Dynamic Right-Sizing: An Automated, Lightweight, and Scalable Technique for Enhancing Grid Performance

Wu-chun Feng, Mike Fisk, Mark Gardner, and Eric Weigle

Research & Development in Advanced Network Technology (RADIANT),
Los Alamos National Laboratory, P.O. Box 1663, Los Alamos, NM 87545,
{feng,mfisk,ehw,mkg}@lanl.gov
http://www.lanl.gov/radiant

Abstract. With the advent of computational grids, networking performance over the wide-area network (WAN) has become a critical component in the grid infrastructure. Unfortunately, many high-performance grid applications only use a small fraction of their available bandwidth because operating systems and their associated protocol stacks are still tuned for yesterday's WAN speeds. As a result, network gurus undertake the tedious process of manually tuning system buffers to allow TCP flow control to scale to today's WAN grid environments. And although recent research has shown how to set the size of these system buffers automatically at connection set-up, the buffer sizes are only appropriate at the beginning of the connection's lifetime. To address these problems, we describe an automated and lightweight technique called dynamic right-sizing that can improve throughput by as much as an order of magnitude while still abiding by TCP semantics.

1 Introduction

TCP has entrenched itself as the ubiquitous transport protocol for the Internet as well as emerging infrastructures such as computational grids [1,2], data grids [3,4], and access grids [5]. However, parallel and distributed applications running stock TCP implementations perform abysmally over networks with large bandwidth-delay products. Such large bandwidth-delay product (BDP) networks are typical in grid-computing networks as well as satellite networks [6,7,8].

As noted in [6,7,8,9], adaptation bottlenecks are the primary reason for this abysmal performance, in particular, flow-control adaptation and congestion-control adaptation. In order to address the former problem,[1] grid and network researchers continue to manually tune buffer sizes to keep the network pipe full [7,10,11], and thus achieve acceptable wide-area network (WAN) performance in support of grid computing. However, this tuning process can be quite difficult, particularly for users and developers who are not network experts, because it involves calculating the bandwidth of the bottleneck link and the round-trip

[1] The latter problem is beyond the scope of this paper.

© Springer-Verlag Berlin Heidelberg 2002

time (RTT) for a given connection. That is, the optimal TCP buffer size is equal to the product of the bandwidth of the bottleneck link and the RTT, i.e., the bandwidth-delay product of the connection.

Currently, in order to tune the buffer sizes appropriately, the grid community uses diagnostic tools to determine the RTT and the bandwidth of the bottleneck link at any given time. Such tools include *iperf* [12], *nettimer* [13], *netspec* [14], *nettest* [15], *pchar* [16], and *pipechar* [17]. However, none of these tools include a client API, and all of the tools require a certain level of network expertise to install and use.

To simplify the above tuning process, several services that provide clients with the correct tuning parameters for a given connection have been proposed, e.g., AutoNcFTP [18], Enable [19], Web100 [20], to eliminate what has been called the *wizard gap* [21].[2] Although these services provide good first approximations and can improve overall throughput by two to five times over a stock TCP implementation, they only measure the bandwidth and delay at connection set-up time, thus making the implicit assumption that the bandwidth and RTT of a given connection will not change significantly over the course of the connection. In Section 2, we demonstrate that this assumption is tenuous at best. In addition, these services "pollute" the network with extraneous probing packets.

A more dynamic approach to optimizing communication in a grid involves automatically tuning buffers over the lifetime of the connection, not just at connection set-up. At present, there exist two kernel-level implementations: auto-tuning [22] and dynamic right-sizing (DRS) [23,24]. Auto-tuning implements sender-based, flow-control adaptation while DRS implements receiver-based, flow-control adaptation.[3] Live WAN tests show that DRS in the kernel can achieve a 30-fold increase in throughput when the network is uncongested, although speed-ups of 7-8 times are more typical. (And when the network is heavily congested, DRS throttles back and performs no better than the default TCP.) However, achieving such speed-ups requires that our kernel patch for DRS be installed in the operating systems of every pair of communicating hosts in a grid.[4]

The installation of our DRS kernel patch requires knowledge about adding modules to the kernel and *root* privilege to install the patch. Thus, the DRS functionality is generally not accessible to the typical end user (or developer). However, in the longer term, we anticipate that this patch will be incorporated into the kernel core so that its installation and operation are transparent to the end user. In the meantime, end users still demand the better performance of DRS but with the pseudo-transparency of Enable and AutoNcFTP. Thus, we propose a more portable implementation of DRS in *user space* that is transpar-

[2] The *wizard gap* is the difference between the network performance that a network "wizard" can achieve by appropriately tuning buffer sizes and the performance of an untuned application.

[3] The Web100 project recently incorporated DRS into their software distribution to enable the dynamic sizing of flow-control windows over the lifetime of a connection [25].

[4] Once installed, not only do grids benefit, but every TCP-based application benefits, e.g., *ftp*, multimedia streaming, WWW.

ent to the end user. Specifically, we integrate our DRS technique into *ftp* (drsFTP). The differences between our drsFTP and AutoNcFTP are two-fold. First, AutoNcFTP relies on NcFTP (http://www.ncftp.com/) whereas drsFTP uses a de-facto standard *ftp* daemon from Washington University (http://www.wu-ftpd.org/). Second, the buffers in AutoNcFTP are only tuned at connection set-up while drsFTP buffers are dynamically tuned over the lifetime of the connection, thus resulting in better adaptation and better overall performance.

The remainder of the paper is organized as follows. Section 2 demonstrates why dynamic, flow-control adaptation is needed over the lifetime of the connection rather than at connection set-up only. Sections 3 and 4 describe the DRS technique and its implementation in kernel space and in user space, respectively. Then, in Section 5, we present our experimental results, followed by concluding remarks in Section 6.

2 Background

TCP relies on two mechanisms to set its transmission rate: flow control and congestion control. Flow control ensures that the sender does not overrun the receiver's available buffer space (i.e., a sender can send no more data than the size of the receiver's last advertised flow-control window) while congestion control ensures that the sender does not unfairly overrun the network's available bandwidth. TCP implements these mechanisms via a flow-control window ($fwnd$) that is advertised by the receiver to the sender and a congestion-control window ($cwnd$) that is adapted based on inferring the state of the network.

Specifically, TCP calculates an effective window ($ewnd$), where $ewnd \equiv min(fwnd, cwnd)$, and then sends data at a rate of $ewnd/RTT$, where RTT is the round-trip time of the connection. Currently, $cwnd$ varies dynamically as the network state changes; however, $fwnd$ has always been static despite the fact that today's receivers are not nearly as buffer-constrained as they were twenty years ago. Ideally, $fwnd$ should vary with the bandwidth-delay product (BDP) of the network, thus providing the motivation for DRS.

Historically, a static $fwnd$ sufficed for all communication because the BDP of networks was small. Hence, setting $fwnd$ to small values produced acceptable performance while wasting little memory. Today, most operating systems set $fwnd \approx 64$ KB – the largest window available without scaling [26]. Yet BDPs range between a few bytes (56 Kbps $\times$ 5 ms $\rightarrow$ 36 bytes) and a few megabtyes (622 Mbps $\times$ 100 ms $\rightarrow$ 7.8 MB). For the former case, the system wastes over 99% of its allocated memory (i.e., 36 B / 64 KB = 0.05%). In the latter case, the system potentially wastes up to 99% of the network bandwidth (i.e., 64 KB / 7.8 MB = 0.80%).

Over the lifetime of a connection, bandwidth and delay change (due to transitory queueing and congestion) implying that the BDP also changes. Figures 1, 2, and 3 support this claim.[5] Figure 1 presents the bottleneck bandwidth between Los Alamos and New York at 20-second intervals. The bottleneck bandwidth

[5] To generate these figures, we used *nettimer* to measure bandwidth and RTT delay.

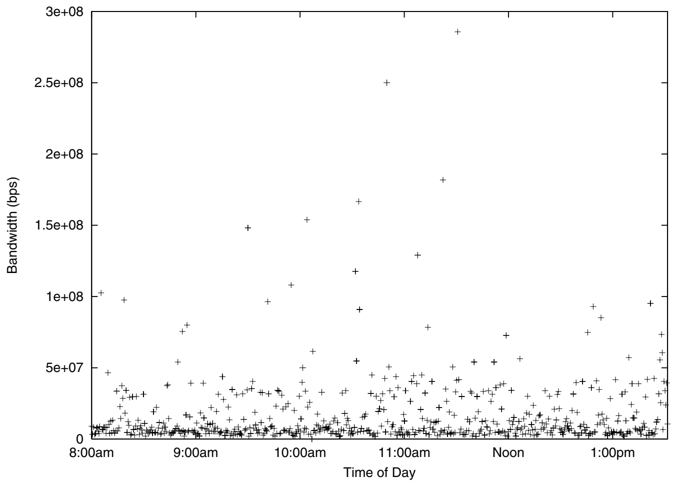

Fig. 1. Bottleneck Bandwidth at 20-Second Intervals

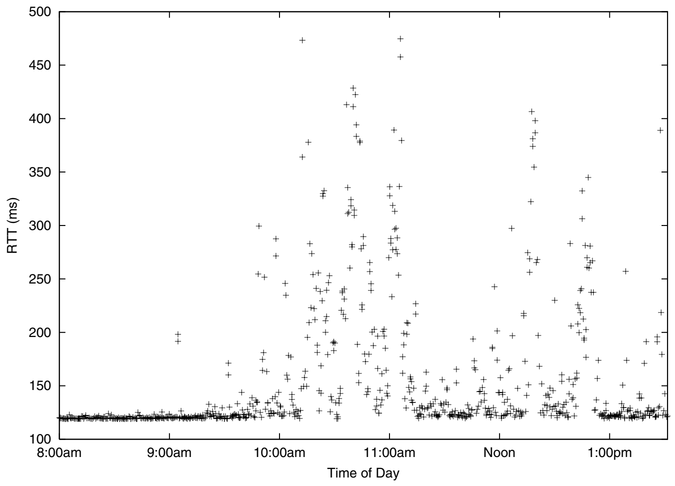

Fig. 2. Round-Trip Time at 20-Second Intervals

averages 17.2 Mbps with a low and a high of 0.026 Mbps and 28.5 Mbps, respectively. The standard deviation and half-width of the 95% confidence interval are 26.3 Mbps and 1.8 Mbps. Figure 2 shows the RTT, again between Los Alamos and New York, at 20-second intervals. The RTT delay also varies over a wide range of [119, 475] ms with an average delay of 157 ms. Combining Figures 1 and 2 results in Figure 3, which shows that the BDP of a given connection can vary by as much as 61 Mb.

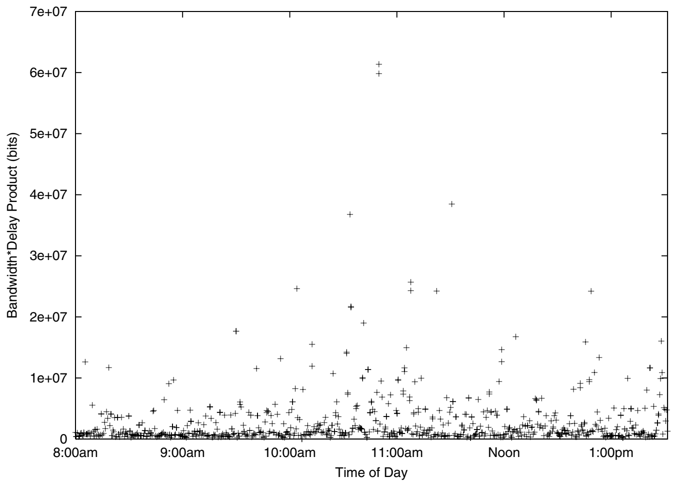

Fig. 3. Bandwidth-Delay Product at 20-Second Intervals

Based on the above results, the BDP over the lifetime of a connection is continually changing. Therefore, a fixed value for $fwnd$ is not ideal; selecting a fixed value forces an implicit decision between (1) under-allocating memory and under-utilizing the network or (2) over-allocating memory and wasting system resources. Clearly, the grid community needs a solution that dynamically and transparently adapts $fwnd$ to achieve good performance without wasting network or memory resources.

3 Dynamic Right-Sizing (DRS) in the Kernel

Dynamic right-sizing (DRS) lets the receiver estimate the sender's *cwnd* and then use that estimate to dynamically change the size of the receiver's window advertisements $fwnd$ (as memory resources will allow on the receiver side). These updates can then be used to keep pace with the growth in the sender's congestion window. As a result, the throughput between end hosts, e.g., as in a grid, will only be constrained by the available bandwidth of the network rather than some arbitrarily set constant value on the receiver that is advertised to the sender.

Initially, at connection set-up, the sender's *cwnd* is smaller than the receiver's advertised window $fwnd$. To ensure that a given connection is not flow-control constrained, the receiver must continue to advertise a $fwnd$ that is larger than the sender's *cwnd* before the receiver's next adjustment.

The instantaneous throughput seen by a receiver may be larger than the available end-to-end bandwidth. For instance, data may travel across a slow link only to be queued up on a downstream 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 *cwnd* and the window advertised by the receiver. Because the sender can send no more than one *cwnd* window's worth of data between acknowledgements, a burst that is shorter than a RTT can contain at most one *cwnd*'s worth of data. Thus, for any period of time that is shorter than a RTT, the amount of data seen over that period is a *lower bound* on the size of the sender's *cwnd*. But how does such a distributed system calculate its RTT?

In a typical TCP implementation, the RTT is measured by observing the time between when data is sent and an acknowledgement is returned. However, during a bulk-data transfer (e.g., from sender to receiver), the receiver may not be sending any data, and therefore, will not have an accurate RTT estimate. So, how does the receiver infer delay (and bandwidth) when it only has acknowledgements to transmit back and no data to send?

A receiver in a computational grid that is only transmitting acknowledgements can still estimate the RTT 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 sending application does not have any data to transmit, the measured RTT could be much larger than the actual RTT. Thus, this measurement acts as an *upper bound* on the RTT and should be used only when it is the only source of RTT information.

For a more rigorous and mathematical presentation for the lower and upper bounds that are used in our kernel implementation of DRS, please see [23,24].

4 DRS in User Space: drsFTP

Unlike the kernel-space DRS, user-space DRS implementations are specific to a particular application. Here, we integrate DRS into `ftp`, resulting in drsFTP. As with AutoNcFTP and Enable, we focus on (1) adjusting TCP's system buffers over the data channel of `ftp` rather than the control channel and (2) using `ftp`'s stream file-transfer mode. The latter means that a separate data connection is created for every file transferred. Thus, during the lifetime of the transfer, the sender *always* has data to transmit; once the file has been completely sent, the data connection closes. We leverage the above information in our design of drsFTP.

The primary difficulty in developing user-space DRS code lies in the fact that user-space code generally does *not* have direct access to the high-fidelity information available in the TCP stack. Consequently, drsFTP has no knowledge of parameters generated by TCP such as the RTT of a connection or the receiver's advertised window.

4.1 Adjusting the Receiver's Window

Because the receiver is running in user space, it is unable to determine the actual round trip time of TCP packets. However, in developing drsFTP, we do know that the sender always has data to send for the life of the connection. It then follows that the sender will send as much data as possible, limited by its idea of congestion- and flow-control windows. So, the receiver can assume that it is receiving data as quickly as the current windows and network conditions allow.

We use this assumption in the following manner. The drsFTP application maintains an application receive buffer of at least twice the size of the current kernel TCP receive buffer (which can be obtained via the `getsockopt()` function call with the `TCP_RCVBUF` parameter). Every time the application reads from the network, it attempts to read an entire receive buffer's worth of data. If more data than some threshold value is read, the assumption can be made that the flow-control window should be increased.

The threshold value depends on the operating system (OS), in particular, how much of the TCP kernel buffer that the OS reserves as application buffer space and how much is used for the TCP transfer (i.e., how much of the buffer is used for the TCP receive window). The threshold value is always less than the value reported by `TCP_RCVBUF`. For example, we based our tests of drsFTP on the Linux operating system. Linux maintains half of the TCP buffer space as the TCP receive window and half as buffer area for the application. Thus, the threshold value we used was $\frac{1}{2}$ the total reported by `TCP_RCVBUF` size.

In the worst case, the sender's window is doubling with every round trip (i.e., during TCP slow start). Thus, when the determination is made that the receiver window should increase, the new value should be at least double the current value. In addition, the new value should take the threshold value into consideration. Thus, for our drsFTP implementation, we increase the receive window by a factor of four. This factor is applied to both the application buffer and the kernel buffer (via `setsockopt()/TCP_RCVBUF`).

4.2 Adjusting the Sender's Window

In order to take full advantage of dynamically changing buffer sizes, the sender's buffer should adjust in step with the receiver's. This presents a problem in user-space implementations because the sender's user-space code has no way of determining the receiver's advertised TCP window size. However, the `ftp` protocol specification [27] provides a solution to this dilemma. Specifically, `ftp` maintains a control channel, which is a TCP connection completely separate from the actual data transfer. Commands are sent from the client to the server over this control channel, and replies are sent in the reverse direction. Additionally, the `ftp` specification does not prohibit traffic on the control channel during data transfer. Thus, a drsFTP receiver may inform a drsFTP sender changes in buffer size by sending appropriate messages over the `ftp` control channel.

Since `ftp` is a bidirectional data-transfer protocol, the receiver may either be the `ftp` server or client. However, RFC 959 specifies that only `ftp` clients may send commands on the control channel, while `ftp` servers may only send replies to commands. Thus, a new `ftp` command and reply must be added to the `ftp` implementation in order to fully implement drsFTP. Serendipitously, the Internet Draft of the GridFTP protocol extensions to `ftp` defines an `ftp` command "SBUF", which is designed to allow a client to set the server's TCP buffer sizes before data transfer commences. We extend the definition of SBUF to allow this command to be specified during a data transfer, i.e., to allow buffer sizes to be set dynamically. The full definition of the expanded SBUF command appears below:

Syntax:

```
sbuf = SBUF <SP> <buffer-size>
buffer-size ::= <number>
```

This command informs the server-PI to set the TCP buffer size to the value specified (in bytes). `SBUF` may be issued at any time, including before or during active data transfer. If specified during data transfer, it affects the data transfer that started most recently.

Response Codes:

If the server-PI is able to set the buffer size to the requested buffer size, a 200 response code may be returned. No response code is necessary if specified during a data transfer, but a response is required if specified outside of the data transfer.

In addition, we propose a new reply code to allow the server-as-receiver to notify the client of changes in the receiver window.

```
126 Buffer Size (xxx)
 xxx ::= buffer size in bytes
```

The 126 Reply may occur at any point when the server-PI is sending data to the user-PI (or a server-PI running concurrently with the user-PI). As with the `SBUF` command during data transfer, this reply is informational and need not be acted upon or responded to in any manner.

This reply code is consistent with RFC 959 and does not interfere with any `ftp` extension or proposed extension.

4.3 TCP Window Scaling

Because the window-scaling factor in TCP is established at connection set-up time, an appropriate scale must be set before a new data connection is opened. Most operating systems allow `TCP_RCVBUF` and `TCP_SNDBUF` to be set on a socket before a connection attempt is made and then use the requested buffer size to establish the TCP window scaling.

5 Experiments

In this section, we present results for both the kernel- and user-space implementations of DRS. In particular, we will show that the throughput for both the kernel- and user-space implementations improves upon the default configuration by 600% and 300%, respectively. The kernel implementation performs better because it has access to fine-granularity information in the kernel and has two fewer copies to perform than drsFTP.

5.1 Experimental Setup

Our experimental apparatus, shown in Figure 4, consists of three identical machines connected via Fast Ethernet. Each machine contains a dual-CPU 400-MHz Pentium II with 128-MB of RAM and two network-interface cards (NICs). One machine acts as a WAN emulator with a 100-ms round-trip time (RTT) delay; each of its NICs is connected to one of the other machines via crossover cables (i.e., no switch).

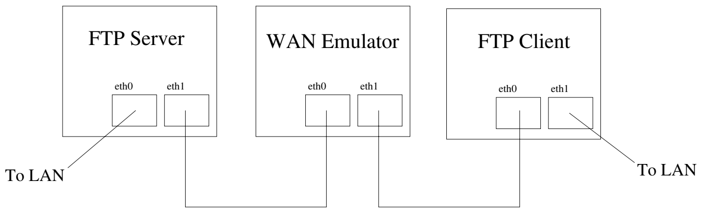

Fig. 4. Experimental Setup

5.2 Kernel-Space DRS

In the kernel implementation of DRS, the receiver estimates the size of the sender's congestion window so it can advertise an appropriate flow-control window to the sender. Our experiments show that the DRS algorithm approximates the actual size quite well. Further, we show that by using this estimate to size the window advertisements, DRS keeps the connection congestion-control limited rather than (receiver) flow-control limited.

Performance. As expected, using larger flow-control windows significantly enhances WAN throughput versus using the default window sizes of TCP. Figure 5 shows the results of 50 transfers of 64 MB of data with `ttcp`, 25 transfers using the default window size of 32 KB for both the sender and receiver and 25 transfers using DRS. Transfers with the default window sizes took a median time of 240 seconds to complete while the DRS transfers only took 34 seconds (or roughly *seven* times faster).

Figures 6 and 7 trace the window size and flight size of the default-sized TCP and the DRS TCP. (The flight size refers to the amount of sent but unacknowledged data in the sender's buffer. This flight size, in turn, is bounded by the window advertised by the receiver.) For the traditionally static, default,

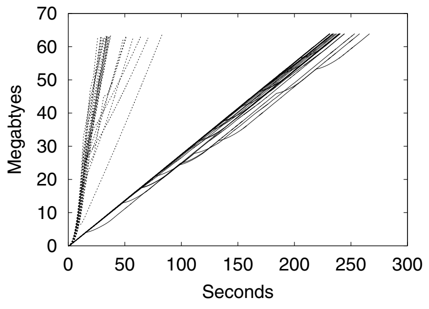

Fig. 5. Progress of Data Transfers

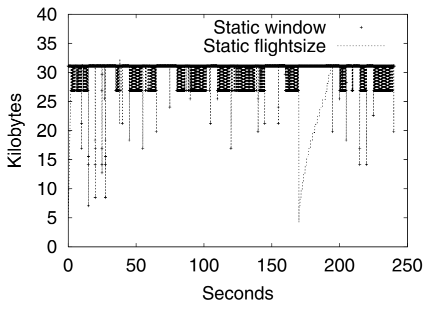

Fig. 6. Default Window Size: Flight & Window Sizes

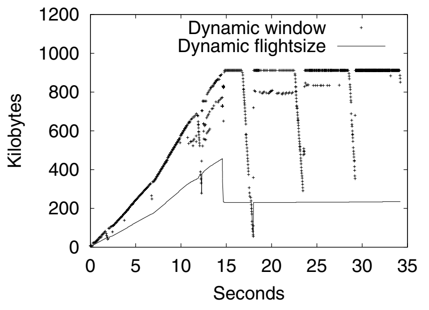

Fig. 7. Dynamic Right-Sizing: Flight & Window Sizes

flow-control window as shown in Figure 6; the congestion window quickly grows and becomes rate-limited by the receiver's small 32-KB advertisement for the flow-control window. On the other hand, DRS allows the receiver to advertise a window size that is roughly twice the largest flight size seen to date (in case, the connection is in slow start). Thus, the flight size is only constrained by the conditions in the network, i.e., congestion window. Slow start continues for much longer and stops only when packet loss occurs. At this point, the congestion window stabilizes on a flight size that is roughly seven times higher than the constrained flight size of the static case. And not coincidentially, this seven-fold increase in the average flight size translates into the same seven-fold increase in throughput shown in Figure 5.

In additional tests, we occasionally observe increased queueing delay caused by the congestion window growing larger than the available bandwidth. This causes the retransmit timer to expire and reset the congestion window to one even though the original transmission of the packet was acknowledged shortly thereafter.

Low-Bandwidth Connections. Figures 8 and 9 trace the window size and flight size of default-sized TCP and DRS TCP over a 56K modem. Because DRS provides the sender with indirect feedback about the achieved throughput rate, DRS actually causes a TCP Reno sender to induce *less* congestion and fewer retransmissions over bandwidth-limited connections. Although the overall throughput measurements for both cases are virtually identical, the static (default) window generally has more data in flight as evidenced by the roughly 20% increase in the number of re-transmissions shown in Figure 10. This ad-

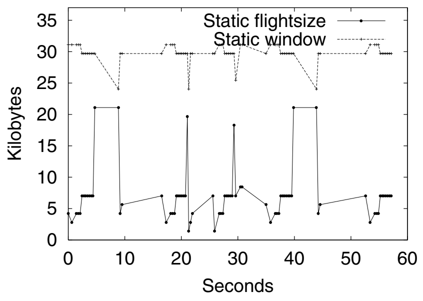

Fig. 8. Default Window Size: Low-Bandwidth Links (56K Modem)

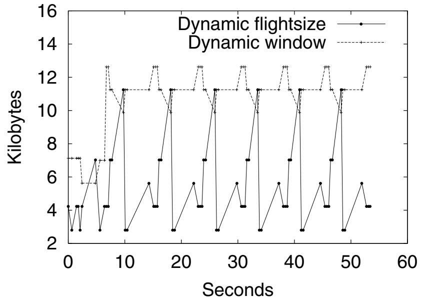

Fig. 9. Dynamic Right-Sizing: Low-Bandwidth Links (56K Modem)

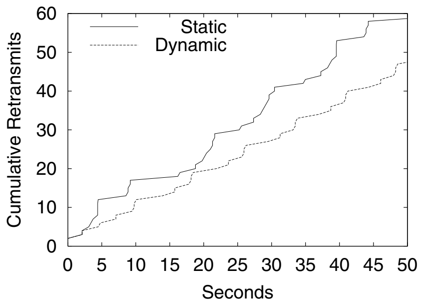

Fig. 10. Retransmissions in Low-Bandwidth Links

ditional data in flight is simply dropped because the link cannot support that throughput.

Discussion. The Linux 2.4.x kernel contains complementary features to DRS that are designed to reduce memory usage on busy web servers that are transmitting data on large numbers of network-bound TCP connections. Under normal circumstances, the Linux 2.4 kernel restricts each connection's send buffers to be just large enough to fill the current congestion window. When the total memory used exceeds a threshold, the memory used by each connection is further constrained. Thus, while Linux 2.4 precisely bounds send buffers, DRS precisely bounds receiver-side send buffers.

5.3 drsFTP: DRS in User Space

For each version of `ftp` (drsFTP, stock FTP, statically-tuned FTP), we transfer 100-MB files over the same emulated WAN with a 100-ms RTT delay. As a baseline, we use stock FTP with TCP buffers set at 64 KB. Most modern operating systems set their default TCP buffers to 64 KB, 32 KB, or even less. Therefore, this number represents the high-end of OS-default TCP buffer sizes. We then test drsFTP, allowing the buffer size to vary in response to network conditions while starting at 64 KB as in stock FTP. Lastly, we benchmark a statically-tuned FTP, one that tunes TCP buffers once at connection set-up time. To test the extremes of this problem, we test the statically-tuned FTP with two different values – one set to the minimum bandwidth-delay product (BDP) and one to the maximum.

Performance. Figure 11 shows the average time to transfer a 100-MB file along with the range of the half-width of the 95% confidence interval centered around the average. Both the drsFTP and statically-tuned FTP produce a four-fold improvement over stock FTP.

When the minimum values are used for the statically-tuned FTP to set $fwnd$, the actual BDP fluctuates significantly below and above the set value over the

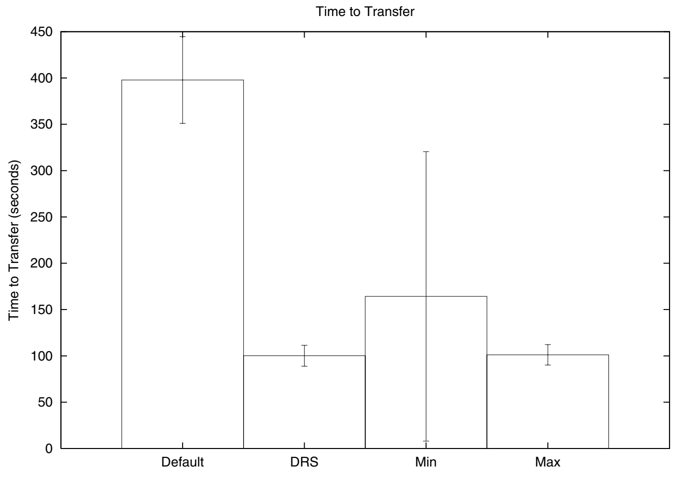

Fig. 11. Transfer Time (Smaller is better)

course of the file transfer. This results in a very large variability in transfer times, as shown in Figure 11. However, the memory usage stays relatively low when compared to the statically-tuned FTP where maximum values are used, as shown in Figure 12.

When the maximum values are used, the transfer time is competitive with drsFTP but at the expense of using a tremendous amount of memory. That is, the buffer sizes are set large enough that the statically set value of $fwnd$ is never exceeded.

As shown in Figures 11 and 12, drsFTP simultaneously achieves fast transfer times (comparable to statically-tuned FTP with maximum values and four times faster than stock FTP) and relatively low-memory usage (comparable to the statically-tuned FTP with minimum values).

6 Conclusion

In this paper, we presented dynamic right-sizing (DRS), an automated, light-weight, and scalable technique for enhancing grid performance. Over a typical WAN configuration, the kernel-space DRS achieves a seven-fold increase in throughput versus stock TCP; and the user-space DRS, i.e., drsFTP, achieves a four-fold increase in throughput versus stock TCP/FTP.

Currently, the biggest drawback of drsFTP is its double-buffered implementation, i.e., buffered in the kernel and then in user space. Clearly, this implementation affects the throughput performance of drsFTP as drsFTP achieves only 57% of the performance of the kernel-space DRS. However, this implementation of drsFTP was simply a proof-of-concept to see if it would provide any benefit over a statically-tuned FTP. Now that we have established that there is sub-

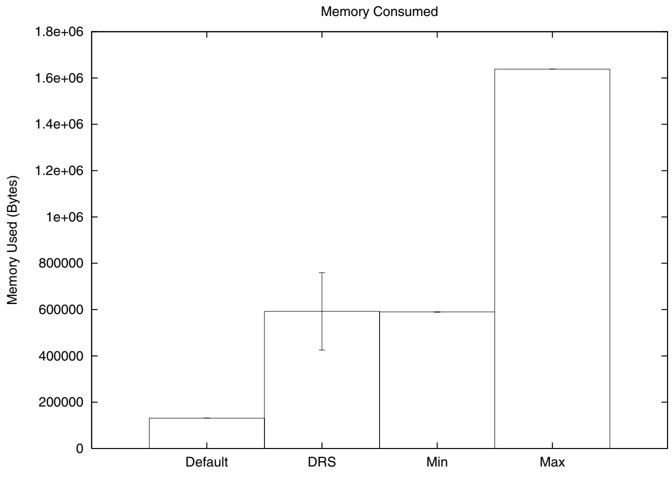

Fig. 12. Memory Usage (Smaller is better.)

stantial benefit in implementing DRS in user space, we are currently working on a new version of drsFTP that abides more closely to our kernel implementation of DRS (albeit at a coarser time granularity) and eliminates the extra copying done in the current implementation of drsFTP.

Lastly, we are working with Globus middleware researchers to integrate our upcoming higher-fidelity version of drsFTP with GridFTP. Currently, GridFTP uses parallel streams to achieve high bandwidth.

References

1. Foster, I. and Kesselman, C.: The Grid: Blueprint for a New Computing Infrastructure. Morgan-Kaufmann Publishers. San Francisco, California (1998).
2. Foster, I., Kesselman, C., Tuecke, S.: The Anatomy of the Grid: Enabling Scalable Virtual Organizations. International Journal of Supercomputer Applications (2001).
3. Chervenak, A., Foster, I., Kesselman, C., Salisbury, C., Tuecke, S.: The Data Grid: Towards an Architecture for the Distributed Management and Analysis of Large Scientific Datasets. International Journal of Supercomputer Applications (2001).
4. Argonne National Laboratory, California Institute of Technology, Lawrence Berkeley National Laboratory, Stanford Linear Accelerator Center, Jefferson Laboratory, University of Wisconsin, Brookhaven National Laboratory, Fermi National Laboratory, and San Diego Supercomputing Center: The Particle Physics Data Grid. http://www.cacr.caltech.edu/ppdg/.
5. Childers, L., Disz, T., Olson, R. Papka, M., Stevens, R., Udeshi, T.: Access Grid: Immersive Group-to-Group Collaborative Visualization. Proceedings of the 4th International Immersive Projection Workshop (2000).
6. Partridge, C., Shepard, T.: TCP/IP Performance over Satellite Links. IEEE Network. **11** (1997) 44–49.

7. Allman, M., Glover, D., Sanchez, L.: Enhancing TCP Over Satellite Channels Using Standard Mechanisms. IETF RFC 2488 (1999).
8. Allman, M. et al.: Ongoing TCP Research Related to Satellites. IETF RFC 2760 (2000).
9. Feng, W., Tinnakornsrisuphap, P.: The Failure of TCP in High-Performance Computational Grids. Proceedings of SC 2000: High-Performance Networking and Computing Conference (2000).
10. Pittsburgh Supercomputing Center. Enabling High-Performance Data Transfers on Hosts. http://www.psc.edu/networking/perf_tune.html.
11. Tierney, B. TCP Tuning Guide for Distributed Applications on Wide-Area Networks. USENIX & SAGE Login. http://www-didc.lbl.gov/tcp-wan.html (2001).
12. Tirumala, A. and Ferguson, J.: IPERF Version 1.2. http://dast.nlanr.net/Projects/Iperf/index.html (2001).
13. Lai, K., Baker, M.: Nettimer: A Tool for Measuring Bottleneck Link Bandwidth. Proceedings of the USENIX Symposium on Internet Technologies and Systems (2001).
14. University of Kansas, Information & Telecommunication Technology Center: NetSpec: A Tool for Network Experimentation and Measurement. http://www.ittc.ukans.edu/netspec.
15. Lawrence Berkeley National Laboratory: Nettest: Secure Network Testing and Monitoring. http://www-itg.lbl.gov/nettest.
16. Mah, B.: pchar: A Tool for Measuring Internet Path Characteristics. http://www.employees.org/ bmah/Software/pchar.
17. Jin., G., Yang, G., Crowley, B., Agrawal, D.: Network Characterization Service. Proceedings of the IEEE Symposium on High-Performance Distributed Computing (2001).
18. Liu, J., Ferguson, J.: Automatic TCP Socket Buffer Tuning. Proceedings of SC 2000: High-Performance Networking and Computing Conference (2000).
19. Tierney, B., Gunter, D., Lee, J., Stoufer, M.: Enabling Network-Aware Applications. Proceedings of the IEEE International Symposium on High-Performance Distributed Computing (2001).
20. National Center for Atmospheric Research, Pittsburgh Supercomputing Center, and National Center for Supercomputing Applications. The Web100 Project. http://www.web100.org.
21. Mathis, M.: Pushing Up Performance for Everyone. http://ncne.nlanr. net/training/techs/1999/991205/Talks/mathis_991205_Pushing_Up_Performance/ (1999).
22. Semke, J., Mahdavi, J., Mathis, M. Automatic TCP Buffer Tuning. Proceedings of ACM SIGCOMM (1998).
23. Fisk, M., Feng, W.: Dynamic Adjustment of TCP Window Sizes. Los Alamos National Laboratory Unclassified Report, LA-UR 00-3221 (2000).
24. Fisk, M., Feng, W.: Dynamic Right-Sizing: TCP Flow-Control Adaptation (poster). Proceedings of SC 2001: High-Performance Networking and Computing Conference (2001).
25. Dunigan, T., Fowler, F. Personal Communication with Web100 Project (2002).
26. Jacobson, V., Braden, R., Borman, D.: TCP Extensions for High Performance. IETF RFC 1323 (1992).
27. Postel, J., Reynolds, J. File Transfer Protocol (FTP). IETF RFC 959 (1985).