Adaptive Data Block Scheduling for Parallel TCP Streams

Thomas J. Hacker
UITS Research Computing
Indiana University
Indianapolis, IN 46202
hacker@iu.edu

Brian D. Noble
Electrical Engineering and
Computer Science
University of Michigan
Ann Arbor, Michigan 48109
bnoble@eecs.umich.edu

Brian D. Athey
Michigan Center for
Biological Information
University of Michigan
Ann Arbor, Michigan 48109
bleu@umich.edu

Abstract

Applications that use parallel TCP streams to increase throughput must multiplex and demultiplex data blocks over a set of TCP streams transmitting on one or more network paths. When applications use the obvious round robin scheduling algorithm for multiplexing data blocks, differences in transmission rate between individual TCP streams can lead to significant data block reordering. This forces the demultiplexing receiver to buffer out-of-order data blocks, consuming memory and potentially causing the receiving application to stall. This paper describes a new adaptive weighted scheduling approach for multiplexing data blocks over a set of parallel TCP streams. Our new scheduling approach, compared with the scheduling approached used by GridFTP, reduces reordering of data blocks between individual TCP streams, maintains the aggregate throughput gains of parallel TCP, consumes less receiver memory for buffering out-of-order packets, and delivers smoother application goodput. We demonstrate the improved characteristics of our new scheduling approach using data transmission experiments over real and emulated wide-area networks.

1. Introduction and motivation

There are considerable efforts within the Grid and high performance computing communities to improve end-to-end network performance for applications that require substantial amounts of network bandwidth. The Atlas project [1], for example, must reliably transfer more than 2 Petabytes of data per year over networks between Europe and the United States. GridFTP [4], widely used for data transmission by scientific applications, takes advantage of the empirically discovered mechanism of striping data transfers across a set of parallel TCP connections to substantially increase TCP throughput.

In previous work [12, 13, 15], we demonstrated that parallel TCP streams could be made fair and effective by reducing the aggressiveness of all but one stream in the set of TCP streams. Our approach, called *Combined Parallel TCP*, relies on the observation that long round trip time (RTT) flows cannot effectively compete with short RTT flows for bottleneck bandwidth.

An application using aggressive parallel TCP streams on one network path can expect that each stream, over a sufficiently long period of time, will transmit approximately the same number of bytes. This is because the TCP congestion avoidance algorithm attempts to deliver an equivalent fair-share of bandwidth to each stream of the set of unmodified parallel TCP streams. In the case of our combined parallel TCP streams, we cannot assume that each stream will receive an equal share of bandwidth over time. This is because the fractional flow components of the set of parallel streams have a much longer virtual RTT than the real RTT of the unmodified stream component, leading to lower throughput for those streams when they compete for bandwidth. When data transmissions are striped over several network paths [31] using multiple TCP streams per path, it is unlikely that each TCP stream will receive an equivalent share of bandwidth over time. If the transmitting application (such as GridFTP) uses a naive round-robin scheduling approach to assign application data blocks to outgoing TCP streams, the difference in throughput between individual streams will cause significant data block reordering.

To solve this problem, we developed a *weighted* round robin data block scheduling approach that reduces data block reordering and maintains the aggregate throughput benefits of parallel TCP. By reducing reordering, our weighted scheduling approach requires significantly less memory to buffer out-of-order packets, and provides a smoother flow of data to the application.

1.1. Impact of reordering on applications

TCP guarantees the in-order delivery of packets to an application on individual TCP streams. Our problem is

the unexpected late arrival of packets on individual TCP streams that disrupts the packet arrival sequence *across* TCP streams. Out of order reception of data blocks may cause the receiving application to stall while waiting for delayed packets to fill gaps in the data stream. Reordering requires memory to buffer out-of-order packets, and disrupts the smooth flow of data to the receiving application.

Many applications require both high performance networking and in-sequence data block delivery for acceptable performance. These applications fall into four broad categories: applications that cannot tolerate an excessive number of processor stalls waiting for network data [6, 28]; applications in which a perceptible degree of latency is unacceptable [14, 17, 18, 23]; applications that require high throughput bulk transfer [1, 2] ; and applications that aggregate data from multiple sources [25, 26].

GridFTP [4] is a widely deployed application that uses parallel TCP to increase end-to-end throughput and is the underlying data transfer mechanism for Globus [11]. Scientific projects, such as Atlas [1] and NEESGrid [2] are using or planning to use GridFTP for bulk data transfer. Since GridFTP is widely used and is representative of the class of applications that use parallel TCP, this paper focuses on improving the unweighted round robin scheduling algorithm used by GridFTP.

The next section describes our weighted round robin scheduling approach.

2. Weighted Round Robin scheduling

In a set of parallel TCP streams on one network path, each stream's share of the total throughput of all of the streams is directly proportional to the ratio of its congestion window (*cwnd*) to the sum of congestion windows from all constituent streams.

If we schedule a set of data blocks (with size $\leq$ $cwnd_{total}$) by allocating an equal portion of packets to each stream, the "fast" high throughput TCP streams will not be allocated enough packets to fill their congestion window (starved), and the "slow" low throughput TCP streams with small congestion windows will require multiple round trip times to transmit their share of packets (over-allocated). The difference in the ability of each stream to transmit its allocation requires the receiver to buffer out of sequence packets and wait for the "slow" TCP sockets to send packets to fill gaps in the data stream. If the network path is a long RTT high-speed network, a significant amount of memory may be consumed to buffer out of sequence packets. This wastes receiver memory and results in a bursty flow of data to the application.

We solve these problems by allocating a portion of packets to each stream in the same proportion as the ratio of the stream's congestion window to the sum of all congestion windows ($cwnd_{total}$). If there are no packet losses during a transmission period, no individual socket will be starved or over-allocated, and all of the packets (assuming that the sender and receiver agree on the sequential order of socket processing) should arrive in consecutive order with no reordering on an individual network path. When a loss occurs, *cwnd* for the stream will be reduced by congestion avoidance, and the portion allocated to the socket by the scheduler will decrease.

Using this adaptive mechanism, our *weighted* scheduling approach results in less reordering than the unweighted scheduling approach.

3. Prior work

Much of the existing work on transmitting data over multiple network connections uses two approaches.

The first approach, called *link striping*, improves performance and reliability by multiplexing a data stream over a set of network adapters on one server that are connected over disjoint network paths. A variety of scheduling algorithms based on round-robin and fair-queueing are used to balance traffic load over the set of network paths. Adiseshu [3] provides an excellent introduction to the topic. Most of the work in this area assumes that the transmission rate is stable and constant (i.e. UDP streams), and does not take into account "bad" network behaviors such as packet reordering. MulTCP [9] describes weighted proportional fairness for a single network path. Phatak [24] discusses link striping across multiple channels for mobile computing, and shows that to fully utilize the network resource, the fraction of data sent along a path should correspond to the ratio of the path's bandwidth to the aggregate sum of available bandwidth. Phatak's analysis assumes that the effective bandwidth of a TCP connection is constant for the lifetime of the TCP connection. Snoeren [29] is the most similar to our work, but differs in that he assumes that congestion avoidance is handled by a transport layer above, and that one transport layer connection is link striped across multiple network links. Our work assumes that *each* link is managed by congestion avoidance, which is more complex for the scheduler. Zhang [31] describes an approach (mTCP) that splits a TCP stream over several independent network paths. Our work is similar to Zhang's, but differs in that we focus on scheduling packets over each stream with the goal of decreasing receiver memory use and improving goodput stability.

The second approach stripes data transmission from several servers over multiple network connections. Nebat [21] examines the implications of performing this type of transfer by breaking down the dataset stride to packet level granularity. Our work differs from Nebat's work because we assume that the complete set of TCP connections exists be-

tween *one* client and *one* server over one or more network links. In some ways, this simplifies the problem, but because we deal with important properties of the network link (such as the congestion avoidance algorithm and packet reordering), our problem has the same level of complexity. Other work that describes TCP improvements include HS-TCP [10], Scalable TCP [19], BIC-TCP [30], and Congestion Manager [7].

4. Weighted scheduling issues

We identified several issues in the implementation of our weighted round robin scheduling approach. Undesired coupling between congestion avoidance and scheduler control systems leads to poor performance. Significant reordering can result from overallocating data to the send socket buffer. Finally, allocation depends on the congestion avoidance phase. This section explores the effects of these issues on reordering and throughput, and describes our solutions to these problems.

4.1. Scheduler and TCP control system interaction

The first issue is the interaction between the scheduler and the TCP congestion avoidance algorithm. Both algorithms are control systems that are coupled by the TCP congestion window *cwnd*. When the congestion avoidance algorithm detects a packet loss, it reduces *cwnd*, which in turn causes the scheduling control system to reduce the number of packets allocated to the socket, which eventually affects the congestion avoidance algorithm. Conversely, an increase in *cwnd* drives the scheduler to allocate more packets to the socket. The cycle between these control systems may cause the coupled control system to overallocate data to the socket or starve it.

If the scheduler persistently underallocates packets to the socket, the goal of maintaining aggregate throughput cannot be met. As long as there is sufficient data in the socket buffer to fully drive the congestion avoidance algorithm, any additional data in the buffer (up to the socket buffer limit) will not exert an upward pressure on *cwnd*.

To prevent socket starvation, the scheduler must maintain an allocation of *at least* two times the current congestion window's worth of data. If the scheduler can provide enough data in every scheduling round to keep the congestion algorithm busy, then the scheduler will not drive a reduction in the TCP congestion window or reduce the effectiveness of the TCP stream.

When a packet loss occurs, *cwnd* is halved, and reordering becomes a problem. Packets ready for transmission in the send socket buffer were scheduled based on the previous *cwnd* value, and at least half of the packets in the send

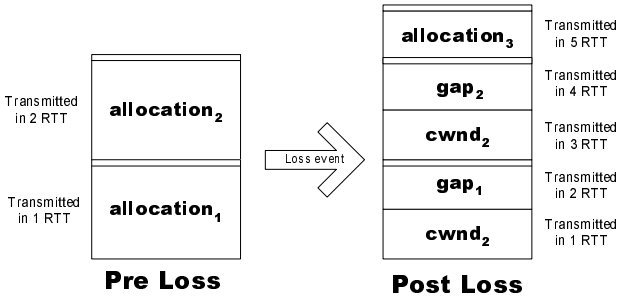

Figure 1. Effects of packet loss effects on send socket buffer transmission

socket buffer will be trapped until at least one additional RTT cycle elapses.

To illustrate this problem, consider the situation in Figure 1, in which the pre-loss socket buffer (one of a set of open parallel TCP sockets) contains two *cwnd*'s worth of data, equally allocated in two scheduling rounds: $allocation_i = f(cwnd_1)$.

When a loss occurs on this socket, *cwnd* will be halved, and only 1/2 of the data in $allocation_1$ will be sent in one RTT, stranding the remaining data in the allocation (gap_1) until the sender receives ACKs for the first half of the allocation. When the receiver receives only half of the allocated data, and then proceeds to read data from the next socket, a gap in the data stream is created. The receiver is forced to buffer subsequent packets from other sockets until the sender transmits the data in gap_1. To solve this problem, we bound the maximum number of packets allocated to the socket buffer to minimize the number of gaps caused by a loss event.

4.2. Optimal TCP transmission socket buffer size

The second issue is determining the optimal size for the TCP send socket buffer. If there are a large number of packets queued in the send socket buffer, it will take too long for a change in *cwnd* to manifest itself. If there are insufficient packets queued in the send socket buffer, the congestion avoidance algorithm will not effectively probe network capacity.

The TCP send socket buffer is used as a buffer between the application and the network, and as a place for holding packets for retransmission when a packet is lost. This dual use requires the socket buffer to be large enough to hold one congestion window's worth of data for transmission and one congestion window's worth of data for fast recovery, and to maintain TCP's ACK clock. Based on this, Semke [27] argued that the TCP send socket buffer should be set to two times the delay bandwidth product to maximize throughput. Consequently, to keep the TCP

congestion avoidance algorithm supplied with a sufficient number of packets, the socket buffer should contain *at least* $2 * cwnd$ worth of data.

To establish the maximum acceptable size (upper bound) for the socket buffer, there are several issues to consider. First, the congestion avoidance algorithm does not require more than $2 * cwnd$ of data in the socket buffer. This is because additional losses beyond an initial loss event (that Selective Acknowledgement (SACK) can't handle) will only further reduce *cwnd*, which leaves sufficient data in the socket buffer to drive congestion avoidance.

The other issue is limiting reordering by calculating the upper bound as a function of *cwnd*. Semke [27] describes a scheme to automatically tune the TCP send buffer size as a function of *cwnd*, maximizing throughput without wasting kernel memory on excessively large buffers. During TCP equilibrium, *cwnd* varies by a factor of two between its minimum and maximum values. In Semke's scheme, the autotuning upper bound (sb_net_target) ranges between 2 * *cwnd* and 4 * *cwnd* when TCP reaches equilibrium. We make use of this idea and apply an upper bound of $4 * cwnd$ for every scheduling round. When the send socket buffer exceeds 4 * *cwnd*, no additional packets are allocated to the socket until it is drained to a value near $2 * cwnd$.

Algorithm 1 Weighted Round Robin Scheduling

Require: Set of tuned parallel TCP sockets P
Require: Operating system support for TCP_INFO option to the *getsockopt* call.
$pktsize \Leftarrow$ Current MSS for the sockets
loop
  Loop forever until termination or socket failure.
  for each socket $S \in P$ **do**
    {Determine scheduler allocation for each socket based on *cwnd* and *ssthresh*.}
    $ssthresh, cwnd \Leftarrow getsockopt(S, ..., TCP\_INFO, ...)$
    if $(cwnd < ssthresh)$ **then**
      {Socket in Slow Start: *cwnd* doubles every RTT}
      $minalloc \Leftarrow 3 * cwnd$ {2 RTT}
      $maxalloc \Leftarrow 15 * cwnd$ {4 RTT}
    else
      {Socket in Linear Increase Phase}
      $minalloc \Leftarrow 2 * cwnd + 1 * pktsize$ {2 RTT}
      $maxalloc \Leftarrow 4 * cwnd + 3 * pktsize$ {4 RTT}
    end if
    {Determine current send socket buffer occupancy}
    $outq \Leftarrow getsockopt(S, SIOCOUTQ, ...)$
    if $(outq > (minalloc * 1.5))$ **then**
      {Current socket buffer loaded beyond low water mark of $1.5 * minalloc$.}
      $newdata \Leftarrow 0$
    else
      {Fill to high water mark of $4 * cwnd$.}
      $newdata \Leftarrow maxalloc - outq$
    end if
    Transmit *newdata* bytes of application data on socket S
  end for
end loop

5. Parallel socket scheduling algorithms

Algorithm 1 describes our weighted round robin scheduler. This algorithm requires a set of connected TCP sockets that have been tuned to disable the Nagle algorithm, enable MTU discovery, set the virtual RTT value, and set the send and receive socket buffer limits high enough to not limit throughput [20].

Algorithm 2 Unweighted Round Robin Scheduling (GridFTP)

Require: Set of tuned parallel TCP sockets P
loop
  Loop forever until termination or socket failure.
  $minalloc = 1048576$ {1 MB block allocation}
  {Maintain a limit of 2 block send socket buffer for each socket}
  $maxalloc = 2 * minalloc$
  for each socket $S \in P$ **do**
    {Determine current send socket buffer occupancy}
    $outq \Leftarrow getsockopt(S, SIOCOUTQ, ...)$
    if $((maxalloc - outq) \geq minalloc)$ **then**
      Transmit *minalloc* bytes of data on socket S via a call to *send*()
    end if
  end for
end loop

To compare the effects of our weighted scheduling approach with the unweighted approach, we used Algorithm 2 to represent the unweighted round robin scheduler used by GridFTP. In the GridFTP scheduler, each socket is allocated a default value of 1 megabyte of data (with an additional 136 bits of overhead per block) every scheduling round [5]. GridFTP waits for sufficient space to become available in the socket buffer to accommodate an entire 1MB block before adding it to the socket buffer. Algorithm 2 maintains a minimum socket buffer allocation of 2 blocks to ensure that the congestion avoidance algorithm is never starved for data. As in the case of Algorithm 1, each socket in P should be tuned.

Note that over a high bandwidth delay product path, 2 MB is insufficient to fill the congestion window. For example, if the round trip time is 68 msec, the congestion window for a 1000 Mb/sec bottleneck throughput is 8.5 MB. A block limit of two 1 MB blocks will starve the congestion avoidance algorithm, which will artificially limit throughput. Consequently, in our experiments we adjusted the minimum block size *minalloc* to ensure that there would always be sufficient data in the socket buffer to drive the congestion avoidance algorithm, ensuring a fair comparison.

6. Experimental methodology

To assess our scheduling approach, we implemented both the unweighted round robin (Algorithm 2) and our

weighted round robin scheduling (Algorithm 1) on a RedHat 9.0 system running Linux kernel 2.4.20. We used unmodified parallel TCP (currently used by GridFTP) along with our Combined Parallel TCP to assess the interaction of both parallel TCP approaches with the WRR and URR algorithms. The sender used the TCP_INFO and SIOCOUTQ *getsockopt*() options available in the Linux kernel to collect the current size of the congestion window and the send socket buffer.

During each experiment, the receiver collected successive packets from each socket until the call to *read*() blocked (due to draining of the socket buffer), or until the packet sequence number of next packet in the socket buffer was not in sequence order. A discontinuity indicated that the scheduler allocation was completely consumed. The packet sequence number was a 64 bit integer, which was large enough to prevent wrap-around during the experiment.

To measure the effect of delayed packets on reordering, the receiver tracked the difference between the sequence numbers of the most recent and last in-order packet received. This difference represented the maximum number of out-of-sequence packets buffered until the arrival of delayed packets necessary to complete the data stream.

Raw throughput was computed by tracking the number of data bytes received (in or out of sequence) during a sampling period, and dividing by the sampling period. Goodput was computed by tracking the number of contiguous in-order bytes received during at least one sampling period, and dividing by the elapsed time. When a sequence gap occurs from a delayed packet, goodput drops to 0 Mb/sec until the late packet arrives. Upon late packet arrival, sequence is restored, and goodput is computed from the number of in-sequence packets stored in the out-of-order buffer up to the end of the sampling period.

To compare the effects of the weighted and unweighted round robin scheduling algorithms on reordering and throughput, we ran transmission experiments on real and emulated wide area networks. Real network experiments were conducted between the University of Michigan in Ann Arbor, Michigan and the California Institute of Technology in Pasadena, California. The sender was a dual-processor AMD Athlon MP 2200+ system with 1 GB of PC2100 memory, and a Tyan 2466 motherboard using an on-board Intel e1000 NIC which was connected with a 100 Mb/sec link to a Cisco 7603 Workgroup switch. The sender ran Linux 2.4.20-8 with the modified fractional TCP congestion avoidance algorithm described in [12, 13, 15]. The receiver was an Origin 200 running IRIX 6.5 with a 100 Mb/sec link through the California Research and Education Network (CALREN) to the Abilene network.

Each experimental trial consisted of two consecutive 230 second transmissions using the weighted round-robin

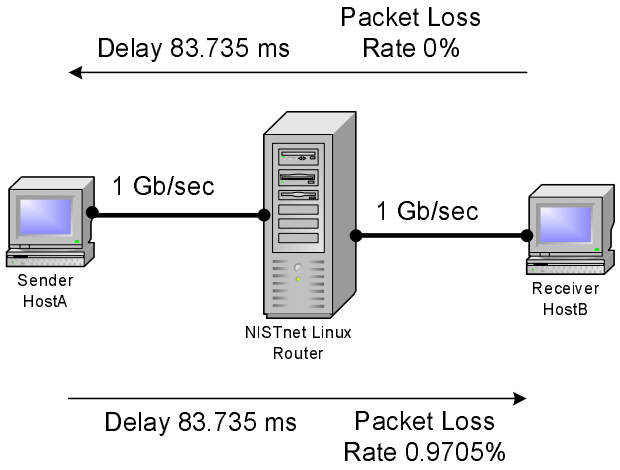

Figure 2. NISTnet single path emulation topology

(WRR) and unweighted round-robin (URR) algorithms with an intervening 30 second gap. For experiments on the real network path, the end-to-end MTU was approximately 1500 bytes. To ensure full use of the frame, we used a 1440 byte data frame in each packet. The send and receive socket buffer sizes were set large enough to ensure that the transmission rate would not be limited by the socket buffer sizes. The URR algorithm was tested using minimum block (*minalloc*) sizes of 1 MB and 2 MB. The bandwidth delay product of the network path to Caltech was 0.8 MB, and the 1 MB minimum block size matches the atomic block-size used by GridFTP. A 15 minute delay was used between each trial to ensure independence between trials. We alternated the order of experiments within each individual trial to eliminate experimental bias between individual experiments (one trial was URR followed by WRR, and the next trial was WRR followed by URR). 372 experimental trials (744 individual tests) were conducted were conducted from April 11 to April 16, 2004 which included diurnal and weekend network loads. Trials used 5 and 10 parallel TCP streams with both unmodified and combined (using a virtual RTT multiplier of 100) parallel TCP, which allowed us to assess each parallel TCP approach.

Figure 2 shows the emulated wide-area network used for the second set of experimental trials. We used the NIST Net network emulation package [22] on a Linux based host router that contained two SysKonnect 9821 V2.0 gigabit ethernet cards connected with a Category-6 crossover cable to a sender and receiver. RedHat Fedora Linux 2.4.22-1.2115.nptlsmp was used on all of the systems in the emulated network. The sender Linux kernel contained the modified fractional TCP congestion avoidance (described previously) to allow the sender to set the virtual RTT for a socket. All of the hosts in the emulated network consisted of dual-processor Athlon MP 1600+ systems with 1

GB of PC 2100 memory, a Tyan 2466 motherboard, and the SysKonnect NICs attached to a 64 bit, 66 MHz PCI bus slot. To instrument the emulation, we used packet loss and round trip time statistics gathered from the Stanford Linear Accelerator Center (SLAC) IEPM project [8] for the month of February, 2004 from Europe to North America. The RTT measured by SLAC was 167.47 ms and the packet loss rate was 0.97%. SLAC measured no packet reordering on the path during the month of February, 2004. We emulated packet loss only on the forward path from the sender to receiver to isolate the effects of ACK packet loss from data packet loss on the TCP sender. Emulated network experiments used an end-to-end MTU of approximately 9000 bytes (Jumbo frame) and a data frame size of 8940 bytes. The experimental design for emulated network trials was identical to real network trials with two exceptions. First, since the emulated network was not subject to any competing traffic, the elapsed time between experimental trials was reduced to 30 seconds. Second, the minimum block sizes (*minalloc*) used for the URR test trials were extended to 1 MB, 2 MB, 4 MB, and 6 MB to accommodate the increased bandwidth delay product of 10.47 MB on the emulated network path.

To assess the effects of our scheduling algorithm on multiple network paths, we created a second emulation testbed (Figure 3) consisting of two separate network paths with differing round trip times and packet loss rates. Four parallel TCP streams were used on each path with a MTU of 1500 bytes.

6.1. Assessing the effects of reordering on the receiver

Packet reordering has two effects on the receiver: additional memory is consumed to buffer out-of-order packets, and the smooth flow of data to the application becomes interrupted and sporadic.

To assess memory consumption, we recorded the mean and maximum amount of buffer memory used to hold out-of-order packets, sampled periodically.

To measure throughput, there are two values to consider. The first value is $throughput_{raw}$, which represents the data bytes per unit time that are received on all of the TCP sockets, independent of reordering. The second value is goodput, which is the total throughput per unit time of in-sequence data delivered to the application (goodput). When a reordering event occurs, throughput will not be affected, but goodput will be reduced to zero until the lost packet is received and the data stream is recovered. When the lost packet is integrated into the out-of-order buffer, and transferred to the application, goodput will become very large, then settle back to a value similar to throughput in the absence of reordering.

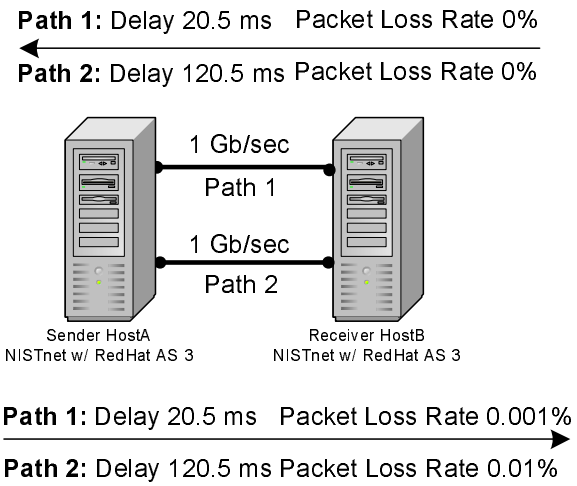

Figure 3. NISTnet multipath emulation topology

To assess the stability of goodput, we used the coefficient of variance (COV).

$$COV = \frac{\sigma}{throughput\ mean_{app}} \quad (1)$$

In this equation, σ is the standard deviation of $throughput_{app}$ and $throughput\ mean_{app}$ is the arithmetic mean of application goodput ($throughput_{app}$) recorded once every sampling period. As $throughput_{app}$ becomes unstable, the measure of dispersion σ increases, which increases the COV. As $thoughput_{app}$ becomes stable, σ decreases, and COV decreases. Thus, larger values of COV indicate more turbulence, and smaller values of COV indicate more stability.

For all the experimental trials, we distilled from each experiment the raw throughput at the first one-second report that occurred after the 230 second mark.

7. Experimental results

This section presents and discusses experimental results for trials on both the real and emulated networks. We first describe the effects of WRR on raw throughput compared with URR. The following section discusses the impact on the amount of memory necessary to buffer out-of-order packets using WRR compared with URR. The final section presents the effects on goodput for WRR versus URR.

The abscissa in the figures in this section are categorical. *P*5, *P*8, and *P*10 represents 5, 8, and 10 parallel TCP streams, and *1M, 2M, 4M* and *6M* indicates the *minalloc* value (in megabytes). Each category indicates the number of streams and the minimum URR block size (*minalloc*) that were used for the experiments in that category.

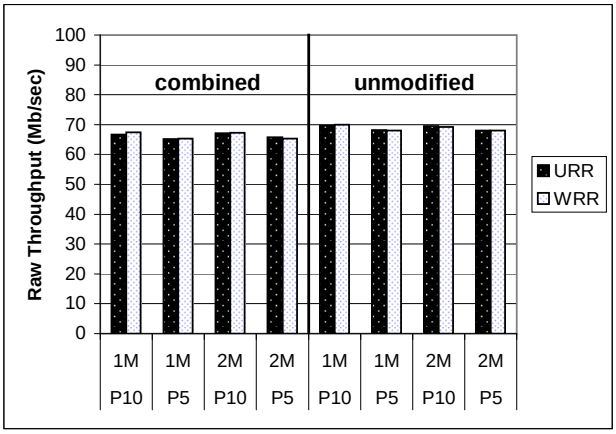

Figure 4. Real raw throughput

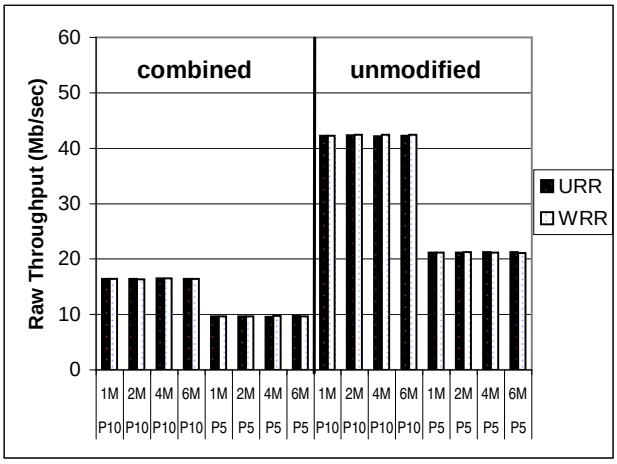

Figure 5. Emulated raw throughput

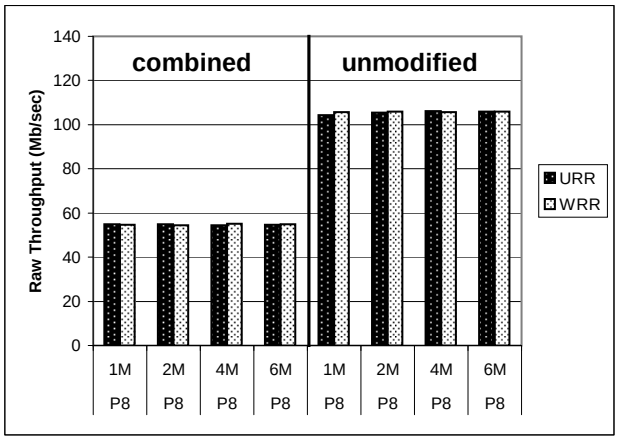

Figure 6. Multipath raw throughput

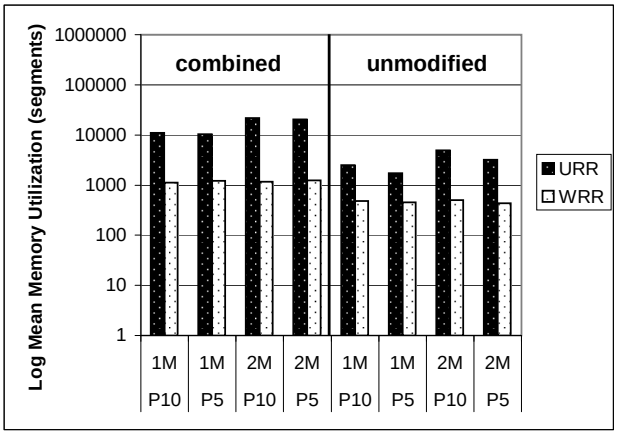

Figure 7. Real buffer memory

7.1. Effects of WRR on raw throughput

Figure 4 shows the mean raw throughput for the URR and WRR scheduling approaches over the period of each experimental trial for both combined and unmodified parallel TCP. Figure 5 shows the mean raw throughput for the emulated network path. Figure 6 shows the mean real throughput for the multipath emulation.

We calculated the 95% confidence interval for the difference in raw throughput between the URR and WRR algorithms for the real, single path, and multipath emulated networks. In all cases, 0 was included in the difference $\pm$ the 95% confidence interval [16]. Consequently, there is no statistical difference in throughput between the URR and WRR scheduling algorithms.

This result confirms several conjectures. First, our WRR scheduling algorithm can keep the congestion avoidance algorithm supplied with enough data to successfully probe the network bandwidth delay product. Second, even through WRR links the scheduler allocation to the size of the congestion window, it delivers the same gains in raw throughput from parallel TCP streams compared with the naive scheduling approach of URR. Thus, the evidence supports our claim that our WRR algorithm successfully maintains the raw throughput gains of URR.

7.2. Effects of WRR on buffer memory consumption

Figure 7 shows the mean memory use for the URR and WRR algorithms for the real network experiments. Figure 8 shows the mean memory use for the URR and WRR algorithms for the emulated network experiments. Figure 9 shows the mean memory use for the emulated multipath network experiments. Please note that the ordinate axis scale is $\log_{10}$.

Figure 10 shows the mean peak memory utilization dur-

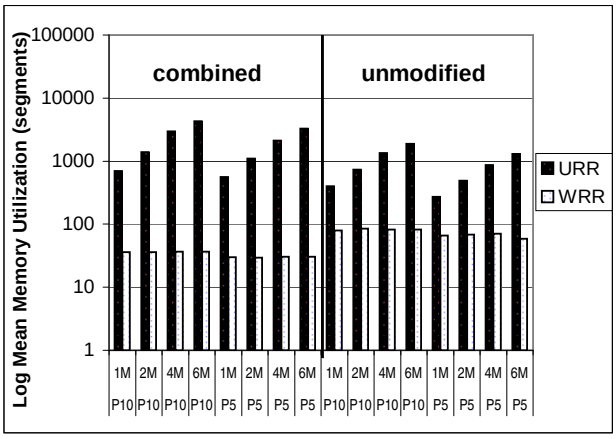

Figure 8. Emulated buffer memory

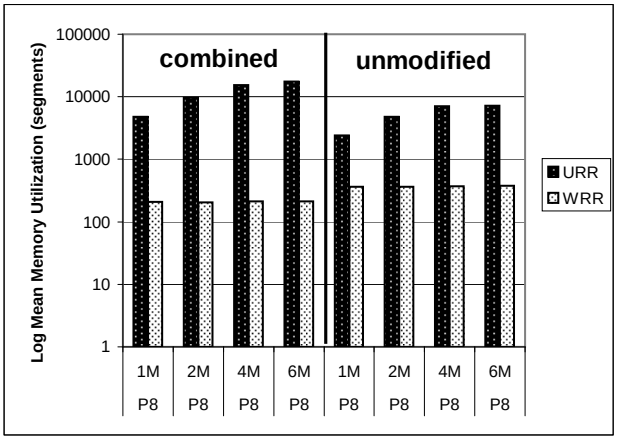

Figure 9. Multipath buffer memory

ing each experiment for combined and unmodified parallel TCP on the real network experiments. Figure 11 shows the peak memory use for combined and unmodified parallel TCP on the emulated network experiments. Figure 12 shows peak memory use for the multipath emulation.

Figure 7 through 9 show that WRR requires **substantially** less mean buffer memory in all cases. These figures show that as the size of the *minalloc* minimum data block size increases for URR, the amount of memory necessary to buffer out-of-order packets increases as well. This has important implications for high bandwidth delay product networks, since *minalloc* must be large enough to keep the congestion avoidance algorithm supplied with enough data to probe the network path. Accordingly, as networks become faster, our WRR scheduling algorithm becomes more necessary to reduce receiver memory consumption.

Figures 10 through 12 show that, in most cases, peak memory consumption for the WRR algorithm is less than peak memory consumption of the URR algorithm.

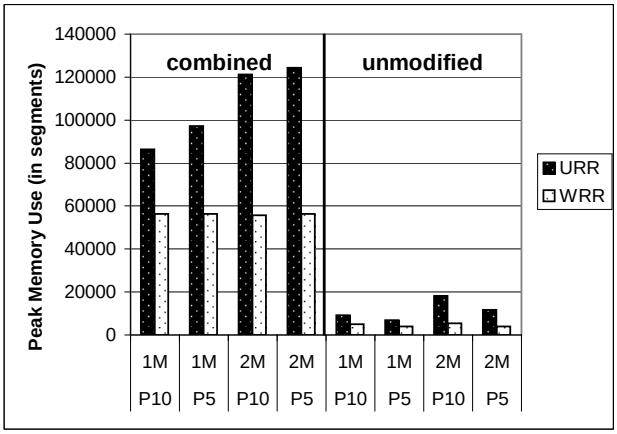

Figure 10. Real peak memory

Figure 11 shows that for a 1MB and 2MB *minalloc* minimum data block size, the mean peak memory consumption of the WRR algorithm is greater than the peak memory consumption of the URR algorithm. We used Student's t-test to verify that the URR peak consumption is less than WRR peak consumption for the P10 case with 1 and 2 MB *minalloc* sizes, and for the P5 case with a 1 MB *minalloc* size. For the P5 case with a 2 MB *minalloc* size, the peak consumption is statistically equivalent. Since the bandwidth delay product on the emulated network path is very large (10.47 MB), the WRR *cwnd* will exceed the maximum URR *cwnd* when the *minalloc* block size is 1 or 2 MB. This difference limits the potential peak memory usage for holding out-of-order packets using the URR algorithm. Although the peak memory usage for WRR exceeds URR in these cases, the mean memory used by WRR over the period of the experiment is still substantially less than URR for these particular cases. We found that WRR improved memory use for both unmodified and Combined Parallel TCP. Figure 12 shows a similar situation for the multipath emulated network.

This evidence supports our hypothesis that the use of WRR scheduling results in less receiver memory consumption to buffer out-of-order packets.

7.3. Effects of WRR on goodput stability

We calculated the coefficient of variance for goodput for the real and emulated networks. Figure 13 shows the goodput stability for combined parallel TCP on the real network experiments. Figure 14 shows goodput stability for unmodified parallel TCP on the real network experiments. Figures 15 and 16 shows the goodput stability for the WRR and URR algorithms for the emulated network experiments. In these figures, a higher coefficient of variance implies less stability, and a lower coefficient is desirable.

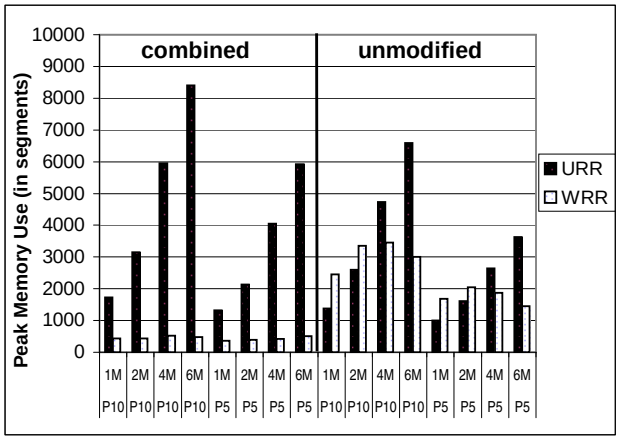

Figure 11. Emulated peak memory

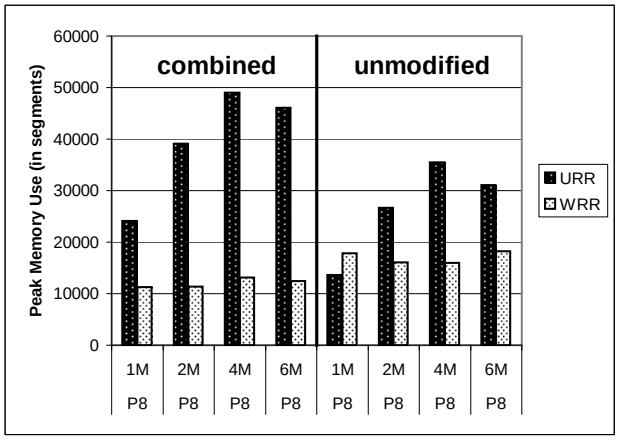

Figure 12. Multipath peak memory

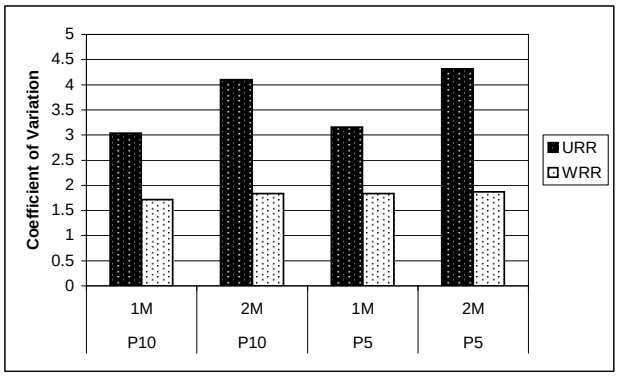

Figure 13. Real goodput stability (combined)

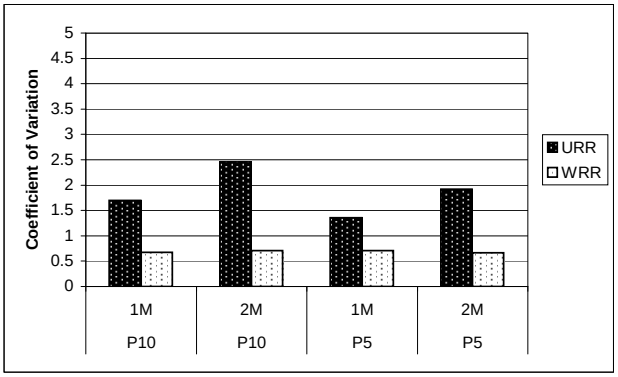

Figure 14. Real goodput stability (unmodified)

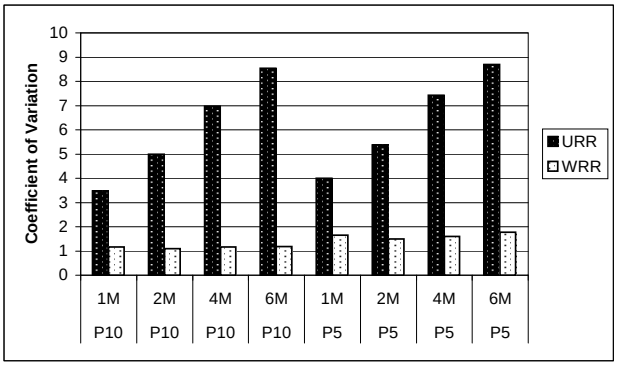

Figure 15. Emulated goodput stability (combined)

Figures 13 and 14 show that the WRR algorithm results in better goodput stability than URR for the experiments conducted on the real network path between the University of Michigan and California Institute of Technology. As the URR *minalloc* minimum data block size increases, the difference in goodput stability between URR and WRR becomes larger. This makes sense, since a larger minimum block size increases the size of the potential gap in the data stream that would result from a delayed packet.

Figures 15 and 16 show that WRR algorithm results in better goodput stability than URR on the emulated network. Similar to the real network, as the *minalloc* URR minimum block size increases, difference in goodput stability between URR and WRR becomes larger. Results for the multipath emulated network (omitted) were similar to the results for the emulated testbed: WRR stability proved to be superior to URR stability in all cases.

These results support our hypothesis that WRR results in better goodput stability that URR.

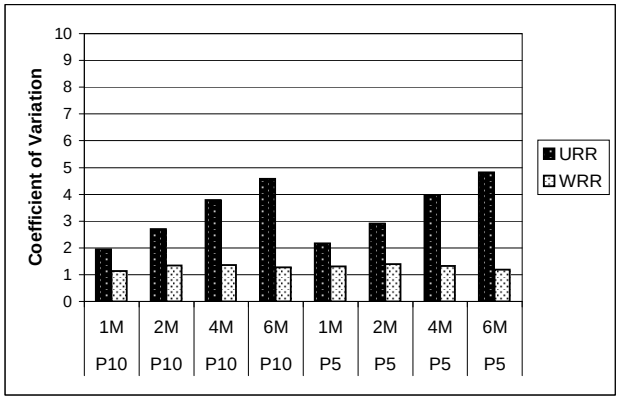

Figure 16. Emulated goodput stability (unmodified)

8. Conclusions

In this paper, we have described a new *weighted* data block scheduler that preserves the gains of parallel TCP, smooths goodput, and reduces memory use. We demonstrated the effectiveness of our approach using a series of experiments over real and emulated wide area networks. Our experimental results indicate that our WRR algorithm results in the same raw throughput, less buffer memory required to hold out-of-order packets, and better goodput stability than the URR algorithm. Based on these results, we believe that applications such as GridFTP that make use of unweighted round robin scheduling over multiple TCP streams on one or more network paths would greatly benefit from the adoption of the weighted round robin scheduling algorithm described in this paper.

Acknowledgments

The authors would like to thank the National Institutes of Health Visible Human Project (Grant N01-LM-0-3511) for their continued encouragement and support. We would also like to thank Dr. Craig Stewart and Malinda Lingwall of Indiana University for their helpful comments and encouragement. The work described in this paper utilized the resources of the National Partnership for Advanced Computational Infrastructure (NPACI).

References

[1] (2003) Atlas high energy physics project. URL: http://pdg.lbl.gov/atlas/atlas.html.

[2] (2004) George E. Brown, Jr. Network for Earthquake Engineering Simulation Project. URL: http://www.neesgrid.org/.

[3] H. Adiseshu, G. Parulkar, and G. Varghese, "A reliable and scalable striping protocol," in *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. New York: ACM Press, Aug. 1996, pp. 131–142.

[4] W. Allcock, J. Bester, A. Bresnahan, A. Chervenak, L. Liming, and S. Tuecke. (2004) Gridftp: Protocol extensions to ftp for the grid. [Online]. Available: http://www-fp.mcs.anl.gov/dsl/GridFTP-Protocol-RFC-Draft.pdf

[5] W. E. Allcock, Personal Communication, 2004.

[6] B. D. Athey. (2003) DARPA virtual soldier project. University of Michigan Virtual Solder Project. [Online]. Available: http://www.virtualsoldier.net

[7] H. Balakrishnan, H. S. Rahul, and S. Seshan, "An integrated congestion management architecture for internet hosts," in *SIGCOMM '99: Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*. New York, NY, USA: ACM Press, 1999, pp. 175–187.

[8] L. Cottrell and W. Mathews. (2003) The Internet End-to-end Performance Monitoring Project, Stanford University. URL: http://www-iepm.slac.stanford.edu.

[9] J. Crowcroft and P. Oechslin, "Differentiated end-to-end internet services using a weighted proportional fair sharing tcp," *SIGCOMM Comput. Commun. Rev.*, vol. 28, no. 3, pp. 53–69, 1998.

[10] S. Floyd. (2003, Feb.) High Speed TCP for Large Congestion Windows. Internet draft. URL: http://www.icir.org/floyd/papers/draft-floyd-tcp-highspeed-02.txt.

[11] I. Foster and C. Kesselman, "Globus: A metacomputing infrastructure toolkit," *International Journal of Supercomputer Applications and High Performance Computing*, vol. 11, no. 2, pp. 115–128, 1997.

[12] T. Hacker, B. Athey, and B. Noble, "The end-to-end performance effects of parallel TCP sockets on a lossy wide-area network," in *16th IEEE-CS/ACM International Parallel and Distributed Processing Symposium, Ft. Lauderdale, FL*. IEEE-CS/ACM, Apr. 2002.

[13] T. Hacker, B. D. Noble, and B. D. Athey, "Improving throughput and maintaining fairness using parallel TCP," in *Proceedings of the 2004 IEEE Computer and Communications Societies Conference on Computer Communications (INFOCOM-04)*, New York, 2004.

[14] T. J. Hacker, B. D. Athey, J. Sommerfeld, and D. Walker, "Experiences using web100 for visible human testbeds," in *Proceedings of the Fourth Visible Human Conference*, Keystone, CO., Oct. 2002.

[15] T. J. Hacker, B. D. Noble, and B. D. Athey, "The effects of systemic packet loss on aggregate TCP flows," in *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*. Los Alamitos, CA, USA: IEEE Computer Society Press, 2002, pp. 1–15.

[16] R. Jain, *The art of computer systems performance analysis*. New York: John Wiley, 1991.

[17] C. Z. Jason Leigh, Rajvikram Singh. (2001, Aug.) The terascope project. URL: http://evlweb.eecs.uic.edu/research/res_project.php3?indi=223.

[18] A. E. Johnson, J. Leigh, and D. T., "Multidisciplinary experiences with cavernsoft tele-immersive applications," in

Proc. of Fourth International Conference on Virtual System and Multimedia, Nov. 1998, pp. 498–503.

[19] T. Kelly, "Scalable TCP: Improving performance in highspeed wide area networks," submitted for publication. [Online]. Available: http://www-lce.eng.cam.ac.uk/~ctk21/scalable

[20] J. Lee, D. Gunter, B. Tierney, W. Allock, J. Bester, J. Bresnahan, and S. Tuecke, "Applied techniques for high bandwidth data transfers across wide area networks." Lawrence Berkeley National Laboratory, Tech. Rep. LBNL-46269, Sept. 2001.

[21] Y. Nebat and M. Sidi, "Resequencing considerations in parallel downloads," in *Proceedings of the 2002 IEEE Computer and Communications Societies Conference on Computer Communications (INFOCOM-02)*. New York: IEEE, June 2002.

[22] NIST Internetworking Technology Group, "NISTNet network emulation package," June 2000. [Online]. Available: http://www.antd.nist.gov/itg/nistnet/

[23] K. S. Park, Y. J. Cho, N. K. Krishnaprasad, C. Scharver, M. J. Lewis, J. Leigh, and A. E. Johnson, "CAVERNsoft G2: A toolkit for high performance tele-immersive collaboration," in *Proceedings of the ACM Symposium on Virtual Reality Software and Technology (VRST-00)*, K. Y. Wohn, Ed. N. Y.: ACM Press, Oct. 22–25 2000, pp. 8–15.

[24] D. S. Phatak and T. Goff, "A novel mechanism for data streaming across multiple ip links for improving throughput and reliability in mobile environments," in *Proceedings of the 2002 IEEE Computer and Communications Societies Conference on Computer Communications (INFOCOM-02)*. New York: IEEE, June 2002, pp. 773–781.

[25] J. Plank, M. Beck, W. Elwasif, T. Moore, M. Swany, and R. Wolski, "The internet backplane protocol: Storage in the network," in *Internet2 NetStore '99: Network Storage Symposium*, Oct. 1999.

[26] J. S. Plank, S. Atchley, Y. Ding, and M. Beck, "Algorithms for high performance, wide-area distributed file downloads," *Parallel Processing Letters*, vol. 13, no. 2, pp. 207–224, June 2003.

[27] J. Semke, J. Mahdavi, and M. Mathis, "Automatic tcp buffer tuning," *SIGCOMM Comput. Commun. Rev.*, vol. 28, no. 4, pp. 315–323, 1998.

[28] L. Smarr. (2002, Sept.) The 'optiputer': California, Illinois researchers fashion new paradigm for data-intensive computing and collaboration over optical networks. URL: http://www.calit2.net/news/2002/9-25-optiputer.html.

[29] A. Snoren, "Adaptive inverse multiplexing for widearea wireless networks," in *IEEE Conference on Global Communications (GlobeCom '99)*, 1999, pp. 1665–1672.

[30] L. Xu, K. Harfoush, and I. Rhee, "Binary increase congestion control for fast long distance networks," in *Proceedings of the 2004 IEEE Computer and Communications Societies Conference on Computer Communications (INFOCOM-04)*, New York, 2004.

[31] M. Zhang, J. Lai, A. Krishnamurthy, L. L. Peterson, and R. Y. Wang, "A transport layer approach for improving end-to-end performance and robustness using redundant paths." in *USENIX Annual Technical Conference, General Track*, 2004, pp. 99–112.