Improving Distributed Application Performance Using TCP Instrumentation

Brian L. Tierney, Jason R. Lee, Dan Gunter, Martin Stoufer
Lawrence Berkeley National Laboratory

Tom Dunigan
Oak Ridge National Laboratory

Abstract

Distributed scientific computing applications are requiring an ever-increasing amount of bandwidth. Raw network bandwidth has been increasing rapidly, but getting the applications to effectively use this additional bandwidth is becoming more and more difficult. In this paper we describe how TCP instrumentation can be used to guide more efficient use of the network, using a monitoring daemon that we have developed to passively collect TCP instrumentation data from the kernel for a given application. We describe how this daemon was used to analyze GridFTP data flows with some experimental TCP protocol enhancements.

1.0 Introduction

One of the primary goals of high performance distributed computing is efficient use of all computing resources, particularly the network. The only way to ensure efficient use of the network is by closely monitoring the behavior of the network protocols such as TCP. In a Grid environment, not only must each network connection be as efficient as possible, but a Grid scheduler or replica selection service must also be able to choose the right set of connections to use. To ensure that a distributed application is running as efficiently as possible is to perform complete end-to-end monitoring [14]. One important component often overlooked in end-to-end monitoring is the TCP stack. However, access to information from the TCP stack is invaluable for monitoring distributed applications.

TCP instrumentation data is useful for many purposes, including:

- Monitoring applications use of the network: TCP instrumentation data can tell us what throughput an application is achieving, averaged over the entire duration of the connection, or tracked in short intervals over time, giving an indication of the burstiness of the application. TCP instrumentation can also tell if an application is network, send host, or receive host limited. This passively collected information can be used in place of active probes and used as input to Grid scheduling or replica selection services.
- Tuning the TCP parameters for a given connection: TCP instrumentation can be used to determine if the TCP send or receive buffers are too small, if parallel streams are helping, if `txqueuelen` on Linux should be increased, and so on.
- Performing analysis on experimental protocols using real Grid applications: TCP instrumentation can be used to examine in detail how new protocols such as High-Speed TCP [6], Scalable TCP [16], or FAST TCP [20] compare to standard TCP using actual running applications on production networks.

To get information from the TCP stack, we are using Web100 [33], an implementation of an IETF Internet Draft TCP MIB [22] for the Linux operating system. Web100 instruments the TCP stack to expose over 100 TCP variables for each active connection. Web100 does not provide, however, an easy way to isolate the TCP instrumentation data for a particular application.

We have developed a monitoring and tuning daemon, called the work around daemon (WAD), that uses Web100 to collect TCP stack information for one or more user-specified applications. The WAD can also modify TCP parameters for a given application. This allows it to automatically tune running applications, thus providing a work around to misconfigured networks or hosts. A previous paper focused on the tuning capabilities of the WAD

[5]. This paper s main focus is on the instrumentation capabilities of the WAD and the types of analysis made easier by using the WAD to collect TCP instrumentation data.

In this paper will describe how the WAD could be used in a Grid environment to provide data needed to make Grid scheduling decisions, and for application tuning and debugging. We also present some results demonstrating the use of the WAD for comparing GridFTP [1] performance using standard TCP, High-Speed TCP, and Scalable TCP.

2.0 Use Cases

The following sections describe several use cases for TCP instrumentation data.

2.1 Achievable Throughput Monitoring

One particularly useful statistic that we can derive directly from TCP instrumentation data is achievable throughput [19] [21]. *Available bandwidth* is the amount of bandwidth currently available on the network, while *achievable throughput* is the throughput actually possible, given the end-system hardware (CPU speed and load, network interface card (NIC), I/O Bus speed, disk speed), operating system, TCP stack, TCP parameters, and so on. Available bandwidth is not particularly useful to Grid middleware, but achievable throughput information is often needed.

For example, a Grid Replica Manager [2] would like to select a data source based on which source can deliver data at the highest rate. This decision should not be based on available network bandwidth if the bottleneck is actually the disk or the network interface card (NIC). The correct way to estimate disk-to-disk throughput is to use disk-to-disk performance data. One could run periodic tests to measure this; this is the approach taken by the IEPM project [4]. However such tests can be quite intrusive to the network and both end hosts. Active probes add load to the network and are often not a good indicator of achievable bandwidth, because they may not accurately simulate the real application. For example, using an *iperf* probe [23], which does memory-to-memory tests, does not accurately simulate GridFTP, which is disk-to-disk.

Using Web100 and the WAD, one can passively monitor the achieved throughput from actual production Grid Services, rather than using active probes.

Of course, one can also measure achievable throughput by instrumenting the Grid Service, such as a GridFTP server [32], directly. But using the WAD ensures that all applications are described with an identical set of measurements, and does not require Grid application developers to worry about this type of instrumentation. It also makes it easier to publish this achievable throughput information via a standard interface.

The resulting TCP instrumentation data can then be used as input to prediction algorithms and made available to Grid scheduling services. For example, using Web100, one can collect the values for `DataBytesOut`, `StartTime`, and `EndTime` for a given TCP session. The achievable throughput for that transfer is then `DataBytesOut / (EndTime - StartTime)`. If desired, the effects of TCP slow start could be factored out of this calculation by using Web100 information to determine when the connection is out of slow start, and only measuring the throughput starting at that point. This is similar to the method used by *iperf quick mode* [31]. Calculation results could be fed into the NWS prediction algorithms [34] and then published in MDS for use in making scheduling decisions.

In addition to a single sample of the TCP connection, the WAD can also periodically sample the Web100 variables over the lifetime of the connection, such as once every second. These samples can be used to characterize a particular application s use of the network. For example, it is useful to see if a remote visualization application requires a fairly steady stream of data, or if data is requested in large bursts followed by a pause. TCP instrumentation can also be used to look for correlations between these bursts of application data requests and packet loss in the network.

2.2 TCP Tuning

There are a number of system parameters that effect TCP throughput. Some of these factors are easy to correct and others are much more subtle. In this section we briefly describe some typical tuning issues for the Linux operating system, and how TCP instrumentation data and the WAD can be used to identify or correct the problem.

The most common TCP tuning problem is the failure to increase the TCP buffer size. This problem has been very well understood for many years [18] [30], yet most applications still do not use large enough TCP buffers. For

example, in April 2003, Internet 2 monitoring of bulk data transfer traffic (TCP transfers of more than 10 MBytes) showed that over 95% of the flows were achieving less that 10 Mbits/second [15]. Web100 includes tools to make it easy to determine if the TCP sending or receiving buffers are too small [3], and the WAD can automatically increase the size of buffers that are too small, as described in [5].

In some situations it may actually be desirable to *decrease* the TCP buffer size. Large TCP buffers allow for large TCP congestion windows, which in turn allow for large bursts of packets to arrive at each router. If a router is congested, this large burst of packets will lead to packet loss. Experiments have shown that reducing the TCP buffer size can be an effective way to reduce packet loss and thus increase overall TCP throughput [19].

Another important tuning parameter for Linux is called `txqueuelen`. In Linux, there is a single queue that all sockets use to pass packets from the operating system to each network interface card. If this queue fills up, Linux generates a `SendStall`, and TCP backs off in the same manner as if it detected a lost packet. For Linux hosts with certain types of 1000 BT NICs, `SendStall` s are actually quite common. Using Web100 it is easy to detect these `SendStall` events, and the WAD can automatically increase the size of `txqueuelen` until they stop occurring. By default, `txqueuelen` is set to 100 packets, but it can be easily increased using the (UNIX) `ifconfig` command. For more information, see http://www-didc.lbl.gov/TCP-tuning/.

Another issue in tuning TCP for applications such as GridFTP is selecting the optimal number of parallel streams. A large amount of work has been done to try to determine the optimal and fair number of parallel streams to use [4] [12] [13] [19]. The WAD makes it easy to associate multiple streams with a particular application (described below), and Web100 can be used to help determine if parallel streams are helping or hurting. For example we can see if all streams are getting an equal share of the bandwidth, and see if additional streams lead to additional congestion events.

To summarize, TCP instrumentation data is necessary to properly tune TCP, and the WAD helps collect this data for particular applications.

2.3 Grid Scheduling

TCP instrumentation data can be used to measure the throughput that a bulk data transfer application such as GridFTP is achieving. This data can then be published in a Grid Information Service such as the Globus MDS. The tuning parameters used to achieve this throughput such as TCP buffer size and number of parallel streams can also be monitored and published.

Using this achievable throughput information, higher-level Grid services can be built to answer queries such as: How long will it take to transfer 5 GBytes from Host A to Host B using GridFTP with 4 data streams? The answer to this question might be based on results for the previous transfer, based on results for a previous transfer that was same time of day and the same day of the week, based on some prediction algorithm, and so on. Grid Scheduling Services and Grid Replica Management Services would use such as service to collect data to make scheduling and data placement decisions.

2.4 Protocol Experiments

In this section we describe some proposed modifications to TCP that allow it to work better on high-speed networks. TCP instrumentation is necessary to observe how these TCP modifications work with real applications on real networks or on testbeds like NISTNet [24]. Results for this are presented in Section 4.0 below.

In order to understand the changes that are proposed to TCP, it would be first necessary to have a basic understanding of standard TCP and how TCP congestion avoidance works. Most TCP implementations in use today are based on TCP Reno [17]. TCP Reno congestion avoidance works as follows.

TCP s congestion management is composed of two algorithms: slow-start and congestion avoidance. The slow-start and congestion avoidance algorithms allow TCP to increase the data transmission rate without overwhelming the network. TCP s congestion window (CWND) is the size of the sliding window used by the sender. TCP cannot inject more than CWND segments of unacknowledged data into the network.

TCP Reno s algorithms are referred to as AIMD (Additive Increase Multiplicative Decrease) and are the basis for its steady-state Congestion Control. TCP Reno increases the congestion window by one packet per window of data acknowledged and halves the window for every window of data containing a packet drop.

It is now well known that standard TCP Reno does not scale to large bandwidth delay product networks. For example, for a TCP connection with 1500-byte packets and a 100 ms round-trip time, achieving a steady-state throughput of 10 Gbps would require an average congestion window of 83,333 segments, and a packet drop rate of at most one congestion event every 5,000,000,000 packets (or equivalently, at most one congestion event every 1 2/3 hours) [6]. This is widely acknowledged as an unrealistic constraint. Solutions to this problem include parallel streams, High-Speed TCP, Scalable TCP, and FAST TCP. In this paper we present results for High-Speed TCP and Scalable TCP.

Network applications can be designed or retrofitted to use parallel TCP streams. Parallel streams take advantage of the fact that TCP tries to share the bandwidth equally among all flows along a path. Performance is improved not only by getting a bigger share of the bandwidth, but *K* parallel streams will have *K* times larger aggregate buffer size. Slow-start will start *K* times faster. If a loss occurs in one stream, the multiplicative decrease (MD) will not be TCP's standard *1/2*, rather the congestion window will be reduced by only *1/(2K)*. The linear recovery can be *K* times faster (*K* segments per RTT rather than one), if all *K* streams are already in linear recovery [13].

High-Speed TCP (HS-TCP) modifies TCP s response function in order to obtain high performance in high-speed environments and in the presence of packet loss. The target environment for HS-TCP is 10Gbps performance in 100ms Round Trip Times (RTT) environments. HS-TCP uses the current congestion window size to calculate the current AI and MD values. High-bandwidth, high-delay paths will have larger window sizes and thence more aggressive values for AI and MD. For more information on HS-TCP behavior in a number of scenarios, see [26].

An alternative solution is Tom Kelly s Scalable TCP. Using Scalable TCP, the TCP response function is proportional to round trip time only, rather than both the window size and round trip time. Instead of adjusting both AI and MD as in HS-TCP, Scalable TCP takes the approach that MD should always be 0.125 (i.e.: reduce window by 1/8 on congestion events), and instead of adding 1/CWND to CWND for each ACK received, (e.g., 1%). As a result, TCP can recover in a fixed number of RTT's. For example, on a 10 Gbps, 200ms RTT link, Scalable TCP will always recover in 2.7 seconds. By comparison, standard TCP takes over 4 hours to recover! A larger MTU or virtual MSS [5] can also improve TCP performance on high bandwidth-delay networks.

3.0 TCP Instrumentation and Analysis Components

We now describe in more detail the various software components used to perform TCP analysis presented in Section 4.0, below.

3.1 Web100

The Web100 project [33] is an NSF funded collaboration between the Pittsburgh Supercomputing Center (PSC), the National Center for Atmospheric Research (NCAR) and The National Center for Supercomputing Applications (NCSA). Web100 exposes the statistics inside the TCP stack itself through an enhanced standard Management Information Base (MIB) for TCP [22]. This MIB uses TCP s ideal vantage point to provide statistics for diagnosing performance problems in both the network and the application. If a network-based application is performing poorly, TCP information from Web100 allows us to determine if the bottleneck is in the sender, the receiver, or the network itself. If the bottleneck is in the network, TCP can provide specific information about its nature. Web100 also provides an interface to set certain TCP settings on active sockets, such as the TCP buffer size.

The current Web100 implementation is based on extensions and modifications to the Linux 2.4 kernel. Web100 variables are contained in a data structure attached to the kernel s socket data structure. An application reads and sets the Web100 variables using the Linux */proc* interface using an API provided in the Web100 distribution. TCP connection start and end events are provided to an application (e.g., the tuning daemon) through the *netlink* service.

3.2 Work-Around Daemon (WAD)

We have developed a monitoring and tuning daemon for the Web100 kernel called the Work-Around Daemon, or WAD. The name comes from the WAD s original goal, which was to use Web100 tuning mechanisms to work around problems with TCP flows in a particular network or application. For more information on the WAD tuning options, see [5]. In this paper we are using the WAD in a read-only mode for monitoring TCP, not for TCP tuning. The WAD daemon is notified asynchronously of socket open/close events, and then data is collected by polling at some user specified interval.

The WAD first detects a TCP connection by listening on the Web100 *netlink socket* -- a communication mechanism used for kernel notifications to user space. The daemon then consults a configuration file that specifies which flows (source, source port, destination, destination port) are of interest. When the WAD is used for tuning a connection, the configuration entry for a tunable flow also includes a set of tuning parameters such as maximum ssthresh, AIMD parameters or algorithms (e.g., HS-TCP, scalable TCP), reordering threshold, and so on.

The WAD can be used to monitor any Web100 variable for any TCP flow. For example, the WAD can measure the congestion window, packet retransmissions, timeouts, and smoothed RTT times of any socket directly from Web100 variables. The WAD can also be configured to generate *derived events* from combinations of Web100 variables. For example, one could generate average and instantaneous bandwidth by placing the following text in the appropriate configuration file:

```
AveBW = (DataBytesOut*8)/(CurrTime - StartTime)
LastIntervalBW = (Delta_DataBytesOut*8)/ (Delta_SndLimTimeRwin +
    Delta_SndLimTimeCwnd + Delta_SndLimTimeSender)
```

The WAD can write any subset of the derived and raw variable values as NetLogger events [28] and send these events to the NetLogger visualization tool, *NLV*, for visual analysis of TCP streams.

The WAD also has the ability to use plug-in modules, that can execute arbitrary additional commands. One use of plug-in modules is to generate *derived* connections, which can in turn be monitored. For example, for GridFTP the WAD only needs to monitor the well-known TCP control port (2811), and a plug-in module can be used to derive the data connections for a given transfer. The plug-in does this by examining the control connection s source host, source port, destination host, and destination port, and then using the fact that the GridFTP server will by default connect back to the same client host/port pair for data connections. The plug-in then sets up the ports to monitor for the data connections, and passes them back to the WAD for monitoring. Using variations on this technique, we can monitor any application that dynamically allocates ports.

It is worth noting that Globus GridFTP service has some features that both help and hinder the technique outlined above. To get around firewall issues, GridFTP has the facility to allow the sysadmin to specify what ports to use for the data channels by setting the GLOBUS_TCP_PORT_RANGE variable. The WAD can by configured to monitor these ports directly, but there is no guarantee that these ports are not used by some other application. GridFTP also supports third party transfers, where the endpoints of the data channels are not the same as the endpoints of the control socket. In order for the WAD to monitor third party transfers the GLOBUS_TCP_PORT_RANGE must be used, and the WAD be configured to monitor those ports.

Other applications of plug-ins allow for the monitoring of system data on a specific host. For example if we are doing GridFTP transfers and we want to correlate the disk and network interface activity with the network flow of the application, we can simply plug-in a module to log disk activity along with the WAD s standard TCP information for the application flow.

3.3 NetLogger

We are using several components of the NetLogger Toolkit to generate, collect, and analyze this TCP instrumentation data, and to correlate it with other sources of monitoring data. Since 1994 researchers at Lawrence Berkeley National Lab have been developing a toolkit for end-to-end instrumentation of distributed applications called NetLogger [28]. Using NetLogger, distributed application components are modified to produce timestamped traces of interesting events at all critical points of the distributed system. The NetLogger Toolkit itself consists of four components: an API and library of functions to simplify the generation of application-level event logs, a service to collect and merge monitoring from multiple remote sources, a monitoring event archive system, and a tool for visualization and analysis of the log files.

NetLogger events can be formatted as an easy to read and parse ASCII format, or an efficient, self-describing binary format [9]. NetLogger also includes reliability support and a remote activation mechanism. The NetLogger Reliability API provides tolerance for temporary network failures that are common in Grid environments. Using the NetLogger Activation Service, the level of monitoring and log destination in a running application can be controlled across the network. For instance, the WAD can be activated to send its NetLogger results to a particular destination during a particular application run.

The NetLogger Toolkit also includes a data analysis component. The NetLogger Visualization tool, *NLV*, provides an interactive graphical representation of system-level and application-level events. NetLogger s ability to correlate detailed application instrumentation data with host and network monitoring data has proven to be a very useful tuning and debugging technique for distributed application developers. As an example, see Figure 1.

4.0 Results

In this section we show how TCP instrumentation data generated by Web100, collected by the WAD, and analyzed using NetLogger and gnuplot can be used to tune TCP, to better understand application use of the network, and to analyze experimental TCP enhancements.

4.1 TCP Analysis Results

To demonstrate the TCP analysis ability, we show the results of using Web100 and the WAD to analyze the performance of *iperf*, a simple tool to test memory-to-memory throughput between two hosts, to determine why *iperf* was not obtaining the available bandwidth on a particular path.

Figure 1 shows a graph of several Web100 variables along with CPU utilization and instrumented *iperf* events. Web100 counters are collected every 0.3 seconds using the WAD. CPU utilization data is collected every second (cpu.utilization.user and cpu.utilization.sys), and *iperf* has been instrumented with NetLogger to generate monitoring events before and after all I/O operations (StartRead/EndRead in the iperf server, and StartWrite/EndWrite in the iperf client). The numbers on the right side of the plot are the range of values for that monitoring event and the numbers without units are a count of the number of times an event occured.

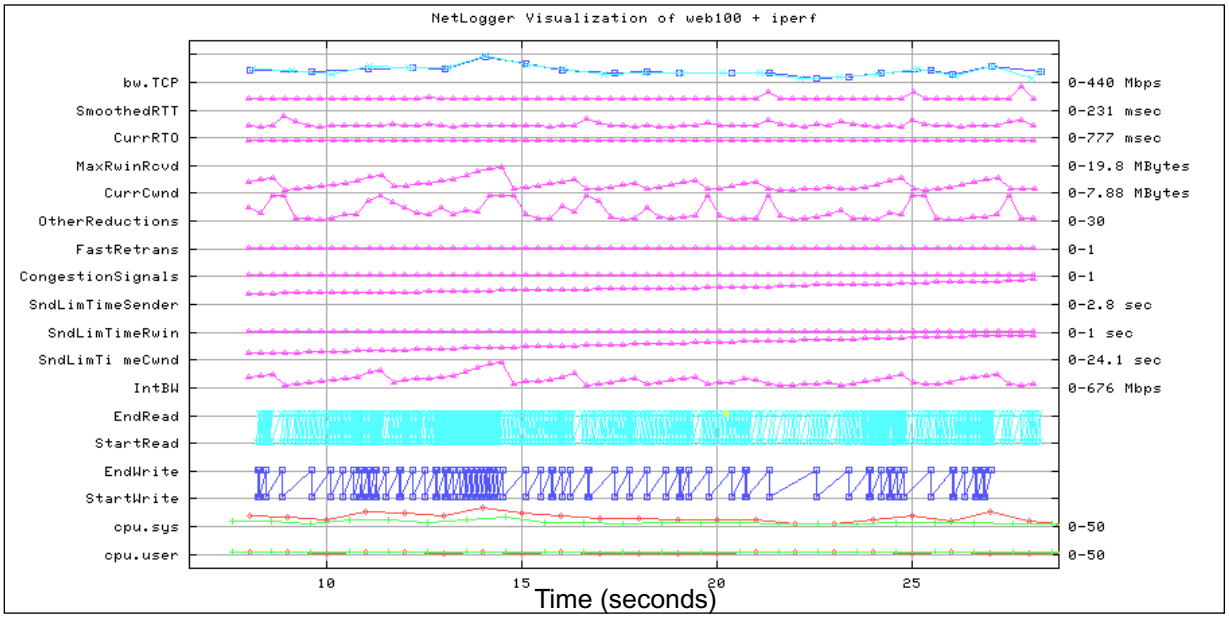

Figure 1: TCP analysis correlated with CPU and application monitoring

There is an immense amount of information in this plot, which we will attempt to explain. From bottom to top:

- CPU user and system load information: the two colors represent the sender and receiver host. Note that system CPU increases when `CurrCWND` is large
- `StartWrite` and `EndWrite` are from the *iperf* client, and represent the time to write a 512 KByte block from user space to kernel space
- `StartRead` and `EndRead` are from the *iperf* server, and represent the time to read a 512 KByte block from kernel space to user space. Note that the client writes are much more bursty than the server reads.
- `IntBW`: a WAD computed value of the bandwidth achieved since the last measurement (0.3 seconds)
- `SndLimTimeCwnd`, `SndLimTimeRwin`, and `SndLimTimeSender`: These are Web100 *sender congestion triage* variables that help determine whether the sender, receiver, or the network is the bottleneck

- `CongestionSignals`: Web100 sum of all types of congestion events, including Fast Retransmit, ECN, and timeouts.
- `OtherReductionsCM`: Other than during congestion, there are a few other situations where the Linux 2.4 TCP implementation reduces the congestion window. One of these is that Linux includes a routine called *tcp_moderate_cwnd*, which reduces CWND whenever it thinks there are more packets in flight than there should be based on CWND. This algorithm appears to be specific to Linux, and based on no known IETF document. Note that since there are no other congestion signals recorded for this run, OtherReductionsCM are clearly the cause of CWND being reduced.
- `CurrCwnd`: current TCP congestion window
- `MaxRwinRcvd`: This is the maximum TCP window size that the receiver is telling the sender it can use.
- `CurRTO`: current value of the TCP RTO (Retransmission Timer) measurement, used to determine when a TCP time-out should occur.
- `SmoothedRTT`: TCP s internal notion of the round-trip time.
- `bw.TCP`: average bandwidth since the start of the test, as reported by *iperf*.

Note that the WAD records both value (current value) and delta (difference from the previous value) for each event. With NLV, you can specify which you wish to graph. For some events (e.g.: `CongestionSignals`) it is better to graph the deltas, but with other events (e.g.: `CurrCWND`) you want to see a trace of the current values.

The careful observer may have noticed that CWND appears to recover from congestion faster than standard TCP would allow. This is because for this test we were using an implementation of the High-Speed TCP algorithm, which more aggressively recovers from congestion events when the congestion window is large.

In Figure 1 we see that the maximum bandwidth peaked around 660 Mbps (`IntBW`), but that the average was only around 200 Mbps. This was due to the fact that CWND was continuously reduced by the Linux 2.4 *tcp_moderate_cwnd* routine. We are trying to determine why this routine exists, but currently believe that it should be removed.

More details on how to produce plots such as the one shown in Figure 1 are given in [29].

4.2 TCP Tuning Results

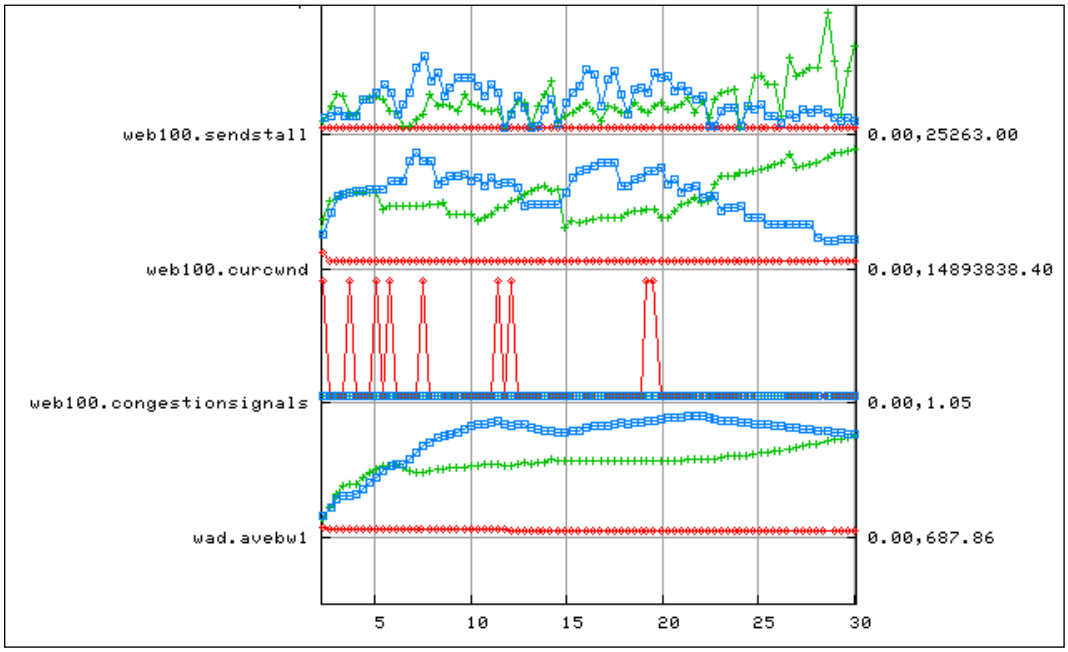

Figure 2: web100 variables for TCP tuning

In this example we show how TCP instrumentation can be used to help tune TCP on a given host. Figure 2 shows the TCP analysis for three parallel streams. Note that curCwnd is constantly being pulled down, and often when there is no congestion event. This is caused by *txqueuelen* being too small, causing several `SendStall` events, and leading to a reduced congestion window. Figure 2 also shows that for this transfer the use of three parallel streams did not really help, as two streams got almost all of the bandwidth.

4.3 GridFTP Results

We now present results investigating how High-Speed TCP and Scalable TCP behave with a real Grid application, GridFTP. Other studies show the results of HS-TCP in a simulated environment [26], but it useful to see what happens in a real environment with real applications.

The test environment is the following. A 2 Gigabyte file is copied from Lawrence Berkeley National Lab to Oak Ridge National Lab. Both end hosts are 1.2 GHz Pentium III hosts running Linux 2.4.20 with the Web100 patch. The network round trip time is 67 ms and the bottleneck link is 622 Mbits/sec. TCP buffers were set to 4 MBytes, *txqueuelen* was set to 2000 and CWND caching was turned off (using the command: `sysctl -w net.ipv4.route.flush=1`). All tests were run several times and typical results are shown below.

The following WAD configuration file was used to collect GridFTP data:

```
[ GridFTP]
src_addr: 0.0.0.0  # capture data for all source addresses
src_port: 2812     # capture data for source port 2812 only
dst_addr: 0.0.0.0  # capture data for all destination addresses
dst_port: 0        # capture data for all destination addresses
plugin: gridftp    # use the gridftp plugin module, which specifies to
                        #  also collect data for data ports created
                        #  after a connection on port 2812
#
[ NetLogger]       # generate the following NetLogger events
wad.AveBWI: (Delta_DataBytesOut*8)/(Delta_SndLimTimeRwin +  \
                  Delta_SndLimTimeCwnd + Delta_SndLimTimeSender)
web100.CongestionSignals: CongestionSignals
web100.SendStall: SendStall
web100.PktsRetrans: PktsRetrans
web100.CurCwnd: CurCwnd
#
[ PyWAD]
outputdest: file:///tmp/gridftp.log     # send results to this file
polltime: 1.0                           # collect results every second
```

Figure 3 shows results comparing standard TCP with HS-TCP, plotting the size of the congestion window, sampled once per second, of the lifetime of the transfer. From this figure one can clearly see that the TCP congestion window recovers from loss much faster using HS-TCP compared to standard TCP. This faster recovery results in higher throughput for the network application.

Figure 4 shows a comparison between HS-TCP and Scalable TCP. The plot for Scalable TCP, single stream (upper left) does not show the expected results for Scalable TCP, as CWND goes back to 1448 (1 packet) instead of being reduced by just 1/8th. Looking at other Web100 data collected by the WAD (`RetranThresh`) for this run showed a large number of out-of-order packets. In the case where the number of out of order packets is greater than 3, Linux does a TCP time-out, and resets CWND to the initial value. For the parallel stream tests, it appears that all of the streams experience loss during slow-start. Further analysis of this is underway, and will be included in the final version of this paper. However, detailed analysis of HS-TCP and Scalable TCP is beyond the scope of this paper. Our goal here is to demonstrate that the combination of Web100, the WAD and NetLogger make this type of analysis possible.

This experiment shows that using the WAD it is easy to collect TCP information, and derived statistics of that information such as `AveBWI`, so that real application performance can be visualized and analyzed.

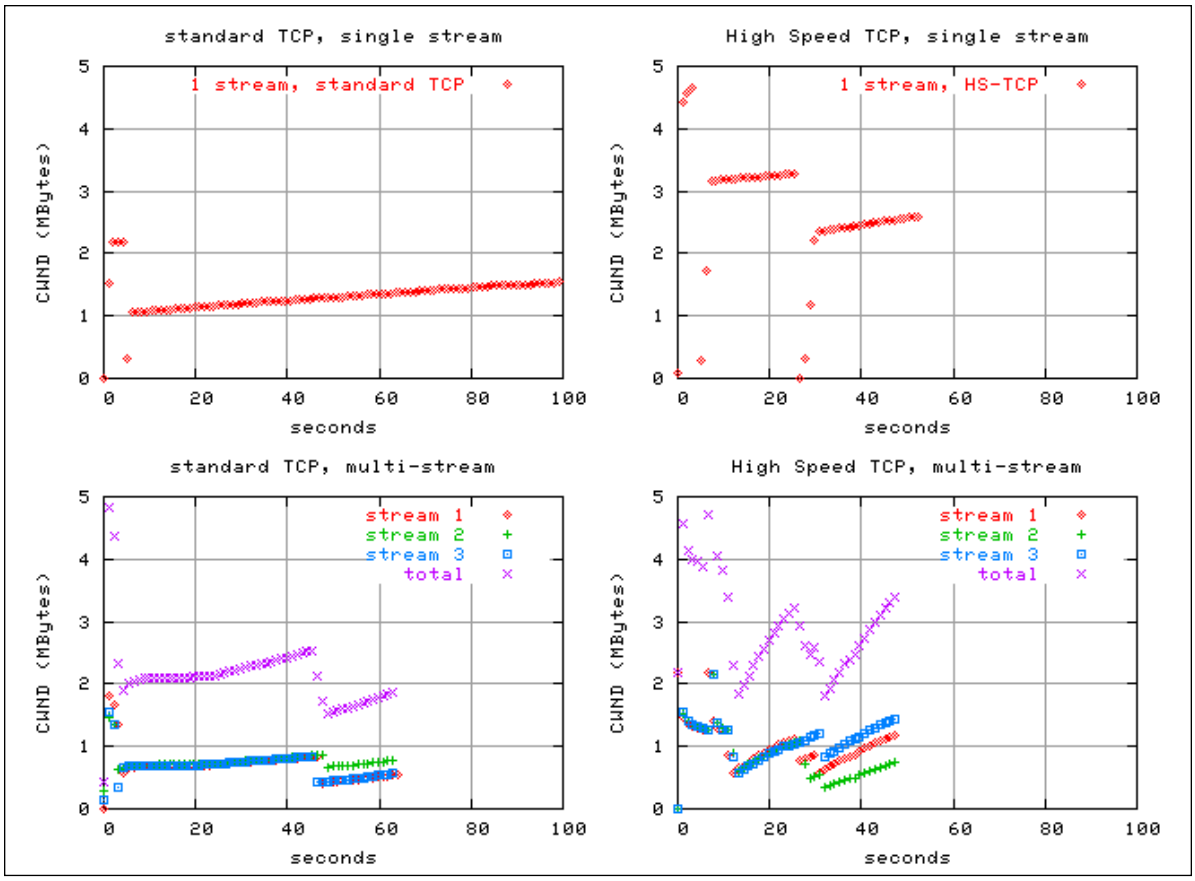

Figure 3: GridFTP Results comparing standard TCP and HS-TCP

5.0 Related Work

Web100 provides a number of tools to analyze TCP connections. For example, the Web100 *gutil* program [10] provides a nice mechanism for exploring Web100 variables for a selected TCP stream. Unlike the WAD, the Web100 tools are all aimed at real-time analysis of active connections and require the user to select which socket to monitor. The WAD can monitor specified applications without any user interaction.

tcptrace [27] is very good for visually exploring tcpdump files. In isolation, though, it does not replace the need for Web100 instrumentation data. For example, events such as `SendStalls` are not collected by tcpdump. Also, *tcptrace* does not provide any ability to correlate this information with other types of monitoring, such as CPU monitoring, a task that is straightforward with the NetLogger visualization tool (NLV).

MAGNeT [7] also provides the mechanisms required to instrument TCP, as well as other aspects of the kernel. MAGNeT has been used in conjunction with profiled applications to determine how the effects of kernel processes, such as scheduling, on network intensive applications. MAGNeT can monitor the data path from the kernel through the TCP stack all the way to the NIC. This amount of detail complements the profiling of the TCP stack to see where spurious values (like large RTO's) may be caused by a kernel scheduling conflict rather then congestion on the network.

There are several system such as Supermon [25] and Ganglia [8] that can read and publish various /proc variables. However only Web100 puts detailed TCP instrumentation information into /proc.

6.0 Conclusions and Future Work

Scaling TCP to very large bandwidth-delay product networks has proven to be very challenging. The only way to ensure efficient use of the network is by closely monitoring the behavior of the network protocols such as TCP.

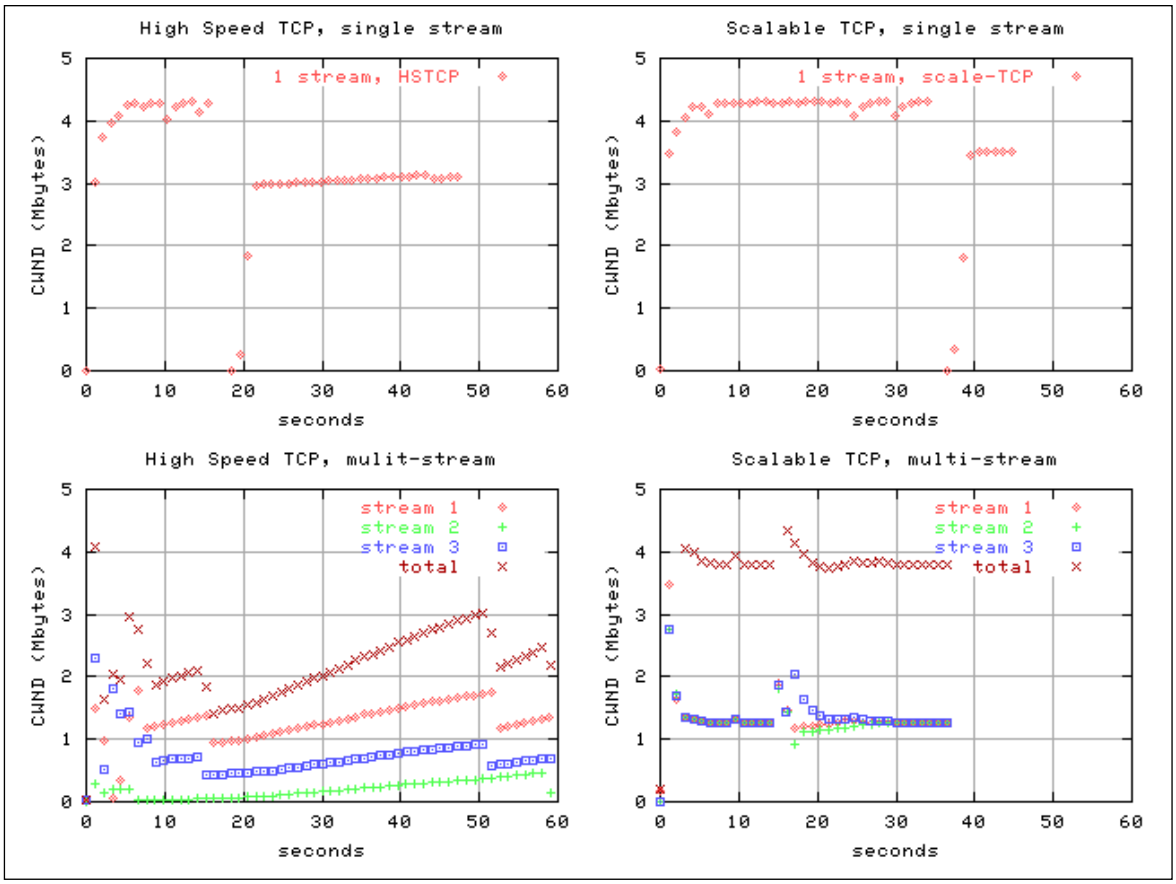

Figure 4: GridFTP Results comparing HS-TCP and Scalable TCP

Although TCP instrumentation is important, it is often omitted from an end-to-end monitoring system. In this paper, we have attempted to show that this omission is a mistake; access to information from the TCP stack can be very valuable for understanding and tuning distributed applications. We have also shown how TCP instrumentation data can be used for performance analysis and tuning, as well as for testing new, experimental protocols.

We plan to add an implementation of FAST TCP to the Web100 kernel, so that we can better compare HS-TCP, Scalable TCP, and FAST TCP.

7.0 Acknowledgments

We wish to thank all the members of the Net100 project (http://www.net100.org/) for their help with this work. This work was supported by the Director, Office of Science. Office of Advanced Scientific Computing Research. Mathematical, Information, and Computational Sciences Division under U.S. Department of Energy Contract No. DE-AC03-76SF00098. See the disclaimer at http://www-library.lbl.gov/disclaimer. This is report no. LBNL-52590.

8.0 References

[1] Allcock, B., J. Bester, J. Bresnahan, et.al., *Data Management and Transfer in High Performance Computational Grid Environments*. Parallel Computing Journal, Vol. 28 (5), May 2002, pp. 749-771.

[2] Chervenak, A., I. Foster, C. Kesselman, et. al. *The data grid: Towards an architecture for the distributed management and analysis of large scientific datasets*. In Network Storage Symposium (NetStore 99), 1999.

[3] Carlson, R., *Developing a Web100 Based Network Diagnostic Tool*, Proceedings of the Passive and Active Monitoring Workshop, San Diego, April 2003. http://www.pam2003.org/listPapers.html

[4] Cottrell, L., Connie Logg, I-Heng Mei, *Experiences and Results from a New High Performance Network and Application Monitoring Toolkit,* Proceedings of the Passive and Active Monitoring Workshop, San Diego, April 2003.

[5] Dunigan, T., M. Mathis and B. Tierney, *A TCP Tuning Daemon*, Proceeding of IEEE Supercomputing 2002 Conference, Nov. 2002, LBNL-51022.

[6] Floyd, Sally, *HighSpeed TCP for Large Congestion Windows*, IETF Internet Draft, http://www.ietf.org/internet-drafts/draft-floyd-tcp-highspeed-02.txt

[7] Gardner, M., W. Feng, and J. Hay, *Monitoring Protocol Traffic with a MAGNeT*, Passive and Active Measurement Workshop, (PAM2002), Fort Collins, Colorado, Mar. 2002.

[8] Ganglia: http://ganglia.sourceforge.net/

[9] Gunter, D., et. al. *Dynamic Monitoring of High-Performance Distributed Applications*. in 11th IEEE Symposium on High Performance Distributed Computing. 2002.

[10] *gutil*, http://www.web100.org/docs/man/gutil.html

[11] Handley, M., J. Padhye, S. Floyd, *TCP Congestion Window Validation,* June 2000, http://www.ietf.org/rfc/rfc2861.txt

[12] Hacker, T., B. Athey, and B. Noble. *The End-to-End Performance Effects of Parallel TCP Sockets on a Lossy Wide-Area Network*. Proc. 16th IEEE-CS/ACM International Parallel and Distributed Processing Symposium (IPDPS), 2002. URL: http://www.cnaf.infn.it/ ferrari/papers/tcp/IPDPS.pdf.

[13] Hacker T. and B. Noble, *The Effects of Systemic Packet Loss on Aggregate TCP Flows*, Proceeding of IEEE Supercomputing 2002.

[14] Hollingsworth J, and B. Tierney, *Instrumentation and Monitoring*, in The Grid, Volume 2. Morgan Kaufman, 2003.

[15] Internet2 NetFlow: Weekly Reports, 2002. URL: http://netflow.internet2.edu/weekly/.

[16] Kelly, T., *Scalable TCP: Improving Performance in HighSpeed Wide Area Networks,* First International Workshop on Protocols for Fast Long-Distance Networks, 2003

[17] Jacobson, V., *Congestion avoidance and control*, in Proceedings of the ACM SIGCOMM 88 Conference on Communications Architectures and Protocols, vol. 18, Stanford, CA, August 1988, pp. 314—329.

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

[19] Jin, G. and B. Tierney, *Netest: A Tool to Measure Maximum Burst Size, Available Bandwidth and Achievable Throughput*, Proceedings of the 2003 International Conference on Information Technology Aug. 10-13, 2003.

[20] Jin, C., D. Wei, S. H. Low, et. al; *FAST TCP: From Theory to Experiments*, submitted to IEEE Communications Magazine, April 1, 2003

[21] Lowekamp, B., B. Tierney, L. Cottrell, R. Hughes-Jones, T. Kielmann, M. Swany, *A Hierarchy of Network Performance Characteristics for Grid Applications and Services*, Global Grid Forum Draft, Network Measurements Working Group, http://www-didc.lbl.gov/NMWG/docs/measurements.pdf

[22] Mathis, M., R. Reddy, J. Heffner, and J. Saperia. *TCP Extended Statistics MIB*. IETF draft, work in progress, November 2002. URL: http://www.ietf.org/internet-drafts/draft-ietf-tsvwg-tcp-mib-extension-03.txt.

[23] NLANR. *iperf*, URL: http://dast.nlanr.net/Projects/Iperf/.

[24] NISTNet, 2001, http://snad.ncsl.nist.gov/itg/nistnet/

[25] Sottile M. and R. Minnich. *Supermon: A High-Speed Cluster Monitoring System*. In Proc. of IEEE Intl. Conference on Cluster Computing, 2002.

[26] Souza, E. and D. Agarwal, *A HighSpeed TCP Study: Characteristics and Deployment Issues*, submitted to IEEE Supercomputing 2003.

[27] *tcptrace*: http://www.tcptrace.org/

[28] Tierney, B., et al. *The NetLogger Methodology for High Performance Distributed Systems Performance Analysis*. in Proc. 7th IEEE Symp. on High Performance Distributed Computing. 1998.

[29] Tierney, B., *Using NetLogger and Web100 for TCP Analysis*, Invited Paper, First International Workshop on Protocols for Fast Long-Distance Networks, 2003, LBNL-51776.

[30] Tierney, B., *TCP tuning Guide for Distributed Applications on Wide Area Networks*, Usenix ;login Journal, Feb., 2001, p33. LBNL-45261. http://www-didc.lbl.gov/TCP-tuning/

[31] Tirumala, A., L. Cottrell, T. Dunigan, *Measuring end-to-end bandwidth with Iperf using Web100,* Proceedings of the Passive and Active Monitoring Workshop, San Diego, April 2003. http://www.pam2003.org/

[32] Vazhkudai, S., J. M. Schopf, I. Foster. *Predicting the Performance of Wide Area Data Transfers*. Proceedings of the 16th International Parallel and Distributed Processing Symposium (IPDPS 2002), April 2002.

[33] *web100* project: http://www.web100.org/

[34] Wolski, R., *Dynamically Forecasting Network Performance Using the Network Weather Service*, in Journal of Cluster Computing, Volume 1, pp. 119-132, January, 1998.