Using NetLogger and Web100 for TCP Analysis

Brian L. Tierney
Lawrence Berkeley National Laboratory

1.0 Introduction

Scaling TCP to very large bandwidth-delay product networks has proven to be very challenging. When diagnosing TCP behavior in these environments, we have found that monitoring various TCP parameters and visually correlating them with host and application information is a very effective analysis technique. In this paper we show how this technique can be implemented with a combination of the Web100 TCP instrumentation capabilities and the NetLogger analysis tools.

Web100 is an implementation of an IETF Internet Draft TCP MIB [6] which allows for low-level instrumentation of the TCP stack within the Linux operating system. The NetLogger Toolkit is a set of tools that provide the ability to visually correlate monitoring data from a variety of sources, such as hosts, operating systems, and applications. The Net100 project has enhanced Web100 with a monitoring and tuning daemon that allows the monitoring of any TCP socket. In this paper, we describe all these components, and then provide several examples of how they can be used together to analyze TCP behavior. *The goal of this paper is not to actually analyze TCP, but rather to show how the combination of web100 and NetLogger create a powerful analysis technique.*

2.0 NetLogger Toolkit

Since 1994 researchers at Lawrence Berkeley National Lab have been developing a toolkit for instrumenting distributed applications called NetLogger [8]. Using NetLogger, distributed application components are modified to produce timestamped traces of interesting events at all critical points of the distributed system. Events from each component are correlated, allowing one to characterize the performance of all aspects of the system and network in detail.

All the tools in the NetLogger Toolkit share a common monitoring event format, and assume the existence of accurate and synchronized system clocks. The NetLogger Toolkit itself consists of four components: an API and library of functions to simplify the generation of application-level event logs, a service to collect and merge monitoring from multiple remote sources, a monitoring event archive system, and a tool for visualization and analysis of the log files. In order to instrument an application to produce event logs, the application developer inserts calls to the NetLogger API at all critical points in the code, then links the application with the NetLogger library.

NetLogger events can be formatted as an easy to read and parse ASCII format, or as a self-describing binary format. The NetLogger binary wire format is very efficient, capable of handling over 600,000 events per second [3]. NetLogger also includes a remote activation mechanism, and reliability support.

The NetLogger Reliability API provides fault-tolerance features that are essential in Grid environments. For distributed monitoring, a particular challenge is that temporary failures of the network between the component being monitored and the component collecting the monitoring data are relatively common, especially when several sites are involved. The NetLogger API included the ability to specify a backup , i.e. fail-over, destination to use. This may be any valid NetLogger destination, but typically is a file on local disk. If the primary destination fails, all data is transparently logged to the backup destination. Periodically, the library checks whether the original destination has come back up . If so, the library reconnects and, if the backup destination was a file, sends over all the data logged during the failure.

The NetLogger Toolkit also includes a data analysis component. One of the major contributions of NetLogger was the concept of linking a set of events together and representing them visually as a lifeline , as shown in Figure 1. Visualizing event traces in this manner makes it easy to determine where the most time is spent.

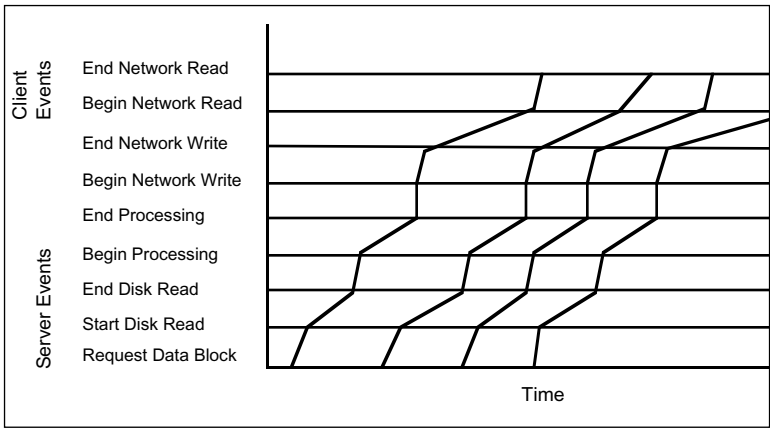

Figure 1: NetLogger Lifelines

The NetLogger Visualization tool, *NLV*, provides an interactive graphical representation of system-level and application-level events. NetLogger s ability to correlate detailed application instrumentation data with host and network monitoring data has proven to be a very useful tuning and debugging technique for distributed application developers.

As an example, see Figure 2. For this figure, we used the NetLogger visualization tool, *NLV*, to correlate client and server instrumentation data with CPU and TCP retransmission monitoring data. The events being monitored are shown on the y-axis, and time is on the x-axis. From bottom to top, one can see CPU utilization events (lines 1-3), application events, and TCP retransmit events all on the same graph. Each semi-vertical line represents the life of one block of data as it moves through the application. The gap in the middle of the graph, where only one set of header and data blocks are transferred in three seconds, correlates exactly with a set of TCP retransmit events. Thus, this plot makes it easy to see that the pause in the transfer is due to TCP retransmission errors on the network. The *NLV* interface allows the user to play, pause, step forward and backward, zoom in and out, select and unselect groups of data, and so on.

3.0 Web100

The Web100 project [9] is an NSF funded collaboration between the Pittsburgh Supercomputing Center (PSC), the National Center for Atmospheric Research (NCAR) and The National Center for Supercomputing Applications (NCSA). The Web100 vision is to enable users running ordinary applications on typical workstations to either saturate a workstation bottleneck or completely fill a network link. In other words, the goal is to make it easy for ordinary users to tune TCP to get the most out of their available resources.

To achieve this goal, Web100 exposes the statistics inside the TCP stack itself through an enhanced standard Management Information Base (MIB) for TCP [6]. This MIB uses TCP s ideal vantage point to provide statistics for diagnosing performance problems in both the network and the application. If a network-based application is performing poorly, TCP information from Web100 allows us to determine if the bottleneck is in the sender, the receiver, or the network itself. If the bottleneck is in the network, TCP can provide specific information about its nature.

The current Web100 implementation is based on extensions and modifications to the Linux 2.4 kernel. Web100 variables are contained in a data structure attached to the kernel s socket data structure. An application reads and sets the Web100 variables using the Linux */proc* interface using an API provided in the Web100 distribution. TCP connection start and end events are provided to an application (e.g., a tuning daemon) through the *netlink* service.

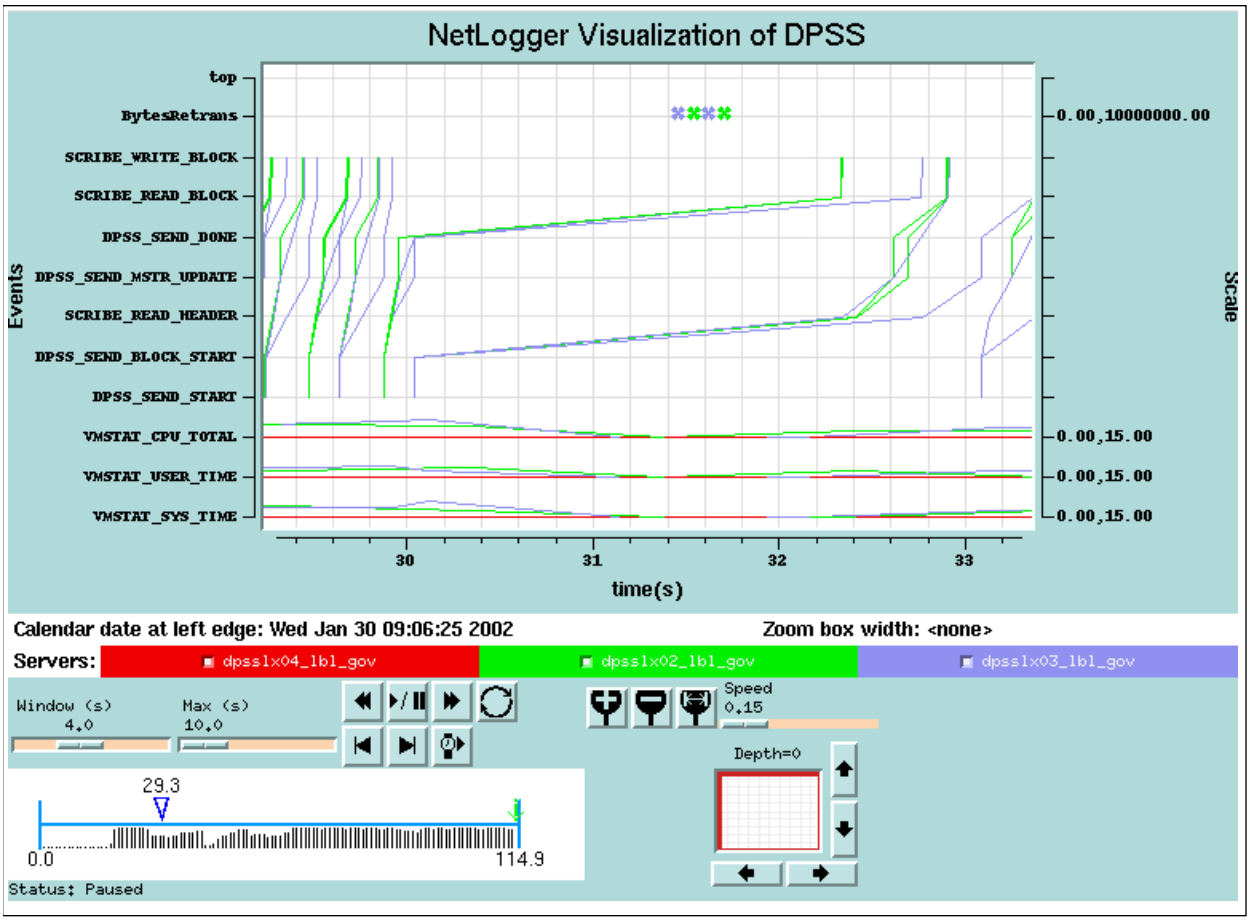

Figure 2: NetLogger Visualization

4.0 Work-Around Daemon (WAD)

We have developed a monitoring and tuning daemon for the Web100 kernel called the Work-Around Daemon, or WAD. The name comes from the WAD s original goal, which was to use Web100 tuning mechanisms to work around problems with TCP flows in a particular network or application. For more information on the WAD tuning options, see [1]. In this paper we are using the WAD in a read-only mode for monitoring TCP, not for TCP tuning.

The WAD first detects a TCP connection by listening on the Web100 *netlink socket* -- a communication mechanism used for kernel notifications to user space. The daemon then consults a configuration file that specifies which flows (source, source port, destination, destination port) are of interest. When the WAD is used for tuning a connection, the configuration entry for a tunable flow also includes a set of tuning parameters such as maximum ssthresh, AIMD parameters, reordering threshold, and so on.

In addition to TCP tuning, the WAD can monitor any Web100 variable for any TCP flow. For example, the WAD can measure the congestion window, packet retransmissions, timeouts, and smoothed RTT times of any socket directly from Web100 variables. The WAD can also be configured to generate *derived events* from combinations of Web100 variables. For example, one could generate average and instantaneous bandwidth as follows:

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

```
LastIntervalBW = (Delta_DataBytesOut*8)/ (Delta_SndLimTimeRwin +
    Delta_SndLimTimeCwnd + Delta_SndLimTimeSender)
```

The WAD can write any subset of the derived and raw variable values as NetLogger events, and send these events to the NetLogger visualization tool, *NLV*, for visual analysis of TCP streams.

5.0 Results

Figure 3 shows a graph of several Web100 variables along with CPU utilization and instrumented *iperf* events. Web100 counters are collected every 0.3 seconds using the WAD. CPU utilization data is collected every second (cpu.utilization.user and cpu.utilization.sys), and *iperf* has been instrumented with NetLogger to generate monitoring events before and after all I/O operations (StartRead/EndRead in the iperf server, and StartWrite/EndWrite in the iperf client). The numbers on the right side of the plot are the range of values for that monitoring event, and the numbers without units are a count of the number of times an event occured.

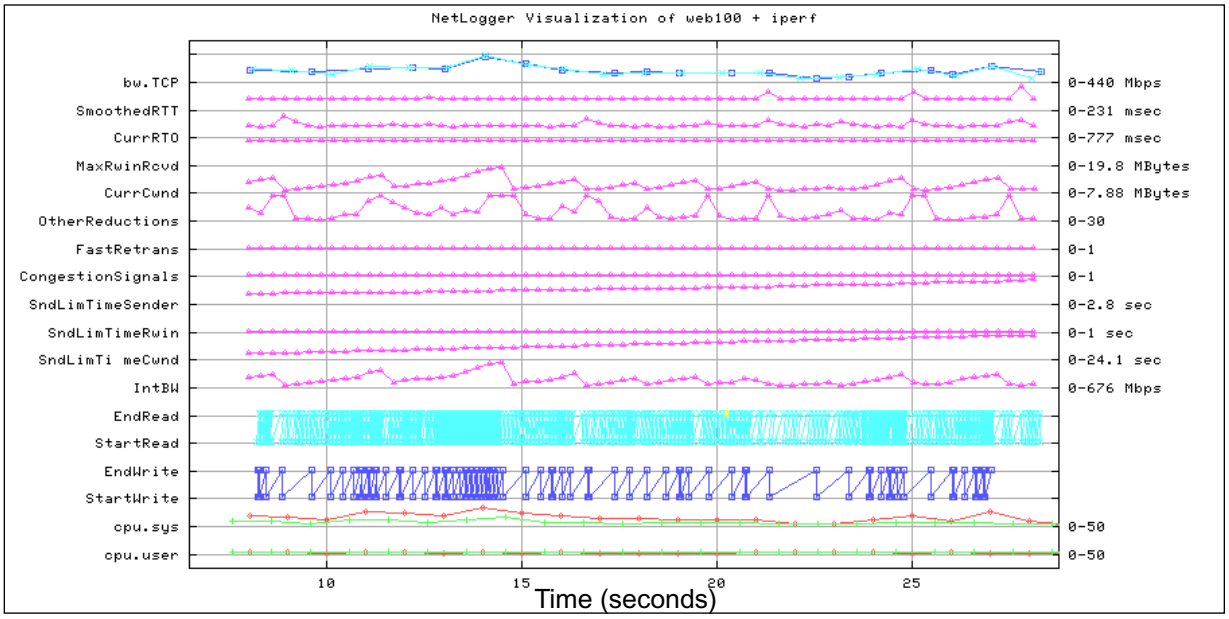

Figure 3: TCP analysis correlated with CPU and application monitoring

There is an immense amount of information in this plot, which we will attempt to explain. From bottom to top:

- CPU user and system load information: the two colors represent the sender and receiver host. Note that system CPU increases when CurrCWND is large
- StartWrite and EndWrite are from the *iperf* client, and represent the time to write a 512 KByte block from user space to kernel space
- StartRead and EndRead are from the *iperf* server, and represent the time to read a 512 KByte block from kernel space to user space. Note that the client writes are much more bursty than the server reads.
- IntBW: a WAD computed value of the bandwidth achieved since the last measurement (0.3 seconds)
- SndLimTimeCwnd, SndLimTimeRwin, and SndLimTimeSender: These are web100 *sender congestion triage* variables that help determine whether the sender, receiver, or the network is the bottleneck
- CongestionSignals: Web100 sum of all types of congestion events, including Fast Retransmit, ECN, and timeouts.

- OtherReductions: Other than during congestion, there are a two other situations where the Linux 2.4 TCP implementation reduces the congestion window. The first is that Linux implements RFC2861 (TCP Congestion Window Validation) [5], which reduces CWND after an extended idle period. The other case is that Linux calls a routine called *tcp_moderate_cwnd*, which reduces CWND whenever it thinks there are more packets in flight than there should be based on CWND. This algorithm appears to be specific to Linux, and based on no known IETF document. Note that since there are no other congestion signals recorded for this run, OtherReductions are clearly the cause of CWND being reduced.
- CurCwnd: current TCP congestion windowMaxRwinRcvd: This is the maximum TCP window size that the receiver is telling the sender it can use.
- CurRTO: current value of the TCP RTO (Retransmission Timer) measurement, used to determine when a TCP time-out should occur.
- SmoothedRTT: TCP s internal notion of the round-trip time.
- bw.TCP: average bandwidth since the start of the test, as reported by *iperf*.

Note that the WAD records both value (current value) and delta (difference from the previous value) for each event. With NLV, you can specify which you wish to graph. For some events (e.g.: CongestionSignals) it is better to graph the deltas, but with other events (e.g.: CurCWND) you want to see a trace of the current values.

The careful observer may have noticed that CWND appears to recover from congestion faster than standard TCP would allow. This is because for this test we were using an implementation [1] of Sally Floyd s High-Speed TCP algorithm [2], which more aggressively recovers from congestion events when the congestion window is large.

In summary, from Figure 3 one can see that the maximum bandwidth peaked around 660 Mbps (IntBW), but that the average was only around 200 Mbps. This was due to the fact that CWND was continuously reduced by whatever was causing the OtherReductions to occur, likely the *tcp_moderate_cwnd* routine. The implementors of web100 have said that the next version will have separate counters for the two types of OtherReductions, which will allow us to know with certainty which case is happening here.

Another Example

Next we look at some data that demonstrates an interesting bug in the Linux TCP stack. Figure 4 shows a rather serious bug in Linux that occurs when the TCP buffers are too large. The path in this graph requires 10 MB TCP buffers to fill the network, but here the user set the TCP buffer to 20 MB. Under these circumstances, something happens when the TCP congestion window (CWND) gets too large. As can be seen in the figure, after about one minute the system CPU utilization rises to 100% and the throughput drops to almost zero. In addition, there is a congestion event at this time.

Figure 5 shows a zoomed-in view of the same graph and an *NLV* annotation window, showing the raw data. In this view, we can see that after the congestion event CWND is clamped at 4344 bytes (see VAL=4344.0 in the annotation window), where it will stay for the remainder of the session. Also note that the WAD stopped generating events for about 1.2 seconds at this time, probably because the CPU was too busy servicing interrupts. Several groups of people are now aware of this bug, so the bug will likely be fixed by the time you read this.

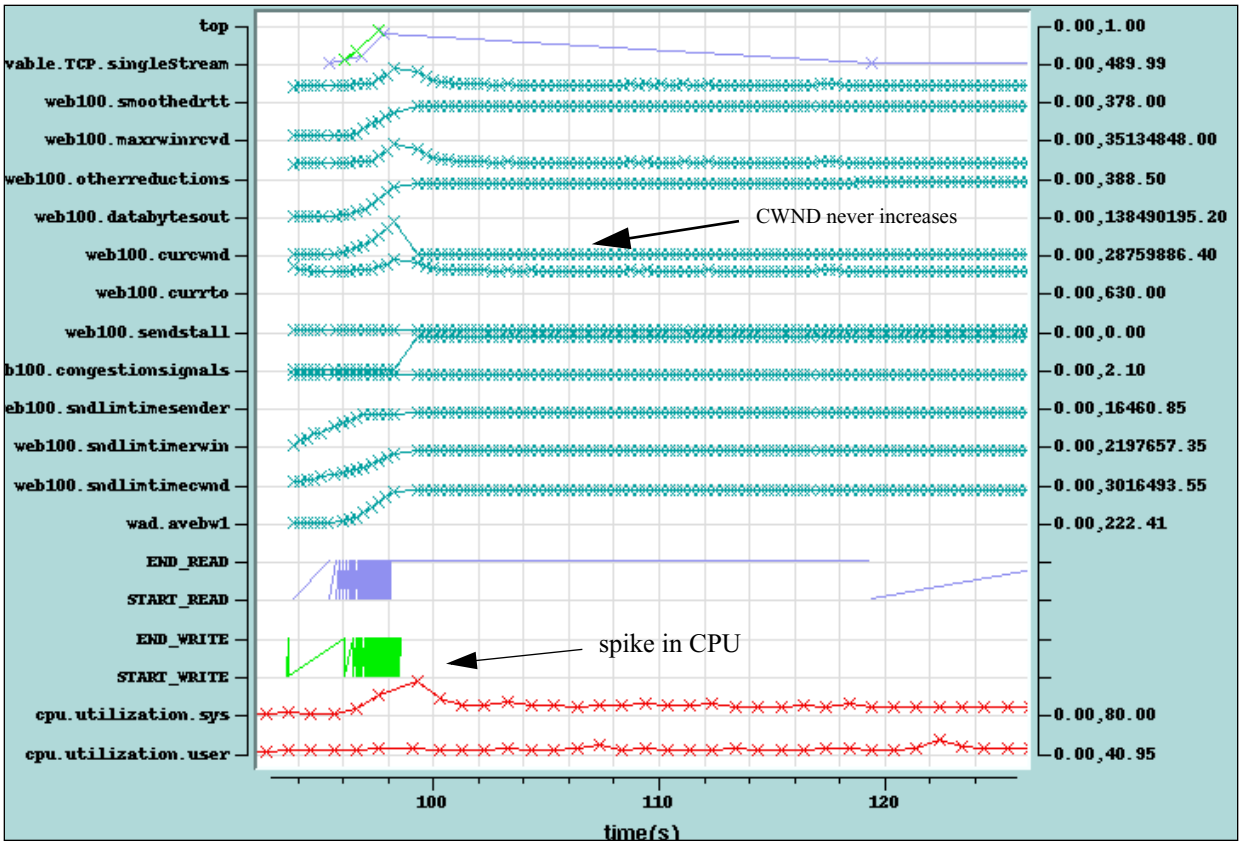

Figure 4: Web100 view of Linux TCP bug

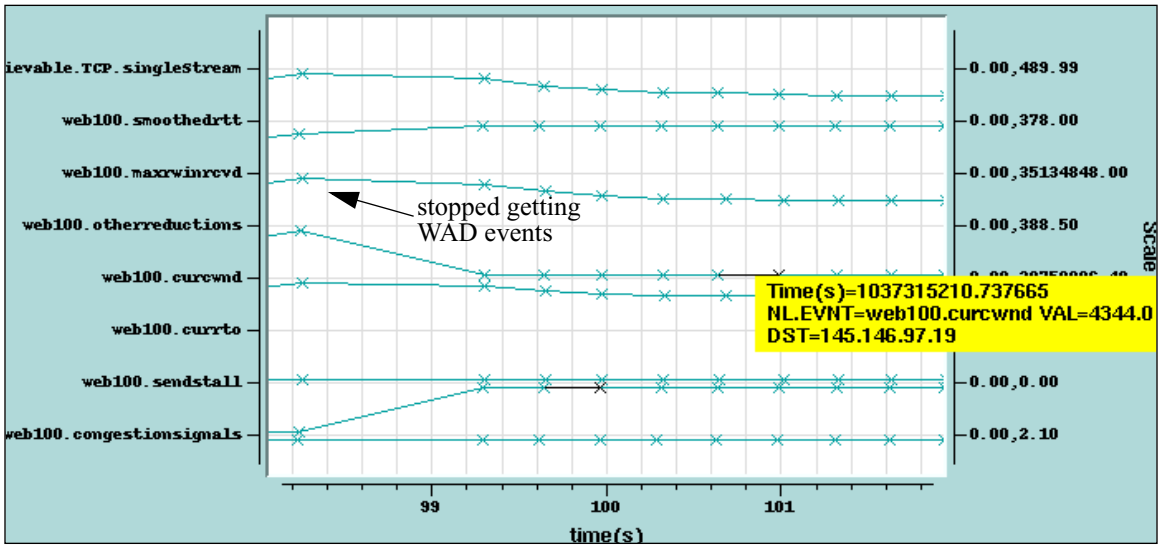

Figure 5: Zoomed View showing clamped CWND

6.0 Related Work

The Web100 *gutil* program [4] provides a nice mechanism for exploring web100 variables for a selected TCP stream. Tcptrace [7] is very good for visually exploring tcpdump files. However, as the results above show, the ability to correlate this information with CPU monitoring is very helpful. Additionally, the flexibility of the *NLV* tool makes it easy to incorporate new kinds of monitoring data.

7.0 Conclusion

The ability to visually analyze TCP flows and correlate their behavior with application and CPU monitoring has proven to be very effective mechanism for understanding and debugging TCP over very large bandwidth-delay product networks. In particular, the ability to track the size of the TCP congestion window over time, along with the various factors that influence CWND such as *congestionSignals* or *otherReductions* is extremely useful for understanding TCP s behavior over high-speed links. The combination of Web100 and NetLogger provide everything needed to perform this type of analysis.

8.0 Acknowledgments

This work was supported by the Director, Office of Science. Office of Advanced Scientific Computing Research. Mathematical, Information, and Computational Sciences Division under U.S. Department of Energy Contract No. DE-AC03-76SF00098. This is report no. LBNL-51776.

9.0 References

[1] Dunigan, T., M. Mathis and B. Tierney, *A TCP Tuning Daemon*, Proceeding of IEEE Supercomputing 2002 Conference, Nov. 2002, LBNL-51022.

[2] Floyd, Sally, *HighSpeed TCP for Large Congestion Windows*, IETF Internet Draft, http://www.ietf.org/internet-drafts/draft-floyd-tcp-highspeed-01.txt

[3] Gunter, D., et. al. *Dynamic Monitoring of High-Performance Distributed Applications*. in 11th IEEE Symposium on High Performance Distributed Computing. 2002.

[4] *gutil*, http://www.web100.org/docs/man/gutil.html

[5] Handley, M., J. Padhye, S. Floyd, *TCP Congestion Window Validation,* June 2000, http://www.ietf.org/rfc/rfc2861.txt

[6] M. Mathis, M., R. Reddy, J. Heffner, and J. Saperia. *TCP Extended Statistics MIB*. IETF draft, work in progress, November 2002. URL: http://www.ietf.org/internet-drafts/draft-ietf-tsvwg-tcp-mib-extension-02.txt.

[7] *tcptrace*: http://www.tcptrace.org/

[8] Tierney, B., et al. *The NetLogger Methodology for High Performance Distributed Systems Performance Analysis*. in Proc. 7th IEEE Symp. on High Performance Distributed Computing. 1998.

[9] *web100* project: http://www.web100.org/