A TCP Tuning Daemon *

Tom Dunigan
Oak Ridge National Laboratory (ORNL)
thd@ornl.gov

Matt Mathis
Pittsburgh Supercomputing Center (PSC)
mathis@psc.edu

Brian Tierney
Lawrence Berkeley National Laboratory (LBNL)
bltierney@lbl.gov

July 29, 2002

Abstract

Many high performance distributed applications require high network throughput but are able to achieve only a small fraction of the available bandwidth. A common cause of this problem is improperly tuned network settings. Tuning techniques, such as setting the correct TCP buffers and using parallel streams, are well known in the networking community, but outside the networking community they are infrequently applied. In this paper, we describe a tuning daemon that uses TCP instrumentation data from the Unix kernel to transparently tune TCP parameters for specified individual flows over designated paths. No modifications are required to the application, and the user does not need to understand network or TCP characteristics.

Keywords: autotuning, TCP, high-performance networking, data grids

1 Introduction

Many high-performance distributed computing applications transfer large volumes of data over wide area networks and require data rates on the order of megabits per second. Even though Internet backbone speeds have increased considerably in the last few years due to projects like Internet 2 and NGI, distributed applications rarely take full advantage of these new high-capacity networks. In fact, recent data for Internet 2 (April, 2002) show that 90% of the bulk TCP flows (defined as transfers of at least 10MB of data) use less than 5 Mbits/sec, and that 99% use less than 20 Mbits/sec out of the possible 622 Mbits/sec. [28]

By design, TCP/IP was built with robustness as the primary goal, and hence hides most problems. When something goes wrong, TCP silently compensates at the expense of reduced performance. Some of these problems are TCP configuration problems (e.g., small buffer space or features such as SACK being improperly negotiated). Other problems are due to the applications (mainly small messages or pauses in the data flow). Many problems are in the network, including routing problems, *leaky networks* (networks with non-congestive packet loss), and excessive packet reordering.

*This research is a collaboration of the Pittsburgh Supercomputing Center (PSC), the National Center for Atmospheric Research (NCAR), Lawrence Berkeley National Laboratory (LBNL), and Oak Ridge National Laboratory (ORNL) and is supported by the High-performance networking program, Office of Science, U.S. Department of Energy under Contract Numbers DE-AC05-00OR22725 (ORNL), DE-AC03-76SF00098 (LBNL), and DE-FC02-01ER25470 (PSC).

There exists a large body of work showing that the end systems can achieve good performance over high-capacity wide-area networks through a number of techniques, including proper manual tuning, system autotuning and network-aware [8, 45] applications (see section 5). Typically, operating system-based approaches to improving TCP throughput are based on "standard" ideas that do not impact assumptions about fairness when competing with other traffic in the network [20, 43]. Application-based approaches are often more aggressive about acquiring network capacity. Examples include most non-TCP based transports [14, 32, 36] and all parallel TCP approaches [19, 25, 44, 49].

This paper describes a daemon-based mechanism to implement both standard and non-standard TCP adjustments for individual flows using network metrics across designated paths. No modifications to the application are required, and the user does not need to understand network and TCP characteristics. We call this tuning daemon the *Work Around Daemon* (WAD), because it provides a transparent mechanism to work around a variety of network issues, including TCP buffer size, MTU size, packet reordering, and leaky network loss. We believe the WAD is the first step toward moving away from the need for *network-aware applications*, and toward *network-aware operating systems*. Many tuning techniques validated by the WAD may become standard for future TCP implementations. However, some of the tuning techniques discussed in this paper are not suitable for future standardization because they are true workarounds, requiring prior out-of-band information about specific limitations of the network path.

The desired outcome of the WAD development work is to eliminate what has been called the "wizard gap" [33], which is the difference between the network performance that a network "wizard" can achieve by manually hand-crafting the optimal tuning parameters, compared to an untuned application. The WAD, as a transparent intermediary, can act as that wizard. This frees the distributed application developer from needing to understand the wide variety of available monitoring tools and tuning methods.

The network tuning parameters that the WAD is initially concentrating on are those required by large bulk data transfer applications, such as global climate modeling and the various Data Grid [12] projects. These include the Particle Physics Data Grid [42], GriPhyN [6], and the EU DataGrid [18]. These projects all require efficient transfer of very large scientific data files across high-delay, high-bandwidth networks.

In the next section, we give a brief review of the TCP transport protocol issues. Section 3 describes the implementation of our tuning daemon using Web100 with a Linux kernel. Section 4 describes experiences with tuning TCP in various network settings. Section 5 describes related work. The final section summarizes our efforts on the WAD and considers future extensions.

2 Background

TCP is a window based protocol: new data is transmitted into the network when old data has been received as indicated by acknowledgments flowing from the receiver to the sender. The data rate is determined by the total amount of outstanding unacknowledged data in the network, referred to as "the window size." The window size (or data rate) is limited by the application, the buffer space at the sender or the receiver and by the congestion window, which ideally reflects only congestion feedback from the network. TCP adjusts the congestion window (using "slow-start", "congestion avoidance", and related algorithms [4]) to find an appropriate share of the network capacity of the path between the source and destination. TCP repairs missing or corrupted data segments by retransmitting data from the retransmit queue within the sender's buffer to the reassembly queue in the receiver's buffer. This recovery process requires that an entire window of data fit into both sender's and receiver's buffers. These buffers generally have default sizes, which can be changed by the application by using a system library call. In addition, the operating system may have limits that restrict the maximum buffer size that the application can request.

Ideally, TCP throughput should be only limited by some intrinsic bottleneck such as disk data rate or network path capacity. However, in practice, throughput is often limited by the send or receive buffer sizes or by an artificially small congestion window induced by some problem in the network. The WAD, on a per flow basis, looks at choosing optimum buffer sizes to eliminate buffer size constrictions and to limit the effect of spurious congestion indications by minimizing loss by using more appropriate buffer sizes for the path, moderating slow-start, and reducing packet bursts. If there is packet loss, the WAD seeks to speed recovery by using a virtual MSS, modifying TCP's congestion control parameters, disabling delayed ACKs, or using tuned parallel streams. These methods are described in detail below.

2.1 TCP Buffer Size

The buffer sizes must be adjusted for both the send and receive ends of the TCP socket. To get maximal throughput it is critical to use optimal send and receive socket buffer sizes for the path. If the buffers are too small, the TCP window can not fully open. It wastes memory to allocate buffers that are too large. Although memory is comparably cheap, the vast majority of the connections are so small that allocating large buffers to each flow can put any system at risk of running out of memory. The optimal TCP window size is usually twice the available bandwidth-delay product for the path, where delay is the one-way latency of the path.

As network throughput capacity has increased in recent years, operating systems have gradually changed the default buffer size from common values of 8 kilobytes to as much as 64 kilobytes. However, this is still far too small for today's high speed networks. For example, there are several hosts of the Particle Physics Data Grid [42] with 1000 Mbits/sec (GigE) network interfaces connected via an OC12 (622 Mbits/sec) WAN, with typical network round trip latencies of about 50 ms. Buffer sizes need to be twice the bandwidth-delay product, so for this network, the TCP buffer should be roughly (600 Mbps / 8 bits/byte x .05 sec) = 3.75 MBytes. Using a default TCP buffer of 64 KB, the maximum utilization of the network path will only be about 2% under ideal conditions.

2.2 Use of Parallel TCP Streams

To work around problems such as buffers that are too small or loss-rates that are too high, some applications have started using parallel data streams. Even with properly tuned TCP buffers, parallel streams can improve performance. Figure 1 shows the advantage of using tuned TCP buffers and parallel streams in the GridFTP program for 100 MByte data transfers between Lawrence Berkeley National Lab in Berkeley, California, and CERN in Geneva, Switzerland. The round trip time (RTT) on the connection was measured with ping to be 180 ms and the bottleneck hop was an OC-3 (155 Mbit/sec) link. With different tuning parameters, actual measured transfer speeds spanned more than an order of magnitude. Tuned TCP buffers alone provided more than a 20x performance increase, and parallel sockets alone yielded about a 30x performance improvement, but it required a very large number of streams. Using just 6 parallel streams with tuned TCP buffers, we were able to outperform 30 untuned streams.

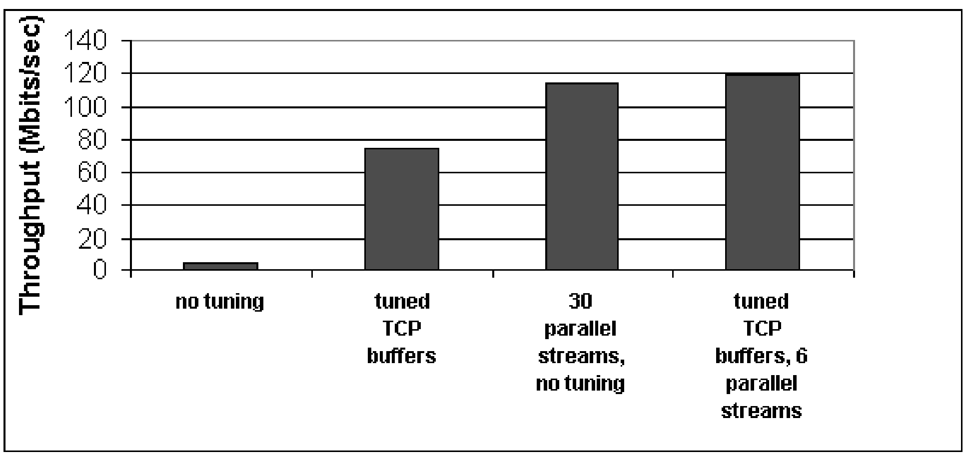

Figure 1: GridFTP results using tuned TCP buffers and parallel streams.

Packet loss is interpreted by TCP as an indication of network congestion between a sender and receiver. Packet loss, however, may be due to other factors, such as intermittent hardware faults or physical layer distortions. In fact, measurements have shown that about 1 in 7500 datagrams pass the CRC checksum but fail the TCP or UDP checksum, indicating that the data was damaged in transit [46]. TCP treats segments lost or damaged in transit as loss and goes into one of its congestion avoidance algorithms. Therefore, the use of parallel streams provides an increase over optimally tuned single stream TCP because even if one or more of the streams experience random loss and slow down, the other streams keep sending and help to keep the pipe full, allowing the application to utilize a greater fraction of the network.

Several data transfer tools now include the ability to use parallel streams. These include GridFTP [1, 2], bbftp

[19], and bbcp [26]. The *psockets* library from the University of Illinois makes it easy for application developers to add parallel sockets to their applications [44]. Note that parallel streams require explicit support from the application. This is the only tuning technique currently under consideration for the WAD that requires application support. One of our objectives is to have a single WAD-tuned TCP flow perform as well as a set of parallel streams over the same path.

In general, using large TCP buffers and parallel streams improves throughput, so it may be tempting for users or developers to simply use big buffers and some parallel streams by default. However, using buffers that are too big can waste system memory and in some cases reduce throughput. Using too many parallel streams or the wrong buffer size for the parallel stream can also reduce throughput and add to network congestion [25].

2.3 MTU Issues

Whenever possible TCP sends data in the largest possible segments. The MSS (maximum segment size) is computed by deducting TCP/IP header sizes from the MTU (maximum transmission unit) of the network interfaces along the path [30, 37]. Today the typical TCP MSS is 1448 bytes, due to the 1500 Byte Ethernet MTU.[1] During congestion avoidance TCP's congestion window,*cwnd*, is opened (raised) by one MSS segment per round-trip time. A larger MSS would improve TCP's recovery speed and reduce the packet interrupt rate of the operating system. Other media support a larger MTU: FDDI has a 4392-byte MTU, GigE jumbo-frames have a 9000-byte MTU, IP-over-ATM uses a 9180-byte MTU, and HiPPI uses a 65535-byte MTU. While acknowledging that the user does not have any control over the MTU's along the path, we propose extensions to the kernel that would allow the WAD to select a virtual MSS which is larger than the actual MTU for a designated path.

3 Tuning daemon design and implementation

Our implementation of a daemon, the *WAD*, to tune individual TCP flows is based on a network measurement component, the *NTAF*, and an instrumented TCP stack, *Web100*. The daemon and its components are described in this section.

3.1 Web100

The Web100 project[35] 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). The Web100 vision is to enable users running ordinary applications on typical workstations to either saturate a workstation bottleneck or completely fill a network link. That is, to make it easy for ordinary users to tune TCP to get the most out of their available resources.

At its inception Web100 was narrowly focused on the TCP buffer tuning problem. However, early on it became apparent that the real problem is actually the flip side of one of the Internet's greatest strengths: TCP and IP work together to decouple details of the application from details of the network. This layering independence is extremely important because it permits old applications to run on new networks and new applications to run on old networks. The down side of layering independence is that TCP/IP also hides all bugs. The only symptom displayed to the user for nearly all problems in the lower layers is less than expected performance.

Web100 helps this situation by providing a mechanism to expose detailed TCP performance statistics through an enhanced standard "Management Information Base" (MIB) for TCP [34]. 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. Our tuning daemon uses the Web100 TCP MIB to collect the necessary information from active flows to perform its "work-arounds."

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

[1] Even though Ethernet speeds have increased by three orders of magnitude over the years, the MTU has remained at 1500 bytes. In fact, 1500 bytes today scaled back by Moore's law is effectively smaller than 53 bytes was 7 years ago. Furthermore, today a packet takes less wire time than 1 byte took 20 years ago (on 3 Mb/s Ethernet).

connection start and end events are provided to an application (e.g., a tuning daemon) through the *netlink* service. In order to dynamically tune buffer sizes during a flow, the window scaling option [29] in the initial TCP packet must be large enough to accommodate the largest tunable buffer size. The Web100 extensions include a *sysctl* variable for setting the default window scale.

3.2 Network Tool Analysis Framework (NTAF)

To provide tuning data for specific network paths, we have developed a framework for running network test tools and storing the results in a database, which we call the Network Tool Analysis Framework (NTAF). The NTAF, a descendant of ENABLE [47], manages and runs a set of network tools and sends the results to a database for later retrieval. Recent results are cached and can be queried via a client API. Data collected by the network tools includes Web100 variables.

The goal of the NTAF is to make it easy to collect, query, and compare results from any set of network or host monitoring tools running at multiple sites in a WAN. This is similar to the goal of NIMI [41], but where NIMI is focused on scalability and security concerns, we are focused on flexibility and ease-of-use and only envision deployment within 10-100 sites. The basic function performed by the NTAF is to run tools at regular intervals and send their results to a central archive system for later analysis or for use by our tuning daemon. Some tools cannot run on the same host without perturbing each other, and the NTAF is designed to accommodate this by never scheduling these tools at the same time.

We have deployed NTAF servers and tuning daemons on hosts at Oak Ridge National Lab (ORNL), Lawrence Berkeley National Lab (LBL), National Energy Research Scientific Computing center (NERSC), Pittsburgh Supercomputing Center (PSC), and National Center for Atmospheric Research (NCAR). We are currently using NTAF to run the following tests: *ping*, *pipechar*, *iperf* (instrumented to collect Web100 information), *GridFTP*, *netest*, and a CPU and memory sensor based on *procinfo*. In our current implementation of the tuning daemon, the NTAF provides TCP socket buffer sizes for a path based on *pipechar* data (round trip time and bottleneck bandwidth).

All tools run at a specified regular interval, plus or minus a randomization factor. For example, *ping* runs for 10 seconds every 10 minutes, plus or minus 1 minute. Since *iperf* is more intrusive, it runs for 10 seconds every 2 hours, plus or minus 10 minutes. Because some tools like *pipechar* can not run in parallel, NTAF never schedules two of them to run at the same time.

The results for all tests are converted into NetLogger events [48]. NetLogger provides us with an efficient and reliable data transport mechanism to send the results to an event archive. For example, if the network connection to the archive goes down, NetLogger transparently will buffer the events on local disk instead, and keep retrying to connect to the archive. When the archive becomes available, the events buffered on disk will be sent automatically. More details are available in [24].

All NTAF-generated NetLogger events contain the following information: timestamp, program name, event name, source host, destination host, and value. For extensibility, arbitrary additional string or numeric data can be included with each event. Using a standard event format with common attributes for all monitoring events allows us to quickly and easily build SQL tables of the results. More details are in [31].

3.2.1 NTAF Design

There are a number of factors that make it difficult to design a reliable distributed monitoring system. These issues are described at length in papers by the NIMI and PingER [15] projects. We have tried to learn from these projects, and incorporate the following principles into the design of the NTAF.

- **Output formats:** Every tool has a different output format, which can be difficult to parse. Then a new version of the tools is released, which has a new output format, and the parsing code breaks. Whenever possible we avoid this by modifying the code to generate NetLogger events directly. That way the author of the tools can change the output format without affecting the generation of NetLogger events. In cases where the source code is not available, we write a simple python wrapper around the tool to generate NetLogger events. This way the NTAF server never needs to understand how to parse anything but NetLogger formatted data.

- **Test scheduling:** A very flexible scheduling mechanism is required. Some tests are very intrusive, and should not be run often. Other tests, like *pipechar*, have the constraint that only one test can run on a single host at one time. To avoid possible test synchronization of many tests running between a group of hosts at one time, we add a randomization factor to the scheduling. For example, run a test every 90 minutes plus or minus a random time between zero and five minutes.

- **Fault tolerance:** Many test tools are experimental and may crash or hang. The NTAF is designed to insure that a bad test does not take down the NTAF server. Each test is run in its own thread, with a time-out to ensure the test eventually ends. All socket I/O is non-blocking to ensure that nothing ever blocks waiting for a message that may never arrive.
- **Error handling:** Errors need to be sent to a convenient location so that some intelligent post-processing can be performed, and the right level of ”alert” can be forwarded to the responsible people. Usually this convenient location is a central server, but the NTAF is flexible in this regard. By default, errors get sent back to the same archive that is holding test results. But the NTAF can also send errors to a network server, into a file, or even to syslog. The NTAF uses the same logging format for errors and test results so parsing tools are already available for processing the errors.
- **Automatic restart:** Certain tools, such as *iperf*, require a remote server. These servers will sometimes crash or hang. It is important to have a mechanism to monitor and restart these servers. We have developed a set of *cron* scripts for this purpose.
- **Update mechanism:** Management of updates quickly becomes very tedious in a monitoring environment such as this. There are a large number of components and configuration files that need to be updated or modified on a regular basis. For this task we are using *PacMan* [51], a tool that automatically checks for updates on an http server, and automatically installs the updated packages as they become available.

3.3 Work Around Daemon (WAD)

The stock Web100 kernel modifications have been extended as part of this research to permit tuning more than just TCP's send and receive buffers. At present, our kernel extensions also allow us to tune TCP's slow-start, providing a threshold to switch to Floyd's limited slow-start [23]. We can tune TCP's additive increase and multiplicative decrease (AIMD) which are used during loss recovery and congestion avoidance. Tuning the additive increase allow us to create a virtual MSS. We can also regulate packet bursts and reordering threshold and disable delayed ACKs. The Linux 2.4 kernel has several built in TCP tuning options, some of which get in the way of WAD-based tuning, so we have added a Web100 mechanism to disable those features.

We have developed a tuning daemon, or work-around daemon (WAD), to apply these new tuning options to designated TCP flows. The WAD listens on the Web100 *netlink socket* (a communication mechanism from the kernel back to user space used for notifications from the kernel) for new TCP connection events. The daemon has a configuration file that specifies what flows (source, source port, destination, destination port) are eligible for tuning. The configuration entry for a tunable flow also includes some static tuning parameters (mode, max ssthresh, AIMD parameters, reordering threshold, etc). An entry for path "bob" in the configuration file looks like the following

```
[bob]
src_addr: 0.0.0.0
src_port: 0
dst_addr: 10.5.128.74
dst_port: 0
mode: 1
sndbuf: 3751239
rcvbuf: 6571899
wadai: 6
wadmd: .3
maxssth: 100
divide: 1
reorder: 9
delack: 0
```

When a new connection event from the kernel awakens the daemon, the daemon checks the configuration file to see if it is a tunable flow. For tunable flows, the configuration file indicates if the static values in the table are to be used or if dynamic tuning from the NTAF is requested.[2] For dynamically tuned flows, a thread is created to monitor and tune the flow for its duration. At present dynamic tuning is provided for the send and receive buffer sizes (based on the bandwidth-delay product for the path provided by the NTAF) and AIMD values. The tuned send buffer size is used to dynamically tune the AIMD values using Floyd's table (Appendix B in [22]) at the start of a flow. The configuration file also indicates whether to override application *setsockopt()* settings for buffer sizes and whether to divide the tuned buffer size among concurrent flows. The tuning requires Web100 kernel modifications, but requires no changes to existing network applications. At flow termination, the daemon logs a summary of its tuning activities.

Besides having the ability to tune TCP connections for applications, the WAD has a number of other useful capabilities. Using the WAD, one can monitor any Web100 variable for any flow. For example, the WAD can measure the average bandwidth, packet retransmissions, timeouts, and round-trip times of any application by watching the Web100 variables. The WAD can also be configured to generate *derived events* from other Web100 variables. For example, one could generate average bandwidth from other Web100 variables as follows:

```
AveBW = (DataBytesOut*8)/(CurrTime - StartTime)
```

The WAD can then generate NetLogger events from these derived values and send them to the NetLogger analysis tool, *nlv*, for real-time analysis of active TCP streams.

We also have developed a tracer daemon that polls the kernel (e.g., every 0.1 seconds) for Web100 variables on configured flows. This data is saved to disk and has proven very useful for diagnostics and tuning. We have used data from the tracer daemon in many of the data plots in this paper.

4 Experimental Results

We have evaluated various TCP optimizations using both *ns* simulation [11] and a TCP-over-UDP test harness [17] as well as the Web100-modified Linux kernel. We have tested various tuning options using a NISTNet [38] test bed as well as the Internet. Even though evaluating different TCP tuning options is problematic over real networks, our first-year goal was to show real performance improvements in TCP bulk transfers over real high delay, high bandwidth networks. So the experiments described below were performed across the US using hosts with GigE interfaces linked to DOE's ESnet (OC12) or Internet2 (OC48).

The results given in this section are not meant to be conclusive in regards to proposed TCP modifications. These results show the utility of Web100 and the WAD for trying out various techniques over various paths. Much more testing and analysis are needed to determine which of these methods should be standardized.

4.1 WAD Buffer Tuning

A number of systems have emerged to address the issue of automatically tuning TCP buffers [43, 20]. Linux 2.4 automatically does sender side tuning, but this only works if the receive buffers have been set to a large value. Dynamic Right Sizing [20] automatically does receiver size buffer tuning, but assumes large send buffers (or a Linux 2.4 autotuning sender).

WAD tuning daemons and NTAF clients are running on our test hosts around the country. When a TCP connection is requested for a tunable path, the Web100 notification mechanism will send a "new connection" event to the WAD when the connection is established. The WAD at each endpoint can then set the optimal buffer size based on static information in the configuration file for the path or using a buffer size provided by the NTAF based on recent network measurements.

Figure 2 shows the throughput for a 10 second data transfer between ORNL and PSC (70 ms round trip time, GigE interfaces) over the ESnet and Internet2 network. The figure shows untuned throughput and WAD-tuned throughput. The network-unaware application runs at less than 10 Mbits/sec, but the same application runs at over 135 Mbits/sec when dynamically tuned by the WAD – more than an order of magnitude improvement in throughput. The tuning daemon at the source and destination uses dynamic data from the NTAF database to increase the TCP buffers to nearly 2 MB as the flow starts. Throughput data was collected from the kernel with a tracer daemon that polls the TCP MIB in the Web100 kernel rather than *tcpdump/tcptrace* collection and analysis.

[2] The daemon periodically queries the NTAF for current path characteristics and saves the information for tuning flows.

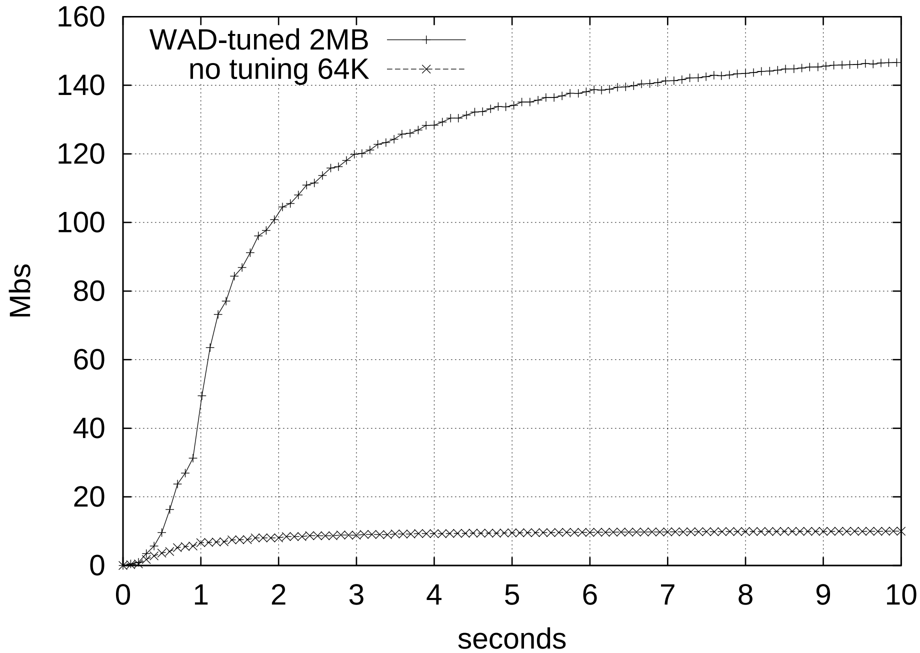

Figure 2: WAD-tuned versus untuned data transfer between ORNL and PSC.

4.2 WAD Parallel Stream Tuning

As noted earlier, a number of applications have been modified to use parallel TCP streams to work around buffer limitations and improve throughput. 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 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. Figure 3 plots the congestion window over time for four parallel TCP streams transferring data from ORNL to LBNL (GigE/OC12, 80 ms RTT) using 2 MB buffers for each stream. The aggregate congestion window is also plotted and illustrates the various parallel speedups. We chose four streams for purposes of illustration. The optimum number of parallel streams and their buffers sizes is an open research question.

All four streams experience loss during initial slow-start, but the aggregate peak was four times larger than the individual flows. The loss in all four streams results in the aggregate responding like standard TCP with the aggregate congestion window halved. At 25 seconds, one of the streams experiences a loss, but the aggregate rate is cut by only $1/8$ and recovery rate is four times faster than for one standard stream.

We considered using the WAD to dynamically reduce the number of parallel streams during heavy congestion by tuning the buffer size to zero for one or more of the parallel streams. However, most of the parallel applications statically allocate the workload to all k streams, so this was not a workable tuning option.

The WAD configuration can indicate that the WAD-tuned buffer size be given to every flow on the path or that the tuned buffer size be subdivided among concurrent flows. With subdivision enabled, the daemon initially sets the buffer size to the tuned value. If a second flow then starts up, then the first flow's buffer size is cut in half, and the second flow's buffer is set to half the original tuned buffer size. So when four flows are in progress, they each have $1/4$ the tuned buffer size. Experiments with each of these tuning methods were inconclusive, sometimes throughput was better for the subdivided flows, sometimes not. However, congestion, as measured by congestion signals and number of retransmitted packets, was almost always reduced when the buffer size was divided among the concurrent flows.

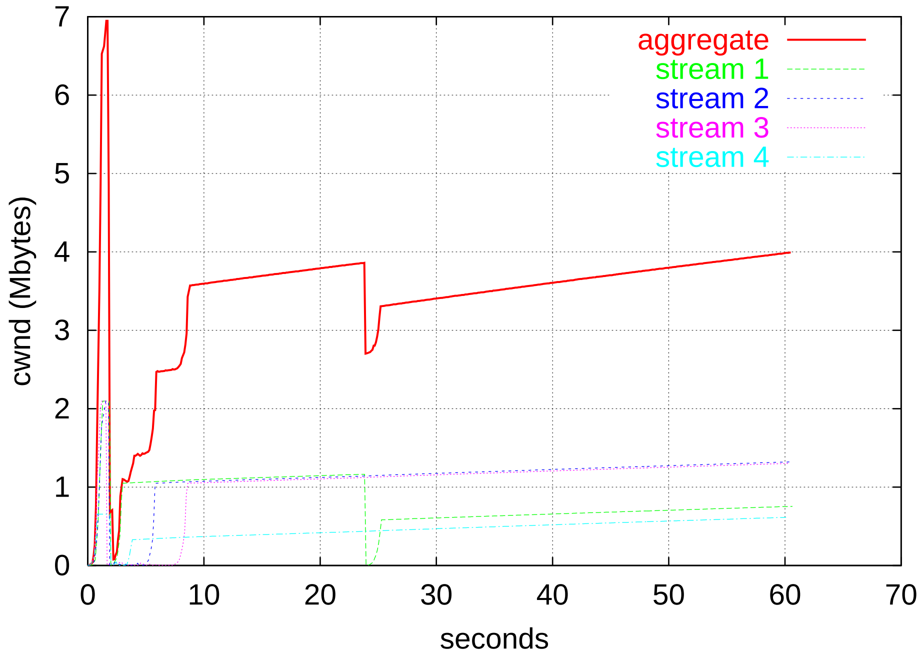

Figure 3: Congestion window for four parallel streams and the aggregate.

4.3 Virtual MSS and AIMD

The behavior of the aggregate congestion window of k parallel streams and the arguments for a larger MTU inspired us to use the WAD to create a "virtual MSS" for designated flows. The virtual MSS is implemented by adding k segments to *cwnd* each RTT during congestion avoidance. The virtual MSS does not cause IP fragmentation [27] or reduce the interrupt overhead.

The effect of the virtual MSS is best illustrated when there is packet loss. Figure 4 illustrates two transfers from ORNL to NERSC with packet loss during slow-start. Both flows use the same TCP buffer sizes, but one flow is dynamically tuned by the WAD to use a virtual MSS of 6 segments (e.g., a virtual jumbo frame). The *cwnd* data was collected from the kernel with a Web100 tracer daemon. The other component of a virtual MSS would be to initiate slow-start with k segments. We have not implemented this with the WAD yet because the WAD does not get notified soon enough to modify the initial window conditions. (Eventually, we may have the WAD store such information in the routing cache and retrieve the data on socket creation.) Note that while the use of a virtual MSS solves some of the AIMD issues, it has no impact on host interrupt issues, which are currently a big obstacle to obtaining very high-speed TCP in a host.

The virtual MSS modifies only the TCP additive increase, the multiplicative decrease can also be configured for a path and tuned by the WAD. Rather than decreasing the window by half on a congestion event, [3] we could designate that for a particular path it only be decreased by 25%. Figure 5 illustrates a standard TCP AIMD (0.5,1) flow and a WAD-tuned AIMD (0.3,6) flow when packet loss is encountered during slow-start. The tuned AIMD starts its linear recovery from a higher point and with a steeper slope resulting in a 40% improvement in throughput for this 10 second TCP transfer from NERSC to ORNL (GigE/OC12, 80ms RTT).

We could try to get a single stream to perform as well as 4 parallel streams by setting the additive increase to 4 and the multiplicative decrease to $1/8$ (also see [16]). (We would also need to set the initial window to 4 to speed slow-start, but as noted above, that is difficult for the WAD at this time.) As an example, Table 1 shows the throughput results of GridFTP transfers of a 200 MByte file from ORNL to LBNL (GigE/OC12, 80 ms RTT). A WAD AIMD-tuned (0.125, 4) single stream transfer using 4 MB TCP buffers gets similar throughput to four streams using 1 MB buffers. Also results for an untuned single stream is included. The tuned stream is 3.5 times faster than the untuned stream and has more congestion events but fewer retransmitted packets. The four parallel streams are about 25% faster

[3] A congestion event is one or more packet losses within a single round-trip time.

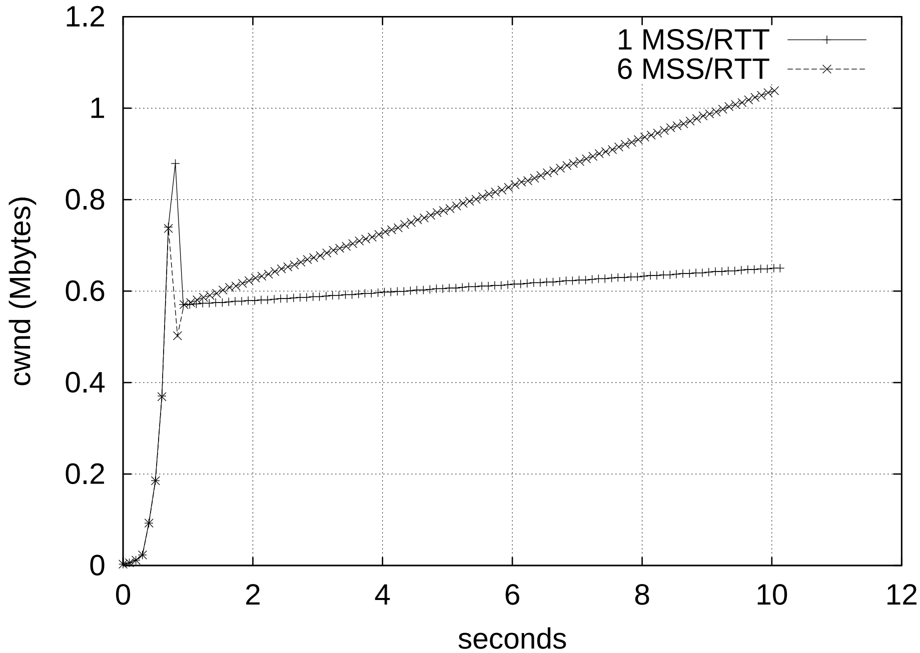

Figure 4: Standard congestion window recovery versus 6x virtual MSS.

tuning	streams	Mbs	cong. events	rexmits
untuned	1	22.4	5	102
tuned	1	78.4	7	64
untuned	4	98.4	15	320

Table 1: GridFTP data rates from ORNL to LBNL.

than the tuned single stream, but have 15 congestion events and 320 retransmitted packets. Even though the parallel and tuned streams have more congestion events than the untuned stream, they perform faster because they recover from loss quicker as a result of their more aggressive AIMD values. In the majority of our experiments, we have not been able to get a single tuned stream to outperform parallel streams, though the tuned stream typically contributes less to congestion. The WAD can, of course, tune each of the parallel streams as well.

One can choose the AIMD parameters statically based on path characteristics, or parallel-stream equivalence, or desired level of service. Floyd proposed a modified AIMD for high-delay, high-bandwidth flows that dynamically adjusts the AIMD parameters as a function of loss probability and current congestion window size [22]. As the congestion window grows and shrinks, so do the AIMD values. We have experimented with Floyd's modified AIMD with *ns* simulation and with our TCP-over-UDP test harness, but have not yet added it to the Linux kernel so that the WAD can select it. However, the WAD does use Floyd's table (Appendix B [22]) to select AIMD values based on the configured buffer size for the path when a connection starts. So if the NTAF recommended a 2 MB buffer for a path, the corresponding AIMD values would be (0.31,11). This is a more aggressive modification than Floyd's, since Floyd's algorithm varies the AIMD values as the congestion window changes. In the future, the WAD will dynamically monitor the congestion window of a flow and reset the AIMD values for the flow according to Floyd's formula and thus avoid having to add Floyd's AIMD code to the kernel.

AIMD tuning usually can more than triple performance over a single, untuned flow and is competitive with parallel streams. The tuned flows can add to congestion but are still "TCP friendly" since they still reduce their transmission rate in response to packet loss. Under heavy congestion, our experience has been that the tuned AIMD and/or virtual MSS flows recover faster but encounter losses more frequently, so the TCP saw-tooth is shallower and steeper but at a higher frequency. In these heavy congestion scenarios, the tuned AIMD flows are not significantly faster than standard TCP over the paths we have tested. Higher bandwidth paths and devices might show more gain from the virtual MSS

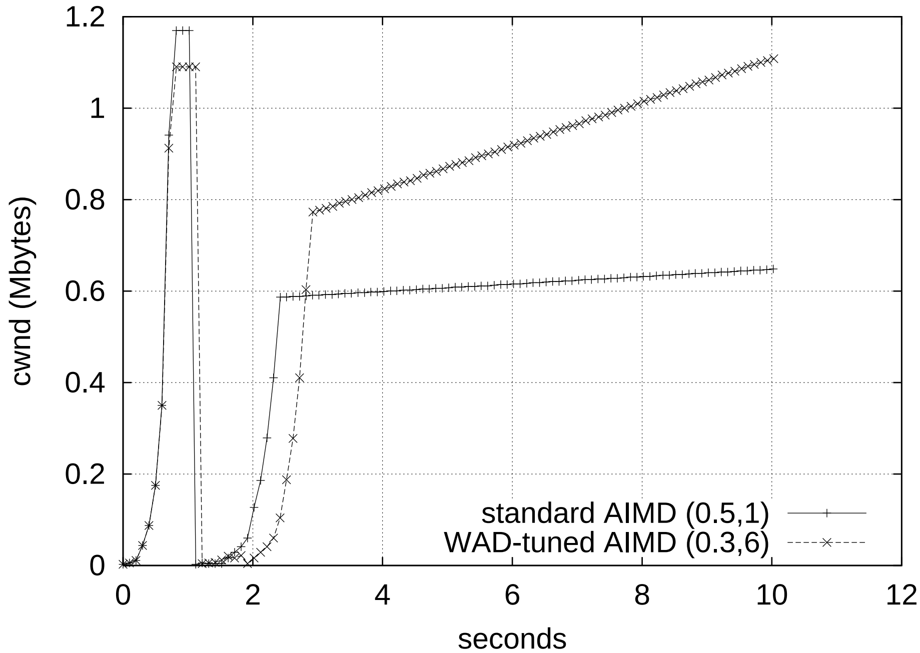

Figure 5: Congestion window over time for standard TCP AIMD (0.5,1) and WAD-tuned AIMD (0.3,6).

and/or tuned AIMD values.

4.4 Delayed ACKs

Most TCP implementations utilize delayed ACKs [9]. Rather than acknowledging every arriving packet, every other packet is acknowledged, or the delayed ACK is sent after 200 ms. Delayed ACKs can slow both slow-start and recovery from a loss by a factor of two, since slow-start opens the congestion window by one segment for each arriving ACK, and during congestion avoidance the congestion window grows with each arriving ACK [3]. The WAD can disable delayed ACKs for designated flows. The speedup of slow-start has a modest effect in our cross-country experiments (10% improvement), though the improvement is a function of the bandwidth-delay product. If there is packet loss, acknowledging every packet doubles the linear recovery rate and can improve throughput typically by 25%.

4.5 Avoiding loss

A virtual MSS and AIMD adjustments can help TCP recover from losses faster. We have also used the WAD to tune flows to reduce packet losses by adjusting TCP's slow-start and TCP's reordering threshold.

TCP starts a flow initially or after a timeout by doubling the congestion window for every ACK received. We have observed that packet losses often occur during this slow-start phase over our high capacity (GigE/OC12) nets. Losses during initial slow-start can have a disastrous effect on throughput over high delay, capacity nets. Recovery from such losses with standard TCP AIMD could take several minutes. Slow-start can overshoot the available bandwidth by a factor of two, so choosing proper buffer size may reduce slow-start losses. However, even with good buffer sizes, slow-start losses are still common. We have used simulation, our TCP-over-UDP test harness, and the WAD to experiment with various modifications to slow-start. First, we tried setting TCP slow-start threshold to a low value, this usually avoided the initial start-up losses, but throughput was not much improved, even with aggressive WAD-tuned AIMD values. Next we added Floyd's limited slow-start modifications [23] to the Linux/Web100 kernel and added the *max_ssthresh* variable to the WAD configuration for a path. During initial slow-start when the congestion window reaches *max_ssthresh*, TCP switches to Floyd's limited slow-start which adds at most $max\_ssthresh/2$ segments per round trip time. This moderates slow-start and in some cases reduces packet losses in slow-start. As an example, a sequence of tests from PSC to LBL (GigE/OC48, 75 ms RTT) were consistently getting slow-start losses about 0.7

seconds into the transfer and the resulting bandwidth was 98 Mbs for a 10 second test with standard TCP. Using Floyd's limited slow-start the losses were eliminated, and the 10 second transfers reached 192 Mbs ($max\_ssthresh = 100$) and 204 Mbs ($max\_ssthresh = 200$) – a factor of two improvement. However during high congestion, the modified slow-start merely delayed the congestion event, and performance was not much better than standard TCP. Figure 6 illustrates three data transfers from ORNL to CERN in Switzerland (150 ms RTT) with 2 MB buffers. One test used

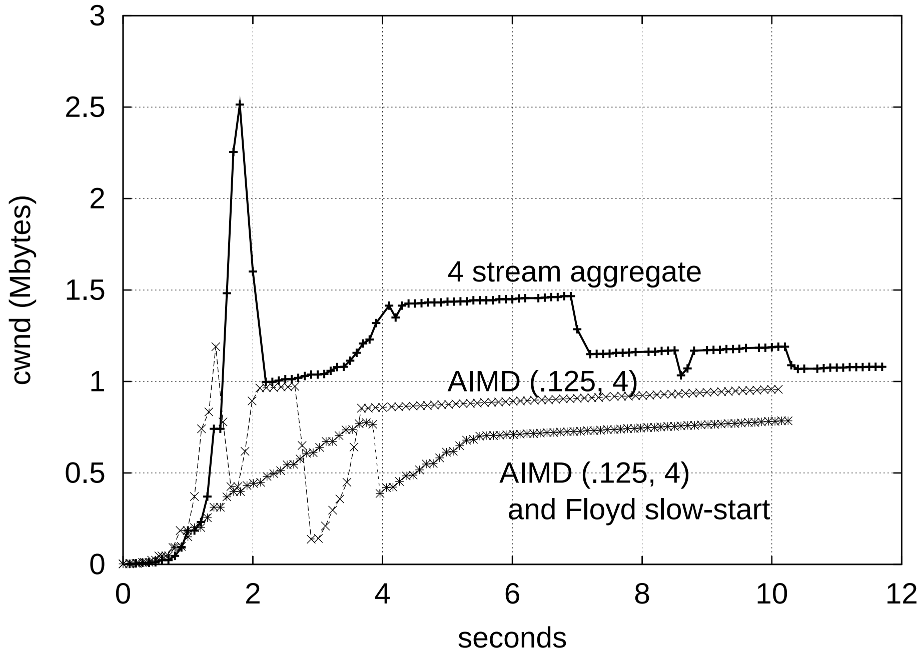

Figure 6: Congestion window for three tests from ORNL to CERN.

four parallel streams and reached an aggregate rate of 58 Mbs with 8 congestion events and 155 retransmitted packets. The other two tests were WAD-tuned single streams with AIMD of (0.125, 4) to try and match the effective AIMD of the parallel streams. One of the single streams used Floyd's limited slow-start ($max\_ssthresh$ of 100 segments), and its more gradual slow-start is evident in the graph. For these tests, the limited slow-start attained 29 Mbs (1 congestion event, 13 segments retransmitted) compared to 38 Mbs (2 congestion events, 41 packets retransmitted) for the other WAD-tuned stream. We continue to experiment with slow-start tuning.

Some of the paths over which we have been collecting NTAF data show significant reordering of packets. Standard TCP will treat out of order packets as a packet loss if the packets are more than three positions out of order. We have added a WAD variable so that the user (or the NTAF) can set the re-ordering threshold for a path. Figure 7 shows the instantaneous and average bandwidth for two tests from LBNL to ORNL (GigE/OC12, 80ms RTT, 2 MB buffers). One test uses the default threshold of 3 and reaches only 18 Mbs after 10 seconds. The flow experienced 9 congestion events and retransmitted 289 packets, but the receiver reported 289 duplicate packets received. So none of the packets were actually lost, they merely arrived out of order. We reran the test using a reordering threshold of 19 and reached 282 Mbs after 10 seconds with no packet loss. This tuning resulted in an order of magnitude improvement in throughput.

5 Related work

NLANR has an auto-tuning FTP [39] that uses a train of ICMP or UDP packets to estimate the available bandwidth and round-trip time between the client and server. Using the bandwidth-delay product, buffer sizes are then set at the client and server. Tierney et al. [47] provide an API and network daemons that allow one to modify a network application to query a daemon for the optimal buffer size to be used for a particular destination. For example, an FTP server and client might issue a call like

```
tcp_buffer_size = EnableGetBufferSize(desthost);
```

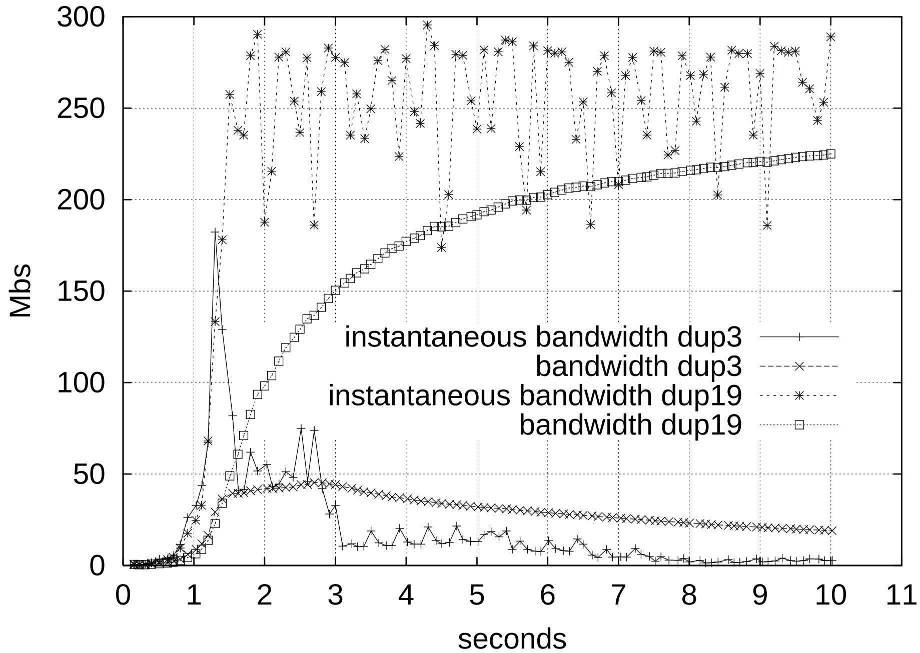

Figure 7: Throughput with standard and tuned reordering threshold.

and then use that value to set the TCP buffer sizes. The Enable daemons periodically measure the bandwidth and latency of designated network paths and provide that information to client as an RPC-like service. It is also possible to generate scripts or command files that would use tuning data from the Enable daemon to tune network applications that accept tuning arguments.

Parallel TCP streams [25] offer another way to get around TCP-size window limits. There are several parallel file transfers protocols including bbftp [19] GridFTP [1] and *psockets* [44] and *iperf* [40] is a path-testing tool that has parallel stream options.

In 1998, Semke, Mahdavi and Mathis[43] describe modifications to a NetBSD kernel that allows the kernel to automatically resize the sender's buffers. (The receiver needs to advertise a "big enough" window.) The kernel modifications allow a host that is serving many clients to more fairly share the available kernel buffer memory and to provide better throughput than manually configured TCP buffers. For applications that have not explicitly set the TCP send buffer size, the kernel will allow the send buffer to grow with the flow's congestion window (*cwnd*) and utilize the available bandwidth of the link (up to the receiver's advertised window). If the network memory load on the server increases, the kernel will also reduce the senders' windows. So no modification are required to sending applications.

Beginning in 2001, Feng and Fisk [20] describe modifications to a Linux 2.4.8 kernel that allow the kernel to tune the buffer size advertised by the TCP receiver. The receiver's kernel estimates the bandwidth from the amount of data received in each round-trip time and uses that bandwidth estimate to derive the receiver's window. The sender's window is not constrained by the system default window size but is allowed to grow, throttled only by the receiver's advertised window. (The Linux 2.4 kernel allows the sender's window buffer to grow. For other OS's, the sender would have to be configured with a "big enough" buffer.) The growth of the sender's congestion window will be limited by currently available bandwidth. High delay, high bandwidth flows will automatically use larger buffers (within the limits of the initial window scale factor advertised by the receiver). No modifications are required to either client or server network applications. Also see Allman/Paxson [5] on receiver-side bandwidth estimation.

In 2001, the Linux 2.4 kernel included TCP buffer tuning algorithms. For applications that have not explicitly set the TCP send and receive buffer sizes, the kernel will attempt to grow the window sizes to match the available bandwidth (up to the receiver's advertised window). Like the Semke work described above, if there is high demand for kernel/network memory, buffer size may be limited or even shrink. Autotuning is controlled by new kernel variables and is disabled if the application does its own setsockopt() for TCP's SND/RCVBUF. The Linux 2.4 advertised receive window starts small and grows with each segment from the transmitter, even if a setsockopt() has been done for the receiver window.

Independent of auto-tuning, the Linux 2.4 kernel has a ”retentive TCP”, caching (for 10 minutes) TCP control information for a destination (cwnd, rtt, rttvar, ssthresh, and reorder). The cached cwnd value is used to clamp the maximum cwnd value for the next transfer to that destination. The cached ssthresh will cutoff the slow-start for the next transfer. For high delay/bandwidth paths, our experience has been that this feature tends to reduce the throughput of subsequent transfers, if an early transfer experiences loss. One of the kernel modifications made for Web100 was to provide a way to disable this caching. Linux 2.4 also does ”speculative” congestion avoidance. If a flow has entered congestion avoidance because of duplicate ACKs or timeout, and the missing packet arrives ”soon”, then *cwnd/ssthresh* will be restored (congestion ”undo”). The reordering threshold will be increased for this path as well. Finally, the Linux 2.4 kernel ACKs every packet during initial slow start.

6 Summary and Future Work

The combination of the Web100 TCP MIB, the Network Tool Analysis Framework (NTAF), and the Work Around Daemon (WAD) have proven effective in transparently tuning TCP flows, often giving an order of magnitude improvement for bulk transfers over standard TCP on high delay, high bandwidth networks. The Web100-based tools and kernel extensions have been effective for evaluating new TCP tuning options on real networks, permitting us to select different tuning options based on path characteristics or desired level of service. Our testing has included emulation and simulation, but more testing is needed to determine which TCP tuning options should be standardized. The NTAF data and Web100 data have also been useful for diagnosing network and application bottlenecks.

Future work will look at how to decide when to use parallel streams, the number of streams to use, and the size of the buffers for each stream. We also will see if our framework can be used to distinguish between random loss and congestive loss. We also plan to investigate additional algorithms for tuning the congestion control parameters (AIMD) for a flow [7, 13, 50] and to add Floyd's AIMD modifications [22] as a tunable option. We will continue to try and avoid packet loss through slow-start modifications and Vegas-like controls [10] and various burst-reduction and rate-smoothing techniques. We will look at WAD-to-WAD communication between sender and receiver for out-of-band tuning of flows. We will also be investigating tuning latency-sensitive distributed applications. We hope to port the Web100 extensions to other operating systems.

Parallel streams and AIMD tuning can be unfair to other network users and are not appropriate in some Internet contexts. We are sensitive to issues of fairness and possible congestion collapse [21], but on an intranet, such as DOE's ESnet, it may be more economical to be unfair (in a strict TCP-friendly sense) than to fix the net. This is an appropriate policy only when supported by explicit coordination with the network owners. A natural extension to this work would be to add policy configuration, controls, or even management protocols to the WAD, such that users can be granted privileged access to network capacity.

References

[1] W. Allcock, J. Bester, J. Bresnahan, A. Chervenak, I. Foster, C. Kesselman, S. Meder, V. Nefedova, D. Quesnel, and S. Tuecke. “gridFTP”, 2000. URL: http://www.globus.org/datagrid/gridftp.html.

[2] W. Allcock, J. Bester, J. Bresnahan, A. Chervenak, I. Foster, C. Kesselman, S. Meder, V. Nefedova, D. Quesnel, and S. Tuecke. Secure, Efficient Data Transport and Replica Management for High-Performance Data-Intensive Computing, 2000. URL: http://www.globus.org.

[3] M. Allman. On the generation and use of tcp acknowledgments. *ACM Comp Commun. Rev.*, 28:4–21, October 1998.

[4] M. Allman, V. Paxson, and W. Stevens. TCP Congestion Control. *RFC 2581*, April 1999.

[5] Mark Allman and Vern Paxson. On estimating end-to-end network path properties. In *SIGCOMM*, pages 263–274, 1999.

[6] P. Avery and I. Foster. The GriPhyN Project: Towards Petascale Virtual Data Grids, 2001. Technical Report GriPhyN-2001-15 URL: http://www.griphyn.org/.

[7] D. Bansal and H. Balakrishnan. Binomial congestion control algorithms, 2001.

[8] W. Bethel, B. Tierney, J. Lee, D. Gunter, and S. Lau. Using high-speed wans and network data caches to enable remote and distributed visualization. In *Proceeding of the IEEE Supercomputing 2000 Conference*, 2000.

[9] R. Braden. Requirements for Internet Hosts – Communication Layers. *RFC 1122*, October 1928.

[10] Lawrence S. Brakmo, Sean W. O'Malley, and Larry L. Peterson. TCP vegas: New techniques for congestion detection and avoidance. In *SIGCOMM*, pages 24–35, 1994.

[11] L. Breslau, D. Estrin, K. Fall, S. Floyd, and J. Heideman. Advances in Network Simulation. *IEEE Computer*, pages 59–67, May 2000.

[12] 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 scientific datasets. In *Network Storage Symposium (NetStore '99)*, 1999.

[13] D. Chiu and R. Jain. Analysis of the increase and decrease algorithms for congestion avoidance in computer networks. *Journal of Computer Networks and ISDN Systems*, pages 1–14, June 1989.

[14] D. Clark, M. Lambert, and L. Zhang. NETBLT: A Bulk Data Transfer Protocol. *RFC 998*, March 1987.

[15] R. L. Cottrell and C. Logg. A new high performance network and application monitoring infrastructure, 2002.

[16] J. Crowcroft and P. Oechslin. Differentiated end-to-end internet services using a weighted proportional fair sharing tcp, 1998.

[17] T. Dunigan and F. Fowler. A TCP-over-UDP test harness. Technical report, Oak Ridge National Laboratory, Oak Ridge, TN, 2002. ORNL/TM-2002/76.

[18] EU. Eu datagrid project, 2001. URL: http://www.eu-datagrid.org/.

[19] G. Farrache. "bbftp", 2000. URL: http://doc.in2p3.fr/bbftp/.

[20] M. Fisk and W. Feng. Dynamic Right-Sizing in TCP. In *Los Alamos Computer Science Institute Symposium*, 2001.

[21] S. Floyd. Congestion Control Principles. *RFC 2914*, September 2000.

[22] S. Floyd. HighSpeed TCP for Large Congestion Windows. *IETF draft, work in progress*, May 2002. URL: http://www.icir.org/floyd/papers/draft-floyd-tcp-highspeed-00c.txt.

[23] S. Floyd. Limited Slow-Start for TCP with Large Congestion Windows. *IETF draft, work in progress*, May 2002. URL: http://www.icir.org/floyd/papers/draft-floyd-tcp-slowstart-00b.txt.

[24] D. Gunter, B. Tierney, K. Jackson, J. Lee, and M. Stoufer. Dynamic monitoring of high-performance distributed applications. In *Proceedings of the 11th IEEE Symposium on High Performance Distributed Computing*, 2002.

[25] T. Hacker, 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.

[26] A. Hanushevsky. "bbcp", 2000. URL: http://www.slac.stanford.edu/ abh/bbcp/.

[27] T. H. Henderson, E. Sahouria, S. McCanne, and R. H. Katz. On improving the fairness of TCP congestion avoidance. *IEEE Globecomm conference, Sydney*, 1998.

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

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

[30] K. Lahey. TCP problems with path MTU discovery. *RFC 2923*, September 2000.

[31] J. Lee, D. Gunter, M. Stoufer, and B. Tierney. Monitoring data archives for grid environments. In *Proceeding of IEEE Supercomputing 2002 Conference*, 2002.

[32] J. Mahdavi and S. Floyd. TCP-Friendly Unicast Rate-Based Flow Control, 1997. URL: http://www.psc.edu/networking/papers/tcp_friendly.html.

[33] M. Mathis. Pushing Up Performance for Everyone, 1999. URL: http://www.ncne.nlanr.net/news/workshop/1999/991205/Talks/mathis_991205_Pushing_Up_Performance/.

[34] M. Mathis, R. Reddy, J. Heffner, and J. Saperia. TCP Extended Statistics MIB. *IETF draft, work in progress*, February 2002. URL: http://www.ietf.org/internet-drafts/draft-ietf-tsvwg-tcp-mib-extension-00.txt.

[35] Matt Mathis. “Web100”, 2000. URL: http://www.web100.org.

[36] K. Miller, K. Roberston, A. Tweedly, and M. White. StarBurst Multicast File Transfer Protocol (MFTP) Specification, 1998. URL: http://www.kblabs.com/lab/lib/drafts/draft-miller-mftp-spec-03.txt.html.

[37] J. Mogul and S. Deering. Path MTU Discovery. *RFC 1191*, November 1990.

[38] NIST. “NISTNet”, 2001. URL: http://snad.ncsl.nist.gov/itg/nistnet/.

[39] NLAR. “Auto-tuning FTP”, 2000. URL: http://dast.nlanr.net/Features/Autobuf/.

[40] NLAR. “iperf”, 2000. URL: http://dast.nlanr.net/Projects/Iperf/.

[41] V. Paxson, A. Adams, and M. Mathis. Experiences with nimi, 2000.

[42] PPDG. Particle physics data grid, 2001. URL: http://www.ppdg.org/.

[43] Jeffrey Semke, Jamshid Mahdavi, and Matthew Mathis. Automatic TCP buffer tuning. In *SIGCOMM*, pages 315–323, 1998.

[44] Harimath Sivakumar, Stuart Bailey, and Robert L. Grossman. PSockets: The case for application-level network striping for data intensive applications using high speed wide area networks. In *Supercomputing SC2000*, 2000. URL: http://www.sc2000.org/techpapr/papers/pap.pap240.pdf.

[45] P. Steenkiste. Adaptation Models for Network-Aware Distributed Computations. In *3rd Workshop on Communication, Architecture, and Applications for Network-based Parallel Computing*, 1999.

[46] J. Stone and C. Partridge. When the CRC and TCP Checksum Disagree, 2000. URL: http://www.acm.org/sigcomm/sigcomm2000/conf/paper/sigcomm2000-9-1.pdf.

[47] B. Tierney, D. Gunter, J. Lee, and M. Stoufer. Enabling network-aware applications. In *10th IEEE Symposium on High Performance Distributed Computing*, 2001.

[48] B. Tierney, W. Johnston, B. Crowley, G. Hoo, C. Brooks, and D. Gunter. The netlogger methodology for high performance distributed systems performance analysis. In *Proceeding of IEEE High Performance Distributed Computing Conference*, 1998.

[49] B. Tierney, W. Johnston, and et. al. Distributed parallel data storage systems: A scalable approach to high speed image servers. In *Proceedings of the ACM Multimedia Symposium*, June 1994.

[50] Y. Yand and S. Lam. General AIMD Congestion Control. Technical report, University of Texas, 2000. TR-2000-09.

[51] Saul Youssef. “PacMAN”, 2002. URL: http://physics.bu.edu/ youssef/pacman/.