Accepted to Computer Communications (DRAFT) / 26 March 2003

User-Space Auto-Tuning for TCP Flow Control in Computational Grids

Mark K. Gardner, Sunil Thulasidasan, Wu-chun Feng

Computer & Computational Sciences 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 their available bandwidth because operating systems and their associated protocol stacks are still tuned for yesterday's network 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 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. We show the performance of two user-space implementations of DRS: drsFTP and DRS-enabled GridFTP.

Keywords: flow-control, buffer tuning, TCP, auto-tuning, dynamic right-sizing, FTP, GridFTP.

1 Introduction

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

As noted in [6–9], congestion- and flow-control adaptation bottlenecks are the primary reason for this abysmal performance. The former is a topic of active research beyond the scope of this paper. (See [10–12].) In order to address the latter problem, grid and network researchers continue to manually tune buffer sizes to keep the network pipe full [7, 13, 14], and thus achieve acceptable wide-area network (WAN) performance in support of grid computing. However, the tuning process can be quite difficult, particularly for users and developers who are not network experts. It involves calculating the bandwidth of the bottleneck link and the round-trip time (RTT) for a given connection. The optimal TCP buffer size is equal to the product of the bandwidth of the bottleneck link and the RTT, i.e., the effective bandwidth-delay product of the connection.

Currently, in order to tune buffer sizes appropriately, the grid community uses diagnostic tools to determine the RTT and the bandwidth of the bottleneck link. Such tools include *pipechar* [15], *nettimer* [16], *nettest* [17], *pchar* [18], *iperf* [19] and *netspec* [20]. However, none of these tools include a client API so applications can tune their TCP connections and all of the tools require a certain level of network expertise to install and use. Furthermore, many of these tools 'pollute' the network with extraneous (probe) packets.

To simplify the tuning process, several services that provide clients with appropriate tuning parameters for a given connection have been proposed, e.g., AutoNcFTP [21], Web100 [22] and Enable [23], in order to eliminate what has been called the *wizard gap* [24]. (The *wizard gap* is the difference between the performance that a network 'wizard' can achieve by appropriately tuning buffer sizes and the performance of an untuned applica-

⋆ This work was supported by the U.S. Dept. of Energy through Los Alamos National Laboratory contract W-7405-ENG-36. Any opinions, findings, and conclusions, or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of DOE, Los Alamos National Laboratory. Los Alamos Unclassified Report (LA-UR) 03-1807.

Email addresses: (mkg@lanl.gov, sunil@lanl.gov, feng@lanl.gov (Wu-chun Feng).

URL: `http://www.lanl.gov/radiant/` (Wu-chun Feng).

tion.) 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. This makes 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.

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 [25] and dynamic right-sizing (DRS) [26–28].

Auto-tuning implements sender-based flow-control adaptation by fairly sharing buffer space on the sender. (It assumes that receiver always have enough buffer space.) DRS, on the other hand, implements receiver-based flow-control adaptation by sizing the flow-control window according to both the buffer space on the receiver and the available bandwidth in the network. (DRS makes no assumptions about the buffer space on the sender. A sender without sufficient buffer space is allowed to transmit at a slower rate than indicated by the flow-control window of the receiver.) Because auto-tuning does not take into account available buffer space on the receiver when sending packets, it allows the sender to (potentially) overrun the receiver, either inadvertently (during a FTP transfer) or maliciously (during a denial-of-service attack). DRS, on the other hand, is fully compatible with regular TCP. [1]

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. Achieving large speed-ups requires DRS to be installed on every pair of communicating hosts in a grid. On the other hand, it also benefits all TCP-based applications, e.g., FTP, multimedia streaming and WWW, not just grid applications.

Installing DRS requires knowledge about modifying, recompiling and installing an operating system (OS) kernel, along with root privilege to do so. Thus, DRS functionality is generally not accessible to the typical end user. While we anticipate that DRS will be incorporated into vendor's kernels so that it is transparent to the end user, users want improved performance now. Thus, we present two portable user-space implementations of DRS: drsFTP [29] and DRS-enabled GridFTP.

drsFTP is similar in many ways to NLANR's AutoNcFTP [30]. Both are modified FTP implementations which adjust buffer sizes to increase performance. The differences are two-fold. First, AutoNcFTP relies on NcFTP [31] whereas drsFTP uses the de-facto standard FTP daemon originally from Washington University in St. Louis [32] and the open-source Netkit FTP client [33]. Second, the buffers in AutoNcFTP are only tuned at connection set-up while drsFTP buffers are dynamically tuned over the lifetime of the connection to provide better adaptation and better overall performance.

The increased performance obtained from drsFTP (Section 6.1) motivates us to integrate dynamic right-sizing into GridFTP [34], a subsystem of the Globus Toolkit [35], for managing bulk-data transfers in grids.

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 by the sender based on the inferred state of the network.

Specifically, TCP calculates an effective window, $ewnd \equiv \min(fwnd, cwnd)$, and then sends data at a rate of $ewnd/RTT$, where RTT is the round-trip time of the connection. The congestion control window, *cwnd*, varies dynamically as the network state changes; however, the flow-control window, *fwnd*, has traditionally 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 $fwnd \approx 64\,\mathrm{KB}$ — the largest window available without scaling [36]. Yet BDPs range between a few bytes ($56\,\mathrm{Kbps} \times 5\,\mathrm{ms} \rightarrow 36\,\mathrm{bytes}$) and a few megabytes ($622\,\mathrm{Mbps} \times 100\,\mathrm{ms} \rightarrow 7.8\,\mathrm{MB}$). In 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. We use *nettimer* to quantify how much they change. (Although we would

[1] As a result of not taking into account the buffer space on the receiver, auto-tuning appears to violate the TCP specification.

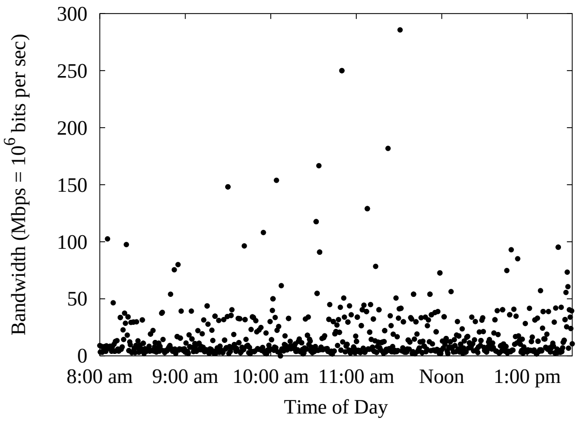

Fig. 1. Bottleneck Bandwidth at 20-Second Intervals

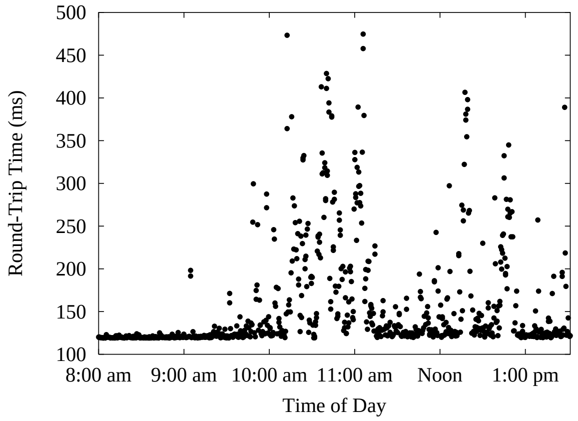

Fig. 2. Round-Trip Time at 20-Second Intervals

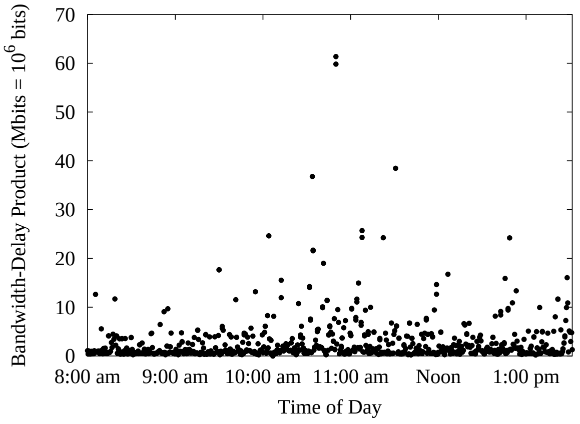

Fig. 3. Bandwidth-Delay Product at 20-Second Intervals

have like to sample at the granularity of the RTT, the overhead of running *nettimer* and other tools in user space prevent us from obtaining the measurements we seek.) Figure 1 presents the bottleneck bandwidth between Los Alamos and New York at 20-second intervals. The bottleneck bandwidth averages 17.2 Mbps with a low and a high of 0.026 Mbps and 28.5 Mbps, respectively. The standard deviation is 26.3 Mbps and the half-width of the 95% confidence interval is 1.8 Mbps. Figure 2 shows the RTT at 20-second intervals, again between Los Alamos and New York. The RTT delay also varies over a wide range 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 Mbit.

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) allows the receiver to estimate the sender's *cwnd* and to dynamically change the flow-control window *fwnd* to match. The estimates are also used to keep pace with the growth in the sender's congestion window.[2] As a result, the throughput between end hosts, e.g., in a grid, will only be constrained by the available bandwidth of the network, rather than some arbitrarily set constant value on the receiver.

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*.

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 estimated by observing the time between when data is sent and an acknowledgement is returned. However, during a bulk-data transfer, the receiver may not be sending any data, and therefore, will not have an accurate RTT es-

[2] Under low-memory conditions, the receiver may advertise a smaller window than the DRS algorithm suggests. The sender is also free to send less than the advertised window.

timate. So, how does the receiver infer delay (and bandwidth) when it only has acknowledgements to transmit back and no data to send?

A receiver 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 estimated RTT could be much larger than the actual RTT. Thus, the estimate acts as an *upper bound* on the RTT and should only be used when there is no other source of RTT information. (For a rigorous presentation of the lower and upper bounds, please see [26, 27].)

We note that DRS is TCP-friendly in the sense that N flows, DRS-enabled or not, will each receive a long-term average of $1/N$-th of the bandwidth of a fully utilized network. Since the congestion-control mechanism governs fairness and because it has the same congestion-control mechanism, DRS responds to congestion the same way as regular TCP. (Here we assume that the end hosts have a fair buffer-allocation policy. If the buffer-allocation policy is not fair, both regular TCP and DRS will be unfair. The fault lies with the buffer-allocation policy, not the transport protocol.) In an uncongested network, however, DRS will attempt to utilize the excess capacity that can exist when all the other connections are artificially limited by their flow windows. As the network becomes congested again, DRS throttles back and performs no better (or worse) than regular TCP.

4 DRS in User Space: drsFTP

As mentioned earlier, deploying DRS requires the OS kernel to be recompiled, which is impractical in may circumstances. Thus, we propose a user-space implementation of DRS.

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 describe the implementation DRS in an FTP client and server, resulting in drsFTP.

4.1 DRS in User-Space

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 has no knowledge of TCP parameters, such as the RTT of a connection, the receiver's advertised window or the sender's congestion window. Information about a connection must be estimated from coarse-grained user-space measurements rather than from fine-grained TCP connection state.

FTP specifies that commands and replies are sent over a control channel that is a completely separate TCP connection from the data channel where the transfer takes place. As with AutoNcFTP and Enable, we focus on (1) adjusting TCP's system buffers over the data channel of FTP and (2) using FTP's stream file-transfer mode. The latter means that a separate data connection is created for every file transferred. If we assume the end-hosts are not bottlenecks (and hence it makes sense to seek higher bandwidth), the sender *always* has data to transmit during the lifetime of the transfer. Once the file has been completely sent, the data connection closes.

4.1.1 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 data as fast 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 that a connection obtains is computed by dividing the number of bytes transmitted by the time required to transmit them.

The difficulty lies in selecting the appropriate sampling interval over which to aggregate the number of bytes transmitted. (Equivalently, we can select a fixed number of bytes to be received periodically and measure how long it takes.) Selecting too short of an interval dramatically increases overhead and reduces performance. It 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 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.

4.1.2 Determining RTT

Unlike the procedure for estimating the bandwidth of a connection, the RTT cannot be inferred in user-space applications without injecting a very small amount of 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, depending on the sampling interval. (Section 4.1.4 gives an optimization which minimizes the impact of RTT probes.)

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 three-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 [37], since commands cannot be sent on the data channel. If probes could be sent on the data channel, then accurate RTT estimates could be obtained in the manner described above.

4.1.3 Adjusting the 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, in general, to determine when slow start ends. Therefore, we increase the receive buffer in drsFTP by a factor of two over the BDP whenever the current buffer size is less than twice the BDP. (In most protocol stacks, buffer space is not allocated until it is actually used so excessive memory usage is not usually a problem in practice.)

4.1.4 Adjusting the Sender's Window

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 code has no way of determining the receiver's advertised window size. The FTP protocol specification [37] does not prohibit traffic on the control channel during data transfer, however. Thus, a drsFTP receiver may inform a drsFTP sender about changes in buffer size by sending appropriate messages over the control channel.

Since FTP is a bidirectional data-transfer protocol, the receiver may be either the server or client. RFC 959 specifies that only clients may send commands on the control channel, while servers may only send replies to commands. Thus, a new command and reply must be added in order to fully implement drsFTP. Serendipitously, the Internet Draft of the GridFTP protocol extensions to FTP [34] defines a 'SBUF' command, 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> <ID> <SP> <size>
  <ID> ::= <number>
  <size> ::= <number>
```

This command requests the server-PI (server protocol interpreter) to set the send-buffer size to `<size>` bytes, assuming sufficient buffer space is available. `<ID>` is provided to match a `SBUF` command to its reply. `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, thus providing interoperability with existing, non-drsFTP, applications.

Response Codes:

```
200 SBUF <SP> <ID> <SP> <size>
```

The server-PI issues a 200 response code containing the `<ID>` of the corresponding command and the new size of the server's buffer. `<ID>` allows the client-PI (client protocol interpreter) to match replies to commands in case multiple `SBUF` commands are outstanding in the active transfer. `<size>` allows the client-PI adjust its buffer usage in case the server-PI chooses to allocate less than the requested amount of buffer space.

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

New Response Code:

```
126 SBUF <SP> <ID> <SP> <size>
```

A 126 response may be sent by the server-PI while it is receiving data from the client-PI. As with the `SBUF` command, this reply is informational and need not be acted upon or responded to in any manner by the client-PI. A non-drsFTP application will simply ignore the reply, guaranteeing interoperability with a drs-FTP server.

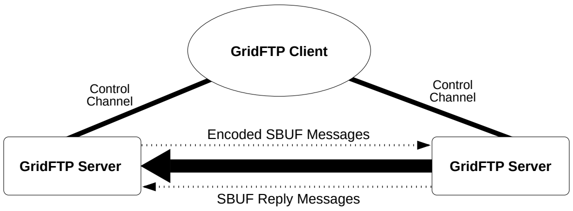

Fig. 4. Data and Control Flow for a Three-Party Transfer.

This response 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 need only contain an `<ID>`, we allow SBUF commands to serve the dual purpose of conveying the receiver's buffer size to the sender and probing for the RTT. Separate RTT probes, as discussed in Section 4.1.2, are not needed in most instances. Separate probes only become necessary if the time between buffer-size changes becomes so large that the RTT becomes stale. Since the mechanism for determining RTT via `SBUF` messages is already in place, "empty" `SBUF` messages with the current buffer size serve as the RTT probe in this case.

4.1.5 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.

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 buffer-size limits of the operating system may need to be increased. The steps involved in increasing the limits are operating system dependent. See [38] for an example of the steps required for a variety of operating systems.

5 DRS in GridFTP

In order to maintain strict compatibility with the FTP specification, drsFTP only supports two-party transfers. Figure 4 shows a three-party transfer in which the client coordinates a data transfer between two servers. Because SBUF messages travel over the control path in drsFTP, the RTT for a three-party transfer is incorrect. Three-party transfers are easily supported if control commands and their replies can be sent and received on the data channel between servers. GridFTP [34], part of the Globus Toolkit [35], already extends the FTP specification in ways that provide most of the needed support for three-party transfers with DRS. The additional features of GridFTP include:

- Support for secure transfers.
- Parallel data transfers — where the data may be transferred in parallel streams between two nodes,
- Striped data transfers — where the data may be transferred to multiple nodes,
- Partial file transfers,
- Automatic negotiation of TCP buffer sizes.

The mechanisms provided by the last feature allow SBUF messages to be exchanged on the data channel. These messages are shown as dashed lines in Figure 4.

Like the drsFTP server, the current GridFTP server is a version of the `wu-ftpd` server, modified to support most of the GridFTP protocol extensions. The GridFTP protocol extends the FTP specification by including an *Extended Block Mode* (*Mode E*). Mode E has 64-bit offset and length fields in the header and supports out-of-order transmission. It also supports parallel or striped transfers.

5.1 Determining RTT over the Data Channel

In drsFTP, RTT probes are sent on the control channel. This implicitly assumes that the path delay on the control channel is the same as on the data channel. If the control and data paths do not follow the same route, as is the case with three-party transfers, the RTT computed by drsFTP will be incorrect. As three-party transfers are an important and oft-used feature of FTP, particularly in scientific computing, a different approach is needed to estimate RTT.

The GridFTP specification adds two commands for setting TCP buffer sizes. The ABUF command sets buffer sizes automatically at connection setup time by actively probing the network for the RTT and available bandwidth. The ABUF command is not implemented in the current version of GridFTP. The SBUF command is used to set the buffer sizes at a remote node. (As mentioned in Section 4.1.4, we extend the semantics of the SBUF command to allow it to be sent at any time during a data transfer.) Because DRS-enabled GridFTP continuously modifies buffer sizes as appropriate during a transfer, the need for the ABUF command is eliminated.

Figure 5 shows structure of the extended block header. The eight-bit descriptor field indicates the type of transfer. We define two new descriptor codes, using two unallocated bits in the descriptor field as shown in Figure 6.

Descriptor	Byte Count = <buffer-size>	Offset Count (unused)

Fig. 5. SBUF Message Encapsulated in a Mode E Header.

Descriptor Code	Meaning
01	SBUF message
02	Reply to SBUF message

Fig. 6. SBUF Descriptors in a Mode E Header.

Upon expiration of an alarm, the receiver sends a extended block header to the sender without a payload. The descriptor field in the header indicates an encoded SBUF message, while the `byte-count` field is set to the value of the receiver's current buffer-size. When the sender receives a SBUF packet on the data channel, it attempts to set its buffers to the value contained in the `<byte-count>` field. It also sends an acknowledgement in an outgoing extended block header in a manner similar to the way TCP piggy-backs acknowledgements.

When the receiver obtains the acknowledgement, it calculates the RTT as the difference between the acknowledgment reception time and the SBUF send time. As in Section 4.1.2, using SBUF messages to compute RTT, as well as for conveying buffer sizes, eliminates the need to inject extra traffic into the network.

Note that estimating RTT requires the data channel to be bidirectional during a data transfer, i.e., full-duplex communication must be supported over the data channel. This is an extension to the current GridFTP implementation. By estimating RTT over the data channel, parallel and striped transfers are seamlessly supported.

5.2 Determining Bandwidth in DRS-enable GridFTP

Bandwidth is computed in the same way as described in section 4.1.1. The average throughput over a sampling interval is calculated on the receiving end upon the periodic expiration of an alarm. The sampling interval can be adjusted to obtain values of sufficient accuracy.

6 Experiments

In this section, we present results for both drsFTP and DRS-enabled GridFTP. In particular, we will show that the throughput improves by over 600%.

6.1 Performance of drsFTP

The experimental apparatus consists of three identical machines connected via Fast Ethernet (100 Mbps). The machines need only be fast enough to ensure that the hosts are not the bottleneck. Each machine contains dual 400 MHz Pentium II processors with 128 MB of RAM

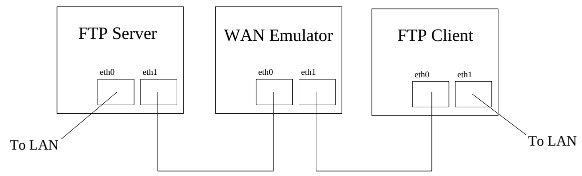

Fig. 7. Experimental Apparatus

and two network-interface cards (NICs). One machine acts as a WAN emulator with a 102 ms round-trip time (RTT) delay; each of its NICs is connected to one of the other machines via a switch.

The WAN emulator, which is implemented using TICKET technology [39], forwards packets at line rate and has a user-settable delay. (In the results that follow, the average round-trip time is 102.1 ms.) All FTP traffic, both data and control, occurs through the WAN emulator.

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 32 or 64 KB. Therefore, this number represents the high end of OS-default TCP buffer sizes.) We next test drsFTP, allowing the buffer size to vary in response to network conditions, starting from 64 KB. Last of all, we test statically-tuned FTP with TCP buffers sizes chosen to represent over- and under-provisioning.

The over-provisioned buffer size, representing the best performance possible, is 16 MB, which is larger than the BDP (12.2 MB). The under-provisioned buffer size is 212.5 KB, which represents a BDP that is sampled when the network is loaded. (The median value of BDP for the data in Figure 3 is 143.3 KB. A buffer size of 212.5 KB is in the 66th percentile.)

For each test, we transfer a set of files, ranging from 8 KB to 64 MB, over the emulated WAN. The drsFTP sampling interval used to estimate the available bandwidth is one second, a conservative configuration with very low overhead. (The performance is not sensitive to the duration of the sampling interval as long as the sampling interval is greater than the round-trip time. This is an artifact of not emulating cross-traffic.)

Figure 8 shows the average FTP bandwidth as a function of the size of the transfer. (The x-axis has markers placed according to the powers-of-two file sizes tested. The width of the 95% confidence interval is less than $\pm 5\%$ 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 by 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

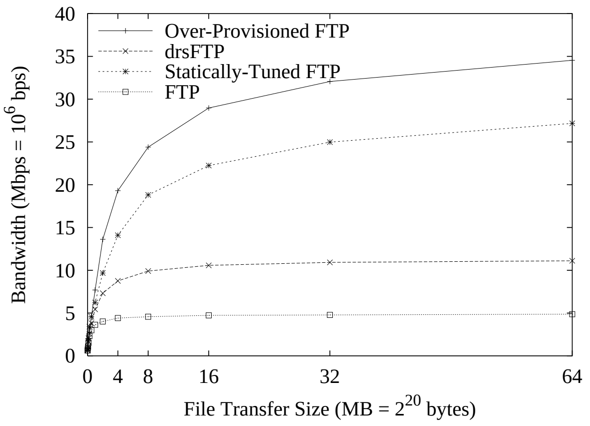

Fig. 8. Comparison of FTP, drsFTP and statically-tuned FTP

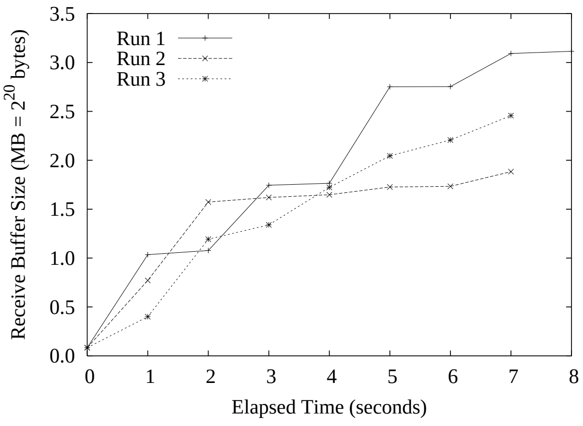

Fig. 9. drsFTP Buffer Sizes over Time

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 grow the buffer sizes quickly enough. This is an inherent limitation indicative of the interim nature of the drsFTP application. Even though its performance is not as good as the kernel-space implementation [26, 27], drsFTP was developed to provide the benefits of DRS to the grid community while vendors implement DRS in their kernels.

Figure 8 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 3. 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 optimizing transfer rates. We now turn our attention to buffer usage. As motivation, we conjecture that memory consumption will become a more serious issue as computational grids become widely used and hence 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 9 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.

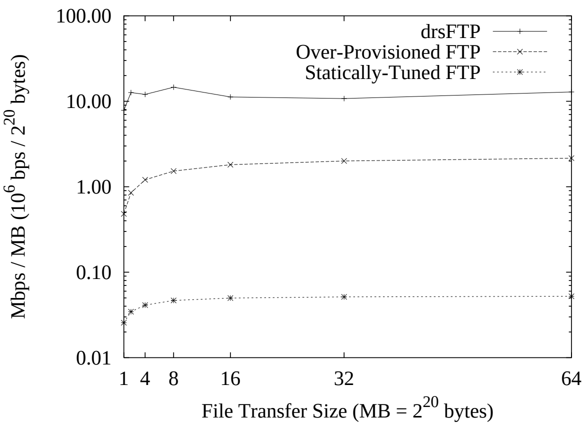

Fig. 10. Mbps per MB of Buffer Space

As Figure 10 shows, drsFTP achieves five times better utilization of the network with respect to memory than the over-provisioned case. Even if the theoretically optimal BDP of 12.2 MB been allocated instead of over-provisioning, drsFTP would still have been able to support more connections with a 3.6 times improvement in Mbps per MB. The difference between drsFTP and the statically-tuned case where the BDP was sampled at an inopportune time is even more dramatic.

6.2 Performance of DRS-enabled GridFTP

In this section, we present the performance of DRS-enhanced GridFTP on three-party transfers.

The experimental apparatus consists of four identical machines, as shown in Figure 11. Each machine contains dual 500 MHz Pentium III processors with 1 GB of RAM and two 100 Mbps network-interface cards (NICs). One machine acts as a WAN emulator with a 102.1 ms RTT. Each of its NICs is connected to one of the server machines via a switch. The final machine acts as the client.

The three-party data transfer is initiated by establishing a control channel connection with the receiving host. The client sends a PASV command to the receiving host to instruct it to listen for a connection from the sending host. The response to the PASV command is a host and port address. Next, the client connects with the sender and issues a PORT containing the host and port addresses it obtained from the receiver. The PORT command instructs the sender to use these values for the data connection.

We test GridFTP with default, statically tuned over-provisioned and DRS-tuned buffer sizes. The files sizes range from 1 MB to 512 MB. As shown in Figure 12, the average bandwidth of GridFTP with default buffer sizes is 4.7 Mbps for the 512 MB transfer. In contrast, the average bandwidth of DRS-enabled GridFTP for the same file size is 30.0 Mbps, an increase of 633%. The average bandwidth of GridFTP with statically over-provisioned buffer sizes is 38.7 Mbps. Thus, DRS-enabled GridFTP achieves 77.5% of the over-provisioned performance without the need to tune the buffers by hand.

Figure 13 shows the performance of GridFTP for various numbers of parallel streams. The average bandwidths on a 512 MB transfer are 25.9, 79.3 and 84.8 Mbps for the default, DRS-tuned and over-provisioned buffer sizes, respectively. Even with parallel streams, DRS-enabled GridFTP delivers 306% of the bandwidth of the stock case using equal numbers of streams. In order to achieve the same performance as DRS-enabled GridFTP with 4 streams, GridFTP with default buffer sizes would require approximately 14 streams. Using multiple streams with stock buffers is not scalable since multiple streams increase operating system (OS) overhead. Finally, we note that the performance difference of DRS-enabled GridFTP with respect to GridFTP using over-provisioned buffers appears to be independent of the number of streams for large files sizes.

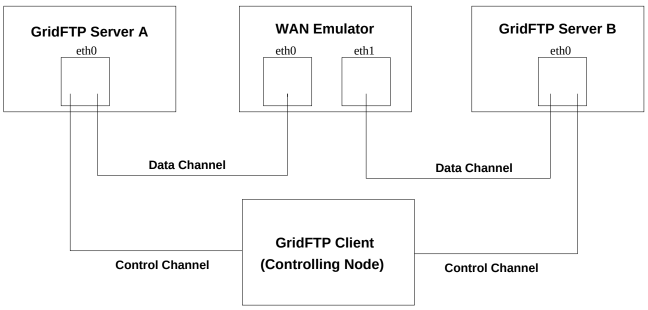

Fig. 11. Experimental Setup for Three-Party Data Transfer

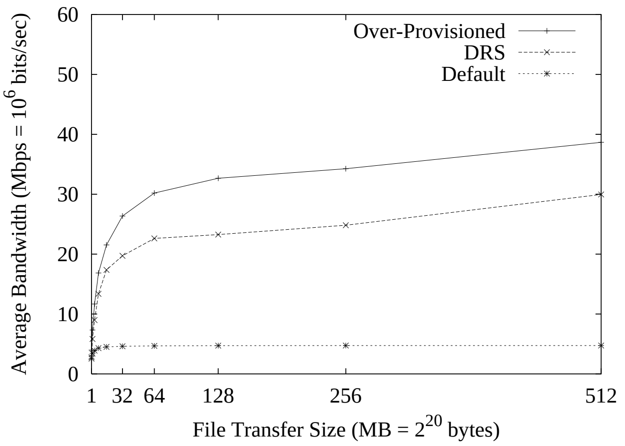

Fig. 12. Bandwidth Comparison for a Three-Party Transfer

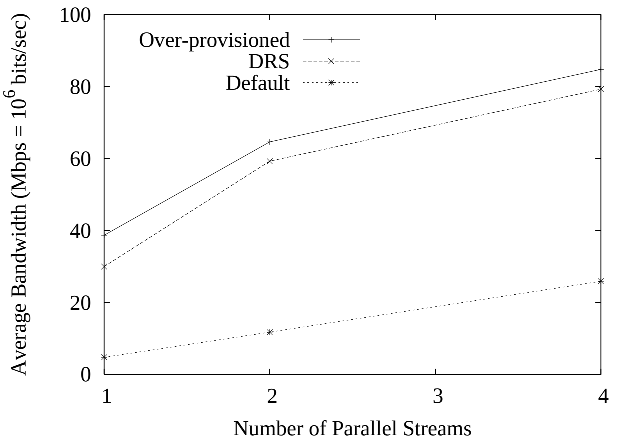

Fig. 13. Bandwidth as a Function of the Number of Streams

Figure 14 compares the delivered bandwidth compared with the amount of buffer space used. We use the Mbps per MB metric we define in Section 6.1. DRS-enabled GridFTP achieves 36.4 Mbps per MB of buffer used, while GridFTP with over-provisioned buffers achieves 23.0 Mbps per MB, a difference of 158%. The ratio is even greater for smaller file sizes (e.g., the performance of DRS-enabled GridFTP is 16.9 times better than GridFTP with over-provisioned buffers at a file size of 1 MB file).

Finally, we note that the performance of DRS-enabled GridFTP for two-party transfers is indistinguishable from the three-party case. This is to be expected as two-party transfers are a degenerate case of three-party transfers in which one of the end points is also the client.

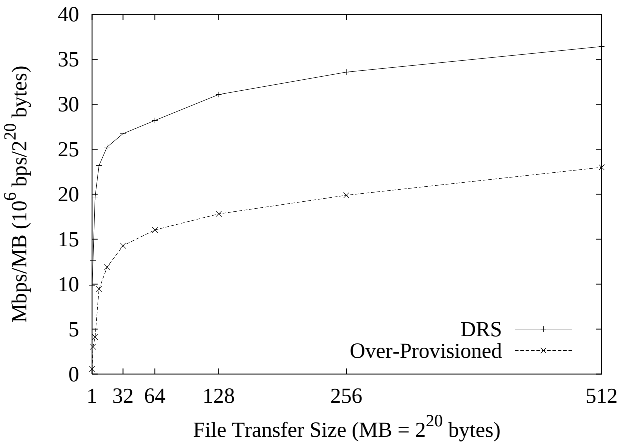

Fig. 14. Delivered Bandwidth vs. Buffer Sizes

7 Future Work

The results of the experiments conducted so far indicate that DRS, both kernel- and user-space, is likely to perform well in the real world. Anecdotally, we have observed very good performance on live networks but still need to rigorously quantify the improvements. We need to do more testing on connections with low and medium bandwidth-delay products. We also need to test DRS with varying amounts of cross traffic.

Although we have shown preliminary results which indicate that DRS is complementary to parallel streams as a means of increasing throughput for large data transfers, the interaction between DRS and parallel streams needs to be better characterized. Does DRS subsume the need for parallel streams or should a combined approach, as we implemented in GridFTP, be used? The results seem to indicate that a hybrid approach will yield the best performance for the fewest numbers of streams (and hence OS resources).

Finally, we are working to get DRS incorporated into the official Linux source tree. Once incorporated, applications will transparently see an increase in delivered bandwidth. In the meantime, we are continuing to develop drsFTP and GridFTP.

8 Conclusion

This paper makes a number of significant contributions to the high-speed networking and grid-computing communities. First, we demonstrated that the bandwidth-delay product (BDP) can vary widely over the lifetime of a connection. Therefore, simply tuning buffers at connection set-up is not good enough; they must be tuned over the lifetime of the connection. This is the motivation for dynamic right-sizing (DRS).

Further, since we used `nettimer` to measure the BDP of a connection, our estimates may be conservative because `nettimer` measures *static* bottleneck bandwidth and *dynamic* delay. With the recent release of `pathload` [40], which measures dynamic available bandwidth and delay, our initial tests indicate that the BDP actually fluctuates by an additional order of magnitude.

Second, we illustrated how a receiver can measure the bandwidth and round-trip time of a connection (i.e., BDP) *without* "polluting" the network with extraneous probing packets. The BDP value is then used as an upper bound for the flow-control window in DRS.

Third, in the context of DRS, we have shown how a TCP receiver can determine the approximate size of the sender's congestion window so that the receiver can advertise a flow-control window that neither needlessly constrains throughput nor unnecessarily over-allocates buffer space. Furthermore, this can be done automatically and transparently while abiding by TCP semantics. We have shown how it can be done in user space and have implemented it in drsFTP.

Fourth, we have demonstrated DRS support for grid computing by extending the GridFTP application. Unfettered by constraints imposed by the FTP specification, we have extended GridFTP to support three-party, parallel and striped transfers that are ubiquitous in grids computing.

Finally, we are making our implementations of DRS available under an open-source license. The DRS kernel implementation has already been incorporated into the Web100 project [41]. We are also working to fold the modifications necessary to support DRS into the GridFTP code base.

References

[1] I. Foster, C. Kesselman (Eds.), The Grid: Blueprint for a New Computing Infrastructure, Morgan Kaufmann Publishers, 1998.

[2] I. Foster, C. Kesselman, S. Tuecke, The anatomy of the grid: Enabling scalable virtual organizations, International Journal of Supercomputer Applications .

[3] The particle physics data grid, `http://www.cacr.caltech.edu/ppdg/`.

[4] A. Chervenak, I. Foster, C. Kesselman, C. Salisbury, S. Tuecke, The Data Grid: Towards an architecture for the distributed management and analysis of large scientific datasets, International Journal of Supercomputer Applications 23 (3) (2001) 187–200.

[5] L. Childers, T. Disz, R. Olson, M. E. Papka, R. Stevens, T. Udeshi, Access Grid: Immersive group-to-group collaborative visualization, in: Proceedings of the 4th International Immersive Projection Workshop, 2000.

[6] M. Allman, et al., Ongoing TCP research related to satellites, IETF RFC 2760 (2000).

[7] M. Allman, D. Glover, L. Sanchez, Enhancing TCP over satellite channels using standard mechanisms, IETF RFC 2488 (1999).

[8] C. Partridge, T. Shepard, TCP/IP performance over satellite links, IEEE Network .

[9] W. Feng, P. Tinnakornsrisuphap, The failure of TCP in high-performance computational grids, in: Proceedings of SC 2000: High-Performance Networking and Computing Conference, 2000.

[10] D. Bansal, H. Balakrishnan, S. Floyd, S. Shenker, Dynamic behavior of slowly-responsive congestion control algorithms, in: SIGCOMM 2001, San Diego, CA, 2001.
URL nms.lcs.mit.edu/publications.html

[11] S. Floyd, Highspeed TCP for large congestion windows, http://www.ietf.org/internet-drafts/draft-floyd-tcp-highspeed-01.txt (Aug 2002).

[12] T. Kelly, On engineering a stable and scalable TCP variant, Tech. Rep. CUED/F-INFENG/TR.435, Cambridge University Engineering Department, http://www-lce.eng.cam.ac.uk/~ctk21/papers/sstcp-variant.pdf (Jun 2002).

[13] Pittsburgh Supercomputing Center, Enabling high-performance data transfers on hosts, http://www.psc.edu/networking-/perf_tune.html/.

[14] B. Tierney, TCP tuning guide for distributed applications on wide-area networks, in: USENIX & SAGE Login, 2001, http://www-didc.lbl.gov/tcp-wan.html.

[15] G. Jin, G. Yang, B. Crowley, D. Agrawal, Network Characterization Service, in: Proceedings of the IEEE Symposium on High-Performance Distributed Computing, 2001.

[16] K. Lai, M. Baker, Nettimer: A tool for measuring bottleneck link bandwidth, in: Proceedings of the USENIX Symposium on Internet Technologies and Systems, 2001.

[17] Lawrence Berkley National Laboratory, Nettest: Secure network testing and monitoring, http://www-itg.lbl.gov/nettest/.

[18] B. Mah, pchar: A tool for measuring internet path characteristics, http://www.employees.org/~bmah/Software/pchar.

[19] A. Tirumala, J. Ferguson, IPERF, http://dast.nlanr.net/Projects/Iperf/index.html (2001).

[20] University of Kansas, Information & Telecommunication Technology Center, NetSpec: A tool for network experimentation and measurement, http://www.ittc.ukans.edu/netspec/.

[21] J. Liu, J. Ferguson, Automatic TCP socket buffer tuning, in: Proceedings of SC 2000: High-Performance Networking and Computing Conference (Research Gem), 2000, http://dast.nlanr.net/Projects/Autobuf.

[22] National Center for Atmospheric Research and Pittsburgh Supercomputing Center and National Center for Supercomputing Applications, The Web100 Project, http://www.web100.org/.

[23] B. Tierney, D. Gunter, J. Lee, M. Stoufer, Enabling network-aware applications, in: Proceedings of the IEEE International Symposium on High-Performance Distributed Computing, 2001.

[24] M. Mathis, Pushing up performance for everyone, http://www.ncne.nlanr.net/news/workshop/19999/991205/Talks/mathis_991205_Pushing_Up_Performance/.

[25] J. Semke, J. Mahdavi, M. Mathis, Automatic TCP buffer tuning, Computer Communications Review, ACM SIGCOMM 28 (4).

[26] M. Fisk, W. Feng, Dynamic adjustment of TCP window sizes, Tech. Rep. Los Alamos Unclassified Report (LAUR) 00-3221, Los Alamos National Laboratory (July 2000).

[27] M. Fisk, W. Feng, Dynamic Right-Sizing: TCP flow-control adaptation, in: Proceedings of SC 2001: High-Performance Networking and Computing Conference, 2001.

[28] E. Weigle, W. Feng, Dynamic Right-Sizing: A simulation study, in: Proceedings of IEEE International Conference on
Computer Communications and Networks, 2001, http://public.lanl.gov/ehw/papers/ICCCN-2001-DRS.ps.

[29] M. Gardner, W. Feng, M. Fisk, Dynamic Right-Sizing in FTP: Enhancing grid performance in user space, in: Proceedings of the IEEE Symposium on High-Performance Distributed Computing, 2002.

[30] National
Laboratory for Applied Network Research, Automatic TCP window tuning and applications, http://dast.nlanr.net/Projects/Autobuf_v1.0/autotcp.html.

[31] NcFTP software client and server, http://www.ncftp.com/.

[32] The washington university archive FTP daemon (wu-ftpd), http://www.wu-ftpd.org/.

[33] The Netkit FTP client, http://freshmeat.net/projects/netkit/.

[34] 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).

[35] The Globus Project, http://www.globus.org/.

[36] V. Jacobson, R. Braden, D. Borman, TCP extensions for high performance, IETF RFC 1323 (1992).

[37] J. Postel, J. Reynolds, File Transfer Protocol (FTP), IETF RFC 959 (1995).

[38] Pittsburgh Supercomputing Center, Enabling high performance data transfers on hosts, `http://www.psc.edu/networking/perf_tune.html`.

[39] E. Weigle, W. Feng, TICKETing High-Speed Traffic with Commodity Hardware and Software, in: Proceedings of the Third Annual Passive and Active Measurement Workshop (PAM2002), 2002.

[40] M. Jain, C. Dovrolis, End-to-end available bandwidth: Measurement methodology, dynamics, and relation with TCP throughput, in: Proceedings of the Annual Conference of the Special Interest Group on Data Communication (SIGCOMM), 2002.

[41] T. Dunigan, F. Fowler, Personal communication with Web100 project, 2002.