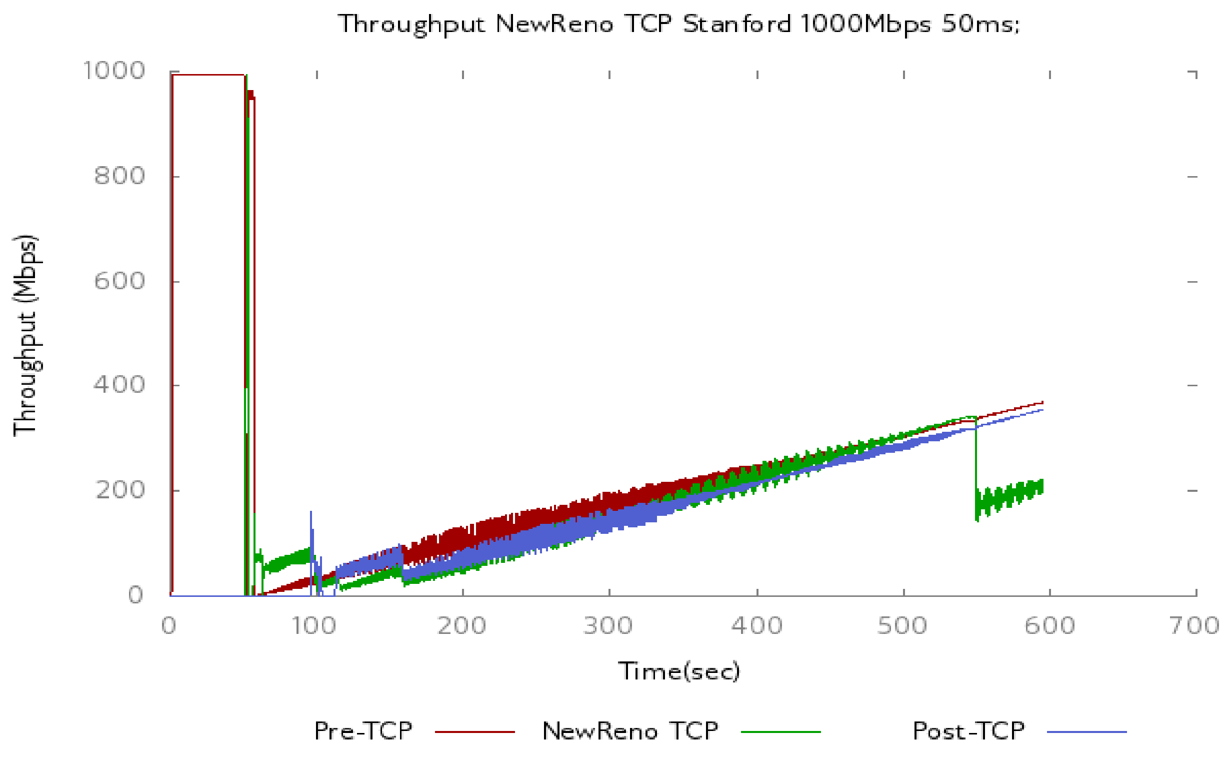


Figure 49: Convergence & TCP fairness of NewReno, 1000Mbps/50ms

Under 1000Mbps/100ms, while the post flow far outperforms the converged pre and test flows, they appear to increase in throughput as the simulation progresses (figure 32).

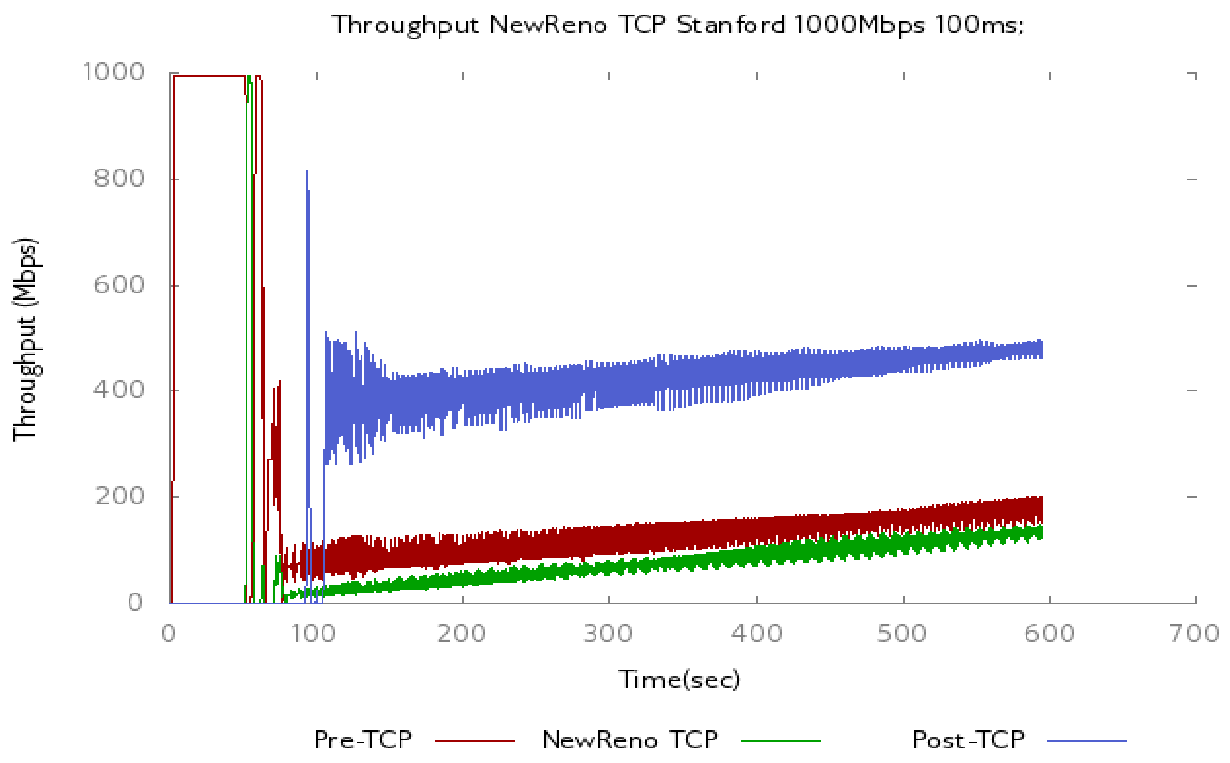


Figure 50: Convergence & TCP fairness of NewReno, 1000Mbps/100ms

7.4.1.3. UMass Buffer Size

Under the UMass router buffer size (20 packets), neither flow is able to achieve any significant throughput. As TCP at this low buffer size would be plagued by consecutive losses, neither flow is able to recover and take advantage of the link capacity. Link utilization remains low, and the flows converge to sub-century throughput (figures 33 – 36).

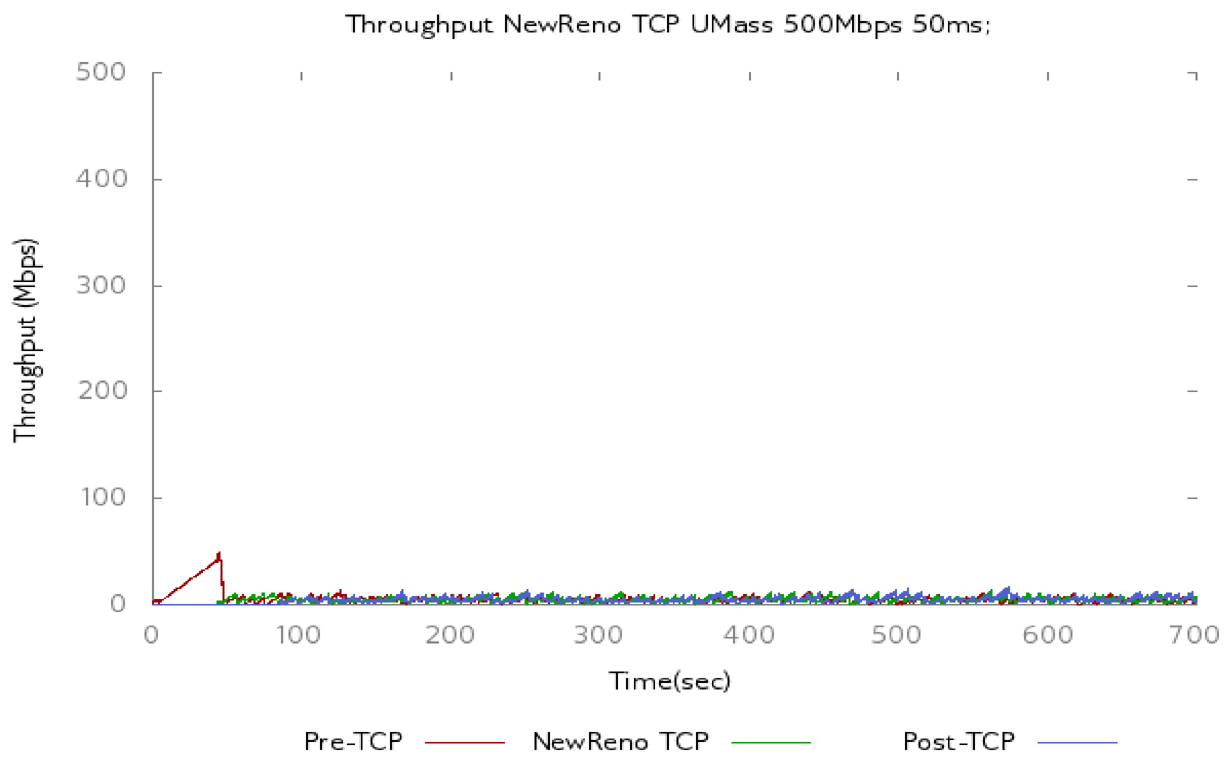


Figure 51: Convergence & TCP fairness of NewReno, 500Mbps/50ms

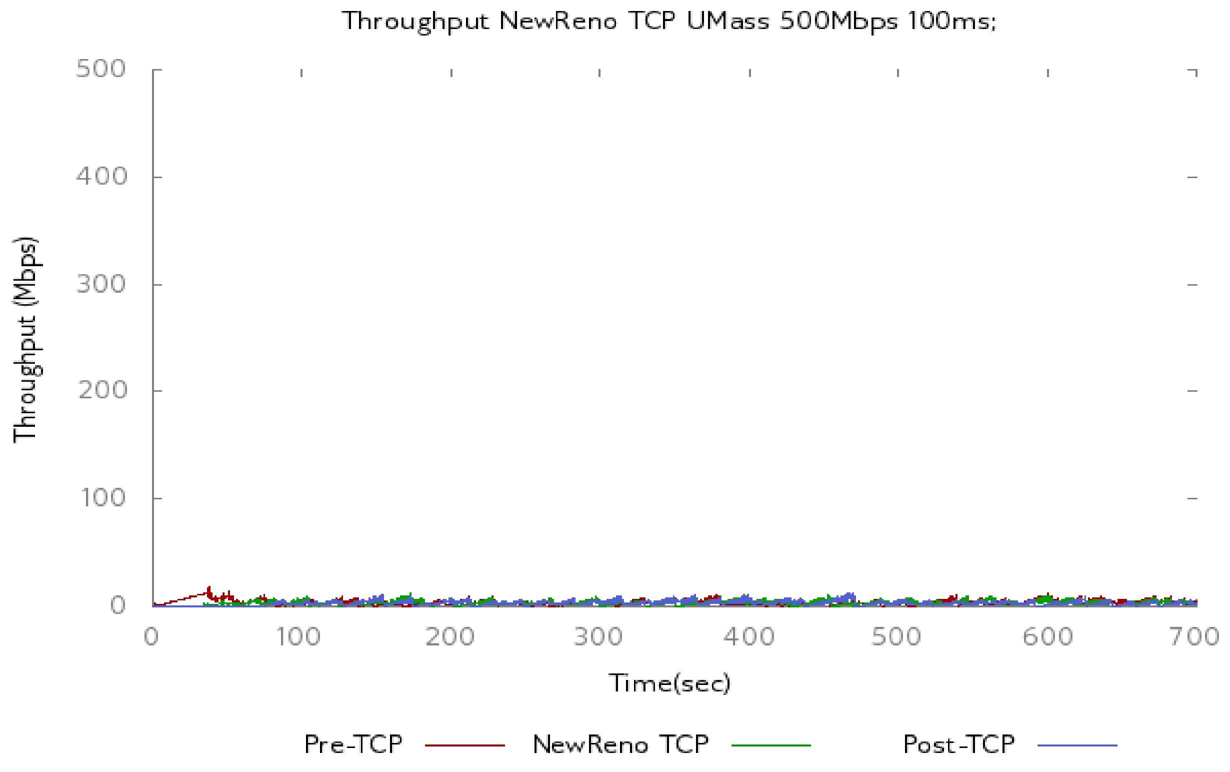


Figure 52: Convergence & TCP fairness of NewReno, 500Mbps/100ms

The only brought spark is the high initial utilization achieved by the flows at 1000Mbps (figures 35 and 36), but this too quickly wanes.

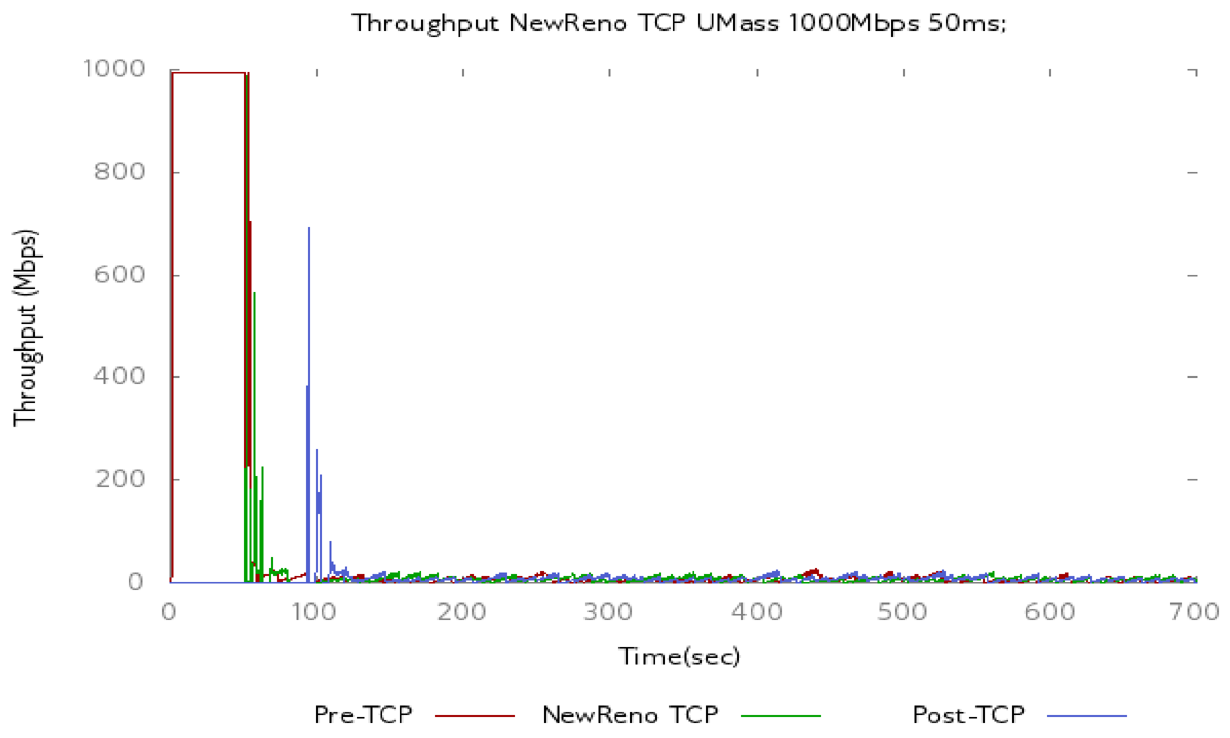


Figure 53: Convergence & TCP fairness of NewReno, 1000Mbps/50ms

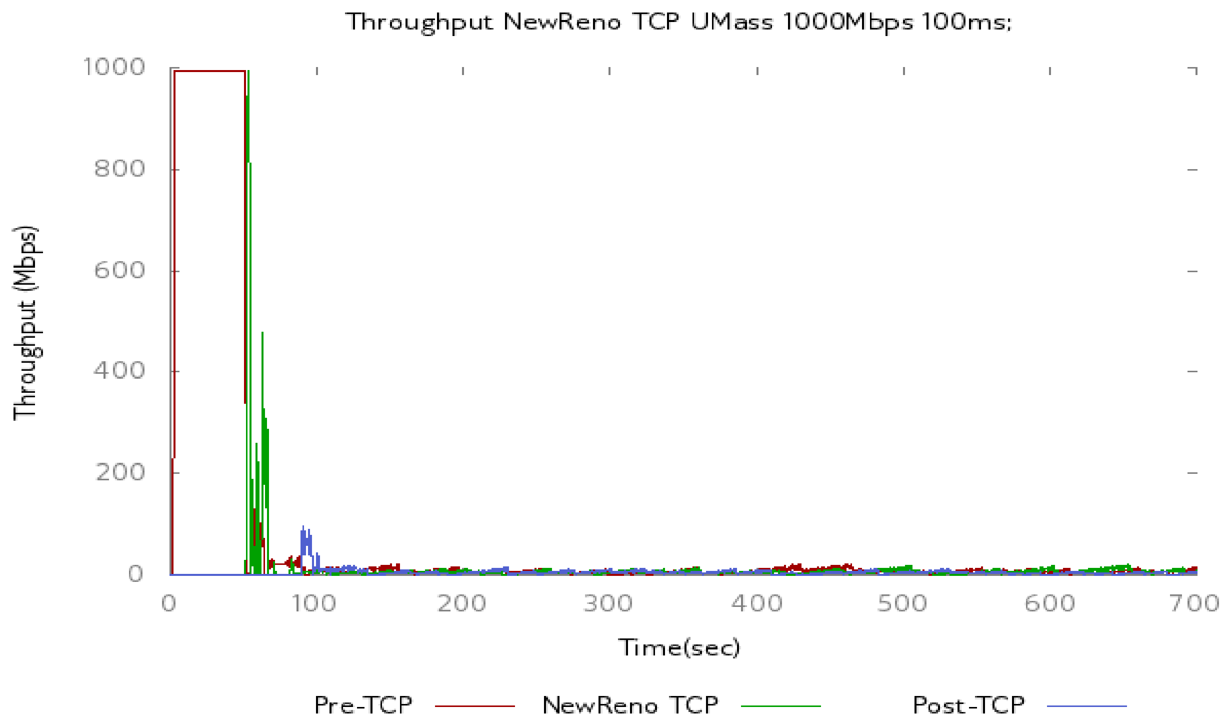


Figure 54: Convergence & TCP fairness of NewReno, 1000Mbps/100ms

7.4.2 Compound TCP

7.4.2.1. BDP Buffer Size

Compound generally appears to display TCP friendliness with NewReno. In small bandwidth-delay environments, the legacy (pre) flow is able to retain most of its bandwidth and easily recover from loss, as shown in figure 23 below. The new (post) NewReno flow is also able to gain significant bandwidth share, despite starting 50 seconds after the high-speed flow.

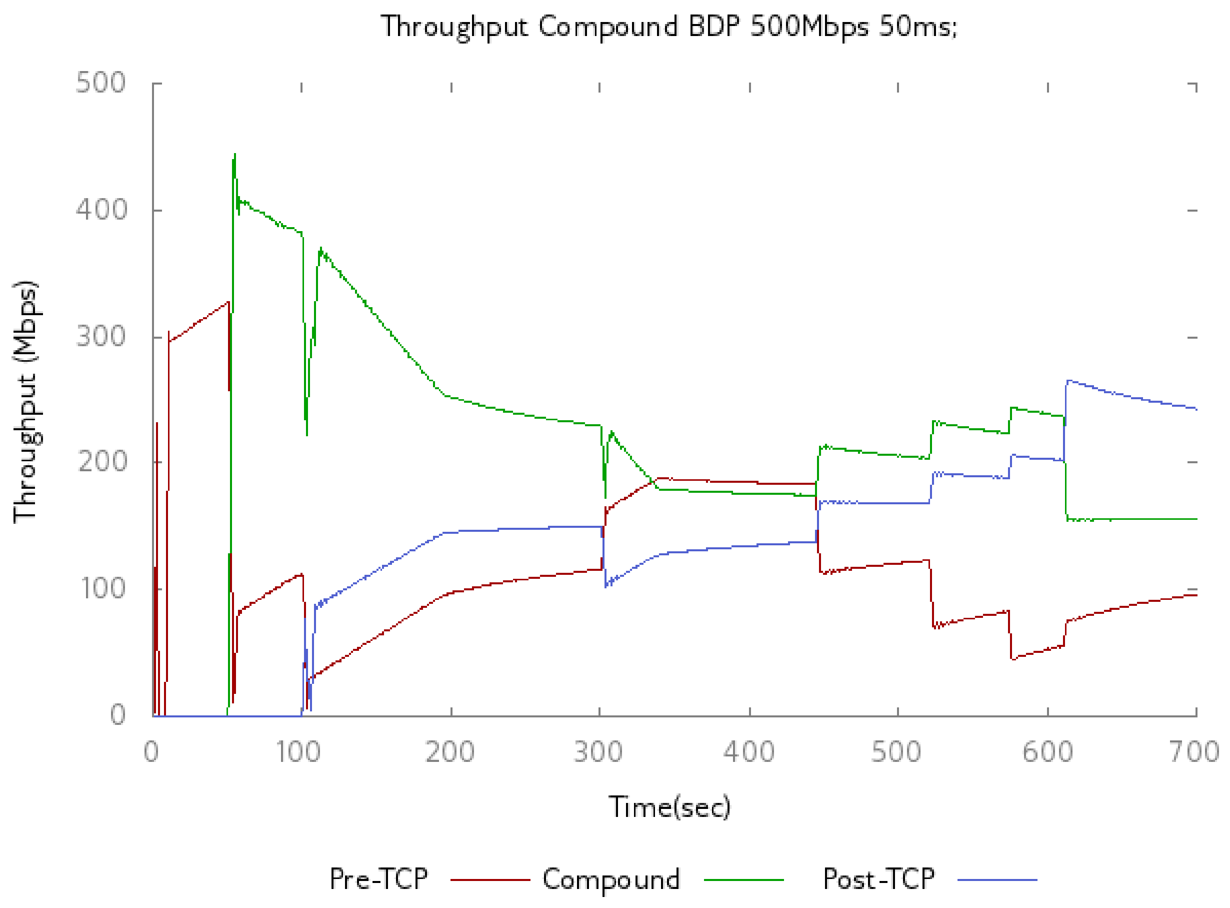


Figure 55: Convergence & TCP fairness of Compound, 500Mbps/50ms

Compound's throughput under larger bandwidth delay product environments behaves in the same manner. As the throughputs of the TCP flows steadily increase, Compound's throughput adjusts accordingly. Further analysis and longer simulations might show whether it maintains the same throughput as the TCP flows, which is the expected behavior of Compound, or whether it suffers the same problems as TCP-Vegas, i.e. losing too much bandwidth to non-delay based flows.

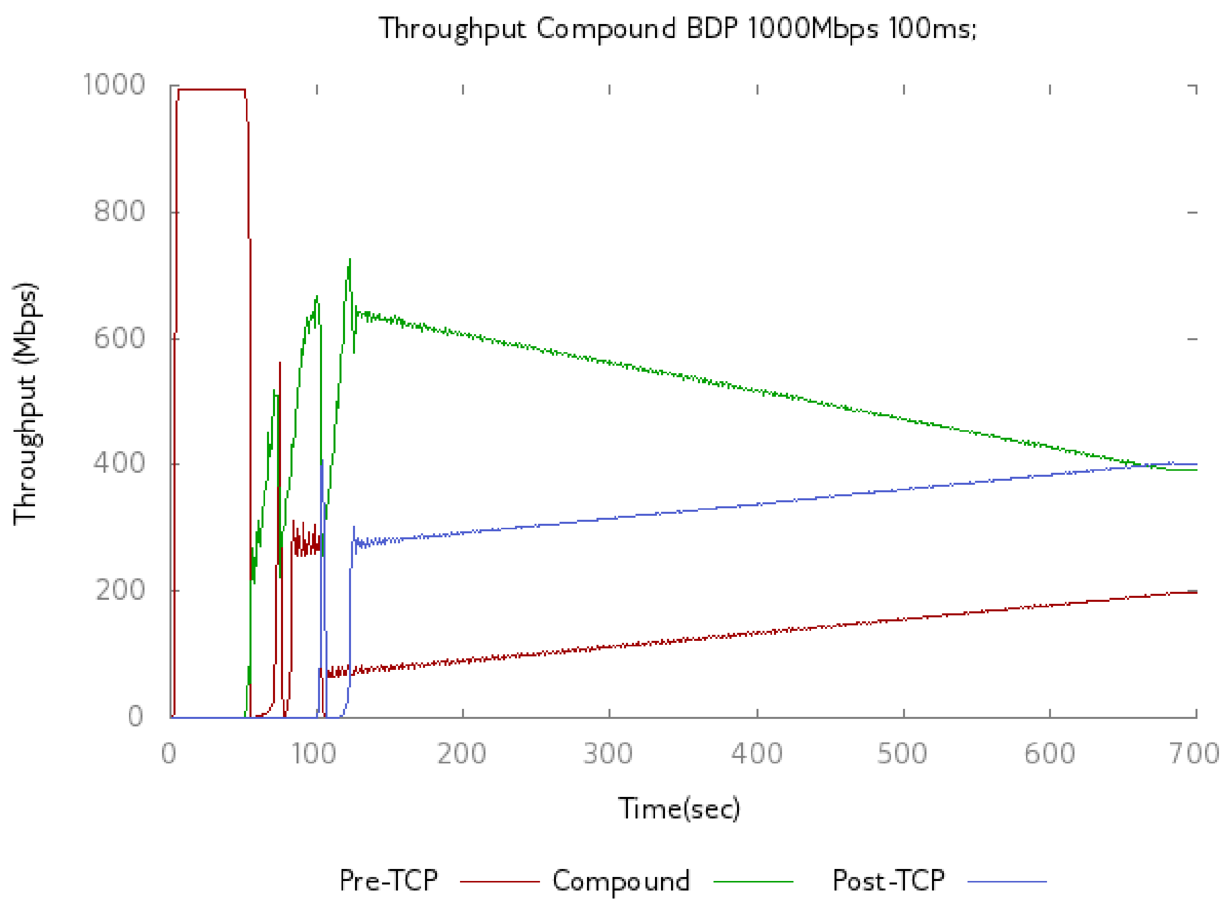


Figure 56: Convergence & TCP fairness of Compound, 1000Mbps/100ms

7.4.2.2. Stanford Buffer Size

Compound's TCP friendliness appears to drop drastically under the much smaller Stanford router buffer size. As shown in previous sections, Compound's throughput is erratic under Stanford buffers, and this appears to greatly affect the ability of the legacy (pre) and post TCP flows to gain and/or retain any bandwidth. This may be due to the fact that under a smaller router buffer size, a loss may occur before the delay change is significant enough to warrant a large reduction in the window size; the delay-based component of its window may not have encountered a significant enough delay before the loss-based component experiences packet loss. Figure 25 below shows the behavior of Compound under Stanford buffers, in environments with both low and high bandwidth delay product.

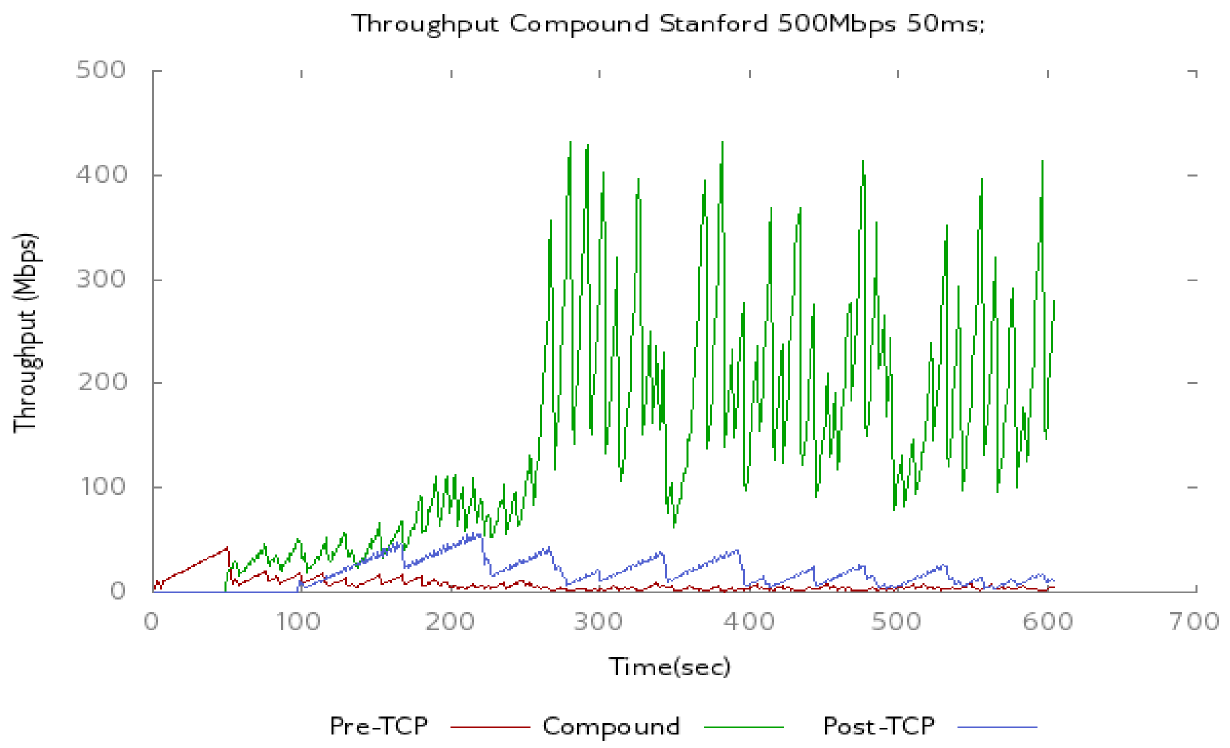


Figure 57: Convergence & TCP fairness of Compound, 500Mbps/50ms

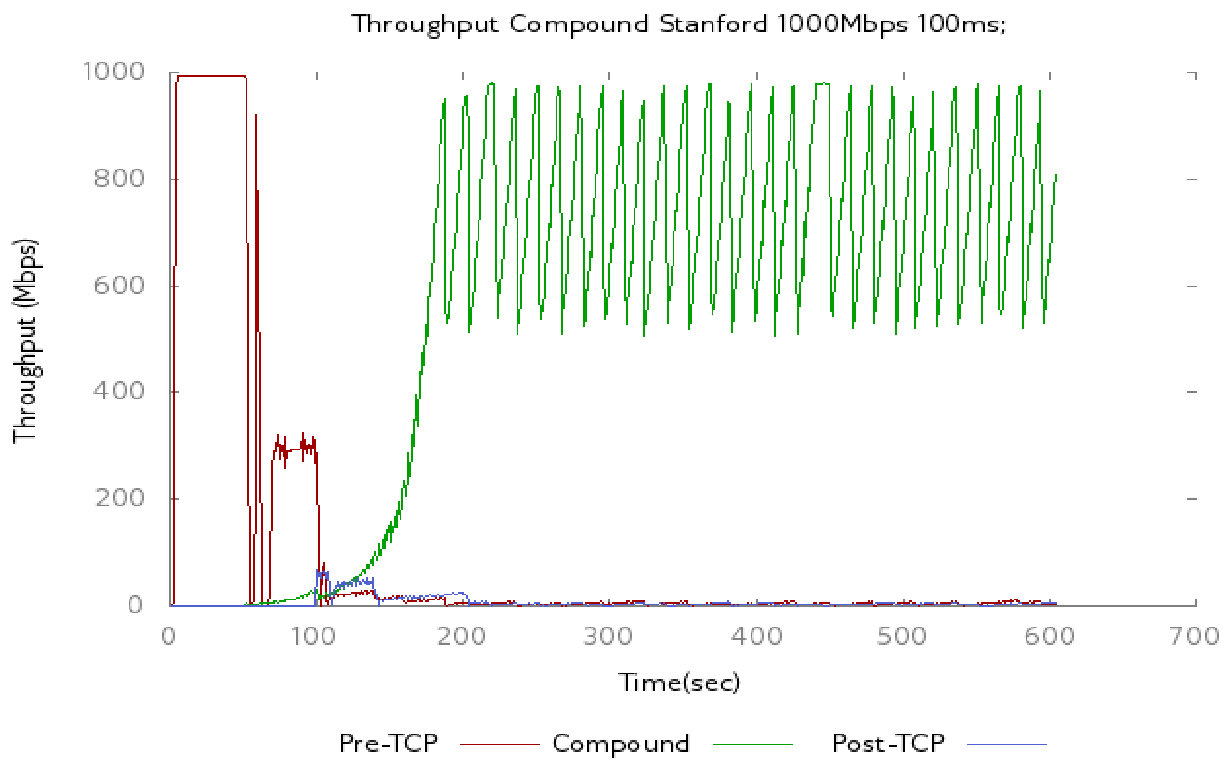


Figure 58: Convergence & TCP fairness of Compound, 1000Mbps/100ms

7.4.2.3 UMass Buffer Size

Under the UMass router buffer, Compound behaves in much the same way as the other TCP flows. Neither the pre nor the post NewReno flows achieve much throughput, and Compound appears to act in much the same way, having difficulty in adapting to such tiny buffer sizes.

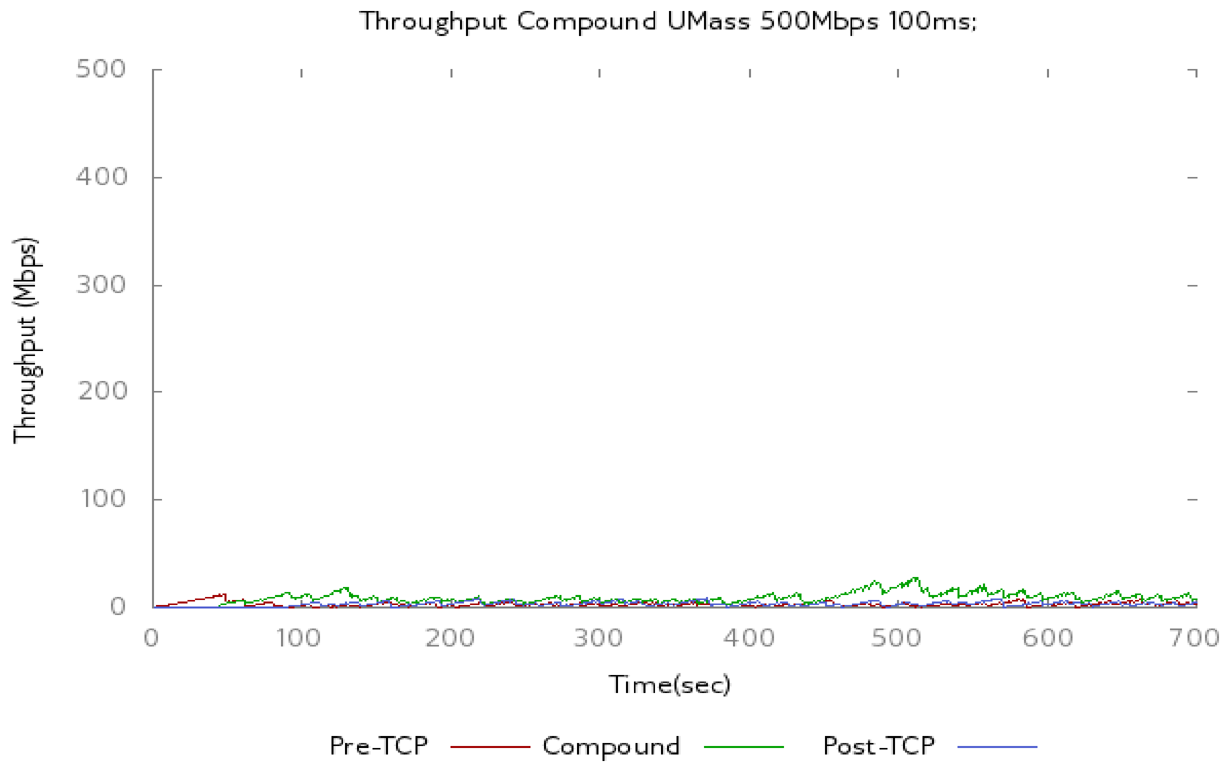


Figure 59: Convergence & TCP fairness of Compound, 500Mbps/100ms

7.4.3 Fast TCP

7.4.3.1 BDP Buffer Size

As previously mentioned, Fast TCP's performance throughout the simulation procedures was wanting. Unlike Compound, and much like Vegas, Fast has a hard limit on the router buffer occupancy (i.e. the number of packets to have in the queue). Its stability in maintaining this therefore becomes its undoing; it doesn't appear to adapt to network conditions and take advantage of available bandwidth. The pre and post TCP flows are therefore able to quickly and easily converge in small bandwidth delay environments, as shown in figure 28 below, without Fast being a hindrance. Under shorter RTTs, TCP's inherent unfairness comes into play, with Fast once again maintaining its buffer occupancy limit (fig. 29). Fast's TCP friendliness thus appears to be highly dependent on the buffer occupancy limit.

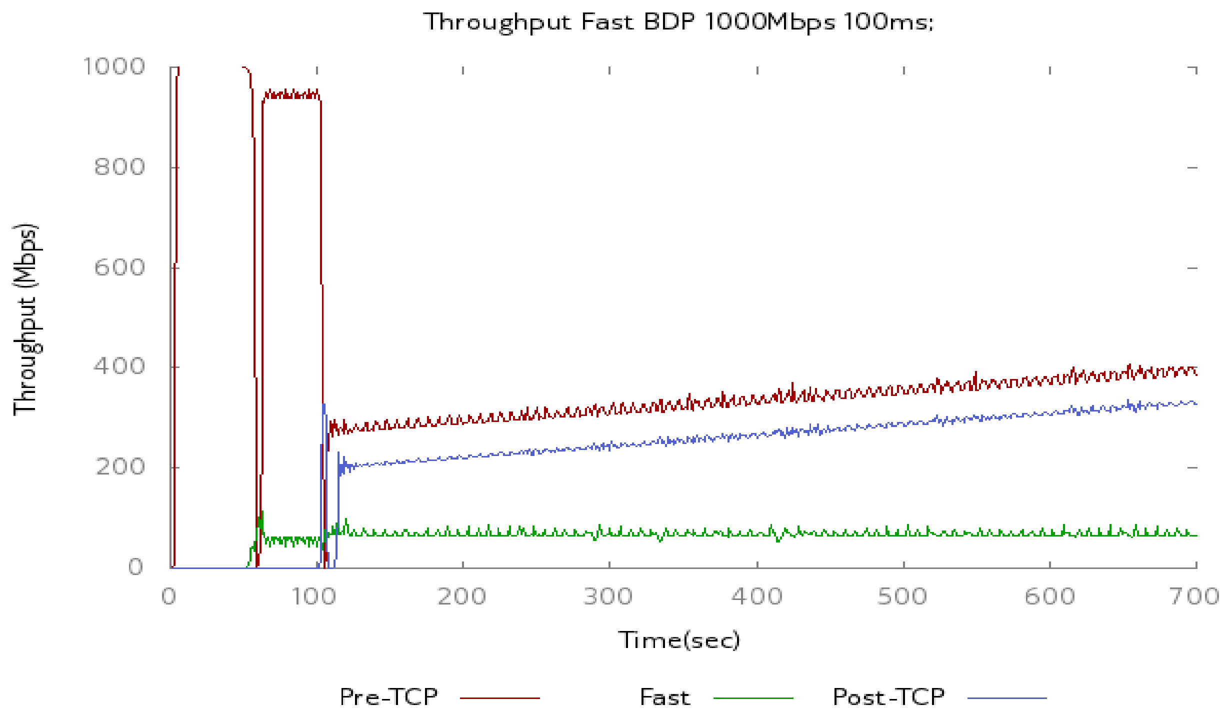


Figure 60: Convergence & TCP fairness of Fast, 1000Mbps/100ms

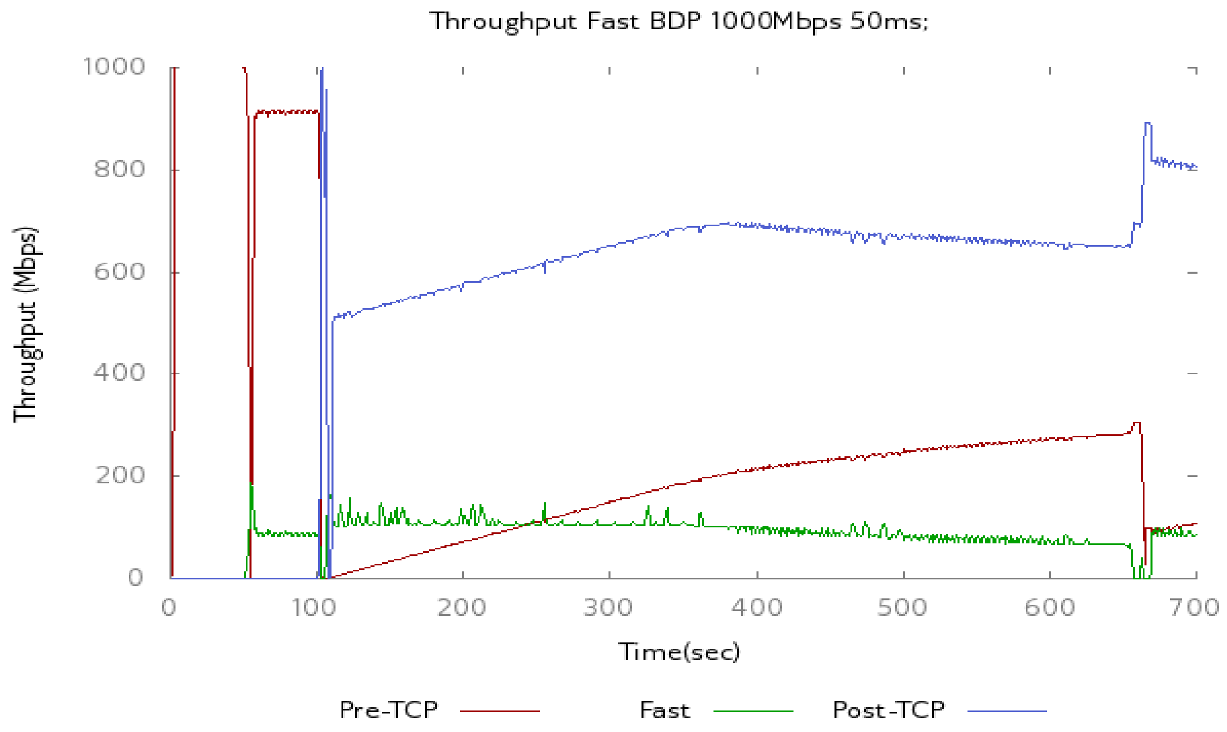


Figure 61: Convergence & TCP fairness of Fast, 1000Mbps/50ms

7.4.3.2. Stanford Buffer Size

Fast’s inability to adapt is again evident in Stanford buffers. The TCP flows only achieve low throughput as they quickly experience loss, but Fast still remains unable to scale above the set router buffer occupancy level. It does, however, appear to show a higher degree of burstiness under higher link capacities and low buffer sizes (Fig. 31 in next section). Figure 30 below shows the behavior of Fast and the pre and post TCP flows under low bandwidth environments.

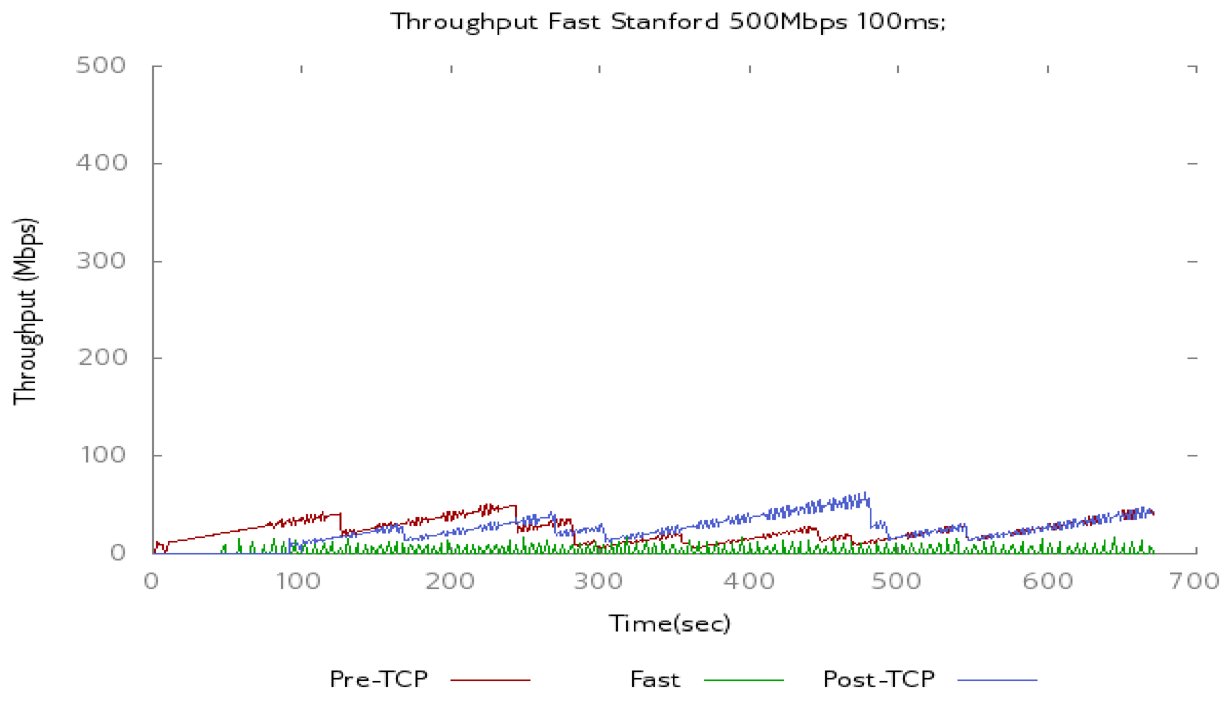


Figure 62: Convergence & TCP fairness of Fast, 500Mbps/100ms

7.4.3.3 UMass Buffer Size

Under the UMass router buffer size, Fast continues to display poor adaptability. The TCP flows, suffering from the low buffer size, are unable to scale to much of a degree that the TCP friendliness of Fast can be determined.
Surprisingly, Fast appears to show a higher degree of burstiness under higher bandwidth/low buffer envirinments, as shown in figure 31 below.

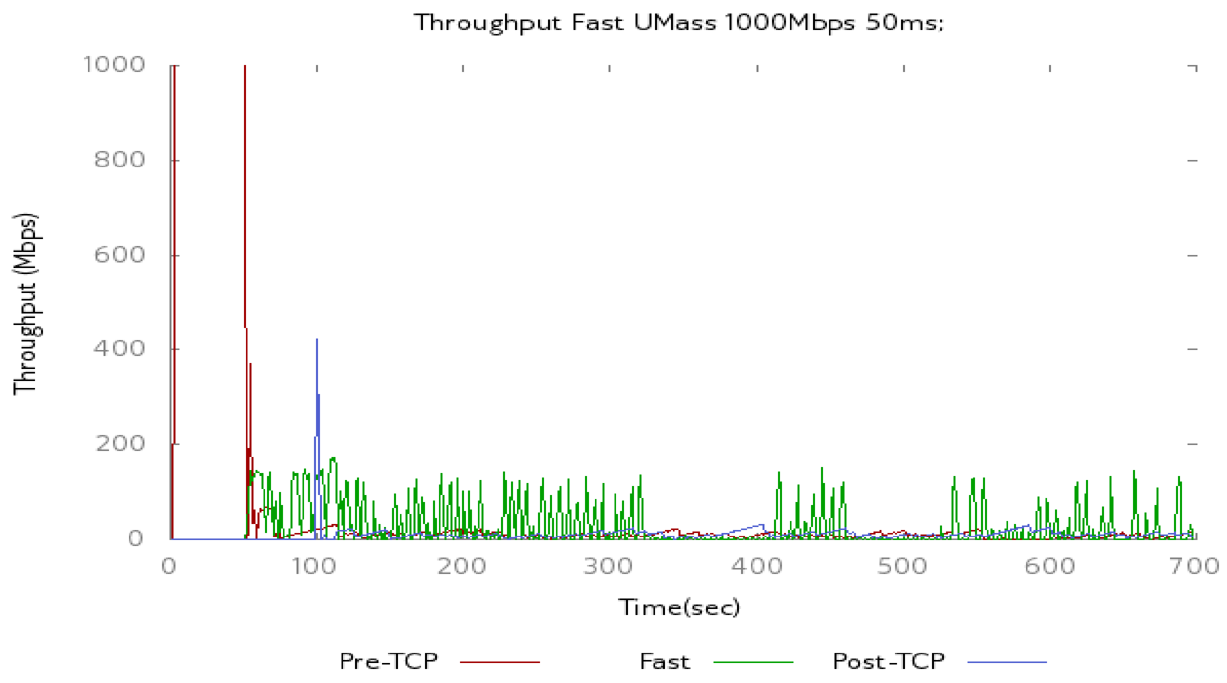


Figure 63: Convergence & TCP fairness of Fast, 1000Mbps/50ms

7.4.4. TCP Illinois

7.4.4.1 BDP Buffer Size

Under the BDP buffer size, Illinois is able to scale much faster than either the pre or post NewReno flows. This, however, makes it rather unfriendly to both flows. According to [lieth 08], this may be to an implementation error in Illinois that over-estimates the RTT of a flow. Since Illinois' window reduction factor β reduces as RTT increases (in order to maintain the Cubic-like concave throughput function), an overestimated RTT would lead to Illinois not backing off as much as it should when a loss occurs. Figure 32 shows examples of Illinois' behavior in a smaller bottleneck-link environment.

Under environments with larger bottleneck bandwidths, Illinois' TCP unfriendliness appears to reduce, as shown in figure 33. The pre NewReno flow is able to regain bandwidth lost when the other flows begin.

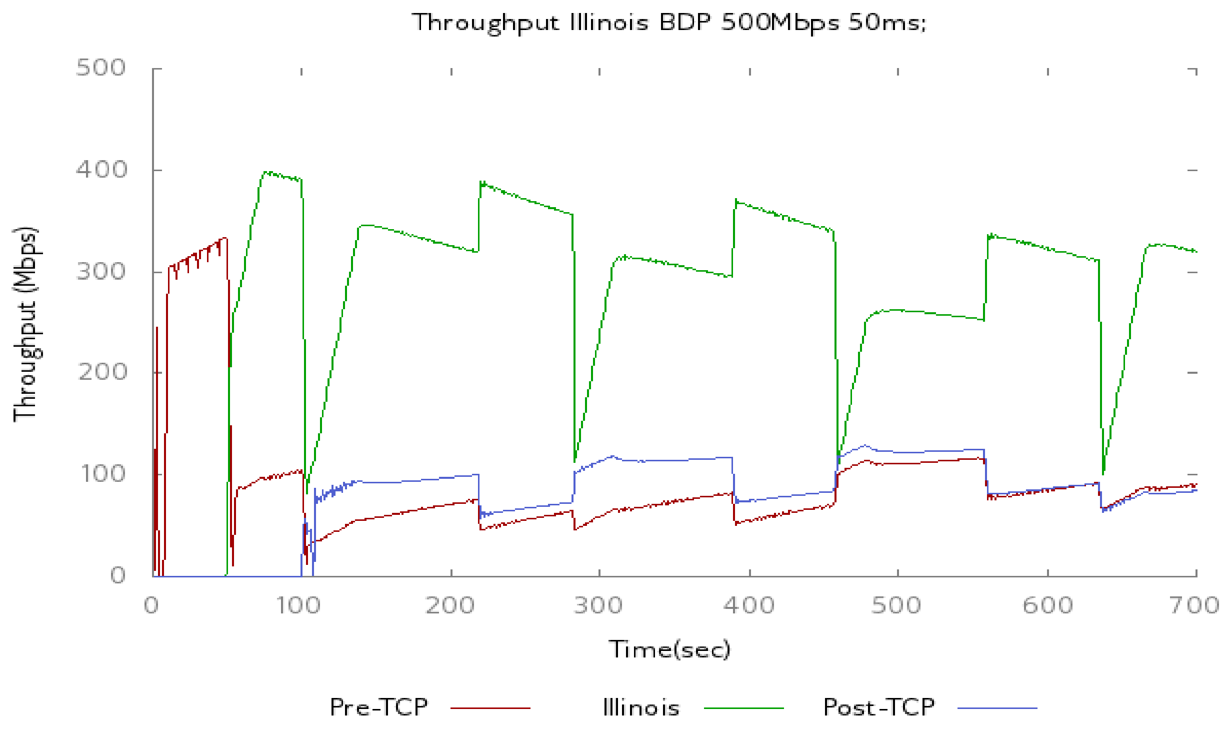


Figure 64: Convergence & TCP fairness of Illinois, 500Mbps/50ms

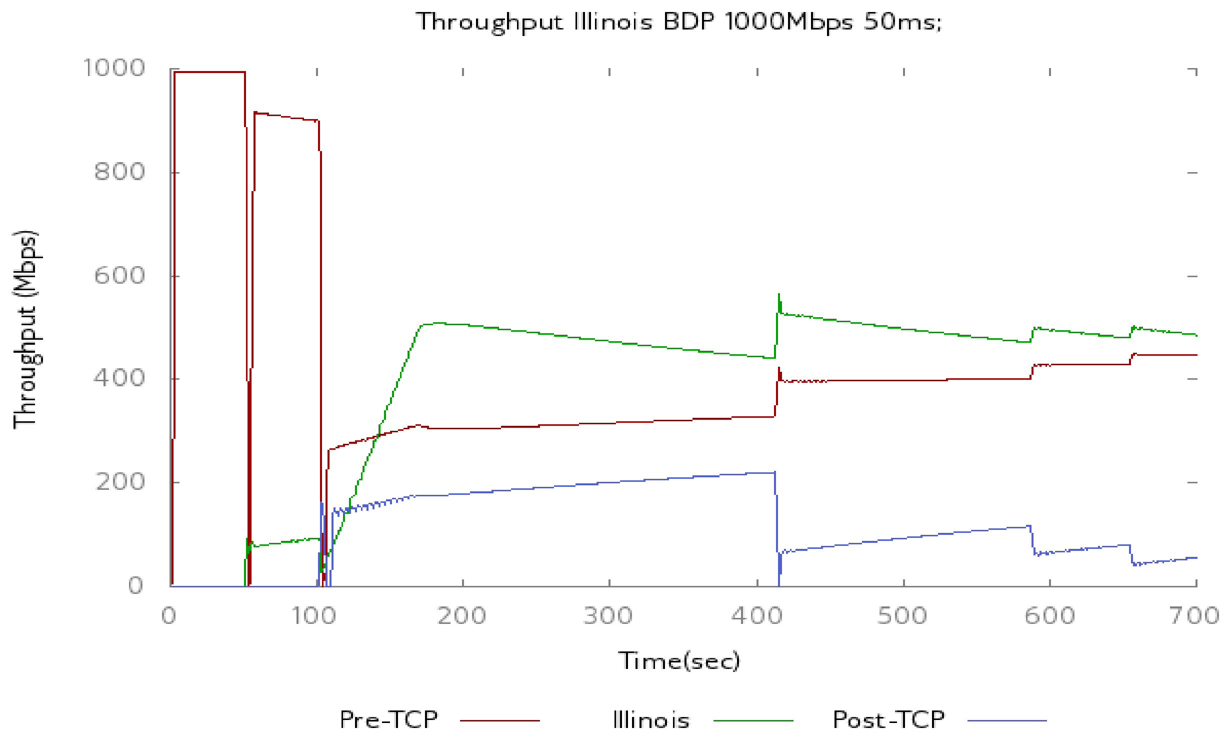


Figure 65: Convergence & TCP fairness of Illinois, 1000Mbps/50ms

7.4.4.2. Stanford Buffer Size

Under the much smaller Stanford buffer size, Illinois' TCP unfairness becomes even more clear as it takes up more bandwidth at the expense of the pre and post NewReno flows. Figures 34 and 35 below display this behavior.

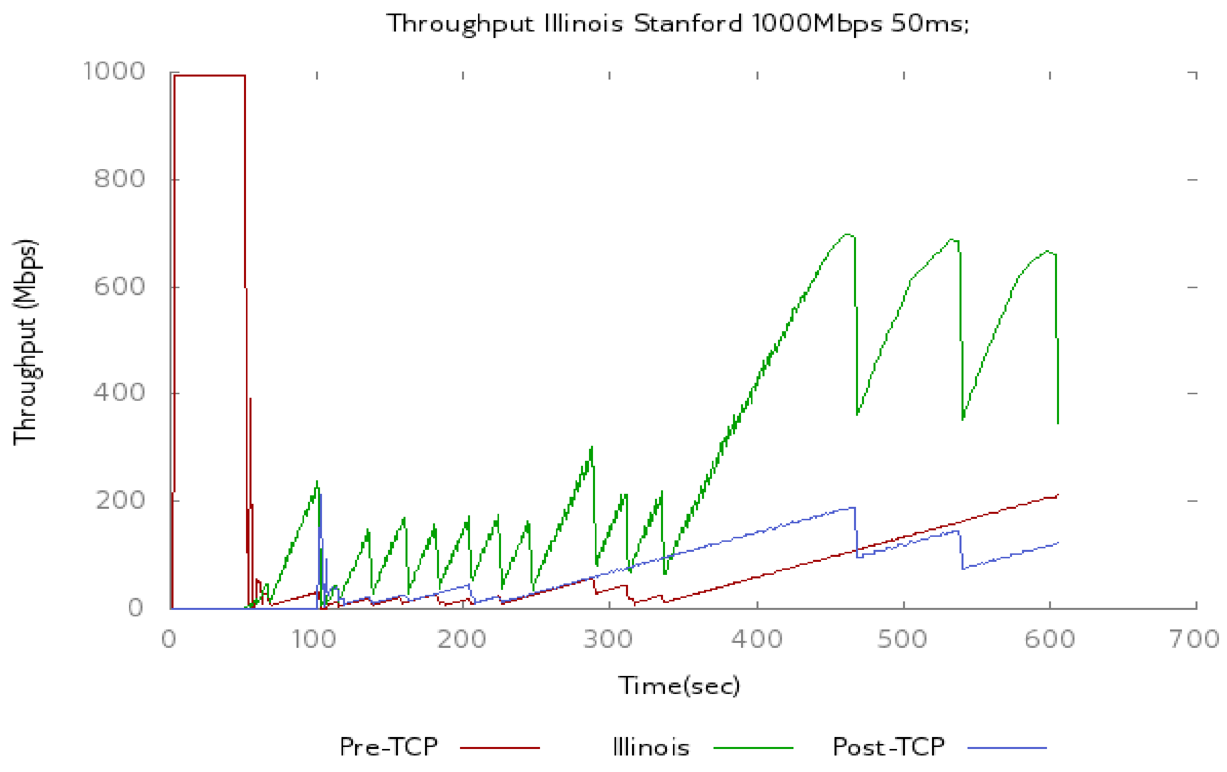


Figure 66: Convergence & TCP fairness of Illinois, 1000Mbps/50ms

As the throughput of the initial TCP flow falls when the new flows begin, Illinois is able to quickly take advantage of the available bandwidth and keep its window growing. Unlike Compound, whose throughput reduces as NewReno's increases, Illinois leaves little room for NewReno to increase. By not using delay as a primary congestion factor, Illinois is able to increase its throughput without much consequence until a loss occurs.

7.4.4.3. UMass Buffer Size

Under the UMass router buffer size, Illinois appears to endure packet loss and revert back to emulating NewReno behavior. Much like the pre and post NewReno flows, Illinois' throughput remains steadily low as it fails to grow its window size. Figure 36 below shows an example of this:

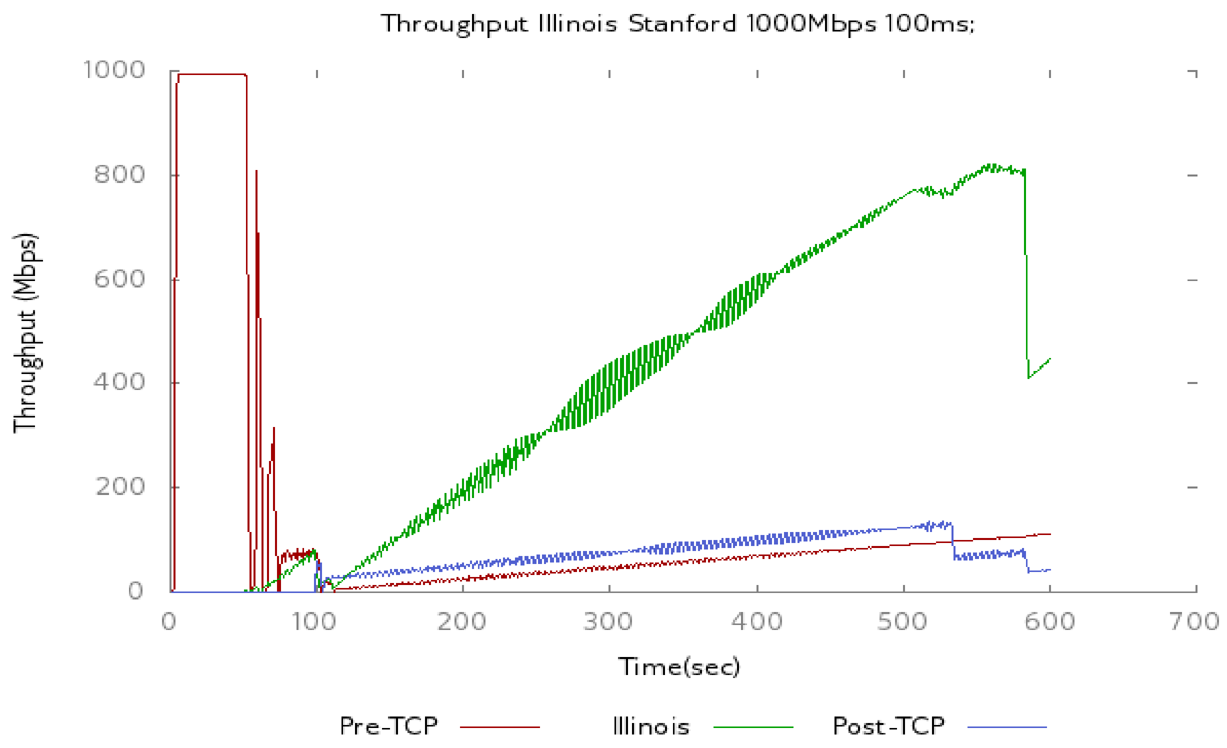


Figure 67: Convergence & TCP fairness of Illinois, 1000Mbps/50ms

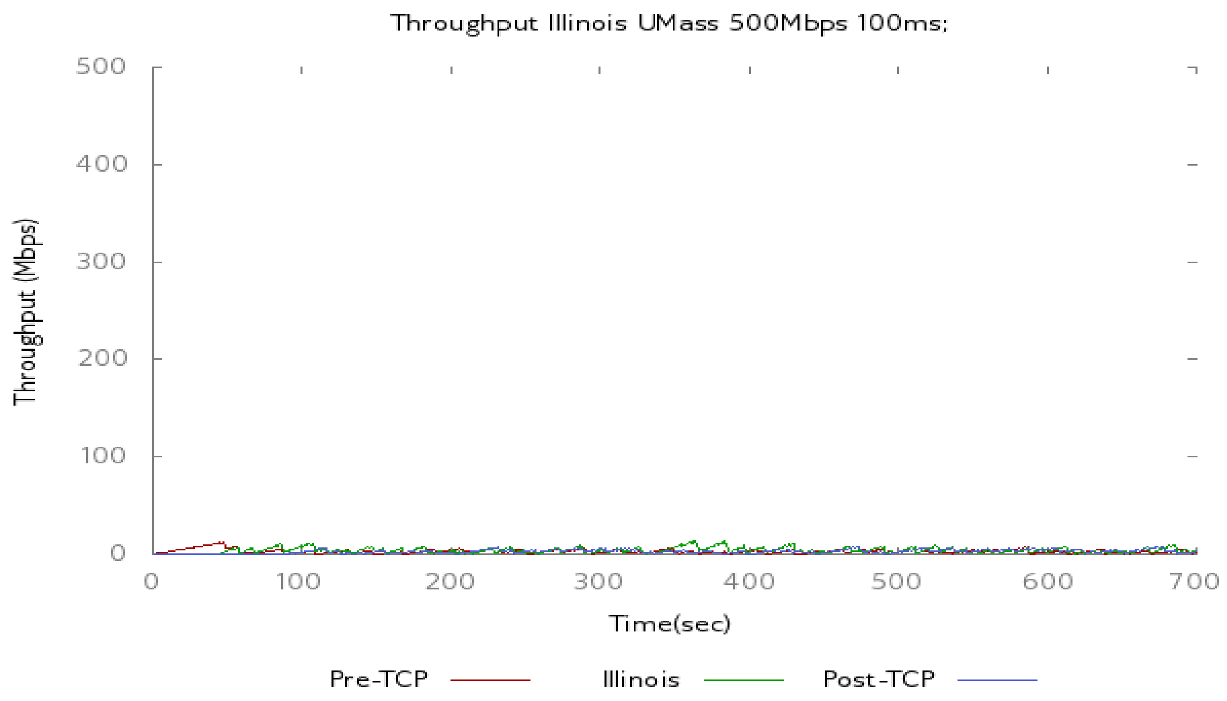


Figure 68: : Convergence & TCP fairness of Illinois, 500Mbps/100ms

7.5 Summary

As the results above have shown, the performance of delay-based high-speed protocols varies greatly with the router buffer size.

- Under a myriad of environments, the throughputs obtained by the protocols appear to have suffered consistently as the router buffer size reduced.
    - In large router buffers, Compound consistently lost bandwidth to long-lived NewReno flows, while Illinois was largely able to outperform NewReno.
    - In smaller buffer sizes, both Compound and Illinois suffer significant throughput loss.
    - Fast, due to its buffer occupancy limit parameter α, consistently disappoints, having far lower throughput than the other protocols.
- All the protocols appear to suffer from extreme amounts of burstiness and instability as the router buffer sizes fall. Fast remains the most stable under larger router buffer sizes, but scales rather poorly. Under small buffer sizes, however, Fast's burstiness outpaces even that of NewReno, while Illinois and Compound appear more muted in their burstiness.
- Of the three protocols, Compound appears to be the friendliest to TCP (perhaps *too* friendly) in large router buffer sizes. Fast, by keeping its output steady, presents a fairly consistent network view to other traffic on the link. Illinois seems to be the most aggressive of the three. This may either be due to the fat that it is more aggressive than Compound (using the delay only to determine how much to reduce its window by), or, as mentioned in [leith 08], due to an implementation error.

In essence, the efficacy of delay-based protocols appears to be drastically affected by smaller router buffer size. The advantage offered by detecting and reacting to imminent congestion is of little service in these environments. While attempts have been made to cater to smaller router buffer sizes (e.g. by pacing the send rate of protocols [tow 06]), more studies need to be done to determine how best to adapt delay-based protocols to smaller router buffers.

VIII. Conclusions

As research continues into finding ways to reduce the router buffer size and increase link capacity, it appears evident that delay-based protocols will have to adapt their mechanisms to the new router buffer regimes. Under smaller router buffer sizes, delay-based protocols appear to achieve low throughput and poor TCP friendliness, either giving up too much throughput, as Compound appears to, or taking up too much, as Illinois has been shown to do. The reliance upon semantics related to router buffer occupancy is fundamentally dependent on the availability and assurance of that occupancy, either through overprovision - larger buffer sizes that guarantee the space will be available – or through protocols setting limits that, like Fast, appear to hinder the protocol's ability to scale and adapt to smaller router buffer sizes and/or larger link sizes. Solutions such as adaptively changing the buffer occupancy limits, or introducing pacing mechanisms may need to be considered to make delay-based protocols viable under smaller router buffer regimes.

IX. REFERENCES

[ien 124] J. Postel. DOD Standard Transmission Control Protocol. Dec 1979.

[rfc 675] V. Cerf, Y. Dalal, C. Sunshine. RFC 675 Specification of the Internet Transmission Control Program. Dec. 1974

[rfc 761] J. Postel. RFC 761 DOD Standard Transmission Control Protocol. Jan 1980

[rfc 768] J. Postel. RFC 768 User Datagram Protocol. Aug. 1980

[rfc 793] J. Postel. RFC 793 Transmission Control Protocol: DAPRA Internet Program Protocol Specification. Sept 1981.

[rfc 896] J. Nagle. *RFC 896 Congestion Control in IP/TCP Internetworks.* Jan 1984.

[rfc 2001] W. Stevens. RFC 2001 TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms. Jan. 1997

[rfc 3649] S. Floyd. RFC 3649 HighSpeed TCP for Large Congestion Windows. Dec. 2003.

[aggar 00] A. Aggarwal, T. Anderson, S. Savage. Understanding the Performance of TCP Pacing. *Proceedings of the 2000 IEEE Infocom Conference, Tel-Aviv, Israel.* March, 2000.

[allman 99] M. Allman, V. Paxson, W. Stevens. RFC 2581 TCP Congestion Control. Apr. 1999.

[app 04] G. Appenzeller et al. Sizing Router Buffers. *Proceedings of ACM SIGCOMM 2004,* Sep. 2004

[brakmo 95] L. Brakmo, L. Peterson, S. O'Malley. TCP Vegas: New Techniques for Congestion Detection and Avoidance. *ACM SIGCOMM Computer Communication Review* Oct. 1994

[beh 08] N. Beheshti. et al. Experimental Study of Router Buffer Sizing. *Proceedings of ACN IMC* 2008, Oct. 2008

[bul 03] H. Bullot, R. L. Cottrell, R. Hughes-Jones. Evaluation of advanced TCP stacks on fast long distance production networks. *J. Grid. Comput.*, vol. 1, no. 4, pp. 345-359, Dec. 2003.

[cai 09] Y. Cai, S. Hanay, T. Wolff. Practical Packet Pacing in Small-Buffer Networks. *IEEE International Conference on Communications*. June 2009.

[cerf 74] V. Cerf, R. Kahn. A Protocol for Packet Network Intercommunication. *IEEE Transactions on Communications*, May 1974

[chiu 89] D. Chiu, R. Jain. Analysis of increase and decrease algorithms for congestion avoidance in computer networks. *Computer Networks and ISDN systems 17: 1–14.*

[clark 81] D. Clark, D. Reed, J. Saltzer. End-to-End Arguments in System Design. *ACM Transactions on Computer Systems*, vol 2, Nov. 1984

[clark 91] D. Clark, S. Shenkar, L. Zhang. Observations on the Dynamics of a Congestion Control Algorithm: the Effects of Two-Way Traffic. *SIGCOMM Comput. Commun. Rev.*, 21(4):133–147, 1991

[coulter 00] R. Coulter et al. A Simulation Study of Paced TCP. NASA Technical Report, 2000.

[dham 05] A. Dhamdere et al. Buffer Sizing for Congested Internet Links. *Proceedings of IEEE INFOCOM 2005, vol. 2,* Mar. 2005

[diot 02] C. Diot et al. Measurement and Analysis of Single-Hop Delay on an IP Backbone Network. *IEEE Journal on Selected Areas in Communications.* 2002

[duki 06] N. Dukkipati, N. McKeown. Why Flow Completion Time is the right metric for Congestion Control. *ACM SIGCOMM Computer Communication Review*, Jan 2006

[enac 05] M. Enachescu et al. Part III: Routers with Very Small Buffers. *ACM SIGCOMM Computer Science Review,* vol. 35, no. 2, July 2005.

[fall 96] K. Fall, S. Floyd. Simulation-Based Comparisons of Tahoe, Reno and SACK TCP. *ACM SIGCOMM Computer Communication Review*, Jul. 1996

[gavaletz 10] E. Gavaletz, J. Jaur. Decomposing RTT-Unfairness in Transport Protocols. *17th IEEE Workshop on Local and Metropolitan Area Networks (LANMAN),* May 2010

[goh 07] K. Goh & A. Barabasi. Burstiness and Memory in Complex Systems. *A lLetters Journal Exploringt he Frontiers of Physics*. Vol 81 no. 4. Dec. 2007

[gor 05] S. Gorinsky et al., Link Buffer Sizing: A New Look at the Old Problem. *Proceedings IEEE Symposium on Computers and Communications (ISCC)*, 2005.

[ha 06] S. Ha et al., A Step Towards Realistic Evaluation of High-Speed TCP Protocols. *Web-based Technical Report, NCSU* 2006

[ha 08] S. Ha, I. Rhee and L. Xu. CUBIC: A New TCP-Friendly High-Speed TCP Variant. *ACM SIGOPS Operating Systems Review*. Volume 42 Issue 5. Jul. 2008

[has 09] S. Hassayoun, D. Ros. Loss Synchronization and Router Buffer Sizing with High-Speed Versions of TCP: Adding RED to the Mix. *IEEE Infocom Workshops.* Apr. 2008.

[has 09] S. Hassayoun, D. Ros. Loss Synchronization, Router Buffer Sizing and High-Speed Versions of TCP: Adding RED to the Mix. *IEEE 34th Conference on Local Computer Networks, 2009.* Oct 2009.

[hc 06] F. Hernandez-Campos. Generation and Validation of Empirically-Derived TCP Application Workloads. PhD thesis, University of North Carolina at Chapel Hill, August 2006

[hoe 95] J. Hoe. Improving the Startup Behavior of a Congestion Control Scheme for TCP. *ACM SIGCOMM Computer Communications Review, Oct. 1996*

[jacobson 88] V. Jacobson. Congestion Avoidance and Control. *ACM/SIGCOMM 1988 Computer Communication Review.*

[jacobson 90] V. Jacobson. Modified TCP Congestion Avoidance Algorithm. Message to end2end-interest mailing list, April 1990.

[jef 05] K. Jeffay, Le, L., D. Smith. Sizing Router Buffers for Application Performance. *Technical report. UNCCS-TR05-111. Department of Computer Science, University of North Carolina*. 2005

[jin 04] C. Jin, D. X. Wei, S. H. Low. FAST TCP: Motivation, Architecture, Algorithms and Performance. *IEEE/ACM Transactions on Networking,* 2007.

[kaur 09] J. Kaur, V. Konda. "RAPID: Shrinking the Congestion Control Timescale". *Proceedings of IEEE INFOCOM*, Rio de Janeiro, Brazil. Apr. 2009

[kel 03] T. Kelly. Scalable TCP. *ACM SIGCOMM Computer Communication Review*. Volume 33 Issue 2, Apr. 2003

[lakshman 97] T. Lakshman, U. Madhow. "The Performance of TCP/IP for Networks with High Bandwidth-Delay products and Random Loss". *IEEE/ACM Transactions on Networking*. Volume 5, no. 3, Jul. 1997

[leith 04] D. Leith, R. N. Shorten. H-TCP: TCP for High-Speed and Long-Distance Networks. *Proc. 2nd Workshop on Protocols for Fast Long Distance Networks. Argonne, Canada.* Feb. 2004

[leith 07] D. Leith, Y. Li, R. N. Shorten. Experimental Evaluation of TCP Protocols for High-Speed Networks. *IEEE/ACM Transactions on Networking*, Oct. 2007

[leith 08] D. Leith et al. Experimental Evaluation of delay/loss-based TCP congestion control algorithms. *Proceedings of the 6th International Workshop on Protocols for Fast Long-Distance Networks* (PFLDnet 2008) , 5-7 March, 2008 [liu 09] Liu, T. Basar, S. Srikant. TCP-Illinois: A Loss and Delay-Based Congestion Control Algorithm for High-Speed Networks. *Proceedings of the 1st International Conference on Performance Evaluation Methodolgies and Tools,* no. 55. 2006.

[log 08] D. Loguinov, Y. Zhang. ABS: Adaptive Buffer Sizing for Heterogeneous Networks. *IWQoS 2008: 16th International Workshop on Quality of Service,* June 2008.

[mckeo 06] N. McKeown, Y. Ganjali. Update on Buffer Sizing in Internet Routers. *ACM SIGCOMM Computer Communication Review, vol. 36, no. 5,* Oct. 2006

[rai 05] G. Raina, D. Wischik. Buffer Sizes for Large Multiplexers: TCP Queuing Theory and Instability Analysis. *Next Generation Internet Networks 2005*, April 2005.

[shak 05] S. Shakkottai et al. A Study of Burstiness in TCP Flows. *Proceedings of the 6th international conference on Passive and Active Network Measurement, pages 13-26*. 2005.

[snep 95] J.L.A. van de Snepscheut. The Sliding Window Protocol Revisited. Formal Aspects of Computing 1995, Volume 7, Issue 1, pp 3-17

[stan 06] R. Stanojevic. et al. Adaptive Tuning of Drop-tail Buffers for Reducing Queuing Delays. *IEEE Communication Letters, vol. 10, no.7,* 2006

[stenning 76] N. V. Stenning. A Data Transfer Protocol. *Computer Networks 1 (2): 99 – 110*. 1976

[tan 06] K. Tan et al. A Compound TCP Approach for High-Speed and Long-Distance Networks. *Proceedings, INFOCOM 2006 IEEE Conference on Computer Communications*. Apr. 2006

[tan 07] L. Tan. On Parameter Tuning for Fast TCP. *IEEE Communication Letters*. 2007

[tow 06] D. Towsley. Study of Buffer Size in Internet Routers. *Final Report prepared for Army Research Lab and DARPA*. Defense Technical Information Center, Aug. 2006. http://www.dtic.mil/docs/citations/ADA457457

[xan 10] C. Xanthopoulos, C. YFoulis. A Robust Dynamic Solution of the Router Buffer Sizing Problem. *Proceedings of the 2010 14th Panhellenic Conference on Informatics,* 2010.