APPLYING TECHNIQUES OF APPLICATION LAYER STRIPING TO IMPROVE THE UTILIZATION OF A SINGLE TCP CONNECTION

Michael Damein Elder

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

Chapel Hill
2009

Approved by:

Dr. Ketan Mayer-Patel

Dr. Kevin Jeffay

Dr. Don Smith

ABSTRACT

Michael Damien Elder: Applying Techniques of Application Layer Striping to Improve the Utilization of a Single TCP Connection
(Under the direction of Dr. Ketan Mayer-Patel)

The Transport Control Protocol (TCP) provides robust consumption of network bandwidth when multiple connections are active across a network link with a high-bandwidth delay product. Yet individual TCP connections can fall short of consuming available bandwidth. Application layer striping allows a single application to open multiple connections to one or more endpoints and parcel or reconstitute data across each link. The benefits of applying this technique to the network transport layer would allow more efficient consumption of network resources. The throughput characteristics could also demonstrate more normalized adjustments based on network events, providing a smooth transmission stream for time-sensitive information like streaming media applications. We examine a design which applies this technique to implement a composite stream with a configurable number of virtual streams with a sender-side only modification. We provide a background survey of existing approaches to this problem, and analysis of our approach.

DEDICATION

To my mother for her unyielding support and encouragement even in challenging times.

To Wendy for her compassion and patience in all things.

ACKNOWLEDGEMENTS

I would like to thank Dr. Ketan Mayer-Patel for his guidance and patience working through the various stages of the proposed design.

I would also like to thank Dr. Kevin Jeffay and Dr. Don Smith for their comments and feedback on the final work.

TABLE OF CONTENTS

LIST OF FIGURES

I. INTRODUCTION

The Transport Control Protocol (TCP) has demonstrated its robustness in a variety of network environments with many active TCP steams. Yet previous studies have shown [0, 12] that individual TCP connections can experience difficulties in high bandwidth-delay product networks, where the capacity of the pipe exceeds the bounds of its congestion control algorithm (as in RFC 2581 [1]). We consider a novel approach to improving the utilization of network bandwidth with a sender side only modification of the TCP protocol. Our proposal takes advantage of the behavior of multiple TCP connections to effectively utilize network bandwidth by simulating multiple TCP connections in a single sender. Our approach could have the advantage of being able to scale up or down based on the particular needs of an application and demonstrates the averaging affect of having multiple virtual congestion control windows.

TCP is challenged to adapt to high bandwidth delay environments largely due to its incremental behavior used to scale the congestion window over a period of time dependent on the round trip time (RTT). As [8] indicates, over 83,333 / 2 RTTs would be necessary for TCP to increase its available window from half utilization to full utilization of 10 Gbps with 1500-byte packets. When further coupled with a 100 ms RTT, the amount of time necessary to complete this increase requires about an hour. The necessary loss rate to sustain such an increase is beyond the theoritcal limit of the network's bit error rates [8].

The motivation for our approach comes from observations about the effectiveness of many TCP sources when utilizing a high-bandwidth delay product network link. Software products like Download Director from IBM [0] or Download Accelerator from SpeedBit [3] already take advantage of opening

multiple connections to a single source in order to improve their ability to quickly transfer data across a network. Often, this technique is referred to as application-level striping. The fact that multiple connections can improve the transfer time of application data indicates that bandwidth is not being effectively utilized. Note that for the purposes of our approach, we are specifically concerned with single source transfers, and not optimizing the transfer of data from multiple sources implemented by applications like BitTorrent [4] or StreamSpider [5].

While application-level striping can be an effective way to scale up network consumption, we submit that its prevalence demonstrates room for TCP to be improved in its ability to utilize available network consumption. In this paper, we examine how the advantages of application layer striping might be applied at the protocol level, to effectively virtualize multiple TCP sources into one composite stream.

To better understand the key idea of our approach, consider what would happen if we treat the calculation of the congestion control window as the sum of various creditors; each creditor extending a credit line to the composite window. Each creditor determines how much credit it is willing to extend the composite stream based on the positive and negative events in a sort of virtual credit history. A positive event indicates that a packet was delivered, and a negative event indicates that a packet was lost. Hence, the overall stream represents that quality of its own virtualized economy. Many positive events will drive the total available credit higher, while negative events can impact the individual components of the composite stream. The more creditors are available to the composite window, the more easily the window can grow, while also being able to stand more loss events before taking a severe decline.

We will now consider a survey of existing approaches and provide a categorization to frame the discussion and highlighting the novel aspects of each of them. One of the advantages to our virtualization approach is that any of these algorithms could be implemented as the specific congestion control algorithm dictating an available credit line. Hence, our approach can be coupled with others to specialize the characteristics of the congestion control window.

We will then follow with a description of the design and operation of our approach and explain our experimental setup. We will conclude with the analysis of our experiments and a discussion of the lessons learned while developing the final design.

II. HYPOTHESIS

When beginning our analysis, we anticipated the following hypotheses:

1. Simulating a number of virtual connection senders should demonstrate the ability to scale network bandwidth consumption proportional to the number of virtual connections.
2. The observed throughput should demonstrate an averaging behavior, avoiding dramatic peaks or valleys experienced by an individual TCP stream.

Ultimately, we were unable to demonstrate a design which supports these hypotheses, but we did encounter behavior which could prove promising with future research.

III. BACKGROUND OF EXISTING TECHNIQUES

The TCP protocol determines congestion window by responding to acknowledgement packets from the receiver. The congestion window, often denoted as *cwnd*, is the number of packets that can be awaiting acknowledgement at any given time. The receiver will also advertise an upper bounded window known as the receiver window, or *rwnd*, which determines the maximum number of amount of data that can be in flight to avoid overloading the receiver. The actual number of packets in flight is the minimum of *cwnd* and *rwnd*. For the purposes of our discussion, we will primarily be focused on *cwnd*.

The value of *cwnd* is determined through a process known as additive increase multiplicative decrease or AIMD. As packets are successfully acknowledged, *cwnd* is increased linearly by a rate often denoted as α. Conversely, if loss is detected, *cwnd* is adjusted downward by the inverse ratio of β. For standard TCP, $\alpha = 1$ and $\beta = 2$.

The AIMD adapts to its environment by first expanding its window, and then falling back to a previously known size if the most recent events warrant it. Alternative techniques to TCP follow the same kind of probe, retreat pattern when calculating an optimal window size. We will now consider a few of these suggestions from the literature to provide a contrast and comparison against our proposal. Each of these describes one possible way to address the slow convergence of an individual TCP sender to a congestion control window size of optimal size, particularly in high-bandwidth delay product links.

For the purposes of this discussion, consider the following categorization suggestion:

- Type I: Minor modifications to the basic TCP congestion control algorithm. Modifications of this nature will tend to adjust the α and β coefficients that decide the increase and decrease of the window size. [6, 7]
- Type II: Clever approaches to compute the window size based on recent observations. Recent observations can include the last observed RTT or the size of the current window relative to the size of the epoch since the last loss event, among others. [8, 9, 10]
- Type III: Application level modifications that use TCP (or other protocols) to implement better utilization than a single connection can consume [2, 3, 4, 5, 11]

The Scalable TCP [6] algorithm is specifically designed to more aggressively increase its congestion window, and respond with greater resilience to loss events. For each successfully acknowledged packet, *cwnd* is increased by 0.01:

$$cwnd := cwnd + 0.01$$

Conversely, for each detected congestion, cwnd is decreased by $1/8^{th}$:

$$cwnd := cwnd - [cwnd * 0.125]$$

The approach taken by Scalable TCP allows it to recover from a loss event in time proportional only to the round trip time, claiming a recovery time proportional to $-log(1-b)/log(1+a)$ rather than $cwnd/2$ as in standard TCP. As described in the respective paper, traditional TCP windows are limited for high bandwidth delay links because of the long convergence times and low drop rates necessary to establish large values for *cwnd*. Scalable TCP reasons about a legacy window size, denoted *lwnd*, and a legacy loss rate, p_l to decide the values of α and β above. Whenever the congestion window is beneath the value of

lwnd, traditional TCP congestion control is employed. When the congestion window exceeds *lwnd*, the more responsive Scalable TCP protocol takes control.

The combination of approaches allows Scalable TCP to be reasonably fair to traditional TCP streams, since a minimum threshold must be reached before a different scheme is employed. It is further reasoned that the threshold is set at a point beyond the theoretical ability of TCP to stabilize for high bandwidth delay product links.

Therefore, we will classify Scalable TCP as a Type I modification. While it can demonstrate better responsiveness for high-bandwidth delay links, its primary modification is an adjustment of the α and β parameters.

The H-TCP [7] algorithm focuses specifically on the rate that new packets are injected into the network (α) and adjusts the rate based on recent observations about the time since the last packet was dropped. Much like Scalable TCP, a two-phase adaptation is employed. In the first phase, a fixed increase, α^L, consistent with traditional TCP streams is used. In the second phase, the ratio, α^H is computed based on the time elapsed since the last dropped packet was detected:

$$\alpha^H(\Delta_i) = 1 + 10(\Delta_i - \Delta^L) + ((\Delta_i - \Delta^L) / 2)^2$$

We note that an interesting approach to the adjustment is that H-TCP is based on real time, rather than on RTT. The result is that it is less likely to be RTT unfair, by scaling more aggressively due to relatively shorter RTTs than its competing flows [8, 9].

While not strictly AIMD, the approach does generate a linear increase for consistent values of RTT.

In addition, the reduction defined by H-TCP is also adaptive in terms of RTT, specifically in the minimum and maximum observed RTT:

$$\beta_i = RTT_{min,\,i} / RTT_{max,\,i}$$

As with the adaptive α^H, the ratio defined above is only valid within a certain threshold band. Outside of the threshold band, β is set to a constant β_{reset} factor.

Therefore, we classify H-TCP as a Type I modification.

More so than protocols discussed thus far, the Binary Increase Congestion Control (BIC) [8] protocol attempts to ensure TCP friendliness and RTT fairness in addition to scalability. Like previously discussed approaches, BIC uses a two-phase adaptation to establish the best congestion window. Unlike prior approaches, BIC uses a more responsive probing function than additive increase for the first phase. In effect, the BIC protocol executes a binary search increase for the first phase, and an additive increase for the second phase. The approach allows BIC to quickly scale up in environments with lots of available bandwidth, but also provide TCP friendliness when other streams are sharing the same connection.

When operating under a *binary search increase* phase, the algorithm uses two sentinel values for the size of the window. The *maximum window size*, denoted W_{max}, generally captures the size of the window before the last loss event. The *minimum window size,* denoted W_{min}, represents the size of the window after the last fast recovery. When in *binary search increase* mode, the protocol sets the size of the congestion window to the midpoint of W_{max} and W_{min}. If packet loss occurs, the midpoint becomes the new W_{max}. The protocol continues in this manner until the difference between W_{max} and W_{min} falls below a threshold known as the *minimum increment*, denoted S_{min}.

Alternatively, if the difference beween W_{max} and W_{min} is beyond a *maximum increment*, denoted S_{max}, the authors reason that increasing directly to the midpoint could be overaggressive and change their

responsive behavior. Instead of adjusting directly to the midpoint, the protocol increments only by S_{max}. Once the difference between the current window and the target fall below S_{max}, the protocol increases directly to the target. When operating under this mode, the protocol is said to be in the *additive increase* phase.

The protocol also provides slow start behavior when the current value of W_{max} is exceeded by probing for a new value in increments of S_{min} until it reaches the value of $W_{max} + S_{max}$, at which point the protocol begins to aggressively probe in increments of S_{max}.

Together, the *binary search increase* and *additive increase* are referred to as *binary increase*, hence the name of the protocol *Binary Increase Congestion Control*.

Another key aspect of BIC is that like previously discussed protocols, it operates under standard TCP up until a certain threshold, after which BIC engages. However, since BIC attempts to adjust the congestion window beyond a fixed linear behavior of AIMD, we classify it as a Type II modification.

The CUBIC [9] algorithm was an approach to expand on the lessons learned from BIC. The authors' goal was to maintain the same favorable properties of BIC while simplifying the behavior of its congestion control function. Beyond making it easier to understand, a more straightforward function would be easier to study than BIC, which has very adaptive behavior based on the current network conditions.

CUBIC defines a congestion window function as a function of t, the time elapsed from the last loss event. Let C denote a constant scaling factor, K denote $\sqrt[3]{(W_{max}\beta/C)}$, and W_{max} denote the *maximum window size*. Then we may compute W_{cubic} as follows:

$$W_{cubic} = C(t-K)^3 + W_{max}$$

As the authors note, the function results in rapid growth when the congestion window is much less than W_{max}, but stabilizes as the window approaches W_{max}. When W_{max} is reached and subsequently

exceeded, the protocol again begins probing for a new value of W_{max}. The authors note that the choice of using the elapsed time since the last window reduction is inspired by the approach of H-TCP, which also exploits this technique. The result is that CUBIC is not RTT unfair with other protocols.

Like its forerunner, BIC, CUBIC also keeps the behavior of TCP in mind for early phases of the protocol. In effect, the window size of TCP is calculated based on the ratio of elapsed time to RTT:

$$W_{tcp} = W_{max}\beta + 3\,[(1 - \beta)/(1+\beta)]\,[t/RTT]$$

As CUBIC extends BIC, we categorize it as a Type II modification.

Other approaches to improve communication across high bandwidth-delay product links make improvements at the application layer, above TCP. Application layer striping is one such approach where multiple connections are managed at the application network layer, and data is chunked up among the various connections. In many cases, the application must be aware of the parceling of data across each connection, and must either break it up for transmission or reconstruct it when received. Previous work in this area [11] demonstrates this approach to be effective at increasing the consumption of available bandwidth in high bandwidth delay product networks.

Alternatively, the data could be compressed for transmission, in affect making better use of the available network bandwidth to transfer more data.

Each of these is an example of a Type III modification. While any program can be implemented using these techniques, the improvements are limited to the specific application, rather than an improvement which benefits all applications using a modified network stack as in Type I or II modifications.

Advantages of Existing Approaches

Existing approaches often modify the TCP protocol in such a way that existing network layers do not need to be aware of the changes. In addition, most existing approaches also consider the fairness of each individual connection when compared to its peers or TCP. These include concerns such as scalability, RTT fairness, TCP friendliness, and fairness and convergence [8]. Designing TCP friendly protocols enables them to be deployed in heterogeneous network environments without impacting standard TCP flows.

In our approach, we do not examine fairness properties as our goal is specifically the scalability of our design. We only expect to be as fair when considered against the characteristics of a comparable number of TCP connections.

Limitations of Existing Approaches

As demonstrated by the modest comparison by Ha et al [12] under real network simulations, these protocols may or may not accomplish their projected behavior. Often, the same properties that make it difficult for TCP connections across high bandwidth-delay links are still a challenge for models of congestion windows regardless of the techniques used.

In particular, some protocols are based on increments of RTT time to modify their window size [6, 8], while others use the real time of the epoch since the last loss event to adjust their window more drastically during times of high loss [7, 9].

Our approach seeks to replicate the advantage of easy deployment to existing networks, without requiring a significant redesign to existing TCP protocols. However, future research could examine the application of any existing TCP protocol with our proposed design.

IV. PROPOSED DESIGN

Concept

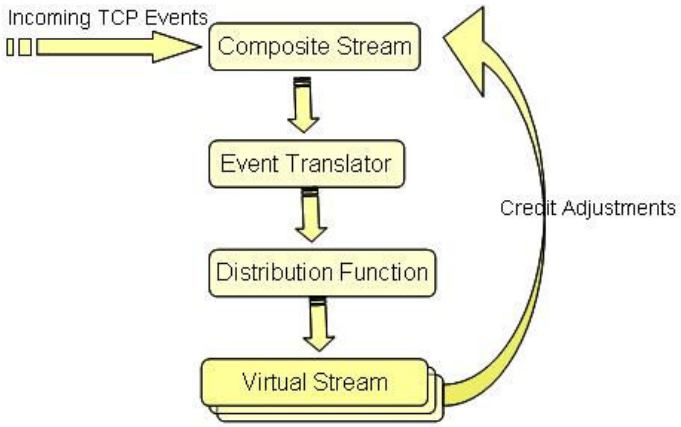
Our approach examines the strategy of simulating multiple virtual TCP connections within a single sender to provide better utilization of network resources, much like network connection striping. The result affords the sender a congestion window proportionate to the number of virtual connections, which can either be defined statically or adjust dynamically with minor modifications of our approach. Each virtual connection acts like a creditor to the composite window, providing more credit (congestion window) or less based on the observed successful or failed deliveries of packets.

The decision to extend or reduce credit derives from the observed event stream provided to the virtual streams. It is this event translation and distribution process that provides the heart of the virtualization. Raw TCP events (in particular, NEWACK, DUPACK, TIMEOUT) are received by the composite stream, converted to a virtual event (SUCCESS, LOSS, SEVERE_LOSS) and provided to a distribution function. The distribution function, which is another configurable point of the approach, decides how to disseminate events among the various virtual streams.

We will now describe the control flow at a conceptual level and then explain each of the important mechanisms behind the algorithm. In addition to new state which is maintained and specific to our algorithm, we refer to various bits of state already maintained by the standard TCP protocol, which is captured in the implementation of the simulation platform we used for our experimental setup. The experimental setup used a fixed *segment size* for the transmission of data, which decided how much data

should be transmitted for each packet. There were several important points in the buffer stream were remembered by the base implementation. The *next transmission sequence*, is the sequence number of the next segment in the sender buffer to be transmitted. The *highest received acknowledgment* is the sequence number of the highest acknowledgment that has been received by the sender. There are many other bits of state maintained by TCP, but the others are not relevant to the course of this discussion.

To better understand the intended control flow, consider the following diagram. Incoming TCP events are received by the composite stream. The composite stream translates each raw event into a virtual event according to a defined workflow, to be described shortly. Once translated, the distribution function selects a virtual stream, and distributes the event to the selected virtual stream. The virtual stream computes its new available congestion control window and computes a delta. If the delta is positive, the credit limit for the composite stream is extended. If the delta is negative, the credit limit is reduced. As more credit is extended to the composite stream, more packets may be transmitted to the receiver. As credit is reduced, the available window shrinks, throttling back transmissions from the composite stream.



Sender Operating Mode

For the sake of this discussion, we refer to data which had not been previously been transmitted as *new data*, and we refer to data which had been previously transmitted but for which no acknowledgment had been received as *retransmitted.*

We can consider the composite stream to be sending a stream of *new* and *retransmitted* data. The virtual streams make the determination for the amount of data allowed to be transmitted. With a standard TCP sender, the sender implicitly knows what state it is in, receiving either DUPACKs that require retransmission after a certain count or NEWACKs. Some variants formalize such state, as in TCP Reno where a Boolean flag indicates whether the sender is in *fast recovery* mode. We found that when responding to events from the raw TCP event stream, we needed a way to manage our behavior to understand whether the event was positive or negative.

In our case, we were specifically concerned about whether the stream of transmitted data should be *new* or *retransmitted*. We therefore introduced a notion of *sender operating mode* and denoted the two states as *transmission mode*, denoted *TX_MODE*, and *retransmission mode*, denoted *RTX_MODE*. With the *sender operating mode* modification, we could be aware of whether we expected to send fresh data or possibly need to patch holes in the composite stream.

Event Translation

Initially, the state of the composite stream accumulates the sum of the initial window sizes of all known virtual streams. The virtual streams then receive virtual events from the distribution function, which is driven from observations of the raw TCP event stream.

Let us take a closer look at the event translation process after the initial state. We will motivate the discussion by describing the response for observed NEWACK, DUPACK, and TIMEOUT events from the raw event stream. We will describe the response and the translation to the appropriate *virtual success* or

virtual loss event for each of these under *transmission* and *retransmission mode*. In addition, within *retransmission mode*, we will describe a particular realization about the design of the algorithm that motivated the introduction of a new variable, *warp sequence*. We will explain how the *warp sequence* is used to improve the recovery of the composite stream to loss events. Finally, we will describe a critical modification to address cases of packets which are retransmitted and subsequently lost.

When a NEWACK is received under *transmission mode*, we know that data was successfully delivered, and therefore can translate the NEWACK into a virtual success event. When we generate a success event, the distribution function will choose which virtual stream will receive the event. The virtual stream will in turn increase the credit limit of the composite stream, allowing more data to be sent. We can then *send pending data* from the composite stream, transmitting *new data* to the receiver.

When a DUPACK is received under *transmission mode,* we operate much like standard TCP. Once three DUPACKs have been received, we retransmit one *segment size* worth of data beginning at the *highest received acknowledgment*. We further interpret it as a loss and generate a virtual loss event to the distribution function. In turn, the distribution function chooses which of the virtual streams will take the loss event. The virtual stream then reduces the available credit to the composite stream.

Under *transmission mode*, the loss event after three DUPACKs is effectively the first loss detected, but others may exist. After the initial loss is detected, the *sender operating mode* transitions to *retransmission mode*. The composite stream also remembers the sequence of the next new data packet to be transmitted as a *warp sequence*.

The *warp sequence* is a special characteristic of our approach that evolves the notion of fast recovery. We realized that we could pre-empt the interpretation of loss events in *retransmission mode* by examining the value of the incoming sequence in NEWACK in relation to the *warp sequence*. When a NEWACK is received with a sequence number less than the *warp sequence*, we know that a loss has occurred. We anticipate that the retransmission of the original lost packet should generate an

acknowledgement for *warp sequence*. If an acknowledgement is received for a packet below the *warp sequence*, we distribute a virtual loss event to the distribution function. The missed packet (*highest received acknowledgement*) is retransmitted.

Until the retransmitted packet is received and acknowledged by the receiver, the composite stream will continue to receive an amount of DUPACK events equal to one congestion window. For each DUPACK event received under *retransmission mode*, we know that a packet was successfully delivered. Therefore, for every DUPACK under *retransmission mode,* we generate a virtual success event to the distribution function, and send more pending data with the extended credit line.

Once the original lost packet is received, the sender should receive an acknowledgement indicating the next packet expected. We receive that packet as a NEWACK under *retransmission mode*. We first check to see if the new packet is less than the *warp sequence*. Remember that the *warp sequence* is the value of the *next transmission sequence* before we retransmitted the original lost packet. We have been sending new data for each DUPACK after the retransmission, and now we have received a new acknowledgement. We know that if the new packet is less than the *warp sequence*, a subsequent loss occurred. We therefore retransmit the packet and generate a virtual loss event to the distribution function, which chooses a virtual stream to take a loss and reduce our credit limit. Thus, an otherwise positive event is interpreted with an additional reversed polarity under *retransmission mode.*

However, since we did receive an acknowledgement, we know that a packet was successfully delivered and also generate a *virtual success event* to the distribution function – even if it was not the one we would have liked. If we have available window to do so, we also send pending data.

We repeat the behavior until the NEWACK exceeds the *warp sequence*, at which point we transition the *sender operating mode* to *transmission mode*. We also restore the *warp sequence* to the sentinel value of 0.

In addition to NEWACK and DUPACK events, the composite stream can also TIMEOUT, where no acknowledgement is received and all transmissions effectively cease. A TIMEOUT is more problematic than packet loss, and therefore we generate *virtual severe loss events* for each TIMEOUT. Like other *virtual events*, the distribution function chooses a virtual stream to take the event. The *virtual stream* will effectively revert its congestion window to one segment size, the initial state. The reduction is taken against the credit limit of the composite stream like any other event.

Interpolating subsequent loss events under *retransmission mode* proved problematic in the initial algorithm. Our experimental results showed that if a packet was retransmitted and lost a second time, our algorithm did not detect the second loss event. We had anticipated a TIMEOUT event would catch this case, but determined that since we were continuously sending new data and receiving acknowledgements (albeit DUPACKs), the stream never timed out, and thus a TIMEOUT was never generated. In effect, the TIMEOUT only indicates situations where all behavior ceases, and we must begin transmission a new, both in terms of adjusted congestion window size and transmission sequence.

We addressed the issue by introducing a new custom timer which runs concurrent with the composite stream timeout (TCP connection timeout). Whenever we retransmit a packet, we set the custom timer using the currently estimated RTT. The RTT indicates the amount of time expected for a packet to travel to the receiver and an acknowledgement to be generated and received by the sender. Therefore, if no acknowledgement for the retransmitted packet occurs in this timeframe, we interpret it as a loss, and retransmit the packet again, generate a severe loss event to the distribution function, and reset the timer. We saw that in some cases, it was possible for a packet to be lost, re-lost, and possibly (but rarely in our simulations), lost for a third time. However, by continually adjusting the timer for each retransmission, we could ensure that no matter how many times were necessary, we could continually attempt delivery of the troublesome packet.

We can summarize the algorithm with the following table which describes an action matrix of the current *sender operating mode* and the interpreted raw event to determine the specific steps to be taken.

	NEWACK<WARP	NEWACK (>=WARP)	DUPACK	TIMEOUT
X	INVALID STATE	CancelRxTimerIfSet() ProcessSuccess() CommonNewAck() -SendPendingData()	c < 3: No-op c == 3: ProcessLoss() warp = nextTx Go to RTX	ProcessSevereLoss() Go to RTX
TX	CancelRxTimerIfSet() ProcessSuccess() ProcessLoss(first) CommonNewAck() -SendPendingData()	CancelRxTimerIfSet() warpSeq = 0; Go to TX	ProcessSuccess() SendPendingData()	ProcessSevereLoss()

Figure 1: Algorithm matrix.

Psuedocode for each composed step is given below.

```
ProcessSuccess()
      distributionFunction.processSuccessfulDelivery()

ProcessLoss(first)
                  Retransmit()
      SetTimer(RETRANSMISSION)
      if (first)
        distributionFunction.processLoss()

ProcessSevereLoss()
                  Retransmit()
      distributionFunction.processSevereLoss()

SendPendingData()
      SetTimer(TIMEOUT)
                  SendPacket(nextTxSeq)
      nextTxSeq += segSize

CommonNewAck(seq)
      highestAckReceived = seq;
      SendPendingData()

Retransmit()
      SetTimer(TIMEOUT)
      SendPacket(highestRxAck)
```

Figure 2: Psuedocode for CompositeStream

Distribution Function

After a raw TCP event is intercepted and translated, it is then injected into a *distribution function*. The *distribution function* chooses which virtual stream will receive the event. The *distribution function* may also choose to adjust its behavior based on previously delivered events.

Our design enables the distribution function to be a point of variation; any algorithm can be used to distribute events, based on the type of behavior being virtualized. In our experimental setup, we focused on a *weighted lottery distribution function.* The weighted lottery distribution function assigns tickets to each virtual stream for each selection event. Streams with more tickets are more likely to be chosen, and streams with fewer tickets are less likely. The approach favors virtual streams which have been least recently chosen. When an event is received by the weighted lottery distribution function, a virtual stream is

selected by selecting a random number between the range of the sum of the total number of tickets available to all streams, and the virtual stream "holding" that ticket wins the lottery, and receives the virtual event.

While we did not focus on the distribution function during our experimentation, the design enables the composite stream to implement the behavior best suited to the particular network traffic environment. For instance, the distribution function might choose to discriminate and deliver a higher proportion of losses to a minority of virtual streams while delivering a higher proportion of successes to another set of streams. We would expect that in this given scenario, the congestion control window of the composite stream would tend to be more resilient to loss events, as the virtual streams paying the cost would never be allowed to contribute to a sizeable portion of the overall composite stream.

Virtual Streams

As a virtual stream receives a virtual event, it computes the change to its local congestion control window and then announces the amount as an adjustment in the available credit limit of the aggregate stream. The virtual stream could be modified to examine different emergent behaviors. Any TCP-variant (Tahoe, Reno) or one of the previously discussed modifications can be implemented at the virtual stream level to control its individual congestion control window.

The number and characteristics of the virtual streams may be defined statically or perhaps driven from observed round trip times, scaling up with more virtual streams if there is reason to believe the network stream is experiencing a high bandwidth-delay situation. We chose to evaluate our protocol using a static number of streams, proportional to the expected background traffic in order to study the scalability properties of the algorithm.

For our initial experimentation, we kept the implementation as consistent as possible by using TCP Reno. The function of a virtual stream is defined by three signatures made available to call by the

distribution function. We will now describe each of these, and how each method adjusts the internal congestion window maintained by the virtual stream.

In ProcessSuccessfulDelivery(), the virtual stream receives a count of how much data has been acknowledged and a Boolean indicator to indicate whether the composite stream is operating under a *retransmission mode*. Two cases are available to adjust the available credit extended to the composite stream. If the *retransmission mode* flag is set or the congestion window is below the slow start threshold, the window is increased by one segment size, as would be done under TCP Reno. Otherwise, a fraction of a segment size is extended proportional to 1 / window size. The difference between the prior congestion window and the new congestion window is extended to the composite stream.

In ProcessLoss(), the slow start threshold is set to the larger of half the size of the congestion window or twice the segment size. The congestion window is then set to the sum of the slow start threshold and three times the segment size. The difference between the prior congestion window and the new congestion window is reduced from the composite stream.

In ProcessSevereLoss(), the slow start threshold is set as in ProcessLoss(), but the congestion window is set to a single segment size. As in ProcessLoss(), the difference between the former and new congestion window is reduced from the composite stream.

We chose TCP Reno because of its well known behavior and properties, but any congestion control algorithm could be applied.

Implementation

We chose to implement our approach on the GTNets simulation platform. [GTNets]. GTNets provides a very robust way to experiment with network configurations using C++ implementations of various network concepts. We used the platform to implement our approach, and examine its behavior using a varying number of virtual streams.

The GTNets implementation supports a common base type for TCP, which defines many standard behaviors and configurable points, such as NewAck(Seq), DupAck(TCPHeader, Count_t), and ReTxTimeout(). We override these methods to implement the behavior described in the previous section.

MultiplexedTCP also adds methods for extending and reducing the credit line, and we expose those methods through a common class, CreditReceiver. We extend CreditReceiver and provide this interface to the DistributionFunction and VirtualConnection.

The base TCP class defines several types of timer events, such as CONNECTION_TIMEOUT, RETRANSMIT_PACKET, DELAYED_ACK, TIMED_WAIT, and LAST_ACK. We define a new timer event, CUSTOM, to implement our retransmission failsafe described previously.

The DistributionFunction aggregates a collection of VirtualConnections. Two implementations were used to understand the behavior characteristics of the protocol. RandomFunction chooses at randomly a connection to receive an event. WeightedLotteryFunction assigns tickets to each connection during each time period, and chooses a connection based on the number of tickets assigned, thus giving connections who have least recently received a connection an advantage.

The diagram below illustrates the class organization of the design.

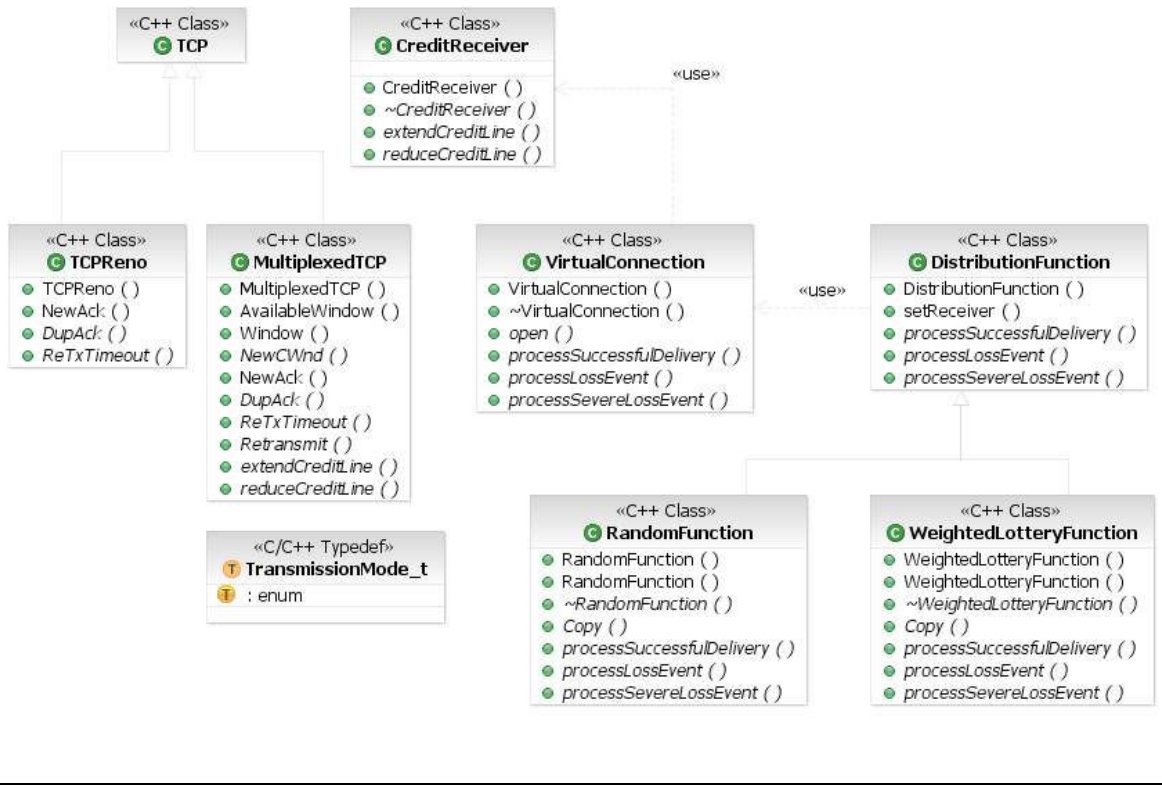


Figure 3: Class design of the proposed approach

Psuedocode the implementation of the critical methods of the MultiplexedTCP is provided below. In particular, we overrode the important methods of the TCP congestion control algorithm, and modified them according to our design. We also found it necessary to implement a custom timer, which required a very minor update to the base TCP type provided by GTNets. We also added methods according to the CreditReceiver interface, which was previously mentioned. These methods (extendCreditLine(), reduceCreditLine()) provide a callback path for VirtualConnections to adjust the available credit to the MultiplexedTCP source.

```
// TCP Overridden methods

void NewAck(Seq seq) {
  // New acknowledgement up to sequence number "seq"

  Count_t numberAck = seq - highestRxAck;
```

```
        if(seq < warpSeq) {
          switch(operatingMode) {
            case TX_MODE:
              cout << "ERROR SHOULD NOT BE IN TX_MODE AFTER WARP" <<
endl;
            break;
          case RTX_MODE:
            experimentLog("NEWACK(RTX)", seq);

            bool first = (lastRetxAck == 0);

            // cancel a timer specific to this retransmit
            if(lastRetxAckTimeout && seq > lastRetxAck) {
              CancelTimer(lastRetxAckTimeout, true);
              lastRetxAck = 0;
            }
            // process succesful event before commonnewack
            distfunc->processSuccessfulDelivery(numberAck, true);
            lastRetxAck = seq;
            highestRxAck = seq;
            ProcessLossEvent(first);

            // calls send pending data
            TCP::CommonNewAck(seq, false);

            break;
          }

        } else {
          switch(operatingMode) {
            case RTX_MODE:
              operatingMode = TX_MODE;
              warpSeq = 0;
              lastRetxAck = 0;
              if(lastRetxAckTimeout) {
                CancelTimer(lastRetxAckTimeout, true);
              }
              // fall through
            case TX_MODE:

               //      positive   event   1   connection   (prior   to
CommonNewAck())
              distfunc->processSuccessfulDelivery(numberAck, false);

              // SendPendingData() occurs from CommonNewAck()
              TCP::CommonNewAck(seq,   false);   //   Complete   newAck
processing
              break;
            }
          }
        }

      void DupAck(const TCPHeader& t, Count_t c) {
        Count_t numberAck = TCP::defaultSegSize;

        switch(operatingMode) {
```

```
        case RTX_MODE: // Indicates a Packet was received

          //  positive event 1 connection
          distfunc->processSuccessfulDelivery(numberAck, true);
          SendPendingData();

          break;
        case TX_MODE:

          if(c == 3) {
            // Go to retransmit mode
            operatingMode = RTX_MODE;
            // remember the next transmission seqno
            warpSeq = nextTxSeq;
            lastRetxAck = highestRxAck;
            ProcessLossEvent(true);
          }
          break;
        }
      }

      void ReTxTimeout() {

        switch (operatingMode) {
          case TX_MODE: // Shouldn't happen
            operatingMode = RTX_MODE;
            break;
          case RTX_MODE: // Indicates a LossEvent
            break;
        }

        nextTxSeq = highestRxAck;
        rtt->IncreaseMultiplier();  //  Double  timeout  value  for  next
retx timer
        distfunc->processSevereLossEvent();
        Retransmit();

      }

      // Custom Protocol design specific methods
      void  ProcessLossEvent(bool first) {
        Retransmit();
        if(first) {
          distfunc->processLossEvent();
        }
        ScheduleTimer(TCPEvent::CUSTOM,   lastRetxAckTimeout,      rtt-
>RetransmitTimeout());
      }

      void Timeout(TimerEvent* te) {
          switch(te->event) {
            case TCPEvent::CUSTOM:
              if(lastRetxAck != 0) {
                ProcessLossEvent(true);
              }
            break;
            default:
```

```
            TCP::Timeout(te);
          }
      }
      void extendCreditLine(Count_t c) {
        if ((cWnd + c) < (initialCWnd*segSize)) {
          cWnd = initialCWnd*segSize;
        } else {
          cWnd += c;   // may increase or reduce cWnd (but not below
initCwnd*segSize)
        }

        SendPendingData(true);
      }

      void reduceCreditLine(Count_t c) {
        if ((cWnd - c) < (initialCWnd * segSize)) {
          cWnd = initialCWnd * segSize;
        } else {
          cWnd -= c; // may increase or reduce cWnd (but not below
initCwnd*segSize)
        }
      }
```

Figure 4: Listing of MultiplexedTCP.cc

The VirtualConnection provided an implementation that closely resembles TCP Reno. Although, we note again that any congestion control strategy could be implemented, and the differences between new and former window sizes announced as credit extensions or reductions to the composite stream. We provide psuedocode for each of the critical methods of VirtualConnection below.

```
      void processSuccessfulDelivery(Count_t numBytes, bool
fastRecovery) {
      if(fastRecovery || newCwnd < ssThresh) {
        // Fast recovery or slow start mode, add one segSize to cWnd
          newCwnd += segSize;
      } else {
      // Congestion avoidance mode, adjust by (ackBytes*segSize) / cWnd
          // Changed by GFR to match RFC2581
          Mult_t adder = (((Mult_t) segSize * segSize) / newCwnd);
          if (adder < 1.0) {
            adder = 1.0;
          }

          newCwnd += (Count_t) adder;
        }

        Count_t availableCredit = newCwnd - cWnd_b;
        cWnd = newCwnd;
        receive->.extendCreditLine( availableCredit);
      }
```

```
void processLossEvent() {
  // dupack count is 3 at this point
  ssThresh = cWnd_b / 2;
  ssThresh = max(ssThresh, 2 * segSize);

  Count_t oldCwnd = cWnd_b;
  // Reset cWnd per rfc2581
  cWnd_b = ssThresh + 3*segSize;

  // reduce available tokens
  Count_t availableCredit = oldCwnd - cWnd_b;
  receiver ->reduceCreditLine(availableCredit);
}

void processSevereLossEvent() {
  ssThresh = cWnd / 2;
  ssThresh = max(ssThresh, 2 * segSize);

  Count_t oldCwnd = cWnd;
  // Per RFC2581, only one segment after timeout
  cWnd = segSize;

  // reduce available credit
  Count_t availableCredit = oldCwnd - cWnd;
  receiver ->reduceCreditLine(availableCredit);
}
```

Figure 5: Listing of VirtualConnection.cc

V. EXPERIMENTAL SETUP

For the purposes of evaluation, a dumbbell network was examined, consisting of one side of senders and one side of receivers. Each sender node was paired with a single receiver node and hosted either a control or experimental source. Each target node hosted a TCP Server which acknowledged the receipt of data following a standard TCP protocol. Each sender would begin sending data at a staggered started time determined by a uniform distribution and continue throughout the lifespan of the experiment. The control sources were configured with a TCP Reno sender using a 512 byte packet size.

A single experiment source was started five (5) seconds into the experiment to allow other targets a chance to begin sending data.

Each fan link was configured with 100 Mb/sec worth of bandwidth and a 10 millisecond delay. The backbone link was designed to provide a bottleneck link and given 10 Mb/sec worth of bandwidth and a 25 millisecond delay. The routers on each of the backbone end points each held a queue of 250 packets and did not implement any specialized congestion avoidance protocols.

Each experiment ran for one hundred (100) seconds in the simulation environment, which did not necessarily correspond to wall-clock time.

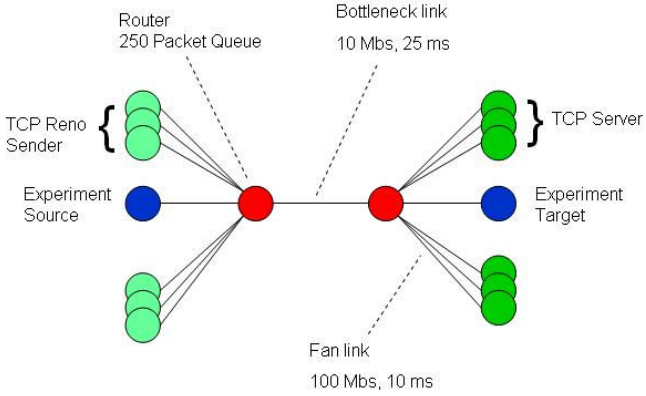


Figure 6: Network topology

To understand the scalability results of our protocol, we chose a fixed number of control sources, and varied the number of virtual streams in our experiment sources. We anticipated the consumption of bandwidth of our experiment source to be proportional to the number of virtual streams to control streams in the system.

We fixed the number of control sources at 16, and examined our experiment source at 8, 16, 32, and 64.

VI. ANALYSIS AND DISCUSSION

We began our analysis with the hypothesis that our approach would demonstrate throughput behavior which scales proportional to the number of virtual streams, and that the observed throughput would remain within a normalized range, demonstrating an averaging behavior since the total window represented the summation of many virtual streams.

In its current form, the algorithm did not support the hypotheses. Rather the observed behavior demonstrates more pronounced peaks in the throughput, and did not converge to a stable throughput level within the bounds of the experiments. We will discuss the results with 8, 16, 32, and 64 sources, and then describe earlier results that motivated certain changes to the algorithm. In particular, we did have earlier results that were promising, and demonstrated the expected scaling behavior, but discovered that they were generating incorrect results with regards to restoring lost packets in the transmission stream.

Numerical Results

Analysis on the performance of our approach was carried out on trace files which were produced from the simulation environment. Each trace file consisted of multiple lines, each line representing one or more incoming or outgoing packets of information. We focused specifically on Layer 4 transmission information, including the direction (in/out), the TCP header, and the length of the data that was transmitted. For each trace file generated from an experiment, we isolated four representative control sources and our experiment source for comparison. We also examined some metrics which accounted for an aggregation of all control sources, particularly examining their aggregate throughput. All trace files were

parsed using the AWK language to create intermediate data files, which were then utilized by a set of plots created for our analysis using gnuplot.

In particular, we will focus the analysis by considering three particular plots from the trace data produced by each experiment. The first graph is an Average Throughput Plot (ATP) which demonstrates the average throughput over a small time window (about 0.01 seconds) of our experiment source compared to the average of the sum of the throughput behavior of all control sources over time. The second graph is a Sent Sequence Plot (SSP) which represents the sequence number of transmitted packet of our experiment source compared to a representative control source control source over time. The last graph is an Acknowledgement Sequence Plot (ASP) which represents the sequence number of acknowledgement packets of our experiment source and a representative control source over time. In each case, the experiment source is denoted by a green line and the control source(s) by a red line.

In the case where 8 virtual streams were simulated, the composite stream began transmitting data with an initial window of 8 times the segment size, and since the congestion window grows with each successful acknowledgement received, the throughput began to climb as would be expected. However, once a loss was experienced, we enter retransmission mode. Once in retransmission mode, fresh acknowledgements (DUPACKs) trigger the sending of new data until we receive an acknowledgement for the lost packet.

In the Average Throughput Plot (ATP) below, we see the averaged throughput of 8 virtual streams and 16 control sources. We would have expected the 8 virtual streams to consume roughly half of the bandwidth of the 16 control sources, leading to a green line at about half the height of the red line. However, we observed something different, so let us examine why this happened.

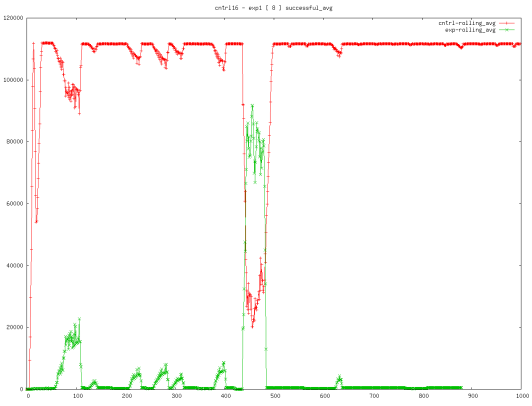


Figure 7: Average Throughput Plot (#79573) - 8 virtual streams

Consider the behavior under retransmission mode. As we receive duplicate acknowledgements, we are transmitting new data, since any acknowledgement represents a successful delivery. At the same time, we only retransmit packets when a new acknowledgement less than the warp sequence arrives. The result is that the protocol is generally sending data far beyond the received acknowledgement sequence. We can see this from the Sent Sequence Plot (SSP) of the 8-virtual stream case, we start with a very sharp arc sending data, dotted with retransmissions, until we finally catch up to the warp sequence. Within the protocol, this series of events leads to a large congestion window (driven up by many successful acknowledgements), and a large amount of unacknowledged data (since we haven't reached the warp sequence, a sizeable portion of data is unacknowledged). Generally in retransmission mode, the available window only affords one new segment for every successfully received acknowledgement. When we catch up to the warp sequence, the available window opens up, and thus the virtual TCP source floods the network. The result is intermediate loss events, which again require an unfortunate amount of time to restore.

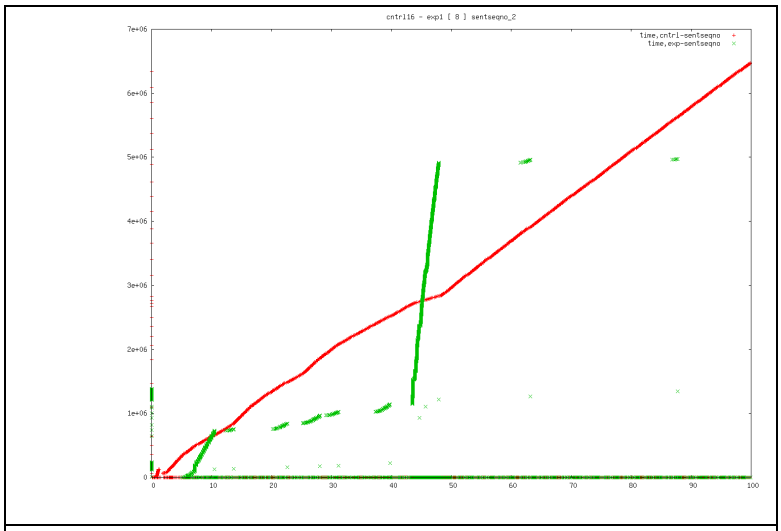


Figure 8: Sent Sequence Plot (#79573) - 8 virtual streams

When we take a look at the Acknowledgement Sequence Plot (ASP) of the same experiment, we see the results of this "flood and recover" behavior. The slope of the acknowledgement sequence progresses slowly, and jumps as we reach the warp sequence, and then returns to a very gradual slope. In the same timeframe, our representative TCP Reno source demonstrates gradual progression,

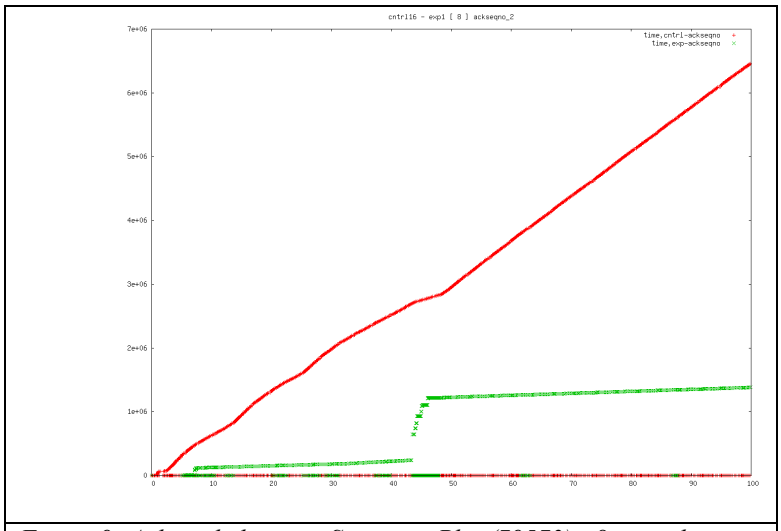


Figure 9: Acknowledgement Sequence Plot (79573) - 8 virtual streams

In the 32-virtual stream experiment, we see pronounced behavior inline with the previous results. Again, the ATP shows a gradual slope and then a dramatic spike, followed by a dramatic decline.

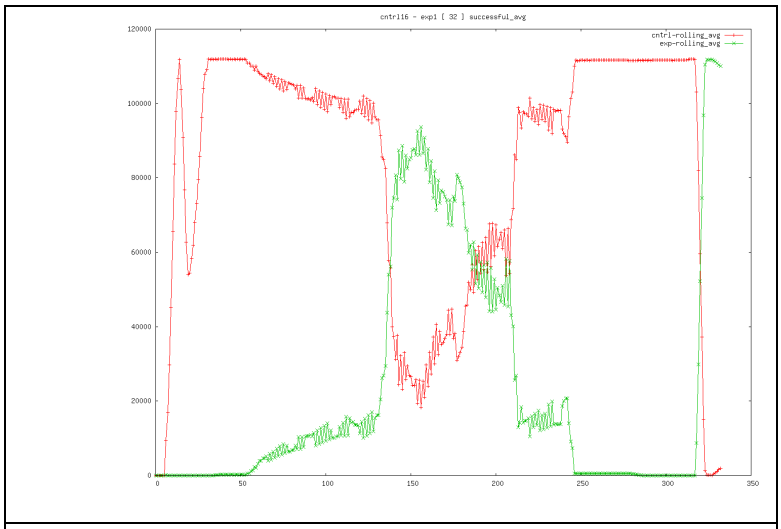


Figure 10: Average Throughput Plot (#78617) - 32 virtual streams

The SSP reflects this behavior, but then (probably as a result of a connection timeout) is forced to retransmit much of the same sequence of data.

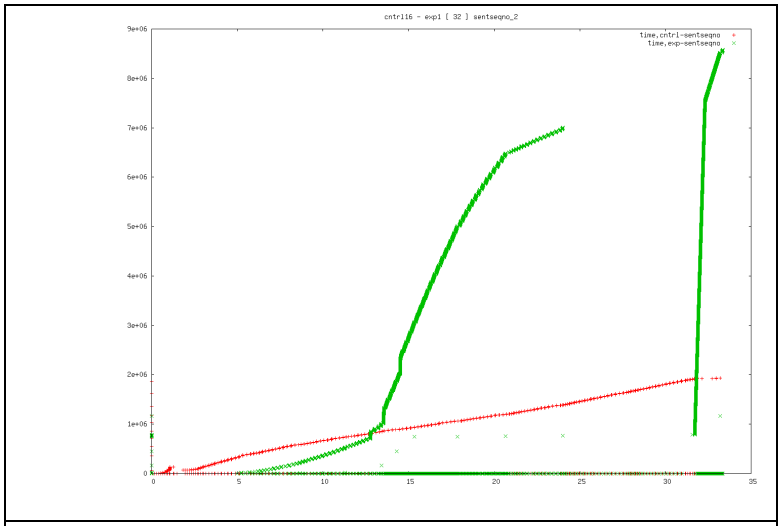


Figure 11: Sent Sequence Plot (78617) - 32 virtual streams

The ASP shows the same kind of step growth function, again driven by the acknowledgement sequence stream sharply lagging the sent sequence stream. Once the acknowledgement sequence reflects the warp sequence, the plot steps, the unacknowledged data count drops to nearly zero, and the full amount of the congestion window is open for transmitting data. The result is that the network is overloaded, packets are lost, and the pattern repeats itself.

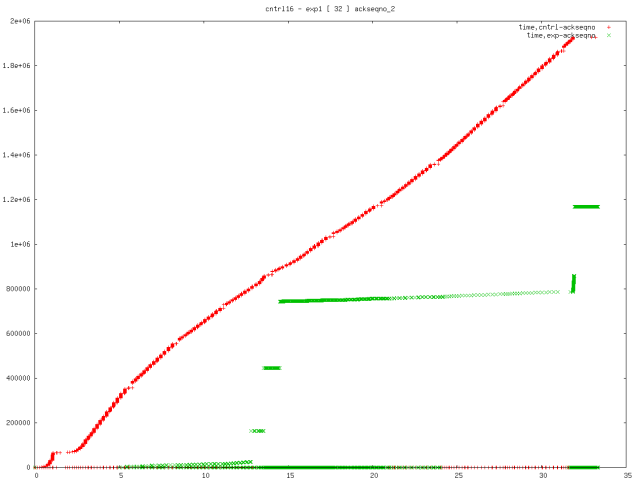


Figure 12: Acknowledgement Sequence Plot (78617) - 32 virtual streams

We omit the results for the 16-virtual stream and 64-virtual stream cases as they were similar enough to the 32-virtual stream case to be uninteresting here.

Effectively, the algorithm appears to gate the congestion window once in retransmission mode. Since the time to receive an acknowledgement is gated by Round Trip Time (RTT) the algorithm effectively gates itself by flooding the network, entering retransmission mode, and only releasing new data as acknowledgements are received.

While the custom timer is effective at detecting lost packets and forcing their retransmission, its prevalence in correcting gaps in the transmission stream indicates that the composite stream is exceeding a stable consumption of network bandwidth.

VII. EVOLUTION OF THE DESIGN

While our current results have been unsatisfying, the evolution of the algorithm did show early signs of success. We will now review a few critical stages in the evolution of the algorithm, and provide our understanding of what caused the behavior, given what we learned with the final evaluation.

Token-based Congestion Window

Our design began with the idea of distributing tokens to the composite stream. The raw events and virtual events were effectively the same, as we distributed NEWACK, DUPACK, and TIMEOUT to virtual streams, as-is. In the design, the composite stream would receive tokens from each virtual stream, based on available window space within the virtual stream. We had two flavors of tokens, one for transmission and one for retransmission. When a NEWACK was received, we informed a virtual stream and it responded by sending the composite stream a transmission token. When the composite stream transmitted data, it would "spend" tokens. If the size of the congestion window was non-zero, we could transmit data, hence the calculation of the available window literally becomes the size of the congestion window, rather than the difference between the *congestion window* and unacknowledged data.

Similarly, when DUPACKs were received, we would hand those off to a chosen virtual stream, and it would make a decision about whether to generate a retransmission token to the composite stream. Each virtual stream would keep track of how many DUPACK events had been distributed to it, and when its internal count reached 3, it would generate a retransmit token. When the retransmission token was

received, the composite stream would retransmit the packet from the highest received acknowledgment. Initially, once a retransmission occurred, any excess retransmission tokens were forgotten.

We ran evaluations against the simulated environment to understand the behavior of this first draft of the design, and found that it performed poorly. The worst case behavior for retransmissions could reach up to $2n+1$ DUPACKs before a retransmission, for n virtual streams. We found this behavior to be severely limiting, and introduced a variable to track whether the composite stream believed a retransmission was necessary. A three-phase state variable allowed the composite stream to remember whether it believed a retransmission was necessary. As before, it would wait for a retransmission token to resend the lost packet but only if the state was set would it actually retransmit data. Otherwise, it could accrue retransmission tokens, and when a future loss was experienced, the composite stream would immediately retransmit the lost packet and spend a retransmission token in so doing. After a few other minor iterations to the algorithm, we reduced the three phase "needs retransmission" state to a two-phase "operating mode", which is still found in the current design.

Credit-based Congestion Window

We found the notion of tokens did not adequately allow us to simulate the aggregate behavior we wanted. We reconsidered the key design elements, and reformulated the design with the notion of a single line of credit, where each virtual stream extended credit or reduced credit from the composite stream. In this design, the virtual streams implemented their own internal congestion windows, and the composite stream represents the sum of all component streams.

We also reformulated how to notify virtual streams of events from the raw TCP sender. A new layer of indirection allowed us to translate raw events into a virtual event. Instead of providing NEWACK, DUPACK, and TIMEOUT directly to each virtual stream, we formulated the notion of a positive or negative virtual event, with the special case of severe loss. While the basic notion of events remains similar, each virtual stream chose how to use the event to adjust its internal congestion window. As the virtual stream congestion window changes, the difference is extended or reduced from the composite

window. Furthermore, we now had the flexibility to decide at the composite layer whether we thought an event was positive or negative, rather than simply its type of acknowledgement.

We submit that the notion of operating mode and the translation of events into positive or negative events is critical for any design which attempts to simulate aggregate behavior with individual components. The individual components may contribute to the rate at which data is transmitted, but only the aggregate stream can be fully aware of whether it needs to transmit new data or retransmit lost data.

Receiver Window Ceiling

Our evaluations with the Credit-based model proved that we quickly scale up and establish a stable state. However, the scaling was not proportional to the number of virtual streams. As shown in the ATP below for 32 virtual streams, we achieved a stable state, but are still only a fraction of the total control throughput. Note that the experiment source (green line) achieves a steady state very quickly, and remains there throughout the course of the experiment. However, since we are simulating 32 virtual streams juxtaposed against 16 individual TCP sources, we would expect the experiment rolling average to be about twice the rolling average of the control sources.

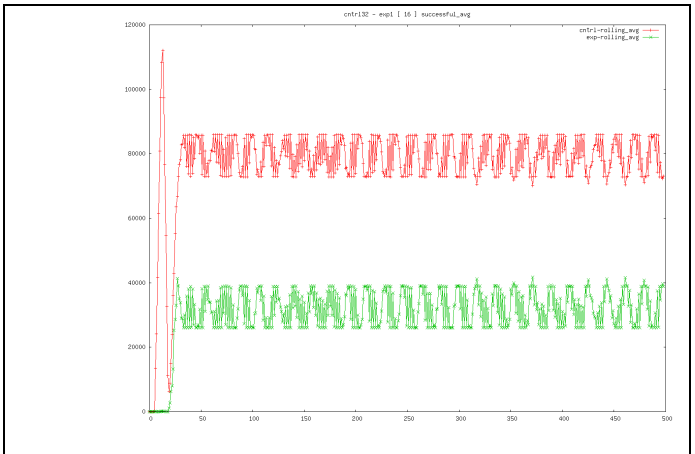


Figure 13: Average Throughput Plot (43861) - 32 virtual streams

Here we also see very predictable SSP and ASPs, showing our experiment source far outpacing a representative control source, as expected.

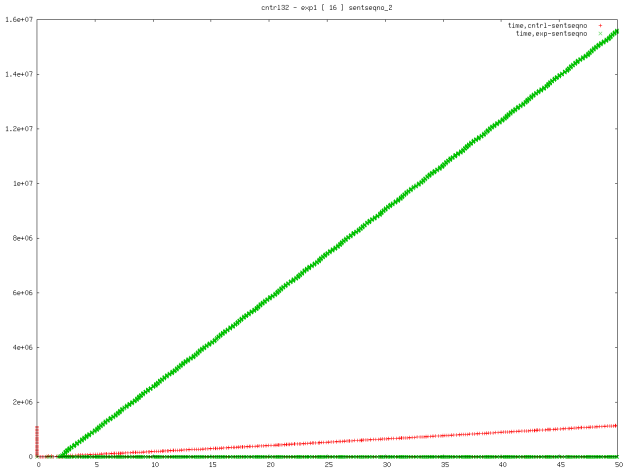


Figure 14: Sent Sequence Plot (43861) - 32 virtual streams

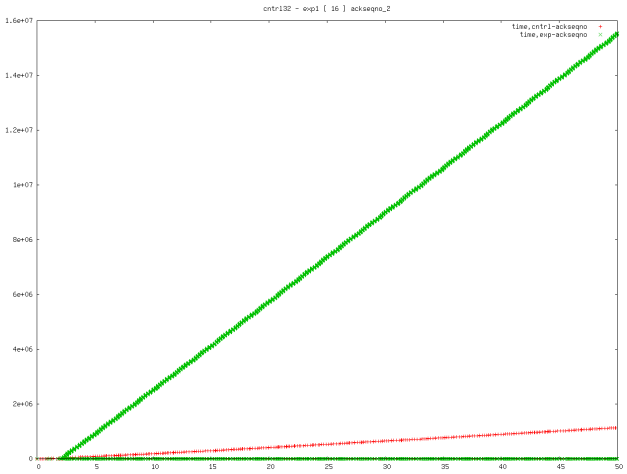


Figure 15: Acknowledgement Sequence Plot (43861) - 32 virtual streams

The puzzling result of the above series is that we are not properly scaling. We found that the window advertised by our receiver was fixed at 2^{16} by the implementation of our simulation platform. We modified our experiment source to ignore the receiver window and rely solely on the computed congestion window. While not expected of the final design, the modification allowed us to gauge whether we could scale as expected, assuming a receiver that could adequately handle the bandwidth.

Expected Scaling Achieved

Our second to last iteration of the design did show promising experimental results. The prior generation of the design did not include a custom timer to watch for packets that were lost a second time once retransmitted. The gap in the transmission sequence caused our experimental source to perpetually remaining in retransmission mode. The composite stream would send new data and receive acknowledgements (which turned out to be on a fixed packet), leading to more available credit provided by the virtual streams. The cycle would repeat itself, causing the congestion window to grow and simultaneously increasing the amount of unacknowledged data. The key aspect was that the difference between the congestion window and unacknowledged data remained relatively consistent, allowing the composite stream to quickly find a steady transmission rate. Because the available credit was directly proportional to the number of virtual streams, the composite stream was able to scale as we had anticipated. The Average Throughput Plots below show this behavior for each of our experiment cases.

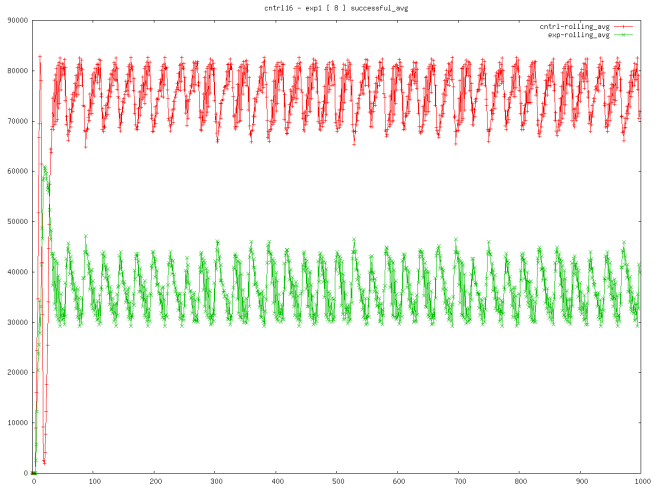


Figure 16: Average Throughput Plot (48172) - 8 virtual streams

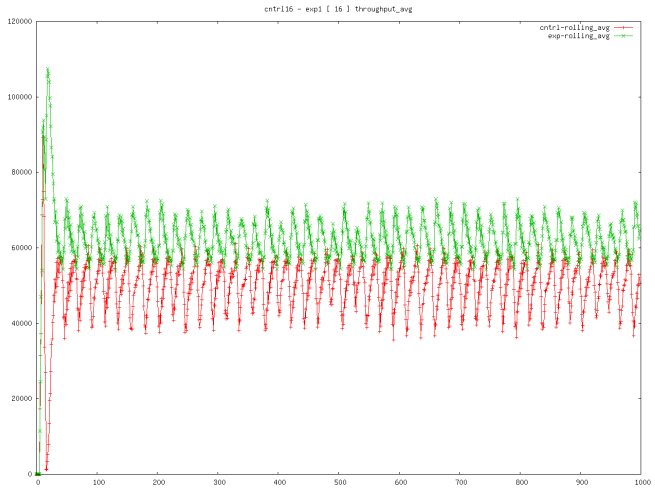


Figure 17: Average Throughput Plot (47109) - 16 virtual streams

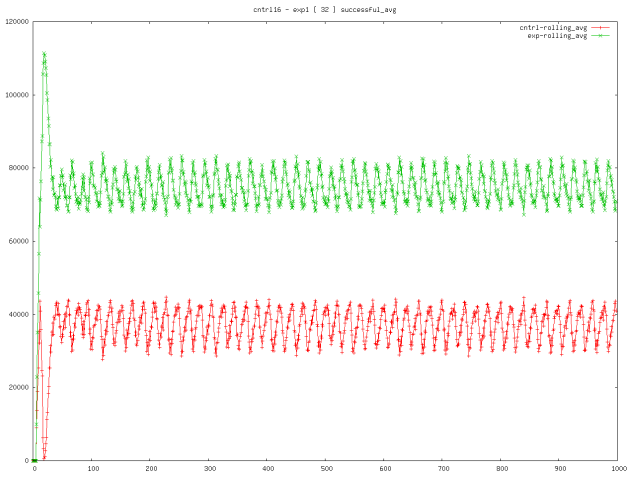


Figure 18: Average Throughput Plot (54433) - 32 virtual streams

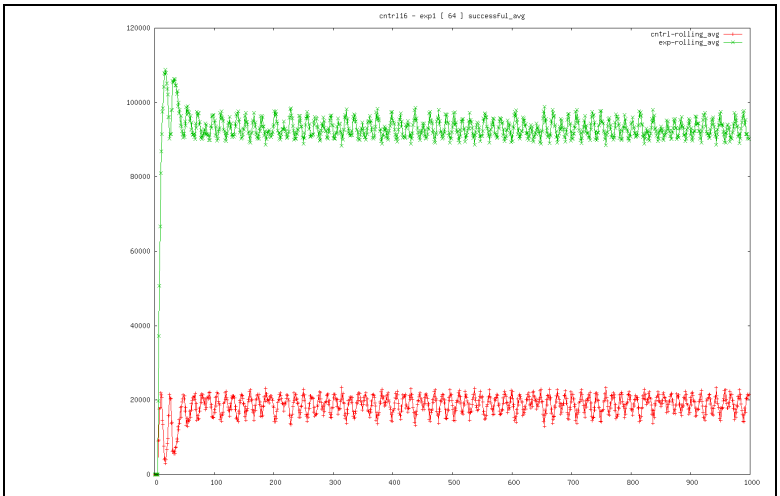


Figure 19: Average Throughput Plot (47426) - 64 virtual streams

As visible in the previous Average Throughput Plots, we clearly consume available bandwidth with a predictable ratio against 16 TCP sources. In the 8-virtual stream case, we consume half of the available bandwidth. In the 16-virtual stream case, we are a little more aggressive, but still roughly in line with the bandwidth consumed by our control sources. Finally, in the 32 and 64 virtual stream cases, we consume roughly double and quadruple the available bandwidth, respectively.

While very exciting, the results proved to hide a dark secret in the correctness of the transmitted data. We saw that the sender sequence remained monotonically increasing, while the Acknowledgement Sequence Plot showed we were perpetually receiving the same acknowledgement, which meant we could never return to transmission mode. Below, a representative sample Sent Sequence Plot and Acknowledgement Sequence Plot demonstrates this disappointing result.

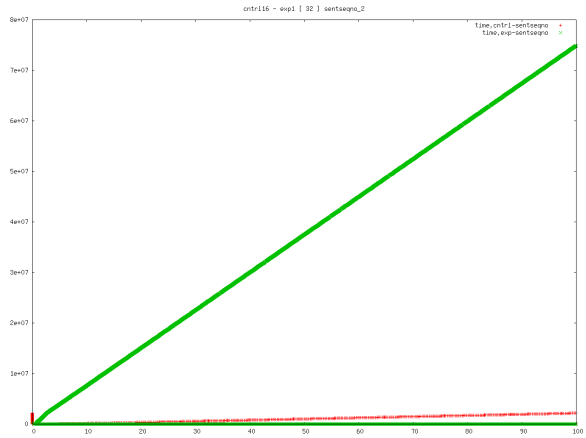


Figure 20: Sent Sequence Plot (54433) - 32 virtual streams

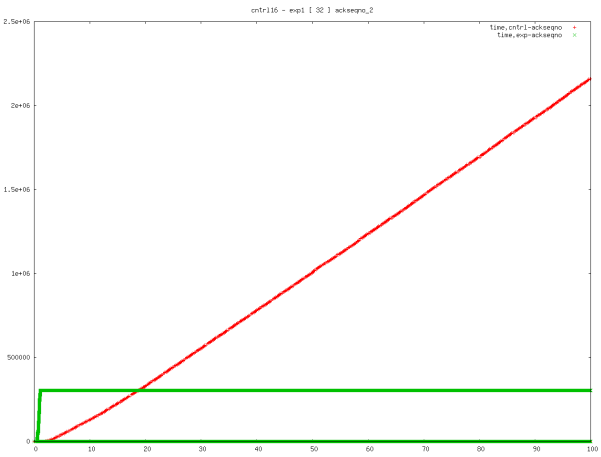


Figure 21: Acknowledgement Sequence Plot (54433) - 32 virtual streams

While the approach demonstrated very effective scaling, the acknowledgement stream was locked into a single lost packet.

When operating in retransmission mode, we expected the standard TCP protocol Retransmission Timeout to provide a failsafe if a packet was lost twice in the transmission sequence. However, upon closer inspection of the implementation in our simulation platform, the Retransmission Timeout was reset for each new transmission, effectively capturing only complete connection failures. We determined that because we were still successfully sending packets and receiving acknowledgements (albeit for the wrong packet), the

TCP stream was not timing out, which would have allowed us to take a severe loss and retransmit the packet again. It was this result which led us to the introduction of the custom timer, to ensure that any lost packet would caught and retransmitted.

Even after the introduction of the custom timer, we found that the congestion window could easily crash, preventing us from sending any data. Hence the ordering of the extension of credit, and retransmissions is critical to the proper function of the design. In general, if we deem a retransmission is necessary, we always Retransmit() before we process any new acknowledgements (e.g. CommonNewAck()).

We also found that taking a loss event on every loss detected under retransmission mode while receiving acknowledgements less than the warp sequence was overly aggressive, causing the congestion window to crash. Instead, we chose to only take a single loss in this case, reasoning that in general we expect that losses will be bursty, losing multiple packets for one independent loss event.

VIII. FUTURE RESEARCH

While our final results did not satisfy our hypothesis, the prior results demonstrate that scaling with a similar model could be possible. In particular, two alternative approaches may demonstrate the expected behavior.

First, in a very early iteration of the design, we treated the congestion window as a queue of tokens, for which every time data was sent, a token was "spent". When we moved to the credit-based model, we choose to treat the congestion window as the sum of all component windows. When we encounter losses, the congestion window grows due to the large number of duplicate acknowledgements that are received. We saw in Section *6.1 Numerical Results*, that the side effect of this behavior causes a dramatic spike once the transmission stream catches up to the warp sequence. If we were to constantly deduct from our congestion window as we transmitted data, we should be able to avoid this negative effect.

Second, we considered the implementation of a timer-based gating function that would pace the transmission of new data when the connection initially began its transmission. In effect, we could avoid such rapid transmission of the entire congestion window early in the life of the connection, and simulate the timing of multiple independent sources sending data. As a result, we might be able to better adjust our transmission speed and grow more steadily, without losses due to initially overburdening the network. We could also use the same timer-based gating function to choose when to transmit data as new acknowledgements cause the extension of new credit. Such an approach would provide a hybrid approach of driving the growth of our congestion window based on real time, rather than strictly based on Round

Trip Time (RTT). Experimental results in other approaches [7, 9] have demonstrated that use of real time rather than RTT can provide a more robust protocol for high bandwidth delay product links.

IX. CONCLUSIONS

The standard TCP algorithm for congestion control has shown tremendous robustness in most network environments, but as network links increase in both bandwidth and distance, the traditional AIMD congestion control algorithm may not be the most efficient means to consume available resources. Existing improvements to the congestion control algorithm make minor modifications to the traditional AIMD approach or clever modifications that uses observed behavior of the network to modify the congestion window size.

Our proposed approach computes the window size as a function of several individual virtual components, in order to scale proportionally to the number of virtual streams. We analyzed network simulations run on the GTNets simulation platform to establish whether we supported our original hypotheses. While, our final design did not achieve our intended results, we did demonstrate the expected scaling behavior in prior iterations of the design, but were plagued by a correctness condition where lost packets were not detected and retransmitted if dropped after an initial retransmission. We concluded with suggested future directions, which could establish a design which met our stated goals, enabling a sender-side only modification for an arbitrarily scalable implementation of the TCP protocol.

X. REFERENCES

[0] T. V. Lakshman and U. Madhow, "The performance of TCP/IP for networks with igh bandwidth-delay products and random loss," *IEEE/ACM Transactions on Networking*, vol. 5 no 3, pp. 336-350, July 1997

[1] M. Allman, V. Paxson, and W. Stevens, "TCP Congestion Control," RFC 2581, April 1999

[2] IBM, *Download Director*. http://www6.software.ibm.com/dldirector/doc/DDfaq_en.html#Q_A1

[3] SpeedBit, *Download Accelerator*. http://www.speedbit.com/

[4] BitTorrent, Inc, *BitTorrent*. http://www.bittorrent.com/

[5] M. Elder, J. Steffy, *Stream Spider*. http://code.google.com/p/streamspider/

[6] T. Kelly, "Scalable TCP: Improving Performance in Highspeed Wide Area Networks," In *Proceedings of PFLDnet 2003*, February 2003, Geneva, Switzerland

[7] R.N.Shorten, D.J.Leith, "H-TCP: TCP for high-speed and long distance networks," In *Proceedings of PFLDnet 2004*, February 2004, Argonne, Illinois

[8] L. Xu, K. Harfoush, and I. Rhee, "Binary Increase Congestion Control (BIC) for Fast Long-Distance Networks," In *Proceedings of IEEE INFOCOM 2004*, March 2004, Hong Kong, China

[9] S. Ha, I. Rhee , and L. Xu, "CUBIC: A New TCP-Friendly High-Speed TCP Variant," In *ACM SIGOPS Operating Systems Review*, July 2008, New York, New York

[10] C. Jin, D. X. Wei and S. H. Low, “FAST TCP: motivation, architecture, algorithms, performance," In *Proceedings of INFOCOM 2004*, March 2004, Hong Kong, China

[11] H. Sivakumar, S. Bailey, and R. L. Grossman, "PSockets: The Case for Application-level Network Striping for Data Intensive Applications using High Speed Wide Area Networks," In *Proceedings of SuperComputing: High-Performance Networking and Computing*, November 2000, Dallas, Texas

[12] S. Ha, Y. Kim, L. Le, I. Rhee, and L. Xu, "A step toward realistic evaluation of high-speed TCP protocols," *Elsevier Computer Networks (COMNET) Journal*, Special issue on "Hot topics in transport protocols for very fast and very long distance networks," 2006