IEEE Symposium on High- Performance Distributed Computing
(HPDC-11/2002), Edinburgh, Scotland, July 2002. LA-UR 02-2799

Dynamic Right-Sizing in FTP (drsFTP): Enhancing Grid Performance in User-Space[*]

Mark K. Gardner[†], Wu-chun Feng[†], and Mike Fisk[‡]
{mkg,feng,mfisk}@lanl.gov

[†] Research & Development in Advanced Network Technology (RADIANT)
Computer & Computational Sciences Division
Los Alamos National Laboratory
Los Alamos, NM 87545

[‡] Network Security Team, Network Engineering Group
Computing, Communications & Networking Division
Los Alamos National Laboratory
Los Alamos, NM 87545

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 the 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. 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 scalable technique called dynamic right-sizing. We implement this technique in user space (in particular for bulk-data transfer) so that end users do not have to modify the kernel to achieve a significant increase in throughput.

1 Introduction

TCP is the ubiquitous transport protocol for the Internet, as well as emerging infrastructures such as computational grids [11, 12], data grids [2, 6], and access grids [7]. 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.

The primary reason for the abysmal performance in large BDP networks is that the default flow-control parameters in TCP are static and are tuned to yesterday's WAN speeds. For any given connection, the optimal TCP buffer size is equal to the product of the bandwidth of the bottleneck link and the round-trip time (RTT), i.e., the bandwidth-delay product of the connection. Grid and network researchers manually tune buffer sizes to keep the network pipe full in order to achieve acceptable WAN performance [4, 24]. To tune the buffer sizes appropriately, the grid community currently uses diagnostic tools to determine the RTT of the connection and the bandwidth of the bottleneck link. Such tools include *iperf* [26], *nettimer* [15–17], *netspec* [27], *nettest* [14], *pchar* [19], and *pipechar* [13]. However, all of the tools require a certain level of network expertise to install and use. Thus, users and developers who are not network experts, e.g., the high-performance visualization community, find the tuning process quite difficult.

To simplify the tuning process and eliminate what has been called the *wizard gap* [20],[1] several services that provide clients with the correct tuning parameters for a given connection have been proposed, e.g., AutoNcFTP [18] and

[*]This work was supported by the U.S. Department of Energy through Los Alamos National Laboratory contract W-7405-ENG-36 with funding provided by the Office of Science Base Program of DOE.

[1]The *wizard gap* is the difference between the network performance that a network "wizard" can achieve by appropriate tuning and the performance with default parameters.

Enable [25]. 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 lifetime of the connection. In Section 2, we demonstrate that this assumption is tenuous at best.

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 [23] and dynamic right-sizing (DRS) [8, 10].[2] The former implements sender-based, flow-control adaptation while the latter implements received-based, flow-control adaptation and abides by TCP semantics. Dynamic right-sizing (DRS) in the kernel exhibits throughput speed-ups of seven to eight over a typical WAN grid [8, 10]. 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.[3]

The installation of our DRS kernel patch requires knowledge about recompiling 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. Thus, we propose a coarser-grained but more portable implementation of DRS in *user space* that is transparent to the end user. Specifically, we integrate DRS technique into FTP (drsFTP). The differences between drsFTP and AutoNcFTP, another user-space auto-tuning implementation, are two-fold. First, AutoNcFTP is based upon NcFTP (http://www.ncftp.com/), which is a commercial product, whereas drsFTP uses relies completely on open-source software, i.e., Debian Linux's FTP client and Washington University's de-facto standard FTP daemon (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.

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 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 the inferred state of the network.

Specifically, TCP calculates an effective window ($ewnd$) as $ewnd \equiv min(fwnd, cwnd)$ 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 connection, thus providing the motivation for dynamic-right sizing (DRS).

Historically, a static $fwnd$ sufficed for all communication because the BDP of networks was small. Thus, 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 [3]. 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 = 0.8%).

Over the lifetime of a connection, bandwidth and delay change (due to transitory queueing, congestion and route changes) implying that the BDP also changes. Figure 1 supports this claim. Here we show the BDP between Los Alamos and New York at 20-second intervals.[4] The bottleneck bandwidth averages 17.2 Mbps with a low and a high of 26 Kbps and 28.5 Mbps, respectively. The RTT delay also varies between [119, 475] ms with an average delay of 157 ms. As a result, the BDP for our connection varies by as much as 61 Mb.

Because the BDP over the lifetime of a connection is continually changing, 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. The implicit decision remains even when the BDP is determined at the start of a connection since the BDP varies widely, even over short time scales, in wide-area networks. Clearly, the grid community needs a solution that dynamically and transparently adapts $fwnd$ to achieve

[2]The Web100 project at http://www.web100.org recently incorporated DRS into their software distribution.

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

[4]We use *nettimer* to measure bandwidth and RTT delay, but note that the actual BDP may be even larger as *nettimer* measures dynamic latency but static bottleneck bandwidth.

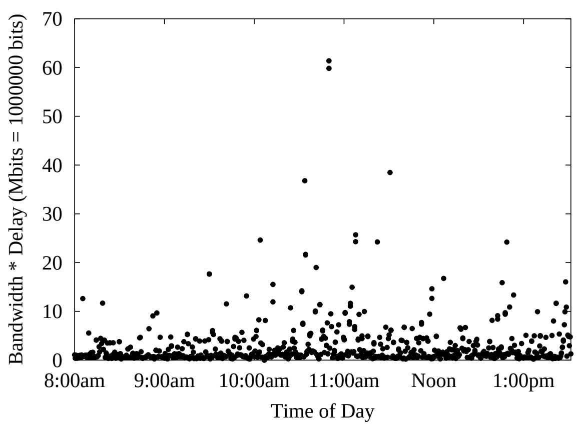

Figure 1. Bandwidth-delay product at 20-second intervals

good performance without wasting network or memory resources. DRS is one such solution.

3 DRS in User Space: drsFTP

The key to maximizing the transfer rate of TCP connections over high bandwidth-delay product networks is to ensure that the transfer rate is limited only by the congestion-control window throughout the lifetime of the connection. DRS in kernel space does this by inferring the “instantaneous” bandwidth-delay product of a connection and setting the flow-control window above that value [8–10]. Unlike the kernel-space version of DRS which benefits all applications transparently, user-space DRS must be implemented by each pair of communicating applications. In this section, we implement DRS in a FTP client and server, resulting in drsFTP.

The primary difficulty in developing user-space DRS applications lies in the fact that user-space code does not have direct access to the state of the TCP stack. Consequently, drsFTP must estimate the bandwidth-delay product of the connection from coarse-grained user-space measurements rather than from fine-grained TCP connection state.

3.1 Determining Available Bandwidth

By definition, we know that the sender always has data to send throughout the life of the FTP data connection. It then follows that the sender will send as much data as possible, limited by its idea of the congestion- and flow-control windows. Furthermore, the receiver is receiving data as quickly as the current windows, network and CPU scheduling conditions allow. Therefore, the average bandwidth available to a connection is computed by dividing the number of bytes received by the time required to receive them.

The difficulty lies in selecting the appropriate sampling interval over which to aggregate the number of bytes received.[5] Selecting too short of an interval dramatically increases overhead and reduces performance. Too short of an interval also leads to erroneous estimates because of scheduling and buffering effects. On the other hand, selecting too long of an interval decreases the responsiveness of DRS to changes in available bandwidth and may reduce performance because the estimated bandwidth-delay product, and hence, the receiver’s advertised window, may be artificially small.

In the current implementation of drsFTP, the available bandwidth is computed through the periodic invocation of a signal handler upon alarm expiration. Different values for the sampling interval can easily be tested by varying the periodic expiration time of the alarm. The average bandwidth available to the connection over the last interval is the number of bytes received since the last alarm signal divided by the length of the interval. An appropriate choice for the sample interval yields estimated bandwidth values of sufficient accuracy.

3.2 Determining RTT

Unlike the procedure for estimating the bandwidth of a connection, the RTT cannot be inferred in user-space applications without injecting extra traffic into the network. User-space code does not have access to the inner workings of the TCP stack and hence cannot know when a given packet is sent nor when its acknowledgement is received.

To sidestep this problem, we send a small packet on the FTP control channel for the sender to echo back. The estimated RTT begins with the sending of a RTT probe packet and ends when its echo is received. The additional load on the network as the result of RTT probe packets is generally small and depends on the sampling interval. (We give an optimization which minimizes the impact of RTT probes in Section 3.5.)

We note that sending the RTT probe packet over the control channel assumes that the control and data channels follow the same route. In the case of third party control of a FTP data transfer, however, the control and data channels are likely to take very different routes. Thus the RTT estimate may be inaccurate. We send RTT probes over the control channel to comply with RFC 959 [21], since commands cannot be sent on the data channel. If RTT probes could be sent out-of-band on the data channel, then RTT estimates could be obtained in the manner described above. Sending data out-of-band is possible within Globus and hence we are working with the GridFTP researchers to integrate drsFTP with GridFTP.

[5]Equivalently, we can select a fixed number of bytes to be received periodically and measure how long it takes.

3.3 Setting Receiver's Advertised Window

User-space applications cannot directly set the flow-control window in most TCP stacks. Instead, they must indirectly set the window by setting the TCP receive buffer size to an appropriate value via a `setsockopt` call.

In the worst case, the sender's window is doubling with every round trip during TCP slow start. When it is determined that the receiver window should increase, the new value should be at least double the current value. (There is no need to double the current value once TCP is out of slow start. However, it is very difficult to determine when slow start ends.) Therefore, we increase the receive buffer in our drsFTP implementation by a factor of two over the value of the estimated BDP whenever the current buffer size is less than twice the BDP.

3.4 TCP Window Scaling

Because the window scale 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. drsFTP sets the send and receive buffer sizes to allow windows of up to 16-MB worth of data before initiating connection set-up. Once the connection has been made (and the window scale factor set properly), drsFTP resets the buffer sizes back to their initial values.

In order to set the window scale factor appropriately, the network limits of the operating system must be increased. The steps involved in increasing the limits are operating system dependent. See [5] for an example of the steps required for a variety of operating systems.

Unfortunately, even after adjusting the network limits in the obvious ways, Linux 2.4 kernels still do not set the window scale factor as expected. For more information, see Appendix A.

3.5 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 window size. However, because the FTP protocol specification (RFC 959 [21]) does not prohibit traffic on the control channel during data transfer,[6] a drsFTP receiver may inform a drsFTP sender about changes in buffer size by communicating over the control channel.

[6]Although RFC 959 does not prohibit traffic on the control channel during data transfer, many implementations do not expect it.

Since FTP is a bidirectional data-transfer protocol, the receiver may be either 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 in both directions.

Serendipitously, the Internet Draft of the GridFTP protocol extensions to FTP [1] 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 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 an active data transfer. If specified during a data transfer, it affects the data transfer that started most recently. The command is informational and need not be acted upon.

Response Codes:

```
200 SBUF <SP> <buffer-size>
```

If the server-PI is able to change its buffer size, a 200 response code is returned. The new size of the server's buffer is also returned in case it is less than the requested size. This allows the client-PI to regulate its buffer usage to keep in step with the server.

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

New Reply Code:

```
126 SBUF <SP> <buffer-size>
```

The 126 Reply may occur at any point when the server-PI is receiving data from the user-PI. As with the `SBUF`, this reply is informational and need not be acted upon or responded to in any manner, thus providing interoperability with non-drsFTP applications.

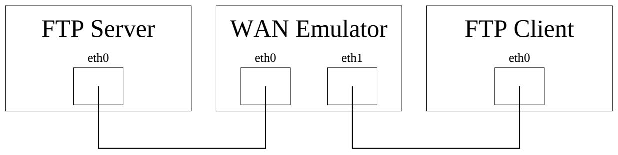

Figure 2. Experimental setup

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

We note that the SBUF command also provides a vehicle for determining RTT without injecting a separate message into the network. Since RTT probes do not contain any data, we allow SBUF commands to serve the dual purpose of conveying the receiver's buffer size to the sender and probing for the RTT. Thus, separate RTT probes, as discussed in Section 3.2, are not needed.

4 Experiments

To test the performance of drsFTP against stock FTP and FTP in which buffer sizes are statically set at the start of the connection, we first need a realistic and reproducible sample of wide-area network (WAN) behavior. To obtain this data, we run `ping` and `nettimer` every twenty seconds to sample the bandwidth and RTT between Los Alamos National Laboratory and a site in New York (a cross-country data transfer). We then use a representative sample of this data to configure our WAN emulator and benchmark each of the aforementioned versions of FTP.

4.1 Experimental Setup

Our experimental apparatus, shown in Figure 2, consists of three identical machines connected via Gigabit Ethernet. Each machine contains dual 933-MHz Pentium III processors with 512-MB of RAM and an Alteon Gigabit NIC in a 64-bit, 66-MHz PCI slot. One machine, containing another Alteon Gigabit NIC, acts as a WAN emulator. Each of its two NICs are connected to one of the other machines and the routes set via hard-coded arp entries on the client and server. The WAN emulator, which is implemented using TICKET technology [29], forwards packets at line rate and has a user-settable delay. All traffic, both data and control, occurs through the WAN emulator.

In the results that follow, the average round-trip time is 102.1 ms. The sampling interval used by the drsFTP implementation to estimate available bandwidth is one second, a conservative configuration with very low overhead. The over-provisioned static buffer size is 16-MB, which is larger than the optimal bandwidth-delay product of 12.2-MB, and represents the best performance possible in this configuration.

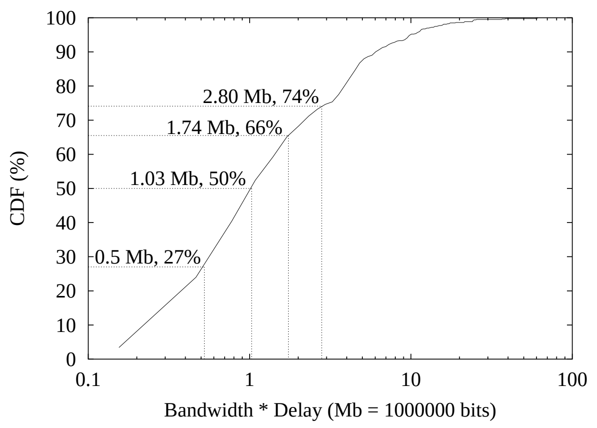

Figure 3. CDF of BDP from Figure 1

4.2 Experimental Method

For each version of FTP (stock FTP, drsFTP and statically-tuned FTP), we transfer a set of files, ranging from 8-KB to 64-MB, over the emulated WAN. As a baseline, we use stock FTP with TCP receive 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. Last of all, we benchmark a statically-tuned FTP with the TCP buffers set to various values at connection set-up time.

4.3 WAN Data

Figure 3 show the cumulative distribution function of the BDP data of Figure 1 which was used to represent the WAN for the experiments. The minimum, median, mean and maximum values of BDP are 3.58-Kb, 1.17 Mb, 2.80-Mb, and 61.4-Mb, respectively. Several representative values are called out on the graph. In particular, the 0.5-Mb value corresponds to a buffer size of 64-KB, which is the default (stock) buffer size tested below. Also, the 1.74-Mb value corresponds with one of the statically-set buffer sizes tested.

4.4 Results

Figure 4 shows the average FTP bandwidth as a function of the size of the transfer. (The x-axis has a linear

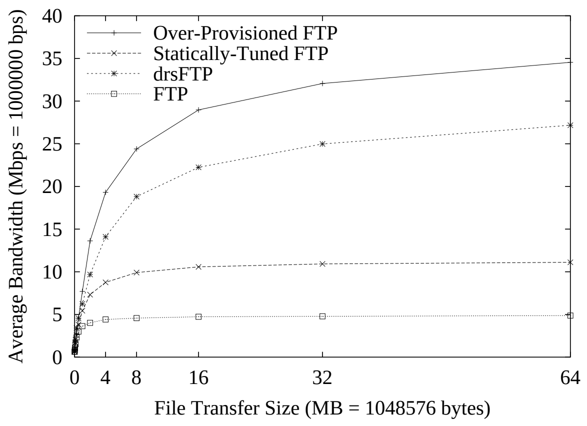

Figure 4. Comparison of FTP, drsFTP and statically-tuned FTP

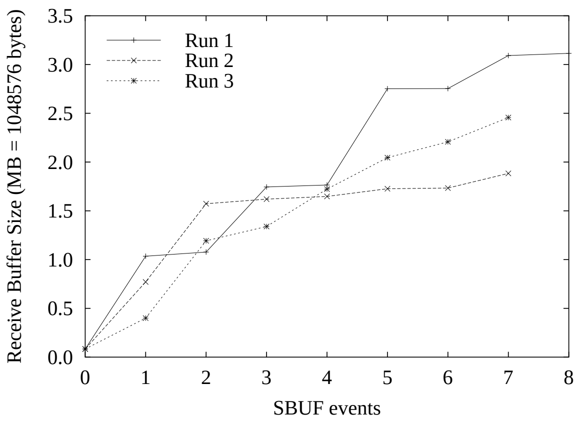

Figure 5. drsFTP Buffer Sizes over Time

scale with markers placed according to the powers-of-two file sizes tested. Also the width of the 95% confidence interval centered around the average is less than 5% in all cases.) The average bandwidth of FTP with stock buffer sizes approaches 5-Mbps for file sizes as small as 8-MB. In contrast, the average bandwidth of drsFTP asymptotically approaches 30-Mbps at over 64-MB file transfers. Thus, the utilization of available bandwidth of drsFTP is approximately six times better than stock FTP.

The best bandwidth (34.5-Mbps) is achieved by the over-provisioned FTP which has larger than required buffer sizes. As shown, drsFTP achieves 78.7% of the over-provisioned bandwidth. The primary reason for the difference in performance is that drsFTP must rely on coarse-grained measurements to infer available bandwidth and round-trip time and hence may not infer the required buffer sizes accurately. This is an inherent limitation indicative of the interim nature of the drsFTP application. The kernel version of DRS has access to fine-grained information and hence performs better [8, 10]. In addition, all applications benefit without modification when DRS is in the kernel. drsFTP was developed to provide the benefits of DRS to the grid community while vendors implement DRS in the kernel.

Figure 4 also compares the average bandwidth of drsFTP to a statically-tuned case where the BDP was sampled at an inopportune time, e.g., at one of the lower data points in Figure 1. As shown in Figure 3, the 212.5-KB (1.74-Mb) buffer size chosen for this test is in the 66th percentile of the BDP data of Figure 1. (The median value of BDP is 143.3-KB or 1.17 Mb.) Here we see that drsFTP utilizes the available bandwidth 2.4 times better than the statically-tuned case. The comparison illustrates the benefit of inferring the available bandwidth and setting the flow-control buffers automatically.

So far, we have only addressed the issue of achieving high transfer rates. We now compare buffer space requirements for drsFTP, statically-tuned FTP and over-provisioned FTP. As motivation, we conjecture that memory consumption will become a more serious issue as computational grids and ubiquitous computing become heavily-used and indispensable parts of the computational infrastructure. While applications are able to use buffer space with abandon now, we envision the time when grid nodes will become heavily loaded with large numbers of potentially diverse applications. One example might be a repository for human genome information which will be accessed simultaneously by thousands of researchers. If each connection over-provisions its buffers, it is likely that the node will run out of buffer space and reject connections which could otherwise be serviced had the connections been more frugal.

Figure 5 shows the growth of the drsFTP receive buffer as a function of time during three transfers of a 512-MB file. The final buffer sizes for the three transfers range from 1.9-MB to 3.1-MB, with an average of 2.7-MB. Due to changing conditions during the transfers, the buffer sizes grow at different rates, particularly during the latter part of the transfer. In contrast, the over-provisioned FTP uses a 16-MB buffer which is statically allocated during connection set-up. Thus drsFTP achieves over three quarters of the over-provisioned performance while only using one sixth the amount of memory. In other words, drsFTP achieves an average of 10.1 Mbps per MB of buffer space used while statically-tuned FTP achieves only 2.2 Mbps per MB of buffer space used.

As Figure 6 shows, drsFTP achieves nearly six times better utilization of the network with respect to memory than the over-provisioned case. Had the theoretically optimal BDP of 12.2-MB been allocated for the statically-tuned FTP

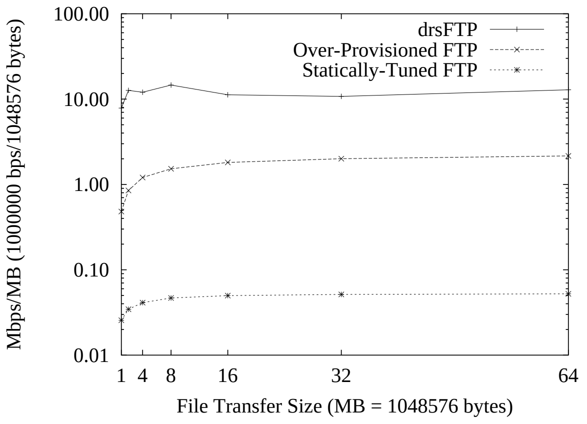

Figure 6. Mbps per MB of Buffer Space

connection instead of 16-MB, drsFTP would still have been able to support more high-performance connections with a 3.6 times Mbps-per-MB advantage. The difference between drsFTP and the statically-tuned case where the BDP was sampled at an inopportune time is even more dramatic.

5 Conclusion

In this paper, we present drsFTP, dynamic right-sizing (DRS) in FTP, an automated technique for enhancing grid performance. This work was inspired by feedback we received at SC 2001 on our kernel implementation of DRS. Specifically, application users wanted the functionality of DRS without modifying their operating system kernels. We have shown that DRS improves FTP throughput by six times over a stock implementation. Furthermore, drsFTP achieves significantly higher bandwidth per buffer space used than the customary over-provisioning approach. Thus applications modified to support DRS provide significant performance improvements to the grid community while we wait for DRS kernel implementations, with even better performance, to become available.

Currently, our implementation of drsFTP is the only DRS-modified application. We will be releasing drsFTP under the GNU Public License after more extensive testing. We are also working with Globus middleware researchers to integrate drsFTP with GridFTP.

Currently, GridFTP uses parallel streams to achieve high bandwidth. However, results from the SC2001 Bandwidth Challenge [22] suggest that under some circumstances having a single stream with appropriately sized buffers may achieve better performance. We have done some preliminary work to investigate this phenomenon [28] but need to thoroughly explore the trade-offs involved. Perhaps, parallel DRS streams will combine the best of both approaches.

6 Acknowledgements

We gratefully acknowledge the assistance of Eric Weigle, not only for the development of the TICKET WAN emulator, but also for many insightful discussions concerning the Linux kernel.

References

[1] W. Allcock et al. GridFTP: Protocol Extensions to FTP for the Grid. `http://www-fp.mcs.anl.gov/dsl/GridFTP-Protocol-RFC-Draft.pdf`, Mar 2001.

[2] ANL, CalTech, LBL, SLAC, JF, U. Wisconsin, BNL, FNL, and SDSC. The Particle Physics Data Grid. `http://www.cacr.caltech.edu/ppdg/`.

[3] D. Borman, R. Braden, and V. Jacobson. TCP Extensions for High Performance (RFC 1323), May 1992.

[4] Pittsburgh Supercomputing Center. Enabling High-Performance Data Transfers on Hosts. `http://www.psc.edu/networking-/perf\_tune.html/`.

[5] Pittsburgh Supercomputing Center. Enabling high performance data transfers on hosts. `http://www.psc.edu/networking/perf\_tune.html`.

[6] A. Chervenak, I. Foster, C. Kesselman, C. Salisbury, and S. Tuecke. The Data Grid: Towards an Architecture for the Distributed Management and Analysis of Large Scientifi c Datasets. *International Journal of Supercomputer Applications*, 23(3):187–200, July 2001.

[7] L. Childers, T. Disz, R. Olson, M. E. Papka, R. Stevens, and T. Udeshi. Access Grid: Immersive Group-to-Group Collaborative Visualization. In *Proceedings of the 4th International Immersive Projection Workshop*, 2000.

[8] M. Fisk and W. Feng. Dynamic Adjustment of TCP Window Sizes. Technical Report Los Alamos Unclassifi ed Report (LAUR) 00-3221, Los Alamos National Laboratory, July 2000.

[9] M. Fisk and W. Feng. Dynamic Right-Sizing in TCP. In *Proceedings of the Los Alamos Computer Science Institute Symposium*, Oct 2001. LA-UR 01-5460.

[10] M. Fisk and W. Feng. Dynamic Right-Sizing: TCP Flow-Control Adaptation (Poster). In *Proceedings of SC 2001: High-Performance Networking and Computing Conference*, November 2001.

[11] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, 1999.

[12] I. Foster, C. Kesselman, and S. Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal of Supercomputer Applications*, 2001.

[13] G. Jin, G. Yang, B. Crowley, and D. Agrawal. Network Characterization Service. In *Proceedings of the IEEE Symposium on High-Performance Distributed Computing*, August 2001.

[14] Lawrence Berkley National Laboratory. Nettest: Secure Network Testing and Monitoring. http://www-itg.lbl.gov/nettest/.

[15] K. Lai and M. Baker. Measuring Bandwidth. In *Proceedings of IEEE INFOCOMM 1999*, March 1999.

[16] K. Lai and M. Baker. Measuring Link Bandwidths Using a Deterministic Model of Packet Delay. In *Proceedings of ACM SIGCOMM 2000*, August 2000.

[17] K. Lai and M. Baker. Nettimer: A Tool for Measuring Bottleneck Link Bandwidth. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, March 2001.

[18] J. Liu and J. Ferguson. Automatic TCP Socket Buffer Tuning. In *Proceedings of SC 2000: High-Performance Networking and Computing Conference (Research Gem)*, November 2000. `http://dast.nlanr.net/Projects/Autobuf`.

[19] B. Mah. pchar: A Tool for Measuring Internet Path Characteristics. `http://www.employees.org/~bmah/Software/pchar`.

[20] M. Mathis. Pushing Up Performance for Everyone. `http://www.ncne.nlanr.net/news/workshop/19999/991205/Talks/mathis_991205_Pushing_Up_Performance/`.

[21] J. Postel and J. Reynolds. File Transfer Protocol (FTP), Oct 1985.

[22] SC2001 Bandwidth Challenge Proposal: Bandwidth to the World. `http://www-iepm.slac.stanford.edu/monitoring/bulk/sc2001/proposal.html`.

[23] J. Semke, J. Mahdavi, and M. Mathis. Automatic TCP Buffer Tuning. *Computer Communications Review, ACM SIGCOMM*, 28(4), October 2001.

[24] B. Tierney. TCP Tuning Guide for Distributed Applications on Wide-Area Networks. In *USENIX & SAGE Login*, February 2001. `http://www-didc.lbl.gov/tcp-wan.html`.

[25] B. Tierney, D. Gunter, J. Lee, and M. Stoufer. Enabling Network-Aware Applications. In *Proceedings of the IEEE International Symposium on High-Performance Distributed Computing*, August 2001.

[26] A. Tirumala and J. Ferguson. IPERF. `http://dast.nlanr.net/Projects/Iperf/index.html`.

[27] Information & Telecommunication Technology Center, University of Kansas. NetSpec: A Tool for Network Experimentation and Measurement. `http://www.ittc.ukans.edu/netspec/`.

[28] E. Weigle and W. Feng. A Comparison of TCP Automatic-Tuning Techniques for Distributed Computing. In *Proceedings of the Eleventh IEEE International Symposium on High Performance Distributed Computing (HPDC-11)*, Jul 2002.

[29] E. Weigle and W. Feng. TICKETing High-Speed Traffi c with Commodity Hardware and Software. In *Proceedings of the Third Annual Passive and Active Measurement Workshop (PAM2002)*, March 2002.

Appendix

A Setting the Window Scale under Linux 2.4

As was discussed in Section 3.4, drsFTP sets the send and receive buffers sizes before establishing a connection in order to set the window scale factor. The network limits in the operating system must also be set appropriately for window scaling to be useful. However, this approach does not work as outlined.

Under Linux 2.4 kernels, writing values to files in the `/proc/sys/net` file system changes the network limits. Table 1 lists the pertinent variables, their initial values (Table 1(a)) and the values used during the drsFTP experiments in Section 4 (Table 1(b)) .

Variable	Minimum	Default	Maximum
`core/rmem_max`			131071
`core/wmem_max`			131071
`ipv4/tcp_rmem`	4096	87380	174760
`ipv4/tcp_wmem`	4096	16384	131072

(a) Default Settings

Variable	Minimum	Default	Maximum
`core/rmem_max`			16777216
`core/wmem_max`			16777216
`ipv4/tcp_rmem`	4096	8488608	16777216
`ipv4/tcp_wmem`	4096	16384	16777216

(b) Settings Used with drsFTP Experiments

Table 1. Increasing Linux Network Limits

Initially, only the maximum values were changed. However, the window scale factor did not increase as expected. It appears that Linux 2.4 determines the receiver's window scale factor from the default value of `ipv4/tcp_rmem` rather than the maximum value. Increasing the default value caused the window scale factor to be set correctly.

Requiring the default `ipv4/tcp_rmem` value to be set in order to set the window scale factor has the consequence of either wasting buffer space, since most of the connections do not perform bulk-data transfers, or limiting the size of the flow-control window for those that do. Using the default value also places implicit limits on the maximum buffer space that can be effectively utilized because too small of default will constrain the window size to a value less than that implied by the maximum. For these reasons, an approach which gives more flexibility is to set the window scale factor based on the maximum `ipv4/tcp_rmem` value.