Proc. of the Los Alamos Computer Science Institute Symposium, October 2001. LA-UR 01-5460

Dynamic Right-Sizing in TCP

Mike Fisk[§†] Wu-chun Feng[§‡]
mfisk@lanl.gov *feng@lanl.gov*

[§]Los Alamos National Laboratory
[†]Department of Computer Science & Engineering, University of California, San Diego
[‡]Department of Computer & Information Science, The Ohio State University

Abstract— **With the widespread arrival of bandwidth-intensive applications such as bulk-data transfer, multi-media web streaming and computational grids for high-performance computing, networking performance over the wide-area network has become a critical component in the infrastructure. Tragically, operating systems are still tuned for yesterday's WAN speeds and network applications. As a result, a painstaking process of manually tuning system buffers must be undertaken to make TCP flow-control scale to meet the needs of today's bandwidth-rich networks. Consequently, we propose an operating system technique called dynamic right-sizing that eliminates the need for this manual process. Previous work has also attacked this problem, but with less than complete solutions. Our solution is more efficient, more transparent, and applies to a wider set of applications, including those that require strict flow-control semantics because of performance disparities between the sender and receiver.**

Keywords— ***Wide-area networking, TCP, dynamic flow control, computational grid, high-performance networking, auto-tuning***

I. Introduction

In previous work [4], we brought up two fundamental problems with TCP in high-performance computational grids and other bandwidth-intensive applications: (1) flow-control adaptation and (2) congestion-control adaptation. In order to address the former problem[1], grid and networking researchers have continued the practice of manually optimizing buffer sizes to keep the network pipe full [14], [10], and thus achieve acceptable performance over the wide-area network, whether for bulk-data transfer or in support of computational grids [5], data grids [1], [12], [3], or access grids [2].

For example, solely by optimizing the buffer sizes of end-hosts at LANL and NCAR, as well as the LANL firewall proxy, we were able to increase the delivered throughput from less than 1 Mb/s to 15-20 Mb/s! However, the appropriate configuration changes could not be made by the end user who desired to transfer data between the systems. Instead, system administrators in three different administrative domains had to separately configure their systems to use larger buffers. Further these systems were manually tuned for the optimal delay-bandwidth product between LANL and NCAR, and were therefore not well tuned for other pairs of sites. The result, is sub-par performance for connections with larger delay-bandwidth products. In addition, [13] shows that using too large a window for small delay-bandwidth connections can result in an under-availability of buffers for a concurrent connection with a large delay-bandwidth product.

This work was supported by the U.S. Dept. of Energy's Next Generation Internet - Earth Systems Grid and Accelerated Strategic Computing Initiative - Distance and Distributed Computing and Communication programs through Los Alamos National Laboratory contract W-7405-ENG-36.

[1]The latter problem is beyond the scope of this paper and will be addressed in a future paper.

A. Related Work

To address these issues, several projects have examined methods for automatically tuning window sizes and buffers. The Web100 project has released a modified FTP client that uses user-space code to send a burst of pings to estimate the latency and bandwidth at the beginning of a connection and adjust the windows accordingly [9]. This approach requires that this measurement code be deployed in each application. Further, it uses a measurement period before data is sent and creates extra network traffic that is not controlled by a congestion avoidance mechanism. Thus, it can only be used sparingly, such as at the beginning of a connection.

Earlier work in [13] presents kernel modifications for 'auto-tuning' a sender's flow-control window based on the congestion window and then using fair-share algorithms to manage competition between connections for buffers. In this scheme, the receiver's window advertisements are superfluous. In contrast, our work shows that properly sizing the receiver's window, and consequently the sender's buffers, can yield better performance than always setting windows to a large value.

B. Contributions

The first contribution of this paper is an analysis of how over-subscription of receiver buffers, as suggested in [13], in conjunction with some bandwidth-intensive applications, can lead to the starvation of competing connections. Second, we show how a receiver can measure the round-trip time without sending any data. Third, we show how a TCP receiver can then measure the approximate size of a sender's congestion window, so that the receiver can advertise a window that does not needlessly constrain throughput. In contrast to the Web100 work, neither measurement requires transmitting any extra measurement traffic. Finally, we present an implementation in the Linux kernel in which a sender and receiver use this information to dynamically adjust their window sizes appropriately. Our analysis shows that not only does dynamically adjusting windows help high-performance connections, but that it reduces the number of retransmits needed on low-bandwidth connections.

The result is a system which, like AutoNCFTP [9], elimi-

nates flow control as a bottleneck to high performance, bulk data transfers. However, our implementation automatically benefits not just FTP, but every application on a host, including the increasing number of applications used to support computational grids, data grids, and access grids. Our technique complements the 'auto-tuning' presented in [13] by providing the receiver with the ability to measure the size of the sender's congestion window. With this information, the receiver can more fairly allocate buffers to connections based on their need for buffers. As we will show in Section II, this becomes an issue when the throughput of the receiving application is bounded by factors other than the network. In contrast to [13], our approach does not change the semantics or usage of TCP flow-control. At the same time, our technique makes no changes to the congestion avoidance mechanisms of TCP or the fairness of the systems using them.

C. Paper Organization

In Section II we describe why dynamic right-sizing is necessary and addresses problems not solved by [13]. In Section III we present the fundamental mechanisms of dynamic right-sizing and in Section IV we describe its implementation. In Section V present our experimental results and we present our conclusions in Section VI.

II. Receivers Need Conservative Windows

The 'Auto-tuning' proposal in [13] suggests that TCP receivers should always advertise arbitrarily large windows. To do so, they must over-subscribe their buffer space under the assumption that receiver-side buffer space is only minimally used under most circumstances since the amount of buffer space used on the receiver due to loss or reordering in the network is bounded by the size of the sender's congestion window.

However, there is another case to consider. When the receiving application, rather than the sender or network, is the bottleneck, the receiver needs flow control. Otherwise, unbounded amounts of receiver buffer space can be consumed by data that has been received and acknowledged by the operating system, but is waiting for the receiving application. In this case, buffer usage is not bounded by the sender's congestion window and a single connection could use all available buffers and starve other connections for an indefinite period of time. One instance of this case is the archival storage used by many data grids. Very large datasets are commonly stored on relatively slow, archival storage media in systems such as HPSS. The high-performance systems and caches that access this storage can saturate numerous storage devices simultaneously, even over a wide-area network [8]. This disparity between receiver and sender performance is characteristic of applications that require flow control.

Unlike a router which can manage its buffers by dropping any packet that arrives or is already queued, a TCP receiver cannot drop already acknowledged data. When a receiver acknowledges data, it is committing that it will hold that data in buffers for as long as necessary until the receiving application is ready. There is no opportunity to later drop that data to make room for higher priority data or other pressures on system memory. Thus, there is a very large penalty for incorrectly speculating that the application will read the data before the system runs out of buffers or memory. One slow application may starve all other connections for indefinite periods of time.

As a result, we claim that variable window advertisements are still necessary to signal to the sender how much more data the receiver is currently willing to buffer for that connection. Otherwise, the receiver will have to resort to dropping packets and unnecessary triggering of the sender's congestion avoidance mechanisms, thus wasting bandwidth and reducing throughput for a prolonged period of time.

But how does the receiver decide what the proper window size is? To obtain full throughput, the advertised window must be at least as large as the delay-bandwidth product and with selective acknowledgements [11], the window should be no less than twice the delay-bandwidth product [13]. This rule of thumb allows us to keep data flowing as fast as possible, while being conservative about over-advertising valuable buffer space. This paper presents the theory and implementation of a receiver that implements this rule.

III. Dynamic Right-sizing

In short, *Dynamic Right-sizing* lets the receiver estimate the sender's congestion window size and use that estimate to dynamically change the size of the receiver's window advertisements. We show how to use these updates to keep pace with the growth in the sender's congestion window. As a result, the sender will be congestion-window-limited rather than flow-control-window-limited. Thus, throughput is constrained by the available bandwidth of the network rather than some arbitrarily set constant value on the receiver.

TCP slow-start causes the sender's congestion window to be smaller than the receiver's advertised window for at least the first portion of the connection. Thus there is no value in having the receiver advertise a window larger than the number of bytes that the sender can send with its current congestion window.[2] Therefore, it is sufficient for the receiver to advertise a window that is larger than the sender's congestion window can become before the receiver's next adjustment. Since the congestion window at most doubles once per round-trip time, knowing the current window size and round-trip time will allow us to bound the congestion window.

A. Receiver-side Delay-Bandwidth Measurement

It is trivial to measure the amount of data received in a fixed period of time and subsequently compute the *average* throughput over that period. However, the instantaneous throughput of a connection seen by a receiver may be larger than the maximum available end-to-end bandwidth. For instance, data may travel across a slow link only to be queued

[2] The congestion window is actually measured in packets, but the amount of bytes is bounded by product of the congestion window and the TCP maximum segment size.

up on a downstream switch or router and then sent to the receiver in one or more fast bursts. The maximum size of such a burst is bounded by the size of the sender's congestion window and the window advertised by the receiver. The sender can send no more than one window's worth of data between acknowledgements. Accordingly, a burst that is shorter than a round-trip time can contain at most one window's worth of data.

Thus, for any period of time that is shorter than a round-trip time, the amount of data seen over that period is a lower-bound on the size of the sender's window. Some data may be lost or delayed by the network, so the sender may have sent more than the amount of data seen. Further, the sender may not have had a full window's worth of data to send. So the window may be significantly larger than this lower-bound, but not if the connection is truly limited by the receiver's window. Measuring this minimum and making sure that the receiver's advertised window is always larger will let the receiver track the congestion window size.

To make these measurements, it is necessary for the receiver to know the round-trip time. In a typical TCP implementation, the round-trip time is measured by observing the time between when data is sent and an acknowledgement is returned [7]. But during a bulk-data transfer, the receiver might not be sending any data and would therefore not have a good round-trip time estimate. For instance, an FTP data connection transmits data entirely in one direction. Thus, we must develop a technique for measuring the delay from a system that sends no data.

A system that is only transmitting acknowledgements can still estimate the round-trip time by observing the time between when a byte is first acknowledged and the receipt of data that is at least one window beyond the sequence number that was acknowledged. If the sender is being throttled by the network, this estimate will be valid. However, if the sending application did not have any data to send, the measured time could be much larger than the actual round-trip time. Thus this measurement acts only as an upper-bound on the round-trip time and should be be used only when it is the only source of round-trip time information.

B. Sender's Buffers

We have outlined a method for a receiver to enlarge its window to match the size of the sender's window. In current practice, the sender's buffers are often limited to a fixed size as well. A functional solution is for the sender and receiver to work their window sizes up in harmony until they are sufficient to fill the delay-bandwidth product.

To maintain strict flow control, the sender cannot send more data than the receiver claims it is ready to receive. These increases must consequently be initiated by the receiver. It has already been stated that the receiver will advertise a window larger than the measured window. Thus the sender may increase its buffer allocations by tracking the receiver's advertisements. This technique will not cause the sender's allocations to be out of line with the useful window size for the particular connection.

In order to keep pace with the growth of the sender's congestion window during slow-start, the receiver should use the same doubling factor. Thus the receiver should advertise a window that is twice the size of the last measured window size.

C. Window scaling

TCP only has a 16-bit field for window size. Newer implementations support Window Scaling [6] in which a binary shift factor (between 0 and 14 bits) is negotiated at connection setup for each end-point. For the remainder of the connection, this shift is implicitly applied to all windows advertised by that end-point.

However, there will be a resulting loss of granularity in expressing window sizes. For example, if the maximum window scaling of 14 bits is used, window sizes of 1 gigabyte can be represented, but the granularity will be 16 kilobytes. This makes it impossible to safely advertise a window size of less than 16 kilobytes as anything other than 0. Consequently, the receiver should use a window that is substantially larger than the quantum specified by the scaling factor.

In order to support dynamically sized buffers, the receiver must request, at connection setup, a window scaling sufficient to represent the largest buffer size that it wishes to be able to use. It must balance this scaling with the ability to represent smaller window sizes.

IV. Implementation

The algorithms described above were implemented in a Linux 2.2.12 kernel. To make the receive throughput measurements, two variables are added to kernel data structures for each connection. These variables are used to keep track of the time of the beginning of the measurement and the next sequence number that was expected at that time.

Upon the arrival of each TCP packet, the current time is compared to the last measurement time for that connection. If more than the current, smoothed, round-trip time has passed, the highest sequence number seen (not including the current packet) is compared to the next sequence number expected at the beginning of the measurement. Assuming that all packets are received in order, the result is the number of bytes that were received during the period. Packets that are received out of order may have lowered the goodput during this measurement, but will increase the goodput of the following measurement which, if larger, will supersede this measurement.

The upper bound placed on the receive buffer space for that connection is then increased, if necessary, to make sure that the next window advertised will be at least twice as large as the the amount of data received during the last measurement period.

A. Receive buffer

The Linux 2.2 kernel advertises a receive window that is half as large as the largest permitted receive buffer for that

connection. Since data may come from the sender in packets with payloads anywhere from 1 byte to the maximum segment size, the receiver can never quite be certain how much buffer space will be used for a given amount of receive window. In practice, high-performance connections use large packets with relatively low storage overhead for headers. Since the right-sizing algorithm described above already keeps the buffer size twice as large as the maximum amount of data received during a round-trip time, the amount of buffer space allocated is actually four times the last measurement. This is a short coming in the Linux kernel and is addressed in the Linux 2.4 kernels which advertise more of their buffer space (the exact fraction is tunable with the `tcp_app_win` parameter). Dynamic right-sizing works equally well in either case.

In addition, the Linux kernel uses the `netdev_max_backlog` parameter to limit the number of received packets that have not been processed yet. This value defaults to 300 and was smaller than the sender's congestion window during the experiments shown below. Due to the bursty nature of TCP and the ability of gigabit Ethernet to deliver packets very fast, we believe that we were passing this limit and changed the value to 3000.

Since the application may not have been ready to send for the entire time period, the estimated window size may be smaller than the actual window. We therefore use the largest measured seen during any round-trip interval. This increasing lower-bound is used to estimate the largest window that the sender may use.

To develop a useful upper-bound on the round-trip time, we keep track of the minimum round-trip time observed using passive measurement by the receiver. This minimum is used only when there is no smoothed, round-trip time (srtt) available from the standard measurements made when transmitting data.

B. Impact of Timer Imprecision

The primary system timer in Linux is *jiffies* which is a global variable incremented periodically as a result of timer interrupts. This value can be efficiently obtained during packet handling, but is limited to a precision of 10ms in a standard kernel. In this section we explore the impact of using this efficient, but imprecise measure of time.

Assume the salient scenario where a sender is always sending as fast as possible, but that the network may delay packets arbitrarily. However, we are concerned with the case were the sender is limited by the window-size, so that network delays are small enough that no timeouts occur. Further assume that the window-size is not fluctuating during a measurement. Then the receiver can determine that the window size limiting the sender is bounded as follows:

$$\frac{d}{n_{max}} \leq w \leq \frac{d}{n_{min}} \tag{1}$$

Where d bytes of data have been received over some number of round-trip times between n_{min} and n_{max}.

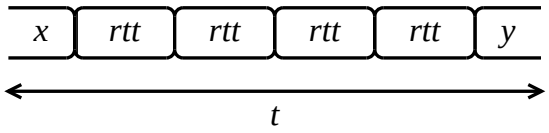

Fig. 1
COMPOSITION OF A SAMPLING PERIOD $(4 < n \leq 6)$

Any measurement period consists of some number of whole round-trip times plus fractional round-trip times preceding and following the complete round-trips. These fractions may be of any duration less than the round trip time. For example, Figure 1 shows a period that must consist of between 4 and 6 round-trip periods.

The possible number of round-trip periods observed is a whole number bounded as follows:

$$\left\lceil \frac{t}{rtt} \right\rceil \leq n < \left\lfloor \frac{t}{rtt} \right\rfloor + 2 \tag{2}$$

Due to similar fence-post problems, a measurement of duration equal to one round-trip time, but measured in jiffies, may actually be up to 20ms longer than the round-trip time:

$$rtt \leq t \leq rtt + 20\mathrm{m}s \tag{3}$$

Combining constraints 3 and 2, we can bound n as follows:

$$\left\lceil \frac{rtt}{rtt} \right\rceil \leq n < \left\lfloor \frac{rtt + 20\mathrm{m}s}{rtt} \right\rfloor + 2 \tag{4}$$

Which reduces to:

$$1 \leq n < \left\lfloor \frac{20\mathrm{m}s}{rtt} \right\rfloor + 3 \tag{5}$$

Further substituting the min and max values of n into bound 1 yields:

$$\frac{d}{\left\lfloor \frac{20\mathrm{m}s}{rtt} \right\rfloor + 3} \leq w \leq d \tag{6}$$

Thus, in no case will the actual window be larger than the measured amount of data received during the period. However, the amount of data received during the period may be three times the actual window size when measurements are made across wide-area networks with $rtt > 20$ms. Further, local networks with small round-trip delays may be grossly over-estimated.

We therefore conclude that measurements made with coarse timers will not cause dynamic right-sizing to underestimate the window size or negatively impact throughput. However, to make more accurate decisions for memory sharing under pressure, it is advantageous to use precise timers. Many CPU architectures now feature hardware time counters that can be used to efficiently obtain a precise timestamp. In future versions of our implementation, we will pursue the use of these counters. However, our use of these counters will also require that the standard TCP round-trip time estimation be done with equivalent precision.

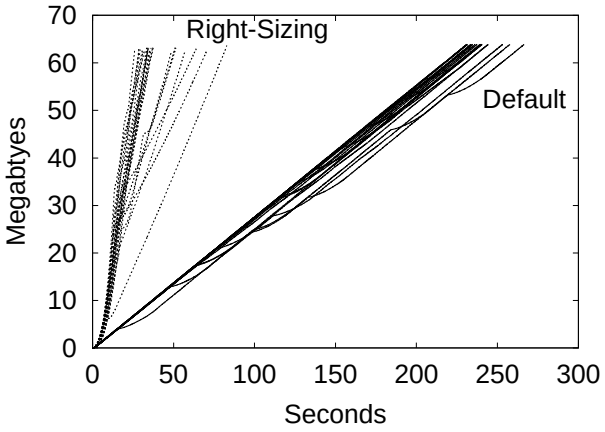

Fig. 2
PROGRESS OF DATA TRANSFERS

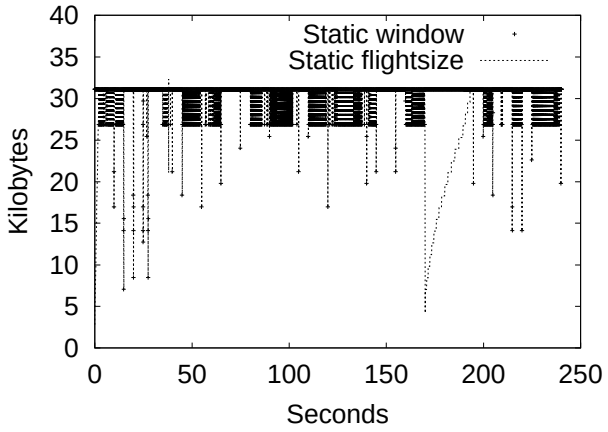

Fig. 3
DEFAULT WINDOW SIZE: FLIGHT & WINDOW SIZES

C. Linux 2.4

It is worth noting that while our implementation was for Linux 2.2.12, the same algorithms can be easily implemented in the newer 2.4 kernels. The 2.4 kernels contain new, complementary features designed to reduce memory usage on busy web servers which are transmitting data on large numbers of network-bound TCP connections. Under normal circumstances, the 2.4 kernels restrict each connection's send buffers to be just large enough to fill the current congestion window. When total memory usage is above some threshold, the memory used by each connection is further restrained. Thus while Linux 2.4 precisely bounds send buffers, Dynamic Right-Sizing precisely bounds receiver-side send buffers.

V. EXPERIMENTAL RESULTS

In this section we demonstrate that the receiver's estimate of the sender's congestion window does approximate the actual size. Further we show that by using this estimate to size window advertisements, we keep the connection congestion window-limited rather than receive window-limited.

As expected, the use of larger windows increases performance compared to using smaller, default window sizes. In Figure 2, 50 transfers of 64 megabytes each were made with the `ttcp` program. The first 25 transfers used the default window sizes of 64 kilobytes for both the sender and receiver. The second 25 transfers, shown in dotted lines, used the dynamically sized windows described above. Both end points have gigabit Ethernet interfaces separated by a WAN emulator that introduces a 100ms delay in the round-trip time. The congestion control mechanisms triggered by packet losses cause the abrupt decreases in slope. The impact of this packet loss has equivalent, inconsequential, effects on the performance of both cases.

A. Performance

The transfers made with default window sizes took a median time of 240 seconds to complete. The transfers with dynamic windows sizes were roughly 7 times faster and took a median time of 34 seconds.

In Figures 3 and 4, we examine the window sizes during two of the above transfers. The amount of sent, but unacknowledged data in the sender's buffer is known as the flightsize. The flightsize is in turn bounded by the window advertised by the receiver.

Figure 3 shows that in the traditional, static case without dynamic right-sizing, the congestion window, and consequently flightsize quickly grow equal to the size of the window advertisements. For most of the duration of the connection, it is limited by the receiver's low window advertisement of 32KB.

In contrast, during the dynamic right-sizing case shown in Figure 4, the receiver is able, during most of the connection, to advertise a window size that is roughly twice the largest flightsize seen to date. As a result, the flightsize is only constrained by the congestion window and the delay-bandwidth product. Slow start continues for much longer and stops only when there is packet loss. At this point the congestion window stabilizes on a flightsize that is 7 times higher than the constrained flightsize of the static case. This 7-fold increase in the average flightsize is the source of the same, 7-fold increase in throughput demonstrated in Figure 2.

Other tests did occasionally see increased queuing delay caused by the congestion window growing larger than the available bandwidth. At this point the retransmit timer expired and reset the congestion window even though the original transmission of the packet was acknowledged shortly thereafter. In this case, the congestion window is reset to 1 rather than just performing a normal multiplicative decrease.

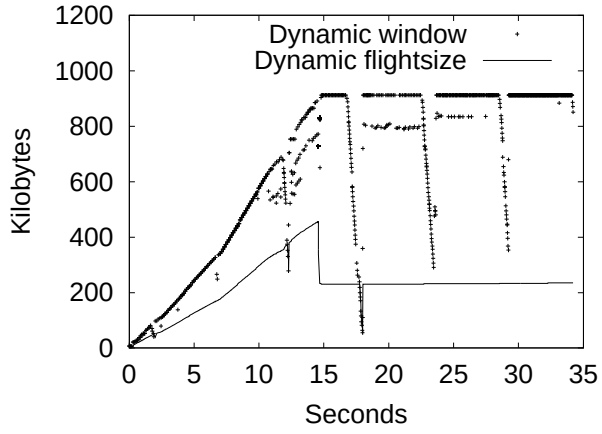

Fig. 4
DYNAMIC RIGHT-SIZING: FLIGHT & WINDOW SIZES

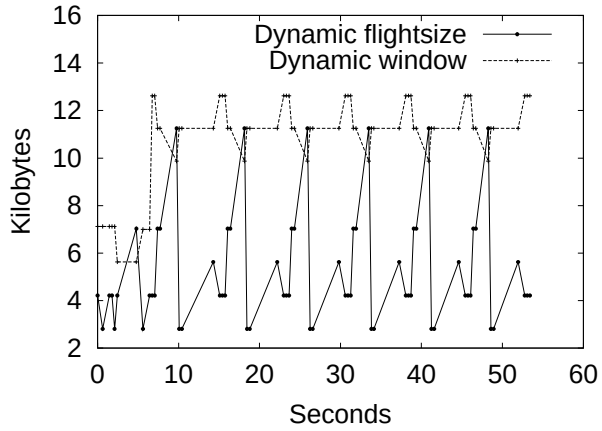

Fig. 5
LOW-BANDWIDTH LINKS: DYNAMIC CASE

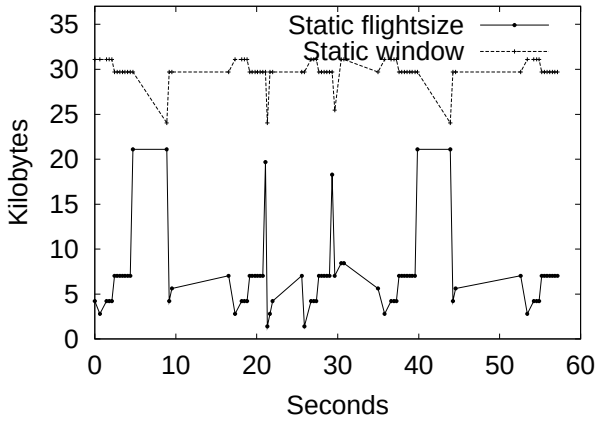

Fig. 6
LOW-BANDWIDTH LINKS: STATIC CASE

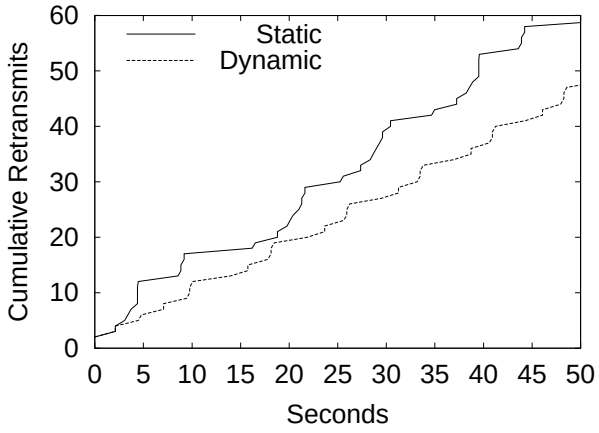

Fig. 7
LOW-BANDWIDTH LINKS: RETRANSMITS

B. Low-Bandwidth Links

In this section, we demonstrate that because dynamic right-sizing provides the sender with feedback about the achieved throughput rate, it actually causes a TCP Reno sender to induce less congestion and fewer retransmissions over bandwidth-limited connections. Figure 5 shows the first part of the same transfer over a link that is simulated to be only 56 kilobits. Here we see that the largest advertised window is under 13 kilobytes.

Figure 6 shows what happens over the same link when default window sizes are used. Other measurements show that the two cases get virtually identical throughput. Yet, the static case appears to usually have more data in flight. As evidenced by the roughly 20% increase in the number of retransmissions shown in Figure 7, this additional data in flight is dropped because the link cannot support that throughput.

In the static case, the sender has no way to estimate the capacity of the link other than to fill the network and its queues and induce packet loss. Using Dynamic Right-sizing, however, the sender gets feedback, in the form of window advertisements, about the actual rate at which packets are received. Because the receiver observes that the throughput is already much less than its window size, it does not increase its window advertisement and the sender does not double the number of packets it will send. All this would achieve is greater queueing delay and probable packet loss.

VI. CONCLUSIONS

The capability for scaling TCP's flow control windows has existed for several years, but the use of this scaling has remained dependent on manual tuning. To address this problem, we have developed a general method for automatically scaling the flow control window in TCP. This scalability allows end systems to automatically support increased performance over high delay-bandwidth networks. In turn, we remove one of the network roadblocks that hinder the use of

present and future distributed computing environments, grid applications, and multi-media.

It has been demonstrated that our method can successfully grow the receiver's advertised window at a pace sufficient to avoid constraining the sender's throughput. For the particular circumstances tested, a 7-fold speedup was achieved. Networks capable of supporting higher throughput would scale equally well. Meanwhile, the receiver remains in full control of the flow-control mechanisms. As a result, implementations that wish to guarantee window availability have the necessary information to strictly allocate buffers or control the degree to which they are over-committed. Additionally, network connections with small delay-bandwidth products are identified and the receiver can avoid allocating unnecessarily large amounts of buffer space for these connections. Thus, our work complements previous work on buffer sharing [13]. In contrast, however, our work preserves the semantics and common usage of TCP flow control and better supports applications that depend on flow control because of performance disparities between the sender and receiver.

Finally, we showed a case where the use of observed throughput information to size windows constrains the sender from raising its congestion window too high. This prevents some unnecessary queueing delay and packet loss. The result is far fewer retransmissions.

In summary, we have shown that TCP flow control can be automatically adapted to support the large delay-bandwidth network connections needed by computational, data, and access grids. In contrast to previous work in this area, we show that operating systems can automatically determine the flow-control requirements of individual connections. As a result, systems can avoid the network performance problems that result from either the under-utilization or over-utilization of buffer space.

REFERENCES

[1] Ann Chervenak, Ian Foster, Carl Kesselman, Charles Salisbury, and Steven Tuecke, "The data grid: Towards an architecture for the distributed management and analysis of large scientific datasets," *Journal of Network and Computer Applications*, vol. 23, no. 3, pp. 187–200, July 2000.

[2] Lisa Childers, Terry Disz, Robert Olson, Michael E. Papka, Rick Stevens, and Tushar Udeshi, "Access grid: Immersive group-to-group collaborative visualization," in *Proceedings of the 4th International Immersive Projection Technology Workshop*, 2000.

[3] "EU-data grid," http://www.eu-datagrid.org/.

[4] Wu-chun Feng and Peerapol Tinnakornsrisuphap, "The failure of TCP in high-performance computational grids," in *SC2000*, Nov. 2000.

[5] Ian Foster and Carl Kesselman, *The Grid: Blueprint for a New Computing Infrastructure*, Morgan-Kaufmann, 1998.

[6] V. Jacobson, R. Braden, and D. Borman, "RFC 1323: TCP extensions for high performance," May 1992.

[7] Van Jacobson, "Congestion Avoidance and Control," in *Proceedings, SIGCOMM '88 Workshop*. ACM SIGCOMM, Aug. 1988, pp. 314–329, ACM Press, Stanford, CA.

[8] Jason Lee, Dan Gunter, Brian Tierney, Bill Allcock, Joe Bester, John Bresnahan, and Steve Tuecke, "Applied techniques for high bandwidth data transfers across wide area networks," Tech. Rep. LBNL-46269, Lawrence Berkeley National Laboratory, Dec. 2000.

[9] Jian Lui and Jim Ferguson, "Automatic TCP socket buffer tuning," in *SC 2000 Research Gems*, Nov. 2000, Awarded "Best Research Gem of the Conference," http://dast.nlanr.net/Features/Autobuf/.

[10] Jamshid Mahdavi, "Enabling high performance data transfers on hosts," Webpage, http://www.psc.edu/networking/perf_tune.html.

[11] Matt Mathis, Jamshid Mahdavi, Sally Floyd, and Allyn Romanow, "RFC 2018: TCP selective acknowledgment options," Oct. 1996, Internet Engineering Task Force.

[12] "Particle physics data grid," http://www.cacr.caltech.edu/ppdg/.

[13] Jeff Semke, Jamshid Mahdavi, and Matt Mathis., "Automatic TCP buffer tuning," *Computer Communications Review*, vol. 28, no. 4, pp. 315–323, Oct. 1998.

[14] Brian L. Tierney, "TCP tuning guide for distributed application on wide area networks," *;login*, vol. 26, no. 1, Feb. 2001.