Automatic TCP Buffer Tuning

Jeffrey Semke and Jamshid Mahdavi and Matthew Mathis *

Pittsburgh Supercomputing Center

{semke,mahdavi,mathis}@psc.edu

Abstract

With the growth of high performance networking, a single host may have simultaneous connections that vary in bandwidth by as many as six orders of magnitude. We identify requirements for an automatically-tuning TCP to achieve maximum throughput across all connections simultaneously within the resource limits of the sender. Our auto-tuning TCP implementation makes use of several existing technologies and adds dynamically adjusting socket buffers to achieve maximum transfer rates on each connection without manual configuration.

Our implementation involved slight modifications to a BSD-based socket interface and TCP stack. With these modifications, we achieved drastic improvements in performance over large bandwidth*delay paths compared to the default system configuration, and significant reductions in memory usage compared to hand-tuned connections, allowing servers to support at least twice as many simultaneous connections.

1 Introduction

Paths in the Internet span more than 6 orders of magnitude in bandwidth. The congestion control algorithms [RFC2001, Jac88] and large window extensions [RFC1323] in TCP permit a host running a single TCP stack to support concurrent connections across the entire range of bandwidth. In principle, all application programs that use TCP should be able to enjoy the appropriate share of available bandwidth on any path without involving manual configuration by the application, user, or system administrator. While most would agree with such a simple statement, in many circumstances TCP connections require manual tuning to obtain respectable performance.

For a given path, TCP requires at least one bandwidth-delay product of buffer space at each end of the connection. Because bandwidth-delay products in the Internet can span 4 orders of magnitude, it is impossible to configure default TCP parameters on a host to be optimal for all possible paths through the network. It is possible for someone to be connected to an FTP server via a 9600bps modem while someone else is connected through a 100Mbps bottleneck to the same server. An experienced system administrator can tune a system for a particular type of connection [Mah96], but then performance suffers for connections that exceed the expected bandwidth-delay product, while system resources are wasted for low bandwidth*delay connections. Since there are often many more "small" connections than "large" ones, system-wide tuning can easily cause buffer memory to be inefficiently utilized by more than an order of magnitude.

Tuning knobs are also available to applications for configuring individual connection parameters [Wel96]. However, use of the knobs requires the knowledge of a networking expert, and often must be completed prior to establishing a connection. Such expertise is not normally available to end users, and it is wrong to require it of applications and users. Further, even an expert cannot predict changes in conditions of the network path over the lifetime of a connection.

Finally, static tuning configurations do not account for changes in the number of simultaneous connections. As more connections are added, more total memory is used until mbuf exhaustion occurs, which can ultimately cause the operating system to crash.

This paper proposes a system for adaptive tuning of buffer sizes based upon network conditions and system memory availability. It is intended to operate transparently without modifying existing applications. Since it does not change TCP's Congestion Avoidance characteristics, it does not change TCP's basic interactions with the Internet or other Internet applications. Test results from a running implementation are presented.

1.1 Prerequisites for Auto-Tuning of TCP

Before adding auto-tuning of buffers, a TCP should make use of the following features to improve throughput:

1. "TCP Extensions for High Performance" [RFC1323, Ste94] allows large windows of outstanding packets for long delay, high-bandwidth paths, by using a window scaling option and timestamps. All the operating systems used in tests in this paper support window scaling.

2. "TCP Selective Acknowledgment Options" [RFC2018] (SACK) is able to treat multiple packet losses as single congestion events. SACK allows windows to grow

*This work is supported by National Science Foundation Grant No. NCR-9415552.

larger than the ubiquitous Reno TCP, since Reno will timeout and reduce its congestion window to one packet if a congested router drops several packets due to a single congestion event. SACK is able to recognize burst losses by having the receiver piggyback lost packet information on acknowledgements. The authors have added the PSC SACK implementation to the hosts used in testing [SAC98].

3. "Path MTU Discovery" [RFC1191, Ste94] allows the largest possible packet size (Maximum Transmission Unit) to be sent between two hosts. Without pMTU discovery, hosts are often restricted to sending packets of around 576 bytes. Using small packets can lead to reduced performance [MSMO97]. A sender implements pMTU discovery by setting the "Don't Fragment" bit in packets, and reducing the packet size according to ICMP Messages received from intermediate routers. Since not all of the operating systems used in this paper support pMTU discovery, static routes specifying the path MTU were established on the test hosts.

1.2 Receive Socket Buffer Background

Before delving into implementation details for dynamically adjusting socket buffers, it may be useful to understand conventional tuning of socket buffers. Socket buffers are the hand-off area between TCP and an application, storing data that is to be sent or that has been received. Sender-side and receiver-side buffers behave in very different ways.

The receiver's socket buffer is used to reassemble the data in sequential order, queuing it for delivery to the application. On hosts that are not CPU-limited or application-limited, the buffer is largely empty except during data recovery, when it holds an entire window of data minus the dropped packets.

The amount of available space in the receive buffer determines the receiver's *advertised window* (or *receive window*), the maximum amount of data the receiver allows the sender to transmit beyond the highest-numbered acknowledgement issued by the receiver [RFC793, Ste94].

The intended purpose of the receive window is to implement end-to-end flow control, allowing an application to limit the amount of data sent to it. It was designed at a time when RAM was expensive, and the receiver needed a way to throttle the sender to limit the amount of memory required.

If the receiver's advertised window is smaller than *cwnd* on the sender, the connection is *receive window-limited* (controlled by the receiver), rather than *congestion window-limited* (controlled by the sender using feedback from the network).

A small receive window may be configured as an intentional limit by interactive applications, such as telnet, to prevent large buffering delays (ie. a ^C may not take effect until many pages of queued text have scrolled by). However most applications, including WWW and FTP, are better served by the high throughput that large buffers offer.

1.3 Send Socket Buffer Background

In contrast to the receive buffer, the sender's socket buffer holds data that the application has passed to TCP until the receiver has acknowledged receipt of the data. It is nearly always full for applications with much data to send, and is nearly always empty for interactive applications.

If the send buffer size is excessively large compared to the bandwidth-delay product of the path, bulk transfer applications still keep the buffer full, wasting kernel memory. As a result, the number of concurrent connections that can exist is limited. If the send buffer is too small, data is trickled into the network and low throughput results, because TCP must wait until acknowledgements are received before allowing old data in the buffer to be replaced by new, unsent data[1].

Applications have no information about network congestion or kernel memory availability to make informed calculations of optimal buffer sizes and should not be burdened by lower layers.

2 Implementation

Our implementation involved changes to the socket code and TCP code in the NetBSD 1.2 kernel [Net96]. The standard NetBSD 1.2 kernel supports RFC 1323 TCP extensions for high performance. Our kernel also included the PSC SACK port [SAC98].

Since NetBSD is based on 4.4 BSD Lite [MBKQ96, WS95], the code changes should be widely applicable to a large number of operating systems.

2.1 Large Receive Socket Buffer

During a transfer, the receiver has no simple way of determining the congestion window size, and therefore cannot be easily tuned dynamically. One idea for dynamic tuning of the receive socket buffer is to increase the buffer size when it is mostly empty, since the lack of data queued for delivery to the application indicates a low data rate that could be the result of a receive window-limited connection. The peak usage is reached during recovery (indicated by a lost packet), so the buffer size can be reduced if it is much larger than the space required during recovery. If the low data rate is not caused by a small receive window, but rather by a slow bottleneck link, the buffer size will still calibrate itself when it detects a packet loss. This idea was inspired by a discussion with Greg Minshall [Min97] and requires further research.

However, the complexity of a receive buffer tuning algorithm may be completely unnecessary for all practical purposes. If a network expert configured the receive buffer size for an application desiring high throughput, they would set it to be two times larger than the congestion window for the connection. The buffer would typically be empty, and would, during recovery, hold one congestion window's worth of data plus a limited amount of new data sent to maintain the Self-clock.

The same effect can be obtained simply by configuring the receive buffer size to be the operating system's maximum socket buffer size, which our auto-tuning TCP implementation does by default[2]. In addition, if an application manu-

[1] One rule of thumb in hand-tuning send buffer sizes is to choose a buffer size that is twice the bandwidth-delay product for the path of that connection. The doubling of the bandwidth-delay product provides SACK-based TCPs with one window's worth of data for the round trip in which the loss was suffered, and another window's worth of unsent data to be sent during recovery to keep the Self-clock of acknowledgements flowing [MM96].

[2] Using large windows to obtain high performance is only possible if the Congestion Avoidance algorithm of the TCP is well behaved. SACK-based TCPs are well behaved because they treat burst losses as single congestion events, so they are not penalized with a timeout when the bottleneck queue overflows [FF96, MM96]. Poor performance of congestion window-limited connections was observed by

ally sets the receive buffer or send buffer size with setsockopt(), auto-tuning is turned off for that connection, allowing low-latency applications like telnet to prevent large queues from forming in the receive buffer. Immediately before connection establishment, auto-tuned connections choose the smallest window scale that is large enough to support the maximum receive socket buffer size, since the window scale can not be changed after the connection has been established [RFC1323].

It is important to note that in BSD-based systems, the buffer size is only a *limit* on the amount of space that can be allocated for that connection, not a preallocated block of space.

2.2 Adjusting the Send Socket Buffer

The send socket buffer is determined by three algorithms. The first determines a target buffer size based on network conditions. The second attempts to balance memory usage, while the third asserts a hard limit to prevent excessive memory usage.

Our implementation makes use of some existing kernel variables and adds some new ones. Information on the variables used in our implementation appears below.

NMBCLUSTERS An existing kernel constant (with global scope) that specifies the maximum number of mbuf clusters in the system.

AUTO_SND_THRESH A new kernel constant (with global scope) that limits the fraction of NMBCLUSTERS that may be dedicated to send socket buffers. This is NMBCLUSTERS/2 in our implementation.

cwnd An existing TCP variable (for a single connection) that estimates the available bandwidth-delay product to determine the appropriate amount of data to keep in flight.

sb_net_target A new TCP variable (for a single connection) that suggests a send socket buffer size by considering only *cwnd*.

hiwat_fair_share A new kernel variable (with global scope) that specifies the fair share of memory that an individual connection can use for its send socket buffer.

sb_mem_target A new TCP variable (for a single connection) that suggests a send socket buffer size by taking the minimum of *sb_net_target* and *hiwat_fair_share*.

2.2.1 Network-based target

The sender uses the variable *cwnd* to estimate the appropriate congestion window size. In our implementation, *sb_net_target* represents the desired send buffer size when considering the value of *cwnd*, but not considering memory constraints. Following the $2 * bandwidth * delay$ rule of thumb described in footnote 1, (and keeping in mind that *cwnd* estimates the bandwidth-delay product), auto-tuning TCP will increase *sb_net_target* if *cwnd* grows larger than *sb_net_target*/2.

In the Congestion Avoidance phase, *cwnd* increases linearly until a loss is detected, then it is cut in half [RFC2001, Jac88]. When equilibrium is reached during Congestion Avoidance, there is a factor of two between the minimum and maximum values of *cwnd*. Therefore, auto-tuning TCP doesn't decrease *sb_net_target* until *cwnd* falls below *sb_net_target*/4, to reduce the flapping of *sb_net_target* during equilibrium[3].

others [VS94, MTW98] when a bug in TCP caused Congestion Avoidance to open the window too aggressively. TCPs without this bug do not suffer severe performance degradation when they are congestion window-limited.

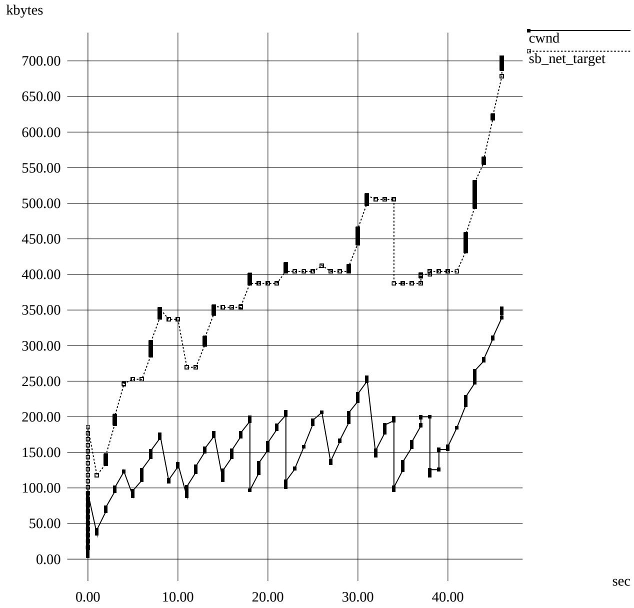

Figure 1: *sb_net_target* and *cwnd* over time

The operation of *sb_net_target* is illustrated in Figure 1. The rough sawtooth pattern at the bottom is *cwnd* from an actual connection. Above *cwnd* is *sb_net_target*, which stays between $2 * cwnd$ and $4 * cwnd$.

2.2.2 Fair Share of Memory

Since Congestion Avoidance regulates the sharing of bandwidth through a bottleneck, determining the buffer size of connections from *cwnd* should also regulate the memory usage of all connections through that bottleneck. But basing the send buffer sizes on *cwnd* is not sufficient for balancing memory usage, since concurrent connections may not share a bottleneck, but do share the same pool of (possibly limited) memory[4].

A mechanism inspired by the Max-Min Fair Share algorithm [MSZ96] was added to more strictly balance the memory usage of each connection. *hiwat_fair_share* represents the amount of memory that each connection is entitled to use. Small connections ($sb\_net\_target < hiwat\_fair\_share$) contribute the memory they don't use to the "shared pool", which is divided up equally among all connections that desire more than the fair share. The fair share is calculated in the *tcp_slowtimo*() routine twice per second, which should

[3] It is not known if the flapping of *sb_net_target* would cause performance problems, but it seems wise to limit small, periodic fluctuations until more is known.

[4] The problem of balancing memory usage of connections also presents itself for hosts which are not attempting to auto-tune, but simply have very large numbers of connections open (e.g. large web servers). As mentioned earlier, BSD-based systems use the buffer size as a *limit*, not as an allocation, so systems that are manually tuned for performance can become overcommitted if a large number of connections are in use simultaneously.

be low enough frequency to reduce the overhead caused by traversal of the connection list, but fast enough to maintain general fairness. Appendix A contains the detailed calculation of $hiwat\_fair\_share$.

Each time $hiwat\_fair\_share$ is calculated, and each time sb_net_target changes, sb_mem_target is updated to indicate the intended value of the send socket buffer size.

$$sb\_mem\_target = min(sb\_net\_target, hiwat\_fair\_share)$$

sb_mem_target holds the intended buffer size, even if the size cannot be attained immediately. If sb_mem_target indicates an increase in the send buffer size, the buffer is increased immediately. On the other hand, if sb_mem_target indicates a decrease in the send buffer size, the buffer is reduced in size as data is removed from the buffer (i.e. as the data is acknowledged by the receiver) until the target is met.

2.2.3 Memory Threshold

Even though memory usage is regulated by the algorithm above, it seems important to include a mechanism to limit the amount of system memory used by TCP since the authors are not aware of any operating system that is reliably well-behaved under mbuf exhaustion. Exhaustion could occur if a large number of mbuf clusters are in use by other protocols, or in the case of unexpected conditions. Therefore, a threshold has been added so that buffer sizes are further restricted when memory is severely limited.

On the sender side a single threshold, $AUTO\_SND\_THRESH$, affects the send buffer size[5]. If the number of mbuf clusters in use system-wide exceeds $AUTO\_SND\_THRESH$, then auto-tuned send buffers are reduced as acknowledged data is removed from them, regardless of the value of sb_mem_target.

Choosing the optimal value for $AUTO\_SND\_THRESH$ requires additional research.

3 Test Environment

The test environment involved a FDDI-attached sender PC in Pittsburgh running NetBSD 1.2 (see Figure 2, top). While it had 64MB of RAM to allow many concurrent processes to run, the amount of memory available for network buffers was intentionally limited in some tests. The NetBSD sender kernels for the tests in Sections 4.1 and 4.3 were compiled with NMBCLUSTERS of 4MB, allowing a total of 4MB of mbuf clusters to be used for all network connections, and 2MB to be used for send socket buffers. The test in Section 4.2 used a kernel compiled with NMBCLUSTERS of 2MB, allowing 2MB of mbuf clusters for all connections, and limiting the total memory used by send socket buffers to only 1MB.

The kernel was modified to include PSC SACK and the auto-tuning implementation described in Section 2. In addition to the auto-tuning code, kernel monitoring was added to be able to examine the internal effects of auto-tuning.

The kernel monitor logged TCP connection variables to a global table upon encountering events in the TCP stack[6].

[5] We make a slight approximation here. The thresholds are based on $NMBCLUSTERS$, a rigid upper bound on the number of 2048 byte mbuf *clusters*. Additional network memory is available from 128-byte mbufs, which do not have a rigid upper bound. Future work may refine how upper bounds on memory are determined.

[6] It was decided to log variables upon events rather than periodically in order to pinpoint important events, and to reduce the size of the data files.

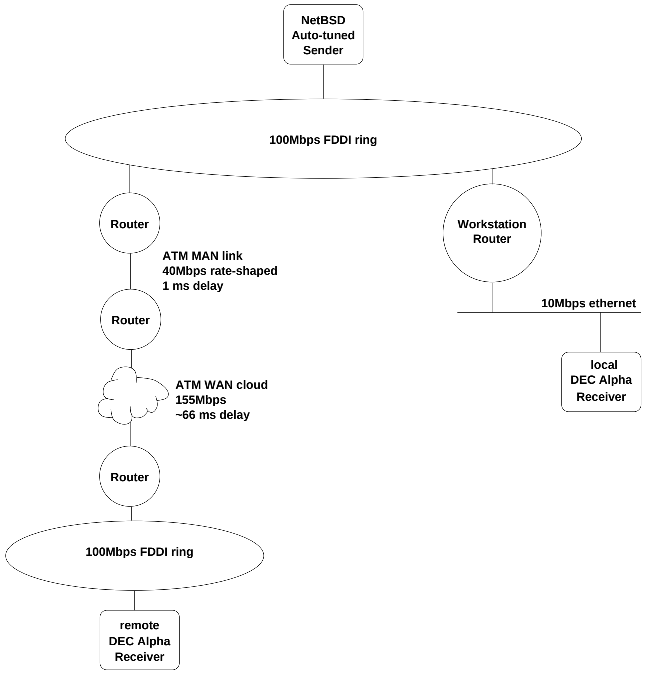

Figure 2: Test Topology

Events included entering or exiting recovery, triggering a retransmit timeout, or changing sb_net_target. The kernel log file was read periodically by a user-level process using NetBSD's *kvm* kernel memory interface.

The receivers were DEC Alphas running Digital Unix 4.0 with PSC SACK modifications. The remote receiver, which was used in all the tests, was a DEC 3000 Model 900 AXP workstation with 64MB of RAM located on the vBNS Test Net at San Diego (see Figure 2, bottom left). Between the sender and the remote receiver there was a site-internal 40Mbps bottleneck link and a minimum round-trip delay of 68ms, resulting in a bandwidth-delay product of 340kB.

The local receiver was a DEC Alphastation 200 4/166 with 96MB of RAM located in Pittsburgh (see Figure 2, bottom right). It was connected to the same FDDI ring as the sender via a 10Mbps private ethernet through a workstation router. The path delay was approximately 1ms, resulting in a bandwidth-delay product of 1.25kB.

Since pMTU discovery was not available for NetBSD 1.2, the MTUs were configured by hand with static routes on the end hosts to be 4352 bytes for the remote connection, and 1480 bytes for the local connection.

Since the receivers were not modified for auto-tuning, their receive socket buffers were set to 1MB by the receiving application. The connections were, therefore, not receive window-limited, and simulated the behavior of an auto-tuning receiver.

4 Auto-tuning Tests

For all of the tests below, a modified version of nettest [Cra92] was used to perform data transfers. The modifications involved stripping out all but basic functionality for unidirectional transfers, and customizing the output. Concurrent transfers were coordinated by a parent process that managed the data-transfer child processes, the kernel monitoring process, and the archiving of data to a remote tape archiver. In order to separate the time required to fork a process from the transfer time, all the data-transfer processes were started in a sleep-state ahead of time, awaiting a "START" signal from the parent. The parent would then send a "START" signal to the appropriate number of processes, which would open the data transfer TCP connections immediately. When all running children signaled to the parent that they were finished, the kernel monitor data from the run was archived, then the next set of concurrent transfers was set in motion with a "START" signal from the parent.

Performance was recorded on the receiver-side, since the sender processes believed they were finished as soon as the last data was placed in the (potentially large) send socket buffer.

The transfer size was 100MB per connection to the remote receiver, and 25MB per connection to the local receiver, reflecting the 4:1 ratio of bandwidths of the two paths.

On the DEC Alpha receivers, the maximum limit of mbuf clusters is an auto-sizing number. Since the receivers had sufficient RAM, the operating system allowed the maximum mbuf cluster limit to be large enough that it did not impose a limit on the transfer speeds.

Each TCP connection was one of three types:

default The default connection type used the NetBSD 1.2 static default socket buffer size of 16kB.

hiperf The hiperf connection type was hand-tuned for performance to have a static socket buffer size of 400kB, which was adequate for connections to the remote receiver. It is overbuffered for local connections.

auto Auto-tuned connections used dynamically adjusted socket buffer sizes according to the implementation described in Section 2.

Concurrent connections were all of the same type. Auto-tuned buffers were limited from growing larger than the kernel's maximum socket buffer size, which was set to 1MB on the sender.

4.1 Basic Functionality

The Basic Functionality test involved concurrent data transfers between the sender and the remote receiver.

In Figure 3, the aggregate bandwidth obtained by each set of connections is graphed. Only one type of connection was run at a time to more easily examine the performance and memory usage for each connection type. For instance, first a single auto-tuning connection was run, and its bandwidth was recorded. Next two auto-tuning connections were run simultaneously, and their aggregate bandwidth was recorded.

From the figure, it can be seen that the default tuning underutilizes the link when less than 22 concurrent connections are running. The send socket buffers of the connections were too small, limiting the rate of the transfers.

On the other hand, the hiperf hand-tuned connections get full performance because their socket buffers were configured to be large enough not to limit the transfer rate. As more connections were added, less send buffer space per connection was actually required due to the sharing of the link bandwidth, but since the sending application processes still had data to send, they filled the 400kB send buffers needlessly. Figure 4 illustrates memory usage for each type of connection.

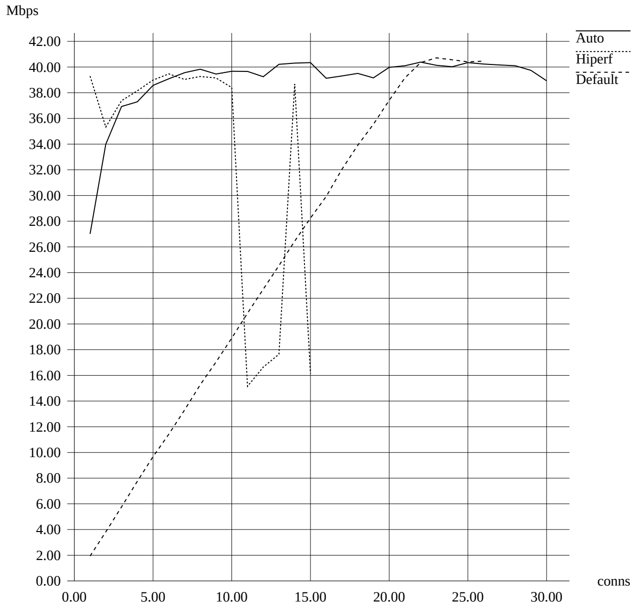

Figure 3: Aggregate Bandwidth comparison

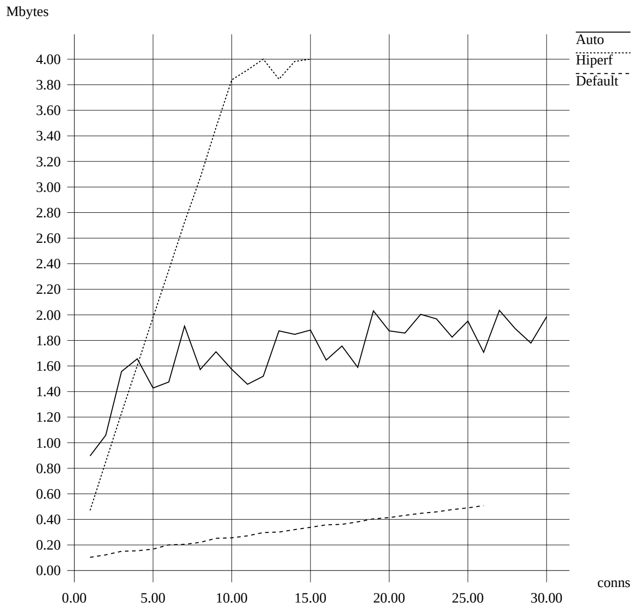

Figure 4: Peak usage of network memory

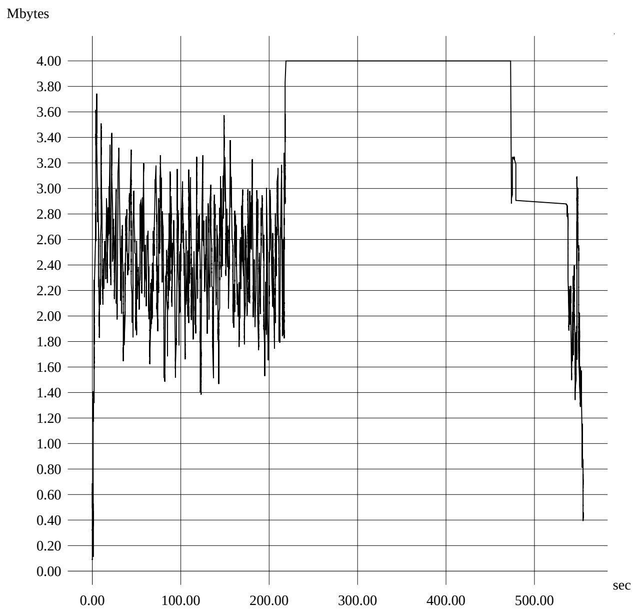

Figure 5: Network memory usage vs. time for 12 concurrent hiperf connections

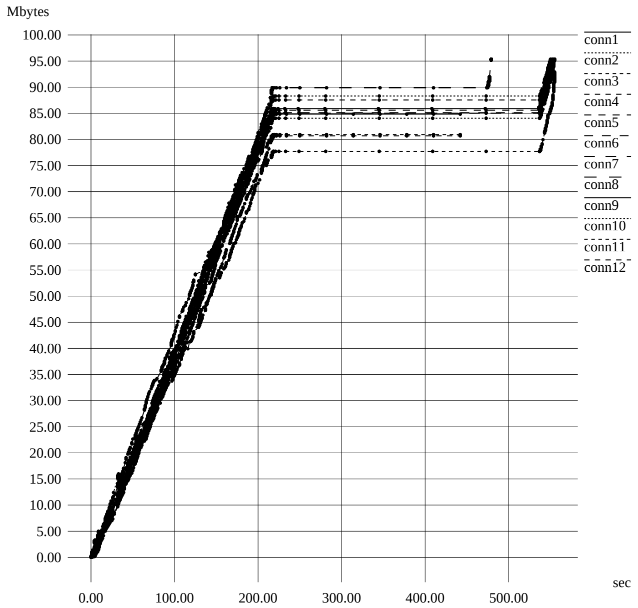

Figure 6: Relative sequence number vs. time for 12 concurrent hiperf connections

As the number of connections was increased, the hiperf sender began to run out of network memory, causing its behavior to degrade until the system finally crashed. Figure 5 shows the system memory being exhausted during the hiperf test with 12 concurrent connections. In Figure 6, it can be seen that around 220 seconds into the test all 12 connections stop sending data until two of the connections abort, freeing memory for the other connections to complete.

As the behavior of the system degraded in progressive hiperf tests, data became unavailable and is not shown in the graphs. The same degradation occurred with default-tuned connections when more than 26 of them were running concurrently.

The auto-tuned connections were able to achieve the performance of the hand-tuned hiperf connections by increasing the size of the send socket buffers when there were few connections, but decreased the send buffer sizes as connections were added. Therefore maximum bandwidth was able to be achieved across a much broader number of connections than either statically-tuned type[7]. In fact, the auto-tuned sender was able to support twice as many connections as the sender that was hand-tuned for high performance.

4.2 Overall Fairness of Multiple Connections under Memory-Limited Conditions

The authors hypothesized that auto-tuning would exhibit more fairness than hiperf tuning, because some small number of hiperf connections were expected to seize all of the available memory, preventing other connections from using any. In order to test the fairness hypothesis, we ran several sets of experiments where the amount of available system memory was quite low in comparison to the hiperf buffer tuning. (Specifically, the total network memory available was 1MB, allowing only two of the 400kB hiperf tuned connections to exist without exhausting memory).

In the course of running the experiments, we searched for evidence of unfairness in the hiperf connections, but only saw fair sharing of bandwidth among parallel connections. Thus, we conclude that although there is no explicit mechanism for enforcing fairness among hiperf connections, parallel hiperf connections appear to achieve fairness.

While the authors were not able to find proof that auto-tuning enhances fairness among concurrent connections, it is worth pointing out that auto-tuning still exhibits a major advantage over hiperf tuning. The hiperf tuned connections caused the system to become unstable and crash with a small number of connections. The auto-tuned connections, on the other hand, were able to run fairly and stably up to large numbers of connections.

4.3 Diverse Concurrent Connections

In the Diverse Concurrent Connections test, concurrent data transfers were run from the sender to both the remote receiver and the local receiver. The bandwidth-delay product of the two paths was vastly different: 340kB to the remote receiver, and 1.25kB locally.

Figure 7 shows the aggregate bandwidth on each path. The x axis represents the number of connections concurrently transferring data to *each* receiver. For each type of

[7] It is believed that an underbuffered bottleneck router is responsible for the reduced performance seen at the left of Figure 3. As more connections are added, each requires less buffer space in the router's queue. Statistically, the connections don't all require the buffer space at the exact same time, so less total buffer space is needed at the router as more connections use the same total bandwidth.

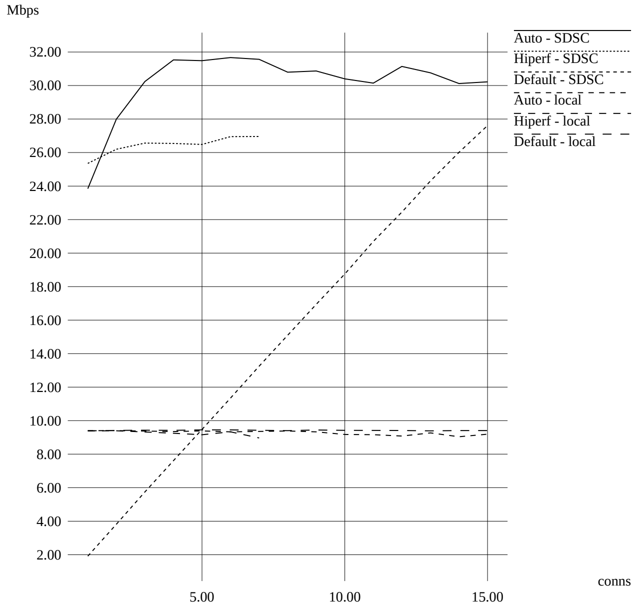

Figure 7: Aggregate Bandwidth over each path
The x axis represents the number of connections to *each* receiver. The three lines around 9.3 Mbps represent connections to the local receiver. The remaining three lines that achieve more than 10 Mbps represent connections to the remote receiver.

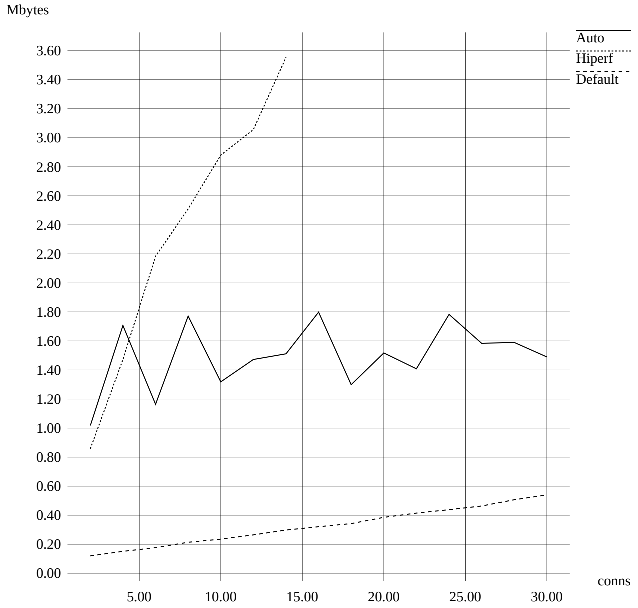

Figure 8: Peak usage of network memory for connections over two paths
The x axis represents the total number of connections to *both* receivers

tuning, the same number of connections were opened simultaneously to the local receiver and to the remote receiver. The aggregate bandwidth of the connections on each path are graphed individually.

In the default case, the 16kB send buffers were large enough to saturate the ethernet of the local path, while at the same time, they caused the link to the remote receiver to be underutilized because of the longer round trip time. In Figure 8, it can be seen that the maximum amount of network memory used in the default tests increased gradually as more connections were active.

Consider now the hiperf case. The 400kB send buffers are overkill for the local ethernet, which can easily be saturated (see Figure 7). However, the local connections waste memory that could be used for the remote connections, which get only about 27Mbps. But, as can be seen from Figure 8, memory is quickly used up needlessly until the operating system crashed.

Auto-tuning, on the other hand, gets improved performance to the remote receiver, while still saturating the local ethernet, because memory not needed by local connections is dedicated to the high bandwidth*delay connections. As can be seen from Figure 8, not enough memory is available to achieve the full 40Mbps, since *AUTO_SND_THRESH* is only 2MB, but the available memory is better utilized to obtain over 30Mbps, while allowing many more concurrent connections to be used.

As mentioned in footnote 7, the bottleneck router is underbuffered, causing reduced performance at small numbers of connections.

5 Open Issues

Several minor problems still remain to be examined. The first is that many TCPs allow *cwnd* to grow, even when the connection is not controlled by the congestion window. For connections that are receive window-limited, send window-limited, or application-limited, *cwnd* may grow without bound, needlessly expanding the automatically-sizing send buffers, wasting memory.

A specific example is an interactive application such as telnet. If a large file is displayed over an auto-tuned telnet connection, and the user types ^C to interrupt the connection, it may take a while to empty the large buffers. The same behavior also exists without auto-tuning if a system administrator manually configures the system to use large buffers. In both cases, the application can resolve the situation by setting the socket buffer sizes to a small size with setsockopt().

Another concern that requires further study is that allowing very large windows might cause unexpected behaviors. One example might be slow control-system response due to long queues of packets in network drivers or interface cards.

A final point that should be made is that all the tests were performed with an auto-tuning NetBSD 1.2 sender. Some aspects of auto-tuning are socket layer implementation-dependent and may behave differently when ported to other operating systems.

6 Conclusion

As network users require TCP to operate across an increasingly wide range of network types, practicality demands that TCP *itself* must be able to adapt the resource utilization of individual connections to the conditions of the network path and should be able to prevent some connections from being starved for memory by other connections. We proposed a method for the automatic tuning of socket buffers to provide adaptability that is missing in current TCP implementations.

We presented an implementation of auto-tuning socket buffers that is straightforward, requiring only about 140 lines of code. The improved performance obtained by auto-tuning is coupled with more intelligent use of networking memory. Finally, we described tests demonstrating that our implementation is robust when memory is limited, and provides high performance without manual configuration. Clearly visible in the tests is one key benefit of auto-tuning over hand-tuning: resource exhaustion is avoided.

It is important to note that none of the elements of auto-tuning allows a user to "steal" more than their fair share of bandwidth. As described by Mathis [MSMO97], TCP tends to equalize the window (in packets) of all connections that share a bottleneck, and the addition of auto-tuning to TCP doesn't change that behavior.

7 Acknowledgements

The authors would like to thank Greg Miller, of the vBNS division of MCI, for coordinating our use of their resources for the tests in this paper. The authors would also like to thank Kevin Lahey, of NASA Ames, and kc claffy and Jambi Ganbar, of SDSC for setting up hosts to allow testing during development of auto-tuning. And finally, the authors would like to thank the National Science Foundation for the continued funding of our network research.

A Detail of Memory-Balancing Algorithm

$hiwat\_fair\_share$ is determined twice per second as follows. The size of the "shared pool" is controlled by the kernel constant $AUTO\_SND\_THRESH$, which is set to $NMBCLUSTERS/2$ in our implementation. Let s be the set of small connections (i.e. those connections for which $sb\_net\_target < hiwat\_fair\_share$) that are currently in ESTABLISHED or CLOSE_WAIT states [RFC793, Ste94]. Let M be the sum of sb_net_target for all connections in s. We denote the number of connections in set s as $|s|$. Let $\overline{s}$ be the set of ESTABLISHED or CLOSE_WAIT connections not in s.

If $M \geq AUTO\_SND\_THRESH$, then too many small connections exist, and the memory must be divided equally among all connections.

$$hiwat\_fair\_share = \frac{AUTO\_SND\_THRESH}{|\overline{s} \cup s|}$$

Note that on the next iteration, $hiwat\_fair\_share$ is much smaller, causing some connections to move from s to $\overline{s}$.

If $M < AUTO\_SND\_THRESH$, the portion of the pool that is left unused by the small connections is divided equally by the large connections.

$$hiwat\_fair\_share = \frac{AUTO\_SND\_THRESH - M}{|\overline{s}|}$$

References

[Cra92] Nettest, 1992. Network performance analysis tool, Cray Research Inc.

[FF96] Kevin Fall and Sally Floyd. Simulations-based comparisons of tahoe, reno and SACK TCP. *Computer Communications Review*, 26(3), July 1996.

[Jac88] Van Jacobson. Congestion avoidance and control. *Proceedings of ACM SIGCOMM '88*, August 1988.

[Mah96] Jamshid Mahdavi. Enabling high performance data transfers on hosts: (notes for users and system administrators), November 1996. Obtain via: http://www.psc.edu/networking/perf_tune.html.

[MBKQ96] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.4 BSD Operating System*. Addison-Wesley, Reading MA, 1996.

[Min97] March 1997. Private conversation between Greg Minshall and the authors.

[MM96] Matthew Mathis and Jamshid Mahdavi. Forward Acknowledgment: Refining TCP congestion control. *Proceedings of ACM SIGCOMM '96*, August 1996.

[MSMO97] Matthew Mathis, Jeffrey Semke, Jamshid Mahdavi, and Teunis Ott. The macroscopic behavior of the TCP Congestion Avoidance algorithm. *Computer Communications Review*, 27(3), July 1997.

[MSZ96] Qingming Ma, Peter Steenkiste, and Hui Zhang. Routing high-bandwidth traffic in max-min fair share networks. *Proceedings of ACM SIGCOMM '96*, August 1996.

[MTW98] Gregory J. Miller, Kevin Thompson, and Rick Wilder. Performance measurement on the vBNS. In *Interop'98 Engineering Conference*, 1998.

[Net96] NetBSD 1.2 operating system, 1996. Based upon 4.4BSD Lite, it is the result of a collective volunteer effort. See http://www.netbsd.org.

[RFC793] J. Postel. Transmission control protocol, Request for Comments 793, September 1981.

[RFC1191] Jeffrey Mogul and Steve Deering. Path MTU discovery, Request for Comments 1191, October 1991.

[RFC1323] Van Jacobson, Robert Braden, and Dave Borman. TCP extensions for high performance, Request for Comments 1323, May 1992.

[RFC2001] W. Richard Stevens. TCP slow start, congestion avoidance, fast retransmit, and fast recovery algorithms, Request for Comments 2001, March 1996.

[RFC2018] Matthew Mathis, Jamshid Mahdavi, Sally Floyd, and Allyn Romanow. TCP Selective Acknowledgement options, Request for Comments 2018, October 1996.

[SAC98] Experimental TCP selective acknowledgment implementations, 1998. Obtain via: http://www.psc.edu/networking/tcp.html.

[Ste94] W. Richard Stevens. *TCP/IP Illustrated*, volume 1. Addison-Wesley, Reading MA, 1994.

[VS94] Curtis Villamizar and Cheng Song. High performance TCP in the ANSNET. *ACM SIGCOMM Computer Communication Review*, 24(5), October 1994.

[Wel96] Von Welch. A user's guide to TCP windows, 1996.
Obtain via: http://www.ncsa.uiuc.edu/People/vwelch/net_perf/tcp_windows.html.

[WS95] Gary R. Wright and W. Richard Stevens. *TCP/IP Illustrated*, volume 2. Addison-Wesley, Reading MA, 1995.