Effect of Receive Buffer Size: An OS-based Perspective

Jerome White and David X. Wei
California Institute of Technology

Abstract—**It is generally accepted that optimal TCP receive buffer size is based on network conditions. We argue that processing limitations at the end hosts should also be a consideration. We present initial analysis as to the validity of this argument, noting process-receive buffer dynamics and its affect on throughput. Based on our observations, we offer suggestions as to how better network performance can be achieved via optimized scheduling from the operating system.**

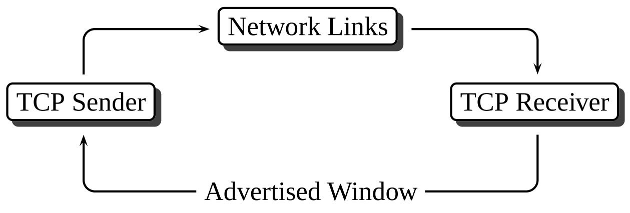


Fig. 1. Established and widely accepted TCP control loop model. Its limitation is that it does not adequately take the sender and receiver operating systems into consideration. In this work, we concentrate our efforts on monitoring the receiver and the effects its operating system can play.

I. Introduction

Receive buffer size plays an important role in any TCP connection. Through the advertised window [1], endpoints calculate the amount of data they can send based primarily on the size of this buffer; thus buffer size can be thought of as a throttle within a connection. To an operating system, receive buffers are the intermediate point between packet reception and packet delivery. Their size, and how they are handled, can have a major impact on TCP performance.

It is widely accepted that optimal receive buffer sizes should be proportional in size to network round trip time (RTT). Namely, that they be at least as large as the bandwidth-delay product (BDP) of a connection. The intuition behind this rule-of-thumb is that there is one RTT worth of delay between the time when an advertised window is sent by the receiver and the time when a new data packet arrives at the receiver. Hence, the advertised window should be at least as large as BDP if we want to fully utilize the available bandwidth. This cycle has commonly been modeled with a control loop, as pictured in Figure 1.

Research in previous years on receive buffer sizes only considers the feedback loop between sender and receiver, as the network is usually the bottleneck and receiver host usually has sufficient CPU power to process the packets. From this standpoint, as long as the rule-of-thumb is satisfied, the effects of receive buffer sizes can be ignored since the receive buffer will not be a bottleneck in the TCP connection.

As networks get faster and router technology advances however, these intermediary aspects that were once bottlenecks become less of a hindrance to data transfer. With the introduction of very high speed networks, 10GE for example, it is becoming more and more common that the host machines are the connection bottlenecks, rather than the network. In these scenarios, data is able to be delivered between sender and receiver with very little loss and at very high throughput, leaving the slowest "link" to be either the sender or the receiver themselves. These types of connections are limited only by their host machines' ability to process data, which includes packet reception.

This trend in network evolution motivates our research. As the bottleneck changes to the end host machines, the interaction between TCP and the operating system becomes very important. Connection and end host resources, such as processor and memory utilization, need to be used properly in order to maximize throughput. Our research focuses on the role of the receive buffer as it pertains to OS and network performance.

Looking at network performance from the vantage of the OS is nothing new, however, doing so in the manner and with the granularity that we do is. We analyze the relationship between throughput and buffer size during the receiving processes active timeslice. Our measurements offer explanation as to why, from the standpoint of the receivers operating system, certain buffer sizes are better than others. Ultimately, receive buffer size should not only be a function of network conditions, such as BDP, but of OS limitations as well—this work is a step in that direction.

II. Methodology

In this study we monitored OS networking on a per process, per cycle basis to identify buffer sizes which produced the largest amount of bandwidth, known as *optimal* buffer sizes. Often, when monitoring system behavior, samples are taken at periodic time intervals; unfortunately, this method was insufficient for the type of analysis we wanted to perform. For this, fine granularity and an emphasis on process scheduling were paramount.

A. Experimental Setup

The goal of our network setup was to isolate the receiving operating system as the only bottleneck within the connection. Our test bed consisted of a single sender and a single receiver separated by zero hops. Both the sender and receiver contained 64-bit AMD processors—the sender having two processors active, while the receiver only had one. Each machine had two Ethernet interfaces: one connecting them to the Internet, the other connecting the two machines directly. The direct connection, used for our tests, used Neterion Xframe II 10GbE network interface cards. The I/O bus was a 64 bit 133MHz PCIX(M1) and both machines ran Linux 2.6.13. When the connection was otherwise idle, ping reported an average round trip time of 0.031 ms. We used the sender to initiate multiple simultaneous connections, allowing us emulate the case where there are multiple upload clients connecting to a server. For brevity and clarity, however, we present only single client results unless multiple client results are significantly different.

Traffic for our tests was generated using *iperf*[1]. We altered the standard 2.0.2 version so that socket measurements could be made (more on this in Subsection II-B). Using *iperf*, we made comparisons not only as buffer size increased, but as the number of parallel flows increased as well. We looked primarily at buffer sizes between 4KB and 60MB. It has been our experience that this range is small enough to observe peaks in bandwidth, yet large enough to see trends that remain consistent through very large ($>$100MB) buffer sizes.

Throughout our experiments, receive buffer size was set via the `/proc` file system. Specifically, we were concerned with the `rmem_default` and `tcp_rmem` variables. The `tcp_moderate_rcvbuf` variable was always turned off. The senders buffer was set in the same manner and always at a size larger than the receive buffer to further help ensure that the sender was not the bottleneck.

B. Implementation of Monitoring System

Socket monitoring works on a per-socket basis. When a socket is created, monitoring of that socket is turned off by default, allowing existing network applications to operate normally. Monitoring is enabled by sending an `SO_MONITOR` option to the `setsockopt` system call. When specified, the file management structure (`files_struct`) of the owning process is altered to contain a pointer to the open socket. These changes allow the kernel to recognize that both the process and socket require monitoring. Socket monitoring is turned off through a similar call to `setsockopt`, and monitoring data is accessed via the `getsockopt` system call. Providing the `SO_MONITOR` option to `getsockopt` copies monitoring data structures from the kernel to the user. For socket monitoring to be most effective, the option was always set immediately after the a call to `accept` or `connect`, and prior to any calls that transmitted data. Moreover, the option was always set by the process performing the socket I/O.

Once a socket was marked for monitoring that socket had its snapshot taken prior to the owning process beginning execution, and again once execution was finished; that is, on process activation and deactivation. We often refer to this period as a “timeslice” or a “cycle.”[2] Again, these socket and scheduler changes were carried out within the Linux 2.6.13 kernel. When set, socket monitoring decreased bandwidth by approximately 3%.

[1]Maintained and distributed by the Distributed Applications Support Team at the National Laboratory for Applied Network Research.

[2]It should be clarified that these terms are meant to describe the time that a process is executing, not necessarily when it is running. In many operating systems, including Linux, a process may be marked as running and even on a run queue, but may not have the the CPU. We are primarily interested in the time that a process is given CPU time, and our use of the terms timeslice, active time, and cycle refer to such.

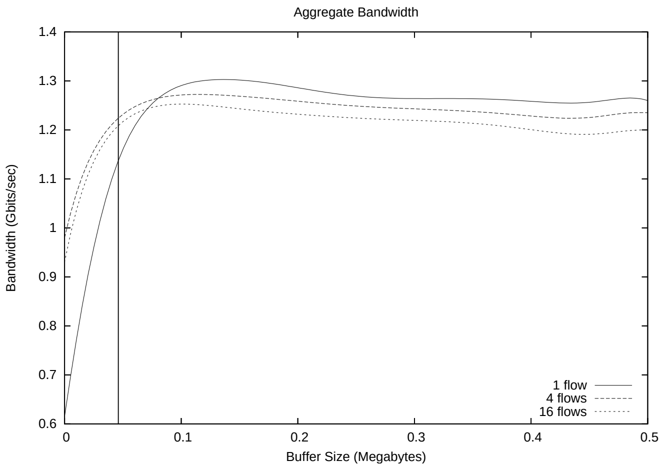


Fig. 2. Resulting bandwidths at varying receive buffer sizes and simultaneous flows. The vertical line near the y-axis represents the BDP based buffer size, which was smaller than optimal. As buffer size increased (even beyond 0.5MB), bandwidth declined.

III. Experimentation and Observation

Traditionally, calculating receive buffer size meant using BDP as a lower bound and total available system resources as an upper bound. This generally leaves a large range of eligible buffer sizes. The relationship between bandwidth and receive buffer size for our network can be seen in Figure 2. We found BDP to underestimate correct buffer size by up to 9%, yet setting the buffer to be significantly larger than this value degraded performance. In general bandwidth declined by an average of 655KB/sec for every 1MB increase in buffer size. Our experiments sought to explain this buffer to bandwidth relationship from the operating systems perspective.

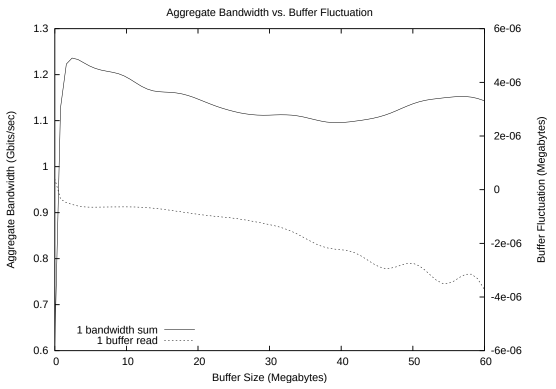
A. Receive Buffer Dynamics

Receive buffer dynamics include statistics pertaining to the receive buffer while its owning process is active. It encompasses two primary statistics: buffer fluctuation and *actual* buffer size. Actual buffer size is the amount of data present on the queue while the process is active. We call this the actual buffer size because it can vary quite largely from what is set in `/proc`.

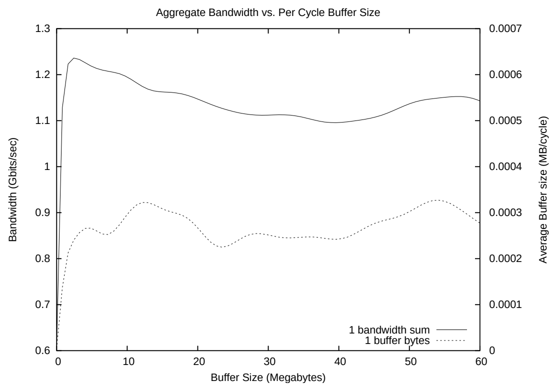
Buffer fluctuation, Figure 3(a), reveals the amount of processing that goes into the receive buffer while a task is active. It is a measurement of the receive queue before and after the receiving processes time slice. When a process becomes active, we note the size of its receive queue. We subtract this value from the size of the buffer at process deactivation, thereby measuring the fluctuation of the buffer while the process was active. Formally:

$$\frac{1}{n}\sum_{i=1}^{n} y_i - x_i \tag{1}$$

where n is the number of timeslices, and x_i and y_i are snapshots of the buffer size before and after, respectively, a



(a) Receive buffer fluctuation during process active time. The negative (sub-0) trend as buffer size increases is a result of more data being taken off of the receive queue during the process active time.



(b) Actual receive buffer size during process active time. This remains relatively stable regardless of set buffer size.

Fig. 3. Buffer size and buffer fluctuation with respect to bandwidth. Bandwidth is plotted against the left y-axis using a solid line; buffer statistics are against the right y-axis using a dotted line.

given timeslice i. A positive value means data was added to the queue during a timeslice, while a negative value corresponds to data being removed.

At very small buffer sizes, including those that were optimal, receive buffer fluctuation remained very close to zero. At these buffer sizes, there is an equilibrium with respect to the amount of data put on the queue and the amount of data taken off during the process active time. It could be the case that with small buffers, there is no data on the queue at process activation and deactivation—that, from Equation 1, $y_i - x_i = 0$. However, our research shows that this is not the case (Figure 3(b)), that data is certainly present on the queue, even at the smallest of buffer sizes. Maintaining this "processability" is important to maintaining optimal bandwidth.

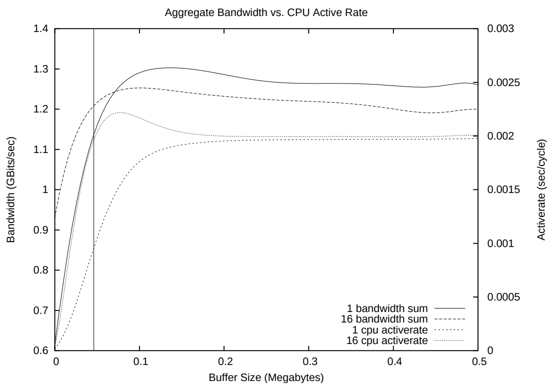
As buffer size increases, we see that both buffer fluctuation, and bandwidth, decline. To blame one on the other is somewhat counterintuitive: that removing data from the queue at a higher frequency results in decreased bandwidth. Our results show that at larger buffer sizes we are able to process our buffer with greater efficiency, but doing so results in sub-optimal bandwidth. It seems that although data is being removed at increased buffer sizes, it is not doing so quickly enough to maintain the bandwidth we see at smaller more optimal buffer sizes. That is, from Equation 1, $x_i > y_i$ for *almost all* $i \in n$. This implies that there is a such thing as too much received data—that although there is room on our receive queue, it is not necessarily a good idea to try and fill it. Too much data is really a disconnect between TCP and the operating system. Although TCP can specify that the receiver has room for more data, it does not take into account whether or not that data can be processed. It is this disconnect that creates the need for CPU load to be taken into account when deciding receiver buffer size.

B. CPU Rate

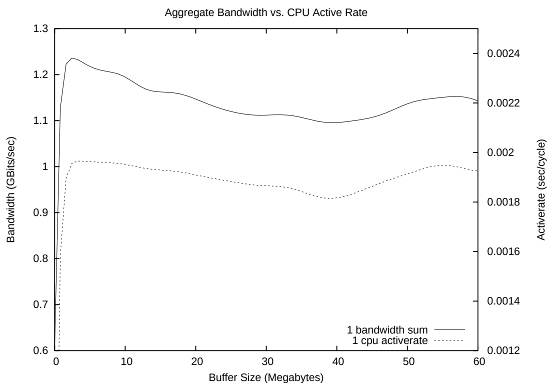
To get a better idea of the relationship between the operating system and the network, we measure the processes active rate, or "CPU rate." CPU rate is the average amount of time a processes active period lasts. Measured in milliseconds per timeslice, CPU rate is the sum of a processes active time divided by the number of timeslices it was allotted. During a network connection, there are several components within the operating system that are competing for CPU resources. Measuring CPU rate allows us to observe the interaction between these components from the standpoint of the scheduler.

The general trend of CPU rate follows the bandwidth curve very closely, as seen in Figure 4(b). Many of the local maxima and minima in the bandwidth curves are mirrored in their corresponding CPU rate curve. A more detailed look at CPU rate with respect to optimal buffer sizes, Figure 4(a), shows just how close the relationship is. When the number of simultaneous connections is greater than 1, CPU rate is maximized at buffer sizes corresponding to BDP. Thus, the CPU rate metric can be a very good indicator of whether or not a connection is attaining optimal bandwidth. By finding the buffer size corresponding to maximized CPU rate, we know we have reached a lower bound on "good" buffer sizes—to go any lower will certainly yield sub-optimal network performance.

1) Process time and cycle count: We can better understand our CPU rate observations by looking at the two components that combine to form the measurement: CPU cycles and CPU time. These components can be seen graphically in Figures 5(a) and 5(b), respectively. As was defined earlier, by "cycle" we mean a processes active period; not necessarily when a process is in a run queue, but when it is actually running. The number of cycles spent in the CPU declines as set buffer size increases. Between buffer sizes of 4KB and 47KB (BDP), we see a decline in the number of cycles by approximately 97% on average, varying with the number of simultaneous flows. From BDP onward, the number of cycles remains relatively stable, averaging a 1.5% decline regardless of how large buffers are set to be. This suggests that at smaller buffers processes are multiplexed at a higher frequency than is required of larger buffers. At smaller buffer sizes, the queue



(a) CPU rate at optimal buffer sizes. Maximums in CPU rate and maximums in bandwidth correspond very closely. CPU rate maxima in fact can be thought of as a lower bound on "good" buffer sizes in much the same way BDP is.



(b) CPU rate through large buffer sizes (general trend). This follows bandwidth very closely.

Fig. 4. Aggregate bandwidth versus process active rate. Active rate can be thought of as the average duration of a processes active period. In each graph, bandwidth is plotted along the left y-axis, while active-rate is along the right y-axis.

must be drained almost immediately to avoid filling up. It is quite possible that the aggregate cost of these context switches hinders application and system performance.

Cycle count turns out to be somewhat the inverse of bandwidth. However, where bandwidth declines after its peak, cycle count does not subsequently increase; rather, it remains constant regardless of buffer size. Why cycle count bottoms out like it does is still not clear, however one possibility is a limitation in the system. More specifically, a (default) setting in the scheduler that prevents processes from receiving too few timeslices. Regardless, by keeping a lower bound on the cycle count as buffer size increases, the system is in turn throttling the bandwidth degradation experienced. It is quite possible that bandwidth would decline at a much more rapid pace than is observed if cycle count was allowed to decline unbounded.

The second factor in our CPU rate calculation is time. We consider time to be the total amount of time a process spends receiving data. This figure mirrors bandwidth very closely, as can be seen in Figure 5(b). Receive time does not display the absolute maximum that bandwidth does, however, as buffer size increases, their declines are very similar. Thus, increased buffer size does lead to less aggregate time spent in the processor, which is to be expected; however, it does not necessarily mean better throughput. This is somewhat counterintuitive: we would expect less time spent in the processor to mean better performance overall. It could be the case that as buffer size increases, the amount of time spent processing incoming packets by the TCP sub-system also increases; something we did not measure in this work. That, or memory management is to blame, as an increase in the memory allotted is the only remaining variable.

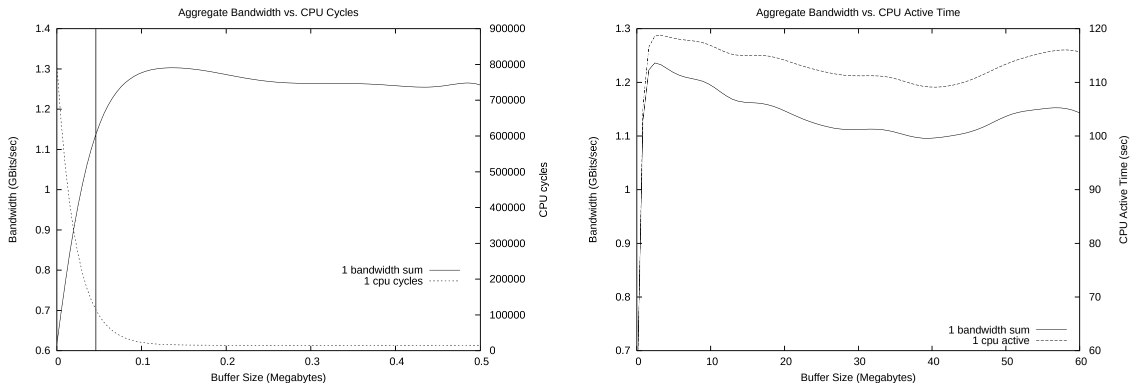
IV. Toward System-based Auto-tuning

In an ideal system, new flows would calculate and offer receive buffer sizes using current and past connection heuristics. For example, keeping a record of typical connection duration and average number of concurrent flows would prove useful. Couple this information with known optimal buffer sizes under these conditions and we have quite a knowledge base with which to work from. Using this information, the system could try and predict optimal buffer sizes required for new connections. How much information to keep and which approximation methods to use to achieve best case throughput are grounds for future work.

Based on our results, the best case for maximizing throughput from a system standpoint is to start by maximizing the CPU rate of receiving processes. In the face of multiple connections, this will give us a lower bound on the correct receive buffer size. Calculating this maximization dynamically, as connections are created, maintained, and terminated, is a seemingly hard problem. An upper bound on optimal buffer sizes can be calculated via buffer fluctuation. When fluctuation begins to sway from zero (within a given threshold: $0 \pm \Delta$) we have reached buffer sizes that are too large. Unfortunately, this method leaves us within a range of good buffer sizes rather than just one. Future work will include narrowing this range.

One very surprising result of our work is the fact that bigger buffers are not necessarily better. It has been observed, in fact, that they can even hinder performance. Thus, determining buffer allocation automatically is not necessarily a recipe for memory consumption. If done properly, system based auto-tuning should recognize this fact, thereby automatically creating its own upper bounded threshold.

Finally, creating optimal receive buffers cannot solely be placed in the hands of the operating system; network parameters are still very necessary. We base the correctness of our system measurements and alterations on a very fundamental network measurement: bandwidth. Thus, without network statistics, we have no way of determining whether or not what we are doing is correct. It should be emphasized that this work is not intended to replace network based auto-tuning, or even static-tuning methods. Rather, system parameters can be used to complement statistics gathered from the network. Our sole intent is to offer an alternative means to achieve best



(a) Average number of timeslices required by a receive process. The vertical line represents the BDP. At very small buffer sizes, processes are context-switched a very high rate.

(b) Aggregate time required by a receive process (CPU active time). This follows bandwidth very closely.

Fig. 5. CPU cycle count and active time as separate components. In each, bandwidth attained is plotted along the left y-axis, using a solid line; the corresponding CPU statistic is plotted along the right y-axis using a dotted line. Each curve represents the aggregate of all clients.

case buffer sizes amidst network conditions that rendered RTT based tuning methods sub-optimal.

V. Related Work

Until recently static values were often used to control the size of TCP buffers. These values were either hand tuned by system administrators or left to the application to control. Finding the optimal size meant either calculating BDP or setting the value as large as system resources would allow. Unfortunately, static-tuning can be prone to resource exhaustion [2] and is not very robust to changing network topologies. Finding ways to tune these buffers automatically has been the goal of recent work.

Dunigan and others propose a daemon that modifies TCP parameters for various open connections [3]. For a given event, the daemon decides whether or not to modify certain parameters of a connection. In order for the daemon to work properly, it must periodically communicate with other remote daemons to get a sense of bandwidth and latency characteristics between end-points. Such added communication not only adds to the traffic on the network, but requires both ends of a connection to be running the daemon and modified kernel.

Semke, Mahdavi, and Mathis offered one of the earliest kernel only solutions to auto-tuning [2]. In their work the receive buffer is set to be as large as possible, limited only by system wide TCP settings. They then use a combination of congestion window and memory usage to determine the optimal send buffer size. Their work inherently assumes that the receiver can consume data at the same rate the network delivers it. We are interested in situations where this is not the case.

In contrast to Semke, Fisk and Feng propose an auto-tuning solution that is throttled by the receiver. They call their solution Dynamic Right-Sizing, or DRS [4]. DRS estimates RTT by measuring the time it takes acknowledgments to be sent and subsequent in-order packets to arrive and uses this measurement to determine the receivers advertised window. Heffner offers an improvement to DRS by calculating RTT with the time stamp value found in most TCP headers [5]. He also decouples the retransmit and send queues, allowing the retransmit queue to grow arbitrarily without limiting the overall window size or performance.

Starting in the 2.4 kernel series, Linux included its own auto-tuning implementation. While not completely documented, the general idea is centered around optimal use of system memory. The kernel not only has the ability to allocate buffers proportional to network properties, but also to re-size buffers based on current memory conditions. We have found kernel auto-tuning to have little affect, with a measured bandwidth ratio between auto-tuning and non auto-tuning be very close to one.

Weigle and Feng offer a comparison of various auto-tuning methods in [6]. They look specifically at manual auto-tuning, Linux 2.4 auto-tuning, and DRS. They also talk about other auto-tuning methods and categorize each, something that had not been done to that point.

The common thread amongst these auto-tuning methods is their focus on network conditions to calculate optimal buffer sizes. Efforts such as this are both adequate and efficient in low speed networks. Our concern, rather, is on very high speed networks, for which BDP calculations are not enough when searching for optimal receive buffer sizes. In this environment, the operating system should be considered.

Work that does have a heavier emphasis on the OS is that of Feng and others. Their Rate Adaptive Protocol [7], [8] allows conditions on receiver to be expressed to the sender via explicit message passing on top of TCP and a simultaneous UDP channel. We are not after a new protocol per-say, moreso a new way for an unaltered TCP to operate. Their method of predicting congestion is also a bit different from ours. Instead of making predictions based on the dynamics of the receive queue, they are trying to predict it based on whether or not the receiving process is active.

VI. Conclusion and Future Work

We have looked at the role in which receive buffer size plays on network performance. We have not only looked at how this parameter affects throughput, but analyzed its relationship with the operating system as well. In doing so we considered buffer dynamics and CPU scheduling. As compared to optimal buffer sizes, we conclude that small buffers require a significant amount of context switches to process, while large buffers are not given enough time per timeslice to be as effective.

We plan to extend our work in two directions. First we would like to gain an even better understanding of how all of the factors we have observed interact with one another. This requires an even more thorough accounting of network processing. We would like to continue to study receive dynamics, and even include similar studies of send dynamics as well. To round out our observations, monitoring of the TCP subsystem is required. This will help to strengthen our existing conclusions and verify many of the hypothesis this work has generated.

We would also like to experiment with different scheduling policies in an effort to maximize CPU rate regardless of the buffer size. We are confident that if this can be accomplished, throughput will benefit. This leads to our overall goal of implementing our system-based auto-tuning proposal into a real system.

References

[1] J. Postel, "RFC 793: transmission control protocol," Sep 1981.

[2] J. Semke, J. Mahdavi, and M. Mathis, "Automatic tcp buffer tuning," in *SIGCOMM '98: Proceedings of the ACM SIGCOMM '98 conference on Applications, technologies, architectures, and protocols for computer communication*. New York, NY, USA: ACM Press, 1998, pp. 315–323.

[3] T. Dunigan, M. Mathis, and B. Tierney, "A tcp tuning daemon," in *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*. Los Alamitos, CA, USA: IEEE Computer Society Press, 2002, pp. 1–16.

[4] M. Fisk and W. chun Feng, "Dynamic right-sizing in TCP," in *Proceedings of the Los Alamos Computer Science Institute Symposium*, Oct 2001.

[5] J. W. Heffner, "High bandwidth TCP queuing," Jul 2002, senior Thesis, Carnegie Mellon University.

[6] E. Weigle and W. chun Feng, "A comparison of tcp automatic tuning techniques for distributed computing," in *HPDC '02: Proceedings of the 11 th IEEE International Symposium on High Performance Distributed Computing HPDC-11 2002 (HPDC'02)*. Washington, DC, USA: IEEE Computer Society, 2002, p. 265.

[7] A. Banerjee, W. chun Feng, B. Mukherjee, and D. Ghosal, "Rapid: an end-system aware protocol for intelligent data transfer over lambda grids," in *Proceedings of the 20th International Parallel and Distributed Processing Symposium (IPDPS)*, April 2006, pp. 25–29.

[8] P. Datta, W. chun Feng, and S. Sharma, "Grid networks and portals—end-system aware, rate-adaptive protocol for network transport in lambdagrid environments," in *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. New York, NY, USA: ACM Press, 2006, p. 112.