NetLogger: A Toolkit for Distributed System Performance Tuning and Debugging

Brian Tierney, Dan Gunter

Lawrence Berkeley National Laboratory

1 Cyclotron Rd., MS 50B-2239

Berkeley, CA 94720

Abstract

Developers and users of high-performance distributed systems often observe performance problems such as unexpectedly low throughput or high latency. Determining the source of the performance problems requires detailed end-to-end instrumentation of all components, including the applications, operating systems, hosts, and networks. In this paper we describe a methodology that enables the real-time diagnosis of performance problems in complex high-performance distributed systems. The methodology includes tools for generating timestamped event logs that can be used to provide detailed end-to-end application and system level monitoring; and tools for visualizing the log data and real-time state of the distributed system. This methodology, called NetLogger, has proven invaluable for diagnosing problems in networks and in distributed systems code. This approach is novel in that it combines network, host, and application-level monitoring, providing a complete view of the entire system. NetLogger is designed to be extremely light-weight, and includes a mechanism for reliably collecting monitoring events from multiple distributed locations. This technical report summarizes most important points of several previous papers on NetLogger, and is meant to be used as a general overview.

keywords: distributed systems performance analysis and debugging

1.0 Introduction

The performance characteristics of distributed applications are complex, rife with "soft failures" in which the application produces correct results but has much lower throughput or higher latency than expected. Because of the complex interactions between multiple components in the system, the cause of the performance problems is often elusive. Bottlenecks can occur in any component along the data's path: applications, operating systems, device drivers, network adapters, and network components such as switches and routers. Sometimes bottlenecks involve interactions between components, sometimes they are due to unrelated network activity impacting the distributed system.

While post-hoc diagnosis of performance problems is valuable for systemic problems, for operational problems users will have already suffered through a period of degraded performance. The ability to recognize operational problems enables elements of the distributed system to use this information to adapt to operational conditions, minimizing the impact on users.

We have developed a methodology, known as NetLogger (short for *Net*worked Application *Logger*), for monitoring, under realistic operating conditions, the behavior of all the elements of the application-to-application communication path in order to determine exactly what is happening within a complex system.

Distributed application components, as well as some operating system components, are modified to perform precision timestamping and logging of "interesting" events, at every critical point in the distributed system. The events are correlated with the system's behavior in order to characterize the performance of all aspects of the system and network in detail during actual operation. The monitoring is designed to facilitate identification of bottlenecks, performance tuning, and network performance research. It also allows accurate measurement of throughput and

latency characteristics for distributed application codes. NetLogger includes a tool for analyzing monitoring events based on visualization of the timestamp correlated event data.

NetLogger has demonstrated its usefulness in a variety of contexts, but most frequently in loosely-coupled client-server architectures. We began developing NetLogger in 1994, and in previous work we have shown that detailed application monitoring is vital for both performance analysis and application debugging [3][17][18][19]. This paper summarizes all NetLogger components, provides details on some recent enhancements, and provides some case studies.

NetLogger has also proven to be a very useful tool for debugging multi-threaded programs, allowing the developer to easily visualize interactions between threaded components, verify that certain tasks are indeed being executed concurrently, and see when threads are blocked waiting for some event.

In section 3 of this paper we present an overview of the main NetLogger Toolkit components. In section 4 we summarize some recent NetLogger enhancements, including a highly efficient binary format that reduces NetLogger overhead; a reliability mechanism that will send NetLogger data to a secondary location if the primary location is unavailable; and an activation mechanism that allows one to start, stop, or change the level of monitoring of a running process. In section 5 we give a couple examples of how NetLogger was used to debug and tune specific applications.

2.0 Related Work

There are a number of systems that address application monitoring. *log4j*, part of the Apache Project [9], has produced a flexible library for Java application logging. However, the performance of *log4j* is far lower than is necessary for detailed monitoring, as is shown section 4.1 below.

Another instrumentation package is the Open Group's Enterprise Management Forum's [14] Application Response Measurement (ARM) API, which defines function calls that can be used to instrument an application for transaction monitoring. ARM provides a way to monitor business transactions, by embedding simple calls in the software that can be captured by an agent supporting the ARM API. ARM was originally made available in 1996 by Hewlett-Packard and Tivoli.

Other related work includes general purpose event handling systems, such as the CORBA event service [4], the JINI distributed event service [8], and the ECHO Event Service [6]. Of all of these, only ECHO is specifically concerned with performance. Our message binary format, described below, is similar in size and efficiency to PBIO [5], which is used in the ECHO system, but is simpler and more dynamic.

3.0 NetLogger Toolkit Component Overview

At Lawrence Berkeley National Lab we have developed the *NetLogger Toolkit*, which is designed to monitor, under actual operating conditions, the behavior of all the elements of the application-to-application communication path in order to determine exactly where time is spent within a complex system. Using NetLogger, distributed application components are modified to produce timestamped logs of "interesting" events at all the critical points of the distributed system. Events from each component are correlated, which allows one to characterize the performance of all aspects of the system and network in detail.

The NetLogger Toolkit itself consists of four components: an API and library of functions to simplify the generation of application-level event logs, a set of tools for collecting and sorting log files, a set of host and network monitoring tools, 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 the critical points in the code, then links the application with the NetLogger library. All the tools in the NetLogger Toolkit share a common log format, and assume the existence of accurate and synchronized system clocks. We have found that for this type of distributed systems analysis, clock synchronization of around one millisecond is required, and that the NTP [13] tools that ship with most Unix systems (e.g.: *ntpd*) can often provide this level of synchronization.

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.

Figure 1 shows sample *nlv* results, using a remote data copy application. 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, 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.

In this section we give an overview of the major components of the NetLogger Toolkit. In the following section we go into more detail on some recent NetLogger enhancements.

3.1 Common Log Format

NetLogger includes options for both an ASCII and binary message format. For the ASCII format, NetLogger uses the IETF-proposed Universal Logger Message format (ULM)[1] for the logging and exchange of messages. Use of a common format that is plain ASCII text and easy to parse simplifies the processing of potentially huge amounts of log data, and makes it easier for third-party tools to gain access to the data. The NetLogger binary format, described below, is much faster, but harder for third-party tools to use. NetLogger includes tools for converting between the ASCII and binary formats.

Figure 1: Sample NetLogger Results

The ULM format consists of a whitespace-separated list of "field=value" pairs. ULM required fields are DATE, HOST, PROG, and LVL; these can be followed by any number of user-defined fields. NetLogger adds the field NL.EVNT, whose value is a unique identifier for the event being logged. The value for the DATE field has six digits past the decimal point, allowing for microsecond precision in the timestamp. Here is a sample NetLogger ULM event:

```
DATE=20000330112320.957943 HOST=dpss1.lbl.gov PROG=testProg
LVL=Usage NL.EVNT=WriteData SEND.SZ=49332
```

This says that the program testprog on host dpss1.lbl.gov performed a WriteData event with a send size of 49,322 on March 30, 2000 at 11:23 (and some seconds) in the morning.

The user-defined events at the end of the log entry can be used to record any descriptive value or string that relates to the event such as message sizes, non-fatal exceptions, counter values, and so on.

3.2 NetLogger API

In order to instrument an application to produce event logs, the application developer inserts calls to the NetLogger API at all the critical points in the code, then links the application with the NetLogger library. This facility is currently available in several languages: Java, C, C++, Python, and Perl. The API has been kept as simple as possible, while still providing automatic timestamping of events and logging to either memory, a local file, syslog, a remote host. Logging to memory is available in the form of a buffer which can be explicitly flushed to one of the other locations (file, host, or syslog), or automatically flushed when the buffer is full. Sample Python NetLogger API usage is shown in Figure 2. As is shown in this example, "interesting" events in the code (such as I/O or processing) are typically wrapped with NetLogger write() calls that generate user-defined start and end instrumentation events.

```
log = NetLogger("program_name","x-netlog://loghost.lbl.gov")
done = 0
while not done:
    log.write("EVENT_START","TEST.SIZE=%d",size)
    # perform the task to be monitored
    done = do_something(data,size)
    log.write("EVENT_END")
```

Figure 2: Sample NetLogger Usage
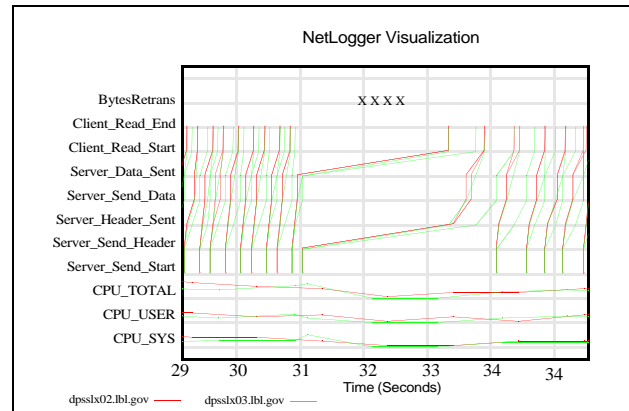
3.3 Event log collection

NetLogger facilitates the collection of event logs from distributed applications by providing automatic logging to a single host and port. A server daemon, called *netlogd*, receives the log entries and writes them into a file on the local disk. Thus, applications can transparently log events in real-time to a single destination over the wide-area network.

3.4 Host and Network monitoring Tools

The NetLogger Toolkit includes wrappers for several standard Unix system and network monitoring tools. These wrapper take the output of the tool and generate NetLogger formatted monitoring events. Current wrappers include *vmstat* (CPU and memory monitoring), *netstat* (network interface monitoring), *iostat* (disk monitoring), and *snmpget* (remote access to a variety of host and network monitoring information).

3.5 Event log visualization and analysis

We have found exploratory, visual analysis of the log event data to be the most useful means of determining the causes of performance anomalies. The NetLogger Visualization tool, *nlv*, has been developed to provide a flexible and interactive graphical representation of system-level and application-level events. *nlv* uses three types of graph primitives to represent different events: points, load lines, and lifelines.

The most important of these primitives is the *lifeline*, which represents the "life" of an object (datum or computation) as it travels through a distributed system, as shown in Figure 3. With time shown on the x-axis, and

Figure 3: NetLogger *lifelines*

ordered events shown on the y-axis, the slope of the lifeline gives a clear visual indication of latencies in the distributed system. *nlv* generates lifelines by correlating monitoring events based on their timestamp and a user-defined *event ID*. The event ID is any unique identifier that is added to all monitoring events in the lifeline. For example, a loop counter, a block ID, or the PID could all be used for the event ID, depending on the application being monitored.
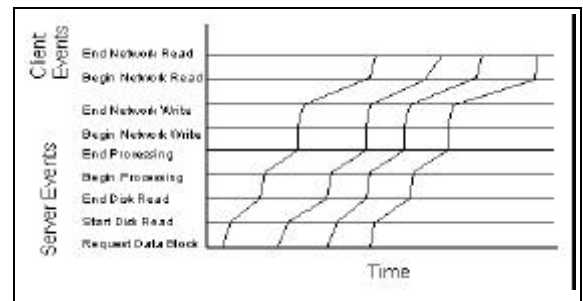
The other two graph primitives are the loadline and the point. The loadline connects a series of scaled values into a continuous segmented curve, and is most often used for representing changes in system resources such as CPU load or free memory. The point data type is used to graph single occurrences of events, often error or warning conditions such as TCP retransmits. In addition, the point datatype can be scaled to a value, producing a scatterplot.

Monitoring event from a variety of sensors, including application, host, and network sensors, are correlated based on their timestamps and can be viewed together using *nlv*. In order to assist correlation of observed system performance with logged events, *nlv* has been designed to allow real-time visualization of the event data as well as historical browsing and playback of interesting time periods. In the real-time mode, the graph scrolls along the time axis (x-axis) in real time, showing data as it arrives in the event log. In historical mode, the user can change the position in the log file, change the scale of the graph, zoom in and out interactively, choose a subset of events to look at, and so on.

4.0 Recent NetLogger Enhancements

We first developed NetLogger in 1994, and first published it in 1996 [19]. Over several years experience, we discovered a few missing features that we needed to add to make NetLogger more useful in a distributed environment, and to be suited for instrumenting middleware as well as client and server code.

In this section we summarize some recent NetLogger enhancements, including a highly efficient binary format that reduces NetLogger overhead; a reliability mechanism that will send NetLogger data to a secondary location of the primary location is unavailable; and an activation mechanism that allows one to start, stop, or change the level of monitoring of a running process.

4.1 NetLogger Binary Log Format

In general we have found that performance analysis of distributed systems requires monitoring events before and after every I/O operation. This can generate huge amounts of monitoring data, and great care must be taken to deal with this data in an efficient and unobtrusive manner.

Consider the simple use-case of monitoring a heavily used FTP server. For example, a user notices downloading files is taking much longer than it did last week. The user has no idea why performance has changed. Is there a problem in the network, disk, end host, FTP server, or FTP client? Monitoring information is needed to pinpoint the bottleneck, and determine what changed to cause this bottleneck. Current performance must be analyzed, and compared against a baseline drawn from previously archived information. This performance analysis requires monitoring data for hosts (CPU, memory, disk), networks (bandwidth, latency, route), and the FTP client and server programs.

In the example above, the amount of monitoring data generated from a well-connected large FTP server could overwhelm a slow logging system. Consider a fast FTP server connected to a Gigabit-Ethernet network, instrumented to log the start and end times for all network and disk read and writes, with 10 simultaneous clients each transferring data at 10 MBytes per second. This server will generate around 6000 events per second of monitoring data. Assuming each monitoring event is 50 bytes, this equates to 300 KBytes/second, or more than a gigabyte per hour, of monitoring data. Clearly, the instrumentation data needs to be very compact and efficient in order to generate and store this much monitoring data without perturbing the system.

Previous versions of NetLogger used the IETF-proposed ULM format, described above. While easy to read and parse, this format imposed a great deal of unnecessary overhead. In order to improve efficiency, we have developed a new binary format that can still be used through the same API but that is several times faster and smaller, with performance comparable or better than binary message formats such as MPI [11], XDR [16], SDDF-Binary [15], and PBIO [5].

Conceptually, the NetLogger binary format carries the same information that the ULM format does; the API makes the underlying format transparent to the application. In reality, NetLogger binary messages are broken into a "header", which is sent only on the first write() call, and a "body". The header message describes the event structure, including the event names, data types, and constant values. The body message has a timestamp and the data values. Five data types are supported: 32 and 64-bit integers, 32 and 64-bit floating point numbers, and 255 byte (maximum) strings. This limited set of data types streamlines the entire NetLogger library but does not, for the purposes of instrumentation and monitoring, greatly restrict functionality. Most instrumentation messages are short and simple. For example, logging a transfer of a block of data requires only a timestamp, host name, integer disk offset, and integer number of bytes.

In order to port the internal data representation across different architectures, NetLogger uses a methodology called "receiver-makes-right", in which the sender uses its native representation and the receiver converts if its native representation is different. This behavior is optimal for instrumentation because it minimizes perturbation at the sender, which is running the application that we do not wish to disturb.

For efficiency, the NetLogger library, by default, employs 128KB buffers that automatically flush when full or when one second has passed, thus limiting latency but minimizing the load on the system at high data rates. This is useful for the small messages typical in monitoring, where buffering can reduce the number of I/O calls by a factor of 10^3.

Figure 4: Binary vs. ASCII performance Results

We gathered performance results for the binary message format in both C and Java. In both languages, we compared the binary format to NetLogger's ASCII ULM and XML formats. In C, we also ran the same test with Pablo's binary SDDF format and with PBIO. In Java we recreated the ULM log format with *log4j*, since log4j is a common solution for instrumentation of Java applications. Results are shown in Figure 4. For both C and Java, the
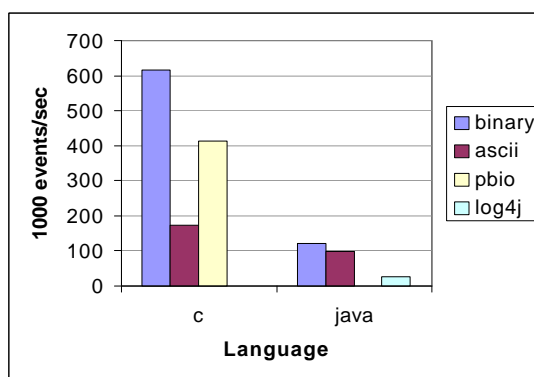
highest throughput is clearly from binary NetLogger. For more information on the testing methodology uses for this results, see [7].

4.2 NetLogger Reliability API

The new NetLogger Reliability API adds to NetLogger fault-tolerance features that are essential in distributed computing environments. For distributed monitoring, the 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 with more than a handful of sites involved. Because NetLogger uses TCP connections to efficiently and reliably transfer the monitoring data, network failures will terminate the data stream.

One solution would be to open the connection only when there is data to transfer. However, this reduces but does not eliminate the possibility of network failure during the data transfer, and is inefficient at high data rates. Instead, we have implemented a more general solution based on the idea of a temporary fail-over destination for the monitoring data. After the NetLogger connection is created, a single API call provides the library with 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 will be transparently logged to the backup destination. Periodically, the library will check whether the original destination has "come back up". If so, the library will reconnect and, if the backup destination was a file, send over all the data logged during the failure.

4.3 Monitoring Activation

Due to the volume of instrumentation data that can be generated by services such as at FTP server, we needed to add a mechanism to NetLogger to control long-running processes such as servers. We need to have the ability to change their logging behavior without command-line arguments, restarting, special signal handlers, or specialized control messages.

The NetLogger API has a new *trigger* function that tells the library to check, at user-specified intervals, for changes to the log destination. Two types of triggers are provided: a *file trigger* that scans a configuration file, and an *activation trigger* that connects to a remote service called the activation service daemon, allowing users to activate various levels of NetLogger instrumentation by sending activation requests to the activation service. Both of these mechanisms allow users to dynamically change NetLogger's behavior inside of a running application. This is very useful for long-lived processes like file servers, which may only occasionally need fine-grained instrumentation turned on. For more information on NetLogger activation, see [11].

5.0 Case Studies

In this section we present two case studies on the use of NetLogger for tracking down problems. The first case study shows how NetLogger helps tune an application, and the second case study shows how NetLogger discovered a "soft failure" that no one even knew existed.

In the first case, NetLogger was used to instrument a 3-dimensional visualization engine called Radiance [3] that read data off disk, rendered it, and sent it out to clients for display. Here, we will show how NetLogger helped tune the Radiance servers. Two graphs of the results are shown in Figure 5. The *lifelines* in these graphs represent the data flow to generate one image. The upper graph shows the results before NetLogger tuning. The line between "BE_LOAD_START" and "BE_LOAD_END" indicates the amount of time to read the data from
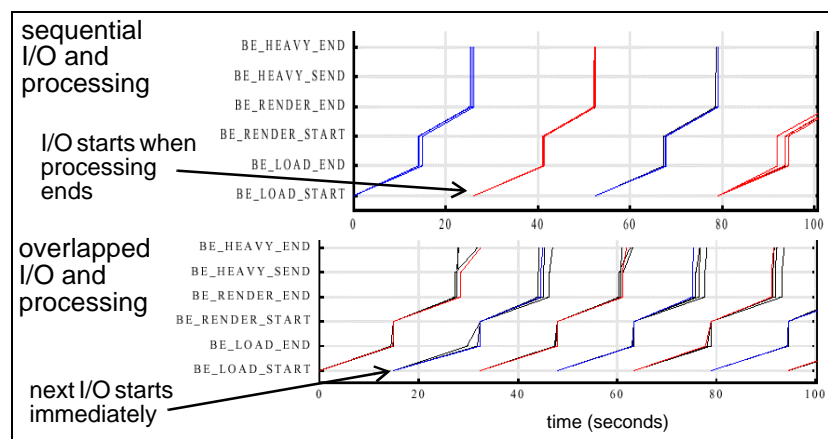

Figure 5: Before and after NetLogger analysis

disk, and the line from BE_RENDER_START" and BE_RENDER_END" is the amount of time to process the data into an image for display. The developer in this case had assumed that the I/O time was greater than the image rendering time, and therefore believed that there would be minimal advantage gained by going through the effort required to make the program multi-threaded, and overlap processing with I/O. However after seeing these results, it was clear to the developer that pipelining I/O and processing was indeed worth the effort, and by doing this, obtained the results in the lower graph; almost double the performance.

In the second case study a high-performance FTP client/server called GridFTP [2] was instrumented. Among other enhancements, GridFTP extends the FTP protocol to transfer a single file across several parallel TCP streams. In some WAN environments this can cause a dramatic (almost linear) speedup. Figure 6 shows NetLogger lifelines for a GridFTP client. The three groups of lifelines show three separate reading sockets in a parallel FTP client, and includes events for reading header and data packets. Data should always be arriving on all three sockets, but clearly the client was not servicing all three sockets equally. There was a bug in the way the Unix *select*() call was being used, and this bug had existed, undetected, for several months until this NetLogger analysis uncovered the problem. Despite the bug, the multi-stream version of the FTP client was faster than the single stream version, so no one had noticed this problem. This is the type of subtle bug that NetLogger is very good at tracking down.
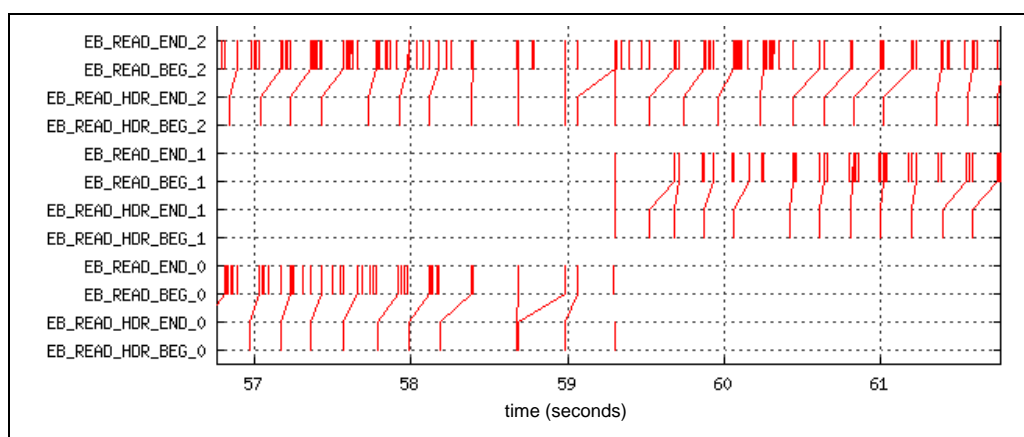

Figure 6: Parallel GridFTP Client Analysis

These two case studies demonstrate the NetLogger's ability to analyze a single application. In both of these cases the NetLogger visualization tool made it easy to spot problems. However, NetLogger's real power is demonstrated by analyzing a distributed application, and correlating monitoring from the application, host, and network. Previously mysterious interactions between components become visible, like the correlation between the block transfers and TCP retransmits shown in Figure 1.

6.0 Conclusions

In order to achieve high end-to-end performance in widely distributed applications, a great deal of analysis and tuning is needed. The top-to-bottom, end-to-end approach of NetLogger has proven to be a very useful mechanism for analyzing the performance of distributed applications in high-speed wide-area networks.

NetLogger can be a valuable "finger pointing" tool. In many cases where lower than expected performance is observed, everyone points the finger to put the blame on someone else. The application developer blames the network, the LAN people blame the WAN, and the WAN people blame the LAN. NetLogger can correlate monitoring events from applications, hosts, and networks to determine where the bottlenecks actually are, thus pointing the finger at the guilty party.

All NetLogger Toolkit components under an Open Source license, and can be downloaded from http://www-didc.lbl.gov/NetLogger/.

8.0 References

[1] Abela, J., T. Debeaupuis. *Universal Format for Logger Messages*. IETF Internet Draft, http://www.ietf.org/internet-drafts/draft-abela-ulm-05.txt

[2] Allcock B., Bester, J., Bresnahan, J., Chervenak, A., Foster, I., et.al. *Secure, Efficient Data Transport and Replica Management for High-Performance Data-Intensive Computing*. IEEE Mass Storage Conference, 2001.

[3] Bethel, W., B. Tierney, J. Lee, D. Gunter, S. Lau. *Using High-Speed WANs and Network Data Caches to Enable Remote and Distributed Visualization*. Proceeding of the IEEE Supercomputing 2000 Conference, Nov. 2000.

[4] CORBA. *Systems Management: Event Management Service*. X/Open Document Number: P437, http://www.opengroup.org/ onlinepubs/008356299/

[5] Eisenhauer G. and Lynn K. Daley. *Fast Heterogeneous Binary Data Interchange*. 9th Heterogeneous Computing Workshop (HCW 2000), May 2000.

[6] Eisenhauer, G., F. Bustamante, K. Schwan. *Event Services in High Performance Systems*. Cluster Computing: The Journal of Networks, Software Tools, and Applications, Vol 4, Num 3, July 2001, pp 243-252.

[7] Gunter, D., B. Tierney, K. Jackson, J. Lee, M. Stoufer, "*Dynamic Monitoring of High-Performance Distributed Applications*", Proceedings of the 11th IEEE Symposium on High Performance Distributed Computing, HPDC-11, July 2002, LBNL-49698.

[8] Jini Distributed Event Specification: http://www.sun.com/ jini/specs/

[9] log4j: http://jakarta.apache.org/log4j/docs/index.html

[10] log4j performance results: http://jakarta.apache.org/log4j/ docs/api/org/apache/log4j/performance/Logging.html

[11] Lee, J., D. Gunter, M. Stoufer, B. Tierney, "*Monitoring Data Archives for Grid Environments*", Proceeding of IEEE Supercomputing 2002 Conference, Nov. 2002, LBNL-50216.

[12] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*. April 1994.

[13] Mills, D., *Simple Network Time Protocol (SNTP)*, RFC 1769, University of Delaware, March 1995. http://www.eecis.udel.edu/~ntp/

[14] Open Group, Enterprise Management Forum. 2002, http://www.opengroup.org/management/arm.htm.

[15] Ribler, R., J. Vetter, H. Simitci, D. Reed. *Autopilot: Adaptive Control of Distributed Applications*. Proceedings of the 7th IEEE Symposium on High-Performance Distributed Computing, Chicago, IL, July 1998.

[16] Sun Microsystems. *XDR: External Data Representation Standard*. IETF RFC 1014, June 1987

[17] Tierney, B., D. Gunter, J. Becla, B. Jacobsen, D. Quarrie. *Using NetLogger for Distributed Systems Performance Analysis of the BaBar Data Analysis System*. Proceedings of Computers in High Energy Physics 2000 (CHEP 2000), Feb. 2000.

[18] Tierney, B., W. Johnston, B. Crowley, G. Hoo, C. Brooks, D. Gunter. *The NetLogger Methodology for High Performance Distributed Systems Performance Analysis*. Proceeding of IEEE High Performance Distributed Computing, July 1998, LBNL-42611. http://www-didc.lbl.gov/NetLogger/

[19] Tierney, B., W. Johnston, G. Hoo, J. Lee, *Performance Analysis in High-Speed Wide Area ATM Networks: Top-to-bottom end-to-end Monitoring*, IEEE Network, Vol. 10, no. 3, May/June 1996, LBL-38246.