IEEE Symposium on High-Performance Distributed Computing (HPDC'02),
Edinburgh, Scotland, July 2002. LA-UR 02-2492

A Comparison of TCP Automatic Tuning Techniques for Distributed Computing

Eric Weigle and Wu-chun Feng

Research and Development in Advanced Network Technology
Computer and Computational Sciences Division
Los Alamos National Laboratory
Los Alamos, NM 87545

{ehw,feng}@lanl.gov

Abstract

Rather than painful, manual, static, per-connection optimization of TCP buffer sizes simply to achieve acceptable performance for distributed applications [8, 10], many researchers have proposed techniques to perform this tuning automatically [4, 7, 9, 11, 12, 14]. This paper first discusses the relative merits of the various approaches in theory, and then provides substantial experimental data concerning two competing implementations – the buffer autotuning already present in Linux 2.4.x and "Dynamic Right-Sizing." This paper reveals heretofore unknown aspects of the problem and current solutions, provides insight into the proper approach for different circumstances, and points toward ways to further improve performance.

Keywords: dynamic right-sizing, autotuning, high-performance networking, TCP, flow control, wide-area network.

1. Introduction

TCP, for good or ill, is the only protocol widely available for reliable end-to-end congestion-controlled network communication, and thus it is the one used for almost all distributed computing.

Unfortunately, TCP was not designed with high-performance computing in mind – its original design decisions focused on long-term fairness first, with performance a distant second. Thus users must often perform tortuous manual optimizations simply to achieve acceptable behavior. The most important and often most difficult task is determining and setting appropriate buffer sizes. Because of this, at least six ways of automatically setting these sizes have been proposed.

In this paper, we compare and contrast these tuning methods. First we explain each method, followed by an in-depth discussion of their features. Next we discuss the experiments to fully characterize two particularly interesting methods (Linux 2.4 autotuning and Dynamic Right-Sizing). We conclude with results and possible improvements.

1.1. TCP Tuning and Distributed Computing

Computational grids such as the Information Power Grid [5], Particle Physics Data Grid [1], and Earth System Grid [3] all depend on TCP. This implies several things.

First, bandwidth is often the bottleneck. Performance for distributed codes is crippled by using TCP over a WAN. An appropriately selected buffer tuning technique is one solution to this problem.

Second, bandwidth and time are money. An OC-3 at 155Mbps can cost upwards of $50,000 a month and higher speeds cost even more. If an application can only utilize a few megabits per second, that money is being wasted. Time spent by people waiting for data, time spent hand-tuning network parameters, time with under-utilized compute resources – also wasted money. Automatically tuned TCP buffers more effectively utilize network resources and save that money, but an application designer must still choose from the many approaches.

Third, tuning is a pain. Ideally, network and protocol designers produce work so complete that those doing distributed or grid computing are not unduly pestered with the "grungy" details. In the real world, application developers must still make decisions in order to attain peak performance. The results in this paper show the importance of paying attention to the network and show one way to achieve maximal performance with minimal effort.

2. Buffer Tuning Techniques

TCP buffer-tuning techniques balance memory demand with the reality of limited resources – maximal TCP buffer space is useless if applications have no memory. Each technique discussed below uses different information and makes different trade-offs. All techniques are most useful for large data transfers (at least several times the $bandwidth \times delay$ product of the network). Short, small transmissions are dominated by latency, and window size is practically irrelevant.

2.1. Current Tuning Techniques

1. Manual tuning [8, 10]
2. PSC's Automatic TCP Buffer Tuning [9]
3. Dynamic Right-Sizing (DRS) [4, 14]
4. Linux 2.4 Auto-tuning [12]
5. Enable tuning [11]
6. NLANR's Auto-tuned FTP (in ncFTP) [7]
7. LANL's DRS FTP (in wuFTP)

Manual tuning is the baseline by which we measure autotuning methods. To perform manual tuning, a human uses tools such as `ping` and `pathchar` or `pipechar` to determine network latency and bandwidth. The results are multiplied to get the $bandwidth \times delay$ product, and buffers are generally set to twice that value.

PSC's tuning is a mostly sender-based approach. Here the sender uses TCP packet header information and timestamps to estimate the $bandwidth \times delay$ product of the network, which it uses to resize its send window. The receiver simply advertises the maximal possible window. PSC's paper [9] presents results for a NetBSD 1.2 implementation, showing improvement over stock by factors of 10-20 for small numbers of connections.

DRS is a mostly receiver-based buffer tuning approach where the receiver tries to estimate the $bandwidth \times delay$ product of the network and the congestion-control state of the sender, again using TCP packet header information and timestamps. The receiver then advertises a window large enough that the sender is not flow-window limited.

Linux autotuning refers to a memory management technique used in the stable Linux kernel, version 2.4. This technique does not attempt any estimates of the $bandwidth \times delay$ product of a connection. Instead, it simply increases and decreases buffer sizes depending on available system memory and available socket buffer space. By increasing buffer sizes when they are full of data, TCP connections can increase their window size – performance improvements are an intentional side-effect.

Enable uses a daemon to perform the same tasks as a human performing manual tuning. It gathers information about every pair of hosts between which connections are to be tuned and saves it in a database. Hosts then look up this information when opening a connection and use it to set their buffer sizes. Enable [11] reports performance improvements over untuned connections by a factor of 10-20 and above 2.4 autotuning by a factor of 2-3.

Auto-ncFTP also mimics the same sequence of events as a human manually tuning a connection. Here, it is performed once just before starting a data connection in FTP so the client can set buffer sizes appropriately.

DRS FTP uses a new command added to the FTP control language to gain network information, which is used to tune buffers during the life of a connection. Tests of this method show performance improvements over stock FTP by a factor of 6 with 100ms delay, with optimally tuned buffers giving an improvement by a factor of 8.

2.2. Comparison of Tuning Techniques

Tuning	Level	Changes	Band	Visibility
PSC	Kernel	Dynamic	In	Transparent
Linux 2.4	Kernel	Dynamic	In	Transparent
DRS	Kernel	Dynamic	In	Transparent
Enable	User	Static	Out	Visible
NLANR FTP	User	Static	Out	Opaque
DRS FTP	User	Dynamic	Both	Opaque
Manual	Both	Static	Out	Visible

Table 1. Comparison of Tuning Techniques

User-level versus Kernel-level refers to whether the buffer tuning is accomplished as an application-level solution or as a change to the kernel (Linux, *BSD, etc.).

Manual tuning tediously requires both types of changes. An "ideal" solution would require only one type of change – kernel-level for situations where many TCP-based programs require high performance, user-level where only a single TCP-based program (such as FTP) requires high performance.

Kernel-level implementations will always be more efficient, as more network and high-resolution timing information is available, but they are complicated and non-portable. Whether this is worth the 20-100% performance improvement is open to debate.

Static versus Dynamic refers to whether the buffer tuning is set to a constant at the start of a connection, or if it can change with network "weather" during the lifetime of a connection.

Generally a dynamic solution is preferable – it adapts itself to changes in network state, which some work has shown to have multi-fractal congestion characteristics [6, 13]. Static buffer sizes are always too large or small given "live" networks. Yet, static connections often have smoother application-level performance than dynamic connections, which is desirable.

Unfortunately, both static and dynamic solutions have problems. Dynamic changes in buffer sizes imply changes in the advertised window, which if improperly implemented can break TCP semantics (data legally sent for a given window is in-flight when the window is reduced, thus causing the data to be dropped at the end host). Current dynamic tuning methods monotonically increase window sizes to avoid this – possibly wasting memory.

In-Band versus Out-of-Band refers to whether $bandwidth \times delay$ information is obtained from the connection itself or is gathered separately from the data transmission to be tuned. An ideal solution would be in-band to minimize user inconvenience and ensure the correct time-dependent and path-dependent information is being gathered.

DRS FTP is "both" because data is gathered over the control channel; usually this channel uses the same path as the data channel, but in some "third-party" cases the two channels are between different hosts entirely. In the first case data collection is "in-band", while in the second not only is it out of band, it measures characteristics of the wrong connection! Auto-ncFTP suffers from the same "third-party" problem.

Transparent versus Visible refers to user inconvenience – how easily can a user tell if they are using a tuning method, how many changes are required, etc. An ideal solution would be transparent after the initial install and configuration required by all techniques.

The kernel approaches are transparent; other than improved performance they are essentially invisible to average users. The FTP programs are "opaque" because they can generate detectable out-of-band data, and require some start-up time to effectively tune buffer sizes. Enable is completely visible. It requires a daemon and database separate from any network program to be tuned, generates frequent detectable network benchmarking traffic, and requires changes to each network program that wishes to utilize its functionality.

3. Experiments

These experiments shift our focus to the methods of direct interest: manual tuning, Linux 2.4 autotuning, and Dynamic Right-Sizing under Linux. The remaining approaches are not discussed further because such analysis is available in the referenced papers.

3.1. Varied Experimental Parameters

Our experiments consider the following parameters:

Tuning (None, 2.4-auto, DRS): We compare a Linux 2.2.20 kernel which has no autotuning, a 2.4.17 kernel which has Linux autotuning, and a 2.4.17 kernel which also has Dynamic Right-Sizing. We will refer to these three as 2.2.20-None, 2.4.17-Auto, and 2.4.17-DRS.

Buffer Sizes (32KB to 32MB): Initial buffer size configuration is required even for autotuning implementations. There are three cases:

1. No user or kernel tuning; buffer sizes at defaults. Gives baseline for comparison with tuned results.
2. Kernel-only tuning; configure maximal buffer sizes only. Gives results for kernel autotuning implementations.
3. User and kernel tuning; use `setsockopt()` to configure buffer sizes manually. Gives results for manually tuned connections.

Network Delay (≈0.5ms, 25ms, 50ms, 100ms): We vary the delay from 0.5ms to 100ms to show the performance differences between LAN and WAN environments. We use TICKET [15] to perform WAN emulation. This emulator can route at line rate (up to 1Gbps in our case) introducing a delay between 200 microseconds and 200 milliseconds.

Parallel Streams (1, 2, 4, 8): We use up to 8 parallel streams to test the effectiveness of this commonly-used technique with autotuning techniques. This also shows how well a given tuning technique scales with increasing numbers of flows. When measuring performance, we time from the start of the first process to the finish of the last process.

3.2. Constant Experimental Parameters

Topology: Figure 1 shows the generic topology we use in our tests. We have some number of network source (S) processes sending data to another set of destination (D) processes through a pair of bottleneck routers (R) connected via some WAN cloud. The "WAN cloud" may be a direct long-haul connection or through some arbitrarily complex network. (In the simplest case, both routers and the "WAN cloud" could be a single very high-bandwidth LAN switch).

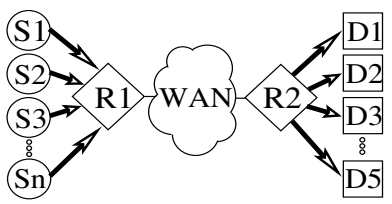

Figure 1. Generic Topology

Our experiments place all processes (parallel streams) on a single host. The results of more complicated one-to-many or many-to-one experiments (common in scatter-gather computation, or for web servers) can be inferred by observing memory and CPU utilization on the hosts. This information shows the scalability of the sender and receiver tuning and whether one end's behavior characterizes the performance of the connection. This distinction is critical for one-to-many relationships, as the "one" machine must split its resources among many flows while each of the "many" machines can dedicate more resources to the one flow.

Unidirectional Transfers: Although TCP is inherently a full duplex protocol, the majority of traffic generally flows in one direction. TCP protocol dynamics do not significantly differ between one flow with bidirectional traffic and two unidirectional flows sending in opposite directions [16].

Loss: Our WAN emulator is configured to emulate no loss (although loss may still occur due to sender/receiver buffer over-runs). All experiments are intended to be the best-case scenario. The artificial inclusion of loss adds nothing to the discussion, as congestion control for TCP Reno/SACK under Linux is a constant for all experiments.

Data Transfer: Rather than using some of the available benchmarking programs we chose to write a simple TCP based program to mimic message-passing traffic. This program tries to send large (1MB) messages between hosts as fast as possible. A total of 128 messages are sent, a number chosen because:

1. 128MB transfers are large enough to allow the congestion window to fully open.
2. 128MB transfers are small enough to occur commonly in practice [1].
3. Longer transfers do not help differentiate among tuning techniques (tested, but results omitted).
4. It is evenly divisible among all numbers of parallel streams.

Hardware: Tests are run between two machines with dual 933MHz Pentium III processors, an Alteon Tigon II Gigabit Ethernet card on a 64bit/66MHz PCI bus, and 512MB of memory.

4. Results and Analysis

We present data in order of increasing delay. With constant bandwidth (Gigabit Ethernet), this will show how well each approach scales as pipes get "fatter."

[1] "In the long run we are all dead." -John Maynard Keynes

4.1. First Case, ≈0.5ms Delay

With delays on the order of half a millisecond, we expect that even very high bandwidth links can be saturated with small windows – the default 64KB buffers should be sufficient.

Figure 2 shows the performance using neither user nor kernel tuning. With a completely default configuration, the Linux 2.4 stack with autotuning outperforms the Linux 2.2 stack without autotuning by 100Mbps or more (as well as showing more stable behavior). Similarly, 2.4.17-DRS outperforms 2.4.17-Auto by a smaller margin of 30-50Mbps. This is due to more appropriate use of TCP's advertised window field and faster growth to the best buffer size possible.

Unexpectedly for such a low-delay case, all kernels benefit from the use of parallel streams, with improvements in performance from 55-70%. When a single data flow is striped among multiple TCP streams, it effectively obtains a super-exponential slow-start phase and additive increase by N (the number of TCP streams). In this case, that behavior improves performance.

Note that limitations in the firmware of our Gigabit Ethernet NICs limit performance to 800Mbps or below, so we simply consider 800Mbps ideal. [2]

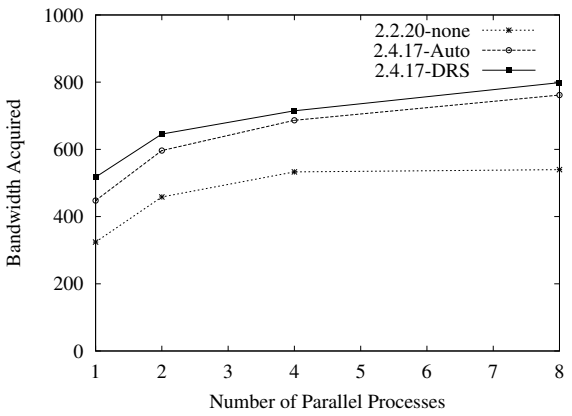

Figure 2. No Tuning, 0.5ms

Figure 3 shows the performance with kernel tuning only; that is, increasing the maximum amount of memory that the kernel can allocate to a connection.

As expected, results for 2.2.20-None (which does no autotuning) mirror the results from the prior test.

2.4.17-Auto connections perform 30-50Mbps better than in the untuned case, showing that the default 64KB buffers were insufficient.

[2] Custom firmware solutions can improve throughput, but such results are neither portable nor relevant to this study.

2.4.17-DRS connections also perform better with one or two processes, but as the number of processes increases, DRS actually performs worse! DRS is more aggressive in allocating buffer space; with such low delay, it overallocates memory, and performance suffers (see Figure 5's discussion below).

Furthermore, performance is measured at the termination of the entire transfer (when the final parallel process completes). Large numbers of parallel streams can lead to the starvation of one or more processes due to TCP congestion control, so the parallelized transfer suffers. Yet this can be a good thing – parallel flows can induce chaotic network behavior and be unfair in some cases; by penalizing users of heavily parallel flows, DRS could induce more network fairness while still providing good performance.

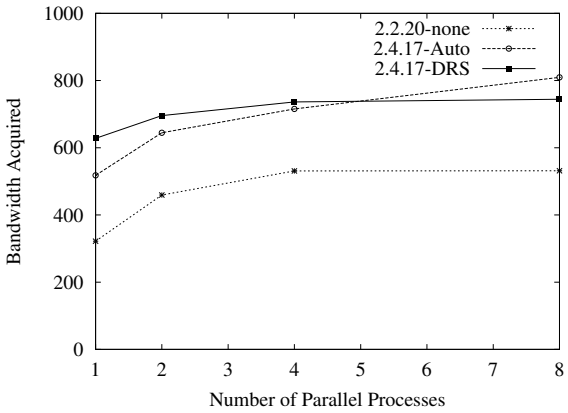

Figure 3. Kernel-Only Tuning, 0.5ms

Figure 4 shows the results for hand-tuned connections. DRS obeys the user when buffers are set by `setsockopt()`, so 2.4.17-Auto and 2.4.17-DRS use the same buffer sizes and perform essentially the same. The performance difference between 2.2.20 and 2.4.17 is due to stack improvements in Linux 2.4.

The "ideal" buffer sizes in the prior graph (Figure 4) are larger than one might expect; Figure 5 shows the performance of 2.4.17-Auto with buffer sizes per process between 8KB and 64MB. We achieve peak performance with sizes on the order of 1MB – much larger than the calculated ideal of 64KB, the $bandwidth \times delay$ of the network. The difference is due to the interaction and feedback between several factors, the most important of which are TCP congestion control and process scheduling.

To keep the pipe full, one must buffer enough data to avoid transmission "bubbles." However, with multi-fractal burstiness caused by TCP congestion control [6, 13], occasionally the network is so overloaded that very large buffers are needed to accommodate it. Also, these buffers themselves can increase the effective delay (and therefore increasing the buffering required) in a feedback loop only terminated by a lull later in the traffic stream. This buffering can occur either in the network routers or in the end hosts.

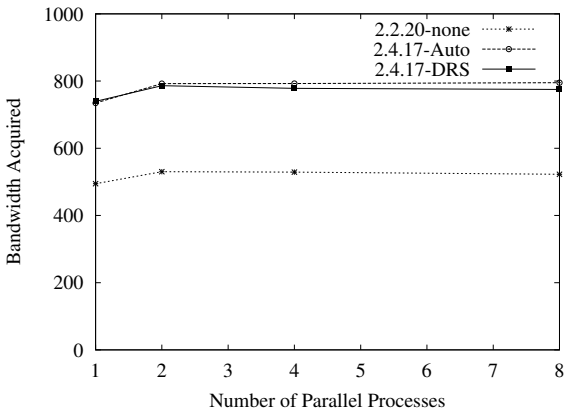

Figure 4. User/Kernel Tuning with Ideal Sizes, 0.5ms

Because of process scheduling, it is incorrect to divide the predicted "ideal" buffer size (the $bandwidth \times delay$) by the number of processes to determine the buffer size per process when using parallel streams. Because only one process can run on a given CPU at a given time, the kernel must buffer packets for the remaining processes until they can get a time slice. Thus, as the number of processes grows, the effective delay experienced by those processes increases, and the amount of required buffering also grows. Beyond a certain point, this feedback is great enough that the addition of additional parallelism is actually detrimental. This is what we see with DRS in Figure 5.

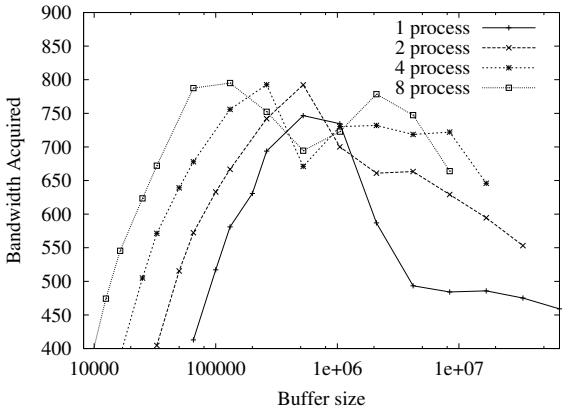

Figure 5. Effect of Buffer Size on Performance, 0.5ms

4.2. Second Case, ≈25ms Delay

This case increases delay to values more in line with a network of moderate size, giving a $bandwidth \times delay$ product of over 3MB. Figure 6 shows how the default configuration is insufficient for high performance, giving less than 20Mbps for a single process with all kernels. As the number of processes increases, our effective flow window increases, and we achieve a linear speed-up. In this case, simple autotuning outperforms DRS, as the memory-management technique is more effective with small windows (it was designed for heavily loaded web servers).

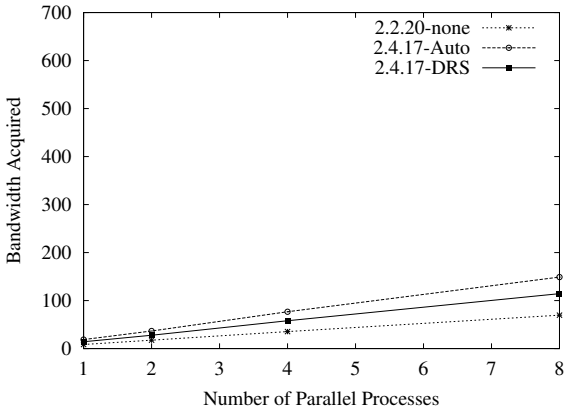

Figure 6. No Tuning, 25ms

Figure 7 shows results with Kernel-Only Tuning. The performance of DRS improves dramatically while the performance of simple autotuning and untuned connections is constant. As we increase the number of processes we again see the performance of DRS fall.

This graph actually reveals a bug in the Linux 2.4 kernel series that our DRS patch fixes; the window scaling advertised in SYN packets is based on initial (default) buffer size, not the maximal buffer size up to which Linux can tune. Thus with untuned default buffers, no window scaling is advertised – even if the kernel is allowed to allocate multi-megabyte buffers, the size of those buffers cannot be represented in TCP packet headers.

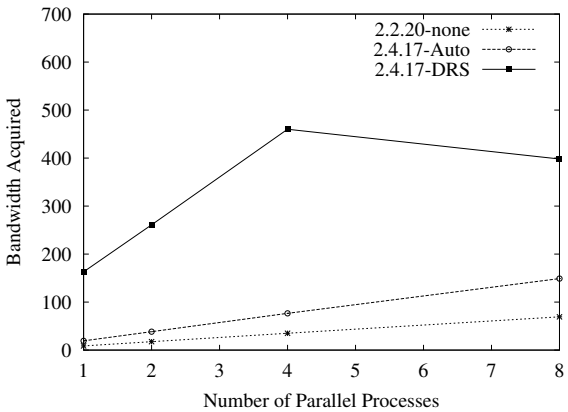

Figure 7. Kernel-Only Tuning, 25ms

With both user and kernel tuning, maximal performance increases for all kernels. However, as shown in Figure 8, performance does fall for DRS in the two and four process case – here we see that second-guessing the kernel can cause problems, and larger buffer sizes are not always desirable.

The other feature of note is that 2.4.17-DRS performance is not identical to 2.4.17-Auto – the only difference in this case is that DRS uses a slightly different algorithm to advertise its receiver window, and this pays off with a 150Mbps performance improvement.

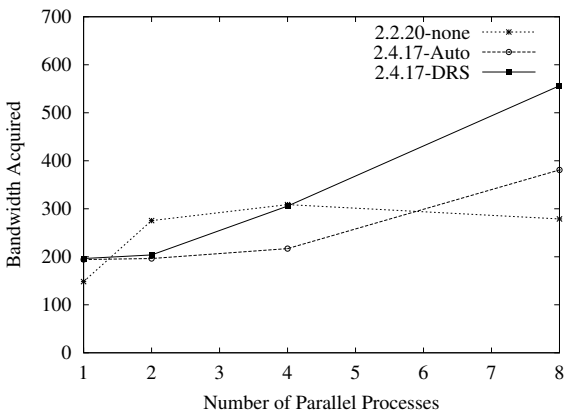

Figure 8. User/Kernel Tuning with Ideal Sizes, 25ms

4.3. Third and Fourth Cases, 50-100ms Delay

The patterns observed in results for the 50ms and 100ms cases do not significantly differ (other than adjustments in scale) from those in the 25ms case – the factors dominating behavior are the same. That is, at low delays (below 20ms), one can find very interesting behavior as TCP interacts with the operating system, NIC, and so forth. At higher delays (25ms and above), the time scales are large enough that TCP slow-start, additive increase, and multiplicative decrease behaviors are most important; interactions with the operating system and so forth become insignificant.

In fact, the completely untuned cases differ so little that the following three equations (generated experimentally) suffice to calculate the bandwidth in Mbps with error uniformly below 20%, given only the number of processes and the delay in milliseconds.

- 2.2.20-None: $(processes \times 214)/delay$
- 2.4.17-Auto: $(processes \times 467)/delay$
- 2.4.17-DRS: $(processes \times 355)/delay$

As in Figure 7, the kernel-only tuning case in Figure 9 shows 2.4.17-DRS significantly outperforming 2.4.17-Auto (by a factor of 5 to 15). DRS at 50ms delay with 8 processes achieves 310Mbps (graph omitted), and at 100ms with 8 process achieves 180Mbps. The performance of 2.2.20-None and 2.4.17-Auto, which do not change with kernel-only tuning, are still limited to the above equations.

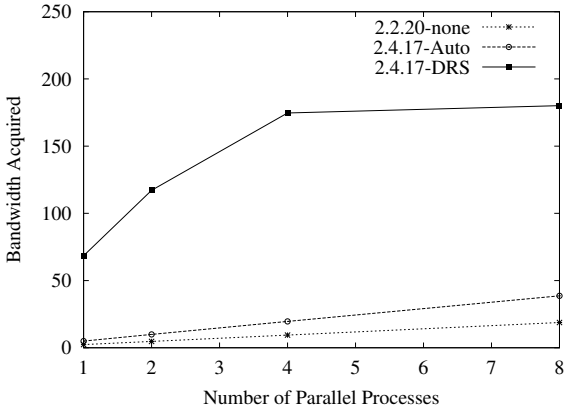

Figure 9. Kernel-Only Tuning, 100ms

Similar to Figure 8, the hand-tuned case in Figure 10 shows 2.4.17-DRS and 2.4.17-Auto performing identically with 2.2.20-None performing slightly worse. Interestingly, at this high delay, the performance difference between DRS and autotuning is insignificant – the factors dominating performance are not buffer sizes but rather standard TCP slow-start, additive increase, and multiplicative decrease behaviors, and the 128MB transfer size is insufficient to differentiate the flows.

With latencies this high, very large (multi-gigabyte, minimum) transfer sizes would be required to more fully utilize the network. It would also help to use a modified version of TCP such as Vegas [2] or one of the plethora of other versions, because a multiplicative decrease can take a ridiculous amount of time to recover on high-delay links.

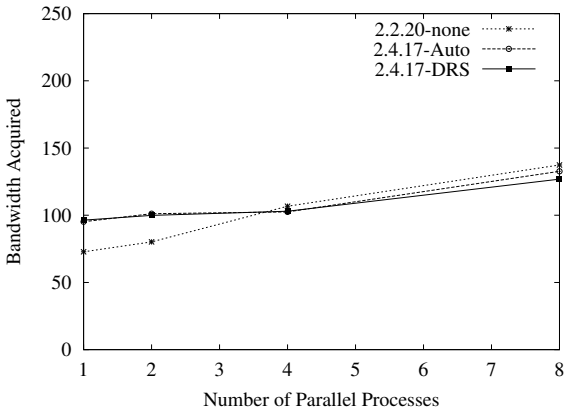

Figure 10. User/Kernel Tuning with Ideal Sizes, 100ms

5. Guidelines on Selecting an Auto-Tuned TCP

This section gives a few practical guidelines for a prospective user of an automatically tuned TCP.

1. You have kernel-modification privileges to the machine. So, you may use a kernel-level solution which will generally provide the best performance. Currently, only NetBSD and Linux implementations exist, so for other operating systems you must either wait or use a user-level solution.

    - If you want to use NetBSD, you must use PSC's tuning.
    - Linux 2.4 autotuning is appropriate for large numbers of small connections, such as web/media servers, or machines where users are willing to tune parallel streams.
    - DRS is appropriate for smaller numbers of large connections, such as FTP or bulk data transfers, or machines where users are not willing to tune parallel streams.

2. You do not have kernel-modification privileges to the machine or are unwilling to make changes, forcing a user-level solution. All user-level solutions perform comparably, so the choice between them is based on features.

    - If all you need is FTP, LANL's DRS FTP or NLANR's Auto-tuned FTP will be the easiest plug-in solutions. Obviously, we are biased in favor of LANL's implementation, which dynamically adjusts the window, over NLANR's implementation, which does not.
    - If you require multiple applications, then the Enable [11] service may fit your needs. This will, however, require source-code level changes to each program you wish to use.

In all cases, initial tuning should be performed to

- Ensure TCP window scaling, timestamps, SACK options are enabled.
- Set the maximum memory available to allocate per connection or for user-level tuning.
- Set ranges for Linux 2.4 autotuning.
- (Optional) Flush caches in between runs so inappropriately set slow-start thresholds are not re-used.

6. Conclusion

We have presented a detailed discussion on the various techniques for automatic TCP buffer tuning, showing the benefits and problems with each approach. We have presented experimental evidence showing the superiority of Dynamic Right-Sizing over simple autotuning as found in Linux 2.4. We have also uncovered several unexpected aspects of the problem (such as the calculated "ideal" buffers performing more poorly than somewhat larger buffers). Finally, the discussion has provided insight into which solutions are appropriate for which circumstances, and why.

References

[1] ANL, CalTech, LBL, SLAC, JF, U. Wisconsin, BNL, FNL, and SDSC. The Particle Physics Data Grid. http://www.cacr.caltech.edu/ppdg/.

[2] L. Brakmo and L. Peterson. TCP Vegas: End to End Congestion Avoidance on a Global Internet. *IEEE Journal on Selected Areas in Communication*, 13(8):1465–1480, October 1995.

[3] W. Feng, I. Foster, S. Hammond, B. Hibbard, C. Kesselman, A. Shoshani, B. Tierney, and D. Williams. Prototyping an Earth System Grid. http://www.scd.ucar.edu/css/esg/.

[4] M. Fisk and W. Feng. Dynamic Adjustment of TCP Window Sizes. Technical Report Los Alamos Unclassified Report (LA-UR) 00-3221, Los Alamos National Laboratory, July 2000. See http://www.lanl.gov/radiant/website/pubs/hptcp/tcpwindow.pdf.

[5] W. E. Johnston, D. Gannon, and B. Nitzberg. Grids as Production Computing Environments: The Engineering Aspects of NASA's Information Power Grid. In *Proceedings of 8th IEEE International Symposium on High-Performance Distributed Computing*, August 1999.

[6] W. Leland, M. Taqqu, W. Willinger, , and D. Wilson. On the Self-Similar Nature of Ethernet Traffic (Extended Version). *IEEE/ACM Transactions on Networking*, 2(1):1–15, February 1994.

[7] G. Navlakha and J. Ferguson. Automatic TCP Window Tuning and Applications. http://dast.nlanr.net/Projects/Autobuf/autotcp.html, April 2001.

[8] Pittsburgh Supercomputing Center. Enabling High-Performance Data Transfers on Hosts. http://www.psc.edu/networking/perf_tune.html.

[9] J. Semke, J. Mahdavi, , and M. Mathis. Automatic TCP Buffer Tuning. *ACM SIGCOMM 1998*, 28(4), October 1998.

[10] B. Tierney. TCP Tuning Guide for Distributed Applications on Wide-Area Networks. In *USENIX & SAGE Login*, http://www-didc.lbl.gov/tcp-wan.html, February 2001.

[11] B. L. Tierney, D. Gunter, J. Lee, and M. Stoufer. Enabling Network-Aware Applications. In *Proceedings of IEEE International Symposium on High Performance Distrubted Computing*, August 2001.

[12] L. Torvalds and The Free Software Community. The Linux Kernel, September 1991. http://www.kernel.org/.

[13] A. Veres and M. Boda. The Chaotic Nature of TCP Congestion Control. In *Proceedings of IEEE Infocom 2000*, March 2000.

[14] E. Weigle and W. Feng. Dynamic Right-Sizing: A Simulation Study. In *Proceedings of IEEE International Conference on Computer Communications and Networks*, 2001. http://public.lanl.gov/ehw/papers/ICCCN-2001-DRS.ps.

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

[16] L. Zhang, S. Shenker, and D. D. Clark. Observations on the Dynamics of a Congestion Control Algorithm: The Effects of Two-Way Traffic. In *Proceedings of ACM SigComm 1991*, September 1991.