TCP tuning guide for distributed application on wide area networks

by Brian L. Tierney

Brian L. Tierney is a Staff Scientist and group leader of the Data Intensive Distributed Computing Group at Lawrence Berkeley National Laboratory.

<bltierney@lbl.gov>

1.0 Introduction

Obtaining good TCP throughput across a wide area network usually requires some tuning. This is especially true in high-speed "next generation Internet"-like networks, where, even though there is no congestion, an application may see only a small percentage of the total available bandwidth. This document describes several techniques required to obtain good throughput, and describes tools for diagnosing problems. This is a printer-friendly version of the Web document: *<http://www-didc.lbl.gov/tcp-wan.html>*. Check the Web page for updates. URLs for all tools mentioned in this document are listed in section 5.

This document is aimed mainly at software developers. All too often software developers blame the network for poor performance, when in fact the problem is un-tuned software. However, there are times when the network (or the operating system, as shown in section 4) really is the problem. This document explains some tools that can give software developers the evidence needed to make network engineers take them seriously.

2.0 TCP Buffer Sizes

TCP uses what it calls the "congestion window," or CWND, to determine how many packets can be sent at one time. The larger the congestion window size, the higher the throughput. The TCP "slow start" and "congestion avoidance" algorithms determine the size of the congestion window. The maximum congestion window is related to the amount of buffer space that the kernel allocates for each socket. For each socket, there is a default value for the buffer size, which can be changed by the program using a system library call just before opening the socket. There is also a kernel enforced maximum buffer size. The buffer size can be adjusted for both the send and receive ends of the socket.

To achieve maximal throughput it is critical to use optimal TCP send and receive socket buffer sizes for the link you are using. If the buffers are too small, the TCP congestion window will never fully open up. If the buffers are too large, the sender can overrun the receiver, and the TCP window will shut down. For more information, see the references on page 38.

Users often wonder why, on a network where the slowest hop from site A to site B is 100 Mbps (about 12 MB/sec), using ftp they can only get a throughput of 500 KB/sec. The answer is obvious if you consider the following: typical latency across the US is about 25 ms, and many operating systems use a default TCP buffer size of either 24 or 32 KB (Linux is only 8 KB). Assuming a default TCP buffer of 24KB, the maximum utilization of the pipe will only be 24/300 = 8% (.96 MB/sec), even under ideal conditions. In fact, the buffer size typically needs to be double the TCP congestion window size to keep the pipe full, so in reality only about 4% utilization of the network is

If you are using untuned TCP buffers you'll often get less than 5% of the possible bandwidth across a high-speed WAN path.

achieved, or about 500 KB/sec. Therefore if you are using untuned TCP buffers you'll often get less than 5% of the possible bandwidth across a high-speed WAN path. This is why it is essential to tune the TCP buffers to the optimal value.

The optimal buffer size is twice the bandwidth * delay product of the link:

buffer size = 2 * bandwidth * delay

The ping program can be used to get the delay, and pipechar or pchar, described below, can be used to get the bandwidth of the slowest hop in your path. Since ping gives the round-trip time (RTT), this formula can be used instead of the previous one:

buffer size = bandwidth * RTT

For example, if your ping time is 50 ms, and the end-to-end network consists of all 100BT Ethernet and OC3 (155 Mbps), the TCP buffers should be 0.05 sec * 10 MB/sec = 500 KB. If you are connected via a T1 line (1 Mbps) or less, the default buffers are fine, but if you are using a network faster than that, you will almost certainly benefit from some buffer tuning.

Two TCP settings need to be considered: the default TCP send and receive buffer size and the maximum TCP send and receive buffer size. Note that most of today's UNIX OSes by default have a maximum TCP buffer size of only 256 KB (and the default maximum for Linux is only 64 KB!). For instructions on how to increase the maximum TCP buffer, see Appendix A. Setting the default TCP buffer size greater than 128 KB will adversely affect LAN performance. Instead, the UNIX setsockopt call should be used in your sender and receiver to set the optimal buffer size for the link you are using. Use of setsockopt is described in Appendix B.

It is not necessary to set both the send and receive buffer to the optimal value, as the socket will use the smaller of the two values. However, it is necessary to make sure both are large enough. A common technique is to set the buffer in the server quite large (e.g., 512 KB) and then let the client determine and set the correct "optimal" value.

3.0 Other Techniques

Other useful techniques to improve performance over wide area networks include:

- Using large data block sizes. For example, most ftp implementations send data in 8 KB blocks. Use around 64 KB instead, since disk reads, memory copies, and network transfers are usually faster with large data blocks. However, be careful on QoS-enabled paths, since large blocks are more likely to overflow router buffers. 32K might be better on these networks.
- Sending lots of data at a time. If there is not enough data sent to keep the pipe full, the TCP window will never fully open up. In general, 0.5 MB or greater is a good amount to send at a time.
- Using multiple sockets. For example, to transfer a large file, send 25% of the file on each of 4 sockets in parallel. On a congested network, this often provides linear speedup! This only helps for large read/writes. Typically 4 sockets per host is a good number to use; with more than 4 the sockets will interfere with each other. The psockets library from the University of Illinois at Chicago makes it easy to add this ability to your applications. However, be careful using this technique with Gigabit Ethernet (1000BT) and a relatively underpowered receiver host. For example, a 500 MHz Pentium needs about 90% of the CPU just to read a single socket using Gigabit Ethernet, and sending data on 2 sockets instead of just 1 will decrease throughput dramatically.

- Using asynchronous I/O, a thread pool, or a select/poll mechanism. There is usually something else the application can be doing while it is blocked waiting for data. For example, use one thread to read data from the network, and a separate thread to write the data to disk. If reading is from multiple sockets, using a thread pool to handle multiple sockets in parallel can also help, especially on multi-CPU hosts.
- Avoiding unnecessary memory copies. Try to read the data straight into the memory location that will later need it. For example, if the data will be displayed by an X Window application, read it directly into the X pixmap structure. Do not read it into a read buffer and then copy it to the X buffer.

4.0 Network Problems

If you still have trouble getting high throughput, the problem may well be in the network. First, use netstat -s to see if there are a lot of TCP retransmissions. TCP retransmits usually indicate network congestion, but can also happen with bad network hardware, or misconfigured networks. You may also see some TCP retransmissions if the sending host is much faster than the receiving host, but TCP flow control should make the number of retransmits relatively low. Also look at the number of errors reported by netstat, as a large number of errors may also indicate a network problem.

If you still have trouble getting high throughput, the problem may well be in the network.

4.1 Use pipechar and pchar

The pchar tool does a pretty good job of giving hop-by-hop performance. If one of the hops is much slower than expected, this may indicate a network problem, and you might think about contacting your network administrator. Note that pchar often gives wrong or even negative results on very high speed links. It's most reliable on links that are OC3 (155 Mbps) or slower.

pipechar is a new tool, developed at LBNL, that will also find your bottleneck hop and seems to give more accurate results than pchar. While pchar attempts to accurately report the bandwidth and loss characteristics of every hop in the path, pipechar only accurately reports the slowest hop; results for all segments beyond the slowest segment will not be accurate. For example, if the first hop is the slowest, pipechar results for all other segments will be meaningless. Another significant difference between the tools is the time to run them. For a typical WAN path of eight hops, pipechar takes about one or two minutes, but pchar may take up to one hour.

If you are trying to determine the optimal TCP window size, the bottleneck hop is the only thing you are interested in. Therefore pipechar is clearly the better tool, since it takes much less time to identify the slowest hop. However, pchar is still a useful debugging tool.

4.2 Check the Duplex Mode

A common source of LAN trouble with 100BT networks is that the host is set to full duplex, but the Ethernet switch is set to half duplex, or vice versa. Most newer hardware will auto-negotiate this, but with some older hardware, auto-negotiation will sometimes fail, with the result being a working but very slow network (typically only 1–2 Mbps). It's best for both to be in full duplex if possible, but some older 100BT equipment only supports half duplex. See Appendix C for some ways to check what your systems are set to.

4.3 Use tcpdump/tcptrace

You can also use tcpdump to try to see exactly what TCP is doing. tcptrace is a very nice tool for formatting tcpdump output, and then xplot is used to view the results.

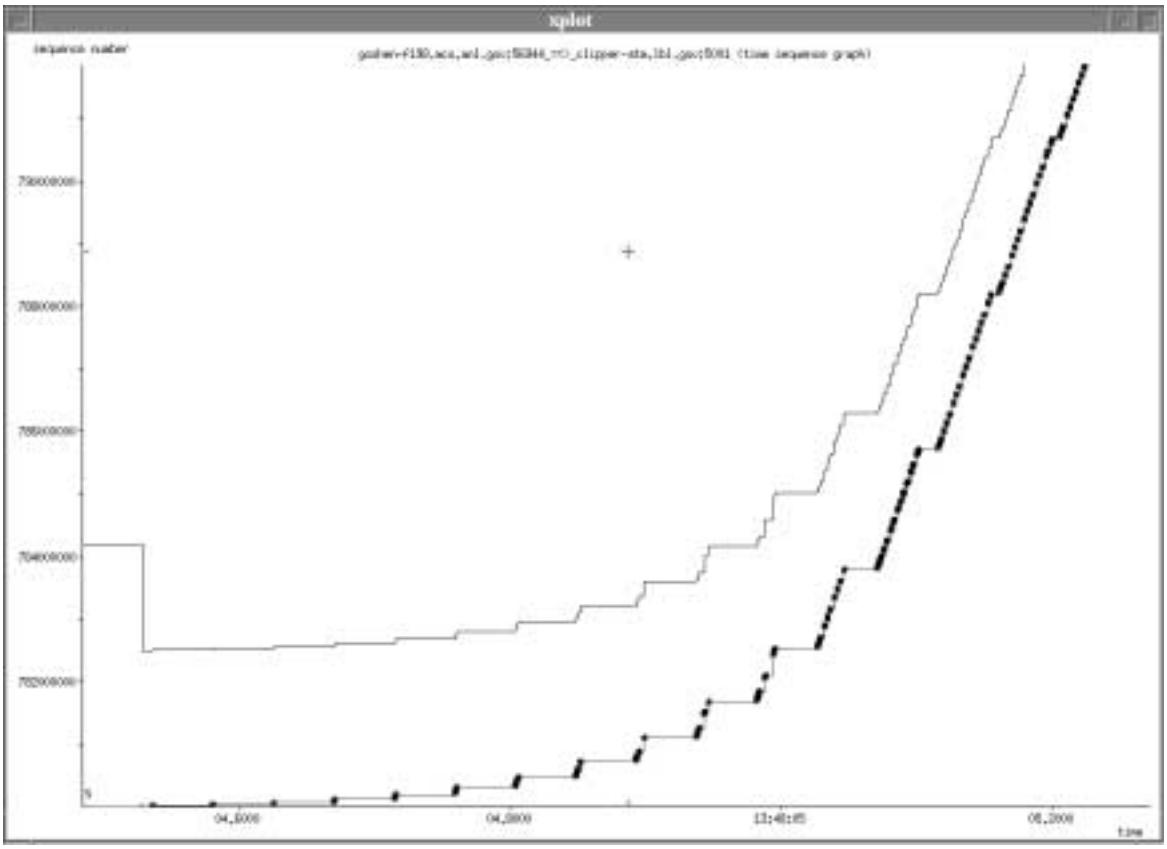

Figure 1. tcptrace *results showing TCP slow-start*

For example:

```
tcpdump -s 100 -w /tmp/tcpdump.out host myhost
tcptrace -Sl/tmp/tcpdump.out
xplot /tmp/a2b_tsg.xpl
```

NLANR's TCP Testrig is a nice wrapper for all of these tools, and includes information on how to make sense out of the results. An example of tcptrace results is shown in Figure 1, which shows the TCP slow start algorithm opening up the TCP congestion windows at the beginning of a data transmission.

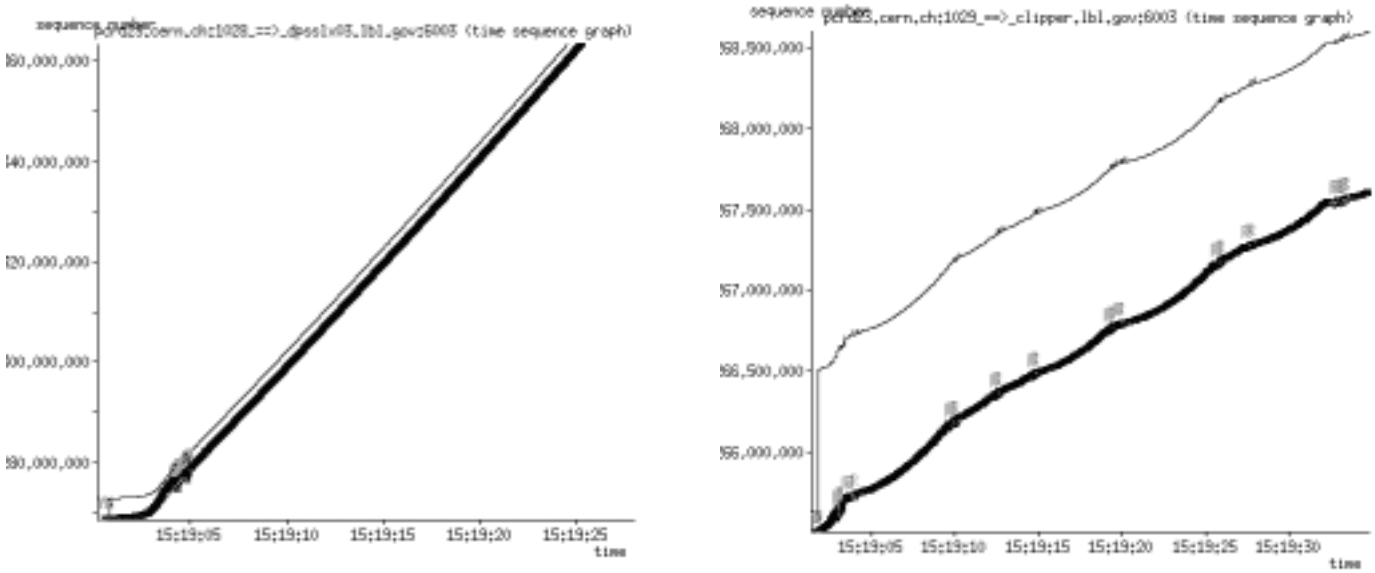

Figure 2. tcptrace *showing Linux TCP bug*

Linux to Linux *Linux to Solaris*

I recently used these tools to help identify a rather severe TCP bug in Linux. On a particular wide area network path I was getting a consistent 20 Mbps throughput with Solaris or FreeBSD sending to Solaris, Linux, or FreeBSD, but only 0.5 Mbps with Linux to Solaris or FreeBSD (Linux to Linux was also fine). Using tcpdump/tcptrace/xplot, I got the following plots. You have to be a serious TCP expert to really understand these plots (which I am not), but it's pretty clear that something strange is going on in the Linux sender. Using this data in Figure 2 as evidence, I was quickly able to convince the Linux TCP developers that there was a bug here, and the Linux 2.2.18 and the Linux 2.4.0-test12 kernels now include a fix for this problem.[1]

1. This Linux sender bug only occurs on networks with at least a 25 ms RTT and an end-to-end network path of at least 10 Mbps, and must have at least some congestion. The bug has something to do with the computation of the TCP RTO timer. If you are running a Linux server in this sort of network environment, I strongly encourage you to upgrade your kernel for find and install patch. For more information, see <*http://www-didc.lbl.gov/Linux-tcp-bug.html*>.

5.0 Tools

Here is the list of tools mentioned in this document, and a few others you may find useful:

- iperf: currently the best tool for measuring end-to-end TCP/UDP performance—<*http://dast.nlanr.net/Projects/Iperf/index.html*>
- NetTune: a library to increase the socket buffer size via an environment variable—<*http://www.ncne.nlanr.net/tools/application.html*>
- pipechar: hop-by-hop bottleneck analysis tool—<*http://www-didc.lbl.gov/pipechar/*>
- pchar: hop-by-hop performance measurement tool—<*http://www.employees.org/~bmah/Software/pchar/*>
- psockets: easy to use parallel sockets library—<*http://www.ncdm.uic.edu/html/psockets.html*>
- tcpdump: dumps all TCP header information for a specified source/destination—<*ftp://ftp.ee.lbl.gov/*>
- tcptrace: formats tcpdump output for analysis using xplot—<*http://jarok.cs.ohiou.edu/software/tcptrace/*>
- NLANR TCP Testrig: Nice wrapper for tcpdump and tcptrace tools—<*http://www.ncne.nlanr.net/TCP/testrig/*>
- traceroute: lists all routers from current host to remote host—<*ftp://ftp.ee.lbl.gov/*>

Many other tools are listed at the NLANR Engineering Tools Repository at <*http://www.ncne.nlanr.net/tools/*>.

6.0 Other Useful Links

- Solaris 2.6 SACK patch: <*ftp://play-ground.sun.com/pub/sack/tcp.sack.tar.Z*> (SACK is part of Solaris >= 2.7 and Linux >= 2.2)
- Pittsburgh Supercomputer Center Tuning Guide: <*http://www.psc.edu/networking/perf_tune.html*>

7.0 Updates

The goal is to continually update this document. Please send additions and corrections to <*bltierney@lbl.gov*>. Note that the Web-based version at <*http://www-didc.lbl.gov/tcp-wan.html*> may be more up-to-date.

8.0 Acknowledgments

The work described in this paper is supported by the US Dept. of Energy, Office of Science, Office of Computational and Technology Research, Mathematical, Information, and Computational Sciences Division (<*http://www.er.doe.gov/production/octr/mics/*

index.html>), under contract DE-AC03-76SF00098 with the University of California. This is report no. LBNL-45261.

9.0 References

1. V. Jacobson, "Congestion Avoidance and Control," Proceedings of ACM SIGCOMM '88, August 1988.

2. J. Semke, M. Mathis Mahdavi, "Automatic TCP Buffer Tuning," *Computer Communication Review*, ACM SIGCOMM, vol. 28, no. 4, October 1998.

3. B. Tierney, J. Lee, B. Crowley, M. Holding, J. Hylton, F. Drake, "A Network-Aware Distributed Storage Cache for Data Intensive Environments," Proceeding of IEEE High Performance Distributed Computing Conference (HPDC-8), August 1999, LBNL-42896.

Appendix A: Changing TCP System Default Values

On Linux, add something like the following to one of your boot scripts. On our systems, we add the following to /etc/rc.d/rc.local to increase the maximum buffers to 8 MB and the default to 64 KB.

```
echo 8388608 > /proc/sys/net/core/wmem_max
echo 8388608 > /proc/sys/net/core/rmem_max
echo 65536 > /proc/sys/net/core/rmem_default
echo 65536 > /proc/sys/net/core/wmem_default
```

For Solaris, create a boot script similar to this (e.g., /etc./rc2.d/S99ndd):

```
#!/bin/sh
# increase max tcp window
# Rule-of-thumb: max_buf = 2 x cwnd_max (congestion window)
ndd -set /dev/tcp tcp_max_buf 4194304
ndd -set /dev/tcp tcp_cwnd_max 2097152
# increase DEFAULT tcp window size
ndd -set /dev/tcp tcp_xmit_hiwat 65536
ndd -set /dev/tcp tcp_recv_hiwat 65536
#
osver=`uname -r`
# Turn on Selective Acks (SACK)
if [ $osver = "5.7" ]; then
      # SACK is on in "passive" mode by default in Solaris.
      # This will set it to "active" mode
      ndd -set /dev/tcp tcp_sack_permitted 2
fi
```

Note that SACK comes as part of Solaris >= 2.7, but for Solaris 2.6, you must install the SACK patch, available from <*ftp://playground.sun.com/pub/sack/tcp.sack.tar.Z*>

For Irix (6.4, 6.5), the maximum TCP buffer doesn't appear to be setable, and is fixed at 4 MB. To modify the default buffer size, edit the file: /var/sysgen/master.d/bsd, and set:

```
tcp_sendspace=65536
tcp_recvspace=65536
```

See the PSC TCP Performance Tuning guide (<*http://www.psc.edu/networking/perf_tune.html*>) for information on setting TCP parameters for other operating systems.

Appendix B: C Code to Set the TCP Buffer Size

Here is how to use the setsockopt call to set TCP buffer sizes within your application using C:

```
int skt, int sndsize;
err = setsockopt(skt, SOL_SOCKET, SO_SNDBUF, (char *)&sndsize,
        (int)sizeof(sndsize));
```

or

```
int skt, int sndsize;
err = setsockopt(skt, SOL_SOCKET, SO_RCVBUF, (char *)&sndsize,
        (int)sizeof(sndsize));
```

Here is sample C code for checking what the buffer size is currently set to:

```
int    sockbufsize = 0; int    size = sizeof(int);
err = getsockopt(skt, SOL_SOCKET, SO_RCVBUF, (char *)&sockbufsize,&size);
```

Note: It is a good idea to always call getsockopt after setting the buffer size, to make sure that the OS supports buffers of that size. The best place to check it is after the server listen() or client connect(). Some OSes seem to modify the TCP window size to their max or default at that time. Also note that Linux mysteriously doubles whatever value you pass to the setsockopt call, so when you do a getsockopt you will see double what you asked for. Don't worry, as this is "normal" for Linux.

Appendix C: Checking for Full vs. Half Duplex Mode

Have your network administrator check what duplex your switch or hub is set to, and then check your hosts.

On Solaris, here is the command to check the duplex:

```
ndd /dev/hme link_mode
```

where a return value of 0 = half duplex, and 1 = full duplex.

To force to full duplex:

```
ndd -sec /dev/hme adv_100fdx_cap ndd -set /dev/hme adv_autoneg_cap 0
```

To force to half duplex:

```
ndd -sec /dev/hme adv_100hdx_cap ndd -set /dev/hme adv_autoneg_cap 0
```

Please send info on other operating systems to <*bltierney@lbl.gov*>, and I'll add them to the version of this document on my Web site.