The NetLogger Methodology for High Performance Distributed Systems Performance Analysis

Brian Tierney, William Johnston, Brian Crowley, Gary Hoo, Chris Brooks, Dan Gunter

Computing Sciences Directorate
Lawrence Berkeley National Laboratory
University of California, Berkeley, CA, 94720

Abstract

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 precision event logs that can be used to provide detailed end-to-end application and system level monitoring; a Java agent-based system for managing the large amount of logging data; and tools for visualizing the log data and real-time state of the distributed system. We developed these tools for analyzing a high-performance distributed system centered around the transfer of large amounts of data at high speeds from a distributed storage server to a remote visualization client. However, this methodology should be generally applicable to any 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.

1.0 Introduction

Developers of high-speed network-based distributed systems often observe performance problems such as unexpectedly low network throughput or high latency. The reasons for the poor performance can be manifold and are frequently not obvious. It is often difficult to track down performance problems because of the complex interaction between the many distributed system components, and the fact that performance problems in one place may be most apparent somewhere else. Bottlenecks can occur in any of the components along the paths through which the data flow: the applications, the operating systems, the device drivers, the network adapters on any of the sending or receiving hosts, and/or in network components such as switches and routers. Sometimes bottlenecks involve interactions among several components or the interplay of protocol parameters at different points in the system, and sometimes, of course, they are due to unrelated network activity impacting the operation of the distributed system.

* Published in the Proceedings of IEEE HPDC-7'98, 28-31 July 1998 at Chicago, Illinois.

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 would enable 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*, 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.

Software agents collect and filter event-based performance information, turn on or off various monitoring options to adapt the monitoring to the current system state, and manage the large amounts of log data that are generated.

The goal of this performance characterization work is to produce high-speed components that can be used as building blocks for high-performance applications, rather than having to "tune" the applications top-to-bottom as is all too common today. This method can also provide an information source for applications that can adapt to component congestion problems.

NetLogger has demonstrated its usefulness with the Distributed Parallel Storage System (DPSS), which is described below. NetLogger-assisted analysis revealed the cause of mysterious intermittent stalls that lowered the system's performance below expectations. Pinpointing the reason for the stalls allowed corrective action to be taken.

Although NetLogger has only been used in a loosely-coupled, client-server architecture, in principle its approach is adaptable to any distributed system architecture, such as the processor pool model. The way in which NetLogger is integrated into a distributed system will, of course, depend on the system's design; however, NetLogger's behavior and utility are independent of any particular system design.

2.0 NetLogger Components

NetLogger consists of *event logs* that represent the raw information about system performance, and the *NetLogger Toolkit* that generates and manipulates the logs.

Events and Event Logs

To analyze the performance of a wide-area distributed system, it is important to log as much information about the state of the system as possible. *Event logs* contain high-resolution, synchronized timestamps taken before and after *events* (activities of interest). Events may include application-, operating system-, or network-level activities; monitoring of operating system and network conditions is an important complement to application-level monitoring. A distributed system should be instrumented to log the time at which data is requested and received, and should record any local processing time.

In this paper, the logical path of an *object* (a particular datum or process flow) through the system is called its *lifeline*. This lifeline is the temporal trace of an object through the distributed system.

NetLogger analysis relies upon every log entry adhering to a common format (syntax), and the clocks of all hosts participating in the distributed system being synchronized.

Common logging format

To aid in the processing of the potentially huge amount of log data that can be generated from this type of logging, all events should be logged using a common format. NetLogger uses the IETF draft standard Universal Logger Message format (ULM).[1]

ULM format consists of a list of "field=value" pairs. ULM required fields are DATE, HOST, PROG, and LVL, followed by optional user defined fields. LVL is the severity level (Emergency, Alert, Error, Usage, and so on). NetLogger adds the field NL.EVNT, which is a unique identifier for the event being logged, and NL.SEC and NL.USEC, which are the seconds and microseconds values returned from the Unix *gettimeofday* system call.

Here is a sample NetLogger ULM event:

```
DATE=19980430133038 HOST=dpss1.lbl.gov
PROG=testprog LVL=Usage NL.EVNT=SEND_DATA
NL.SEC=893968238 NL.USEC=55784
SEND.SZ=49332
```

This says that a program named *testprog* on host *dpss1.lbl.gov* performed the event named SEND_DATA with a size of 49332 bytes at the time given.

The end of every log event can contain any number of user-defined elements. These can be used to store any information about the logged event that may later prove useful. For example, for a NETSTAT_RETRANSSEGS event, the data element would be the number of TCP retransmits since the previous poll time. A SERVER_START_WRITE event data element, on the other hand, might contain a data block ID, data set ID, and a user ID.

2.1 Clock Synchronization: NTP

To analyze a network-based system using timestamps, the clocks of all systems involved must be synchronized. This can be achieved by using the Network Time Protocol (NTP) [10]. By installing a GPS-based NTP server on each subnet of the distributed system, and running the *xntpd* daemon on each host, all host clocks can be synchronized to within about 0.25 ms of each other. It has been our experience that most application-significant events can be accurately characterized by timestamps that are accurate to about 1 ms, well within NTP's tolerances. If the closest time source is several IP router hops away, NTP accuracy will be somewhat less, but probably still accurate enough for many types of analysis. The NTP web site[2] has a list of public NTP servers that one may be able to connect and synchronize with.

2.2 NetLogger Toolkit

The NetLogger Toolkit consists of three components: a library of routines to simplify the generation of application-level event logs (for C, C++, and Java™), a set of modified operating system utilities, a set of tools for managing and filtering the log files, and a set of tools to visualize and to analyze the log files.

Event Log Generation Library

A library has been developed to simplify the creation of application event logs. The library contains routines to open, write, and close log files. Events can be written to a local file, the syslog daemon, or a given TCP port on a given network host.

As previously noted, monitoring of operating system and network activities complements application-level monitoring. Indeed, characterizing a distributed system's performance requires distinguishing between the "applica-

1. Available from:
ftp://ds.internic.net/internet-drafts/draft-abela-ulm-02.txt

2. See http://www.eecis.udel.edu/~ntp/.

tion" and its supporting infrastructure. NetLogger monitors and logs operating system- and network-level events using versions of the Unix utilities *netstat* and *vmstat*[3] that have been modified to support NetLogger. *netstat* reports on the contents of various network-related data structures. *vmstat* reports statistics on virtual memory, disk, and CPU activity. Both programs were modified to present only a relevant subset of their information in the common logging format, and *netstat* was modified to poll and report continuously (it normally provides only a snapshot of current activity). We typically poll at 100 ms intervals. Since the kernel events are not timestamped, the data obtained this way represents all events in this interval.

Agents for Event Logging Management

Management of monitoring programs and event logs for many clients connected to many distributed server components on many hosts is quite difficult. Our approach to this problem is to use a collection of software agents [6] to provide structured access to current and historical information.

For this discussion, an *agent* is an autonomous, adaptable entity that is capable of monitoring and managing distributed system components. Agents provide the standard interface for monitoring CPU load, interrupt rate, TCP retransmissions, TCP window size, etc. Agents can also independently perform various administrative tasks, such as restarting servers or monitoring processes. Collectively, a distributed system's agents maintain a continually updated view of the global state of the system.

A *broker* is a special type of agent that manages system state information, filters this information for clients, or performs some action on behalf of a client. The broker controls agents on each host, telling them which tasks to perform. The broker also collects event logs from each agent and merges them together, sorted by time, for use by the event log visualization tools.

NetLogger's agents are written in Java, use the Java Agent Toolkit (JATLite) from Stanford [7], and use the KQML communication language [4]. JATLite is a set of Java packages that facilitate the agent framework development by providing basic communication tools and templates based upon TCP/IP and KQML messages.

This agent/broker system provides two main functions for NetLogger. The agents can start and stop individual monitoring components, based on which system components are currently active. This reduces the amount of logging data collected. The second function is to filter, collect, and sort the logged events for the analysis tools.

These functions are provided through brokers, which are implemented as Java applets. One broker monitors when a server is accessed, and turns on system monitoring of CPU load, interrupt rate, TCP retransmissions, TCP window size, and server events, for the duration of the connection.

A second broker is used to collect and filter the event logs. An applet shows the user which event log files are currently available and allows the user to specify which events to collect over what time range. The broker then collects the event logs from each agent and merges them, sorted by timestamp, for use by the event log visualization tools.

The agent architecture is a crucial component for NetLogger because without it, run-time management of the immense volume of log event data that can be generated would be infeasible. In addition, the agent architecture is sufficiently general to be useful for a wide variety of distributed system management tasks. For example, we are using the agents to monitor NTP clock synchronization accuracy. We are also using these agents in other projects to make sure servers are up, and to help with load balancing and fault tolerance. We expect to discover new uses for this agent architecture as we gain experience using it with different types of applications.

Event Log Analysis and Visualization Tools

Exploratory, interactive analysis of the log data—especially analysis of the graphical representations of individual, exceptional events—has proven to be the most important means of identifying the causes of specific behavior. In particular, the ability to distinguish, manipulate, and analyze lifelines is critical to isolating the locations of (and thereby the reasons for) unexpected behavior.

NetLogger builds lifelines by combining specified events from a given set of processes, and represents them as lines on a graph. The graph plots time (i.e., the timestamp from the event log) against a set of events. For example, in a client-server distributed system, each request-response transaction might be represented as a lifeline; the events on the lifeline might include the request's dispatch from the client, its arrival at the server, the commencement of server processing of the request, the dispatch of the response from the server to the client, and the arrival of the response at the client.

We have developed a tool called *nlv* (NetLogger Visualization) for interactively viewing the NetLogger event files. *nlv* can display several types of NetLogger events. The user can combine multiple different sequences of events, servers, and graph types (lifeline, load-line, or point) on a single graph; the display can be modified to show an arbitrary subset of these elements. *nlv* graphing primitives are shown in Figure 1. The *point* type is used to graph events such as TCP retransmits that happen at a cer-

3. Both *netstat* (displays network statistics) and *vmstat* (displays virtual memory statistics) are tools available on most Unix systems. We have modified versions for Solaris, FreeBSD, and Linux to date.

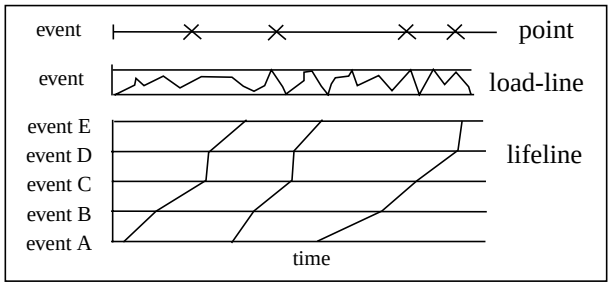

Figure 1:nlv graphing primitives

tain point in time. The *load-line* type is used to graph events such as CPU load that vary over time. The *lifeline* type is used to follow a data object through time. *nlv* provides the ability to play, pause, rewind, slow down, zoom in/out, and so on. Figure 2 shows a sample *nlv* session.

nlv can be run post-mortem on log file collected after the application is finished, or can be run in "real-time," analyzing live applications. In addition, *nlv* allows the user to specify data statistics to display based on formulas which use standard mathematical operators and event keyword names.

nlv is implemented in C and Tcl/Tk [12], and uses the Tcl/Tk extension library called BLT [2]. BLT is a graphics extension to the Tk toolkit, adding new widgets and geometry managers.

NetLogger tools to analyze log files also include *perl* scripts[4] to extract information from log files and to write

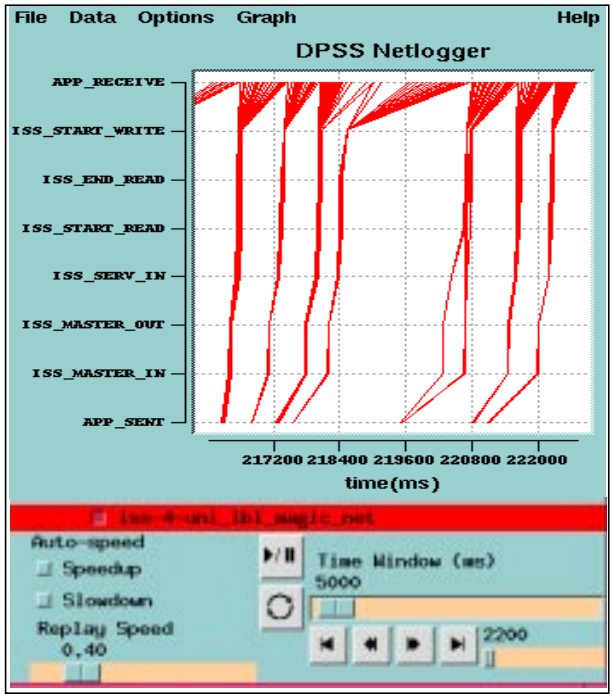

Figure 2: Sample nlv display

data files in a format suitable for using *gnuplot*[5] to graph the results. These tools were used to generate the graph in Figure 2. *gnuplot* is not as interactive as *nlv*, but it is more flexible, and produces cleaner graphs for use in publications.

3.0 Sample NetLogger Toolkit Use

To illustrate the NetLogger approach, we describe its use in a high performance distributed system deployed in the DARPA-funded MAGIC gigabit testbed network[6] [5]. MAGIC has a complex, widely distributed application running over a high-speed ATM network. This is an example of an environment in which a complete monitoring methodology is critical to obtaining expected performance.

3.1 MAGIC Application Overview

A real-time terrain visualization application uses the MAGIC network to access multi-gigabit image data sets from the Distributed-Parallel Storage System (DPSS). The DPSS provides an economical, high-performance, widely distributed, highly scalable architecture for caching large amounts of data that can be accessed by many different users. It allows real-time recording of, and random access to, very large data sets. In the MAGIC testbed, DPSS system components are distributed across several sites separated by more than 2600 Km of high speed network, using IP over ATM.

The terrain visualization application, called *TerraVision*, lets a user explore/navigate a "real" landscape represented in 3D by using ortho-corrected, one meter per pixel images and digital elevation models (see [9]). TerraVision requests from the DPSS, in real time, the sub-images ("tiles") needed to provide a view of a landscape for an autonomously "moving" user. This requires aggregated data rates as high as 100 to 200 Mbits/sec. The DPSS is easily able to supply these data rates. Terravision typically sends a request for 50 to 100 50 KByte tiles every 200 milliseconds.

The combination of the distributed nature of the DPSS, together with the high data rates required by TeraVision and other DPSS clients, make this a good system with which to test and analyze high-speed network-based systems.

3.2 NetLogger and the DPSS

To understand how NetLogger works within the TeraVision-DPSS distributed system, it is necessary to understand the architecture of the DPSS.

4. See: http://www.metronet.com/perlinfo/perl5.html

5. See: http://www.cs.dartmouth.edu/gnuplot_info.html

6. See: http://www.magic.net/

DPSS Architecture

The DPSS is essentially a “logical block” server whose functional components are distributed across a wide-area network. The DPSS uses parallel operation of distributed servers to supply high-speed data streams. The data is declustered (dispersed in such a way that as many system elements as possible can operate simultaneously to satisfy a given request) across both disks and servers. This strategy allows a large collection of disks to seek in parallel, and all servers to send the resulting data to the application in parallel, enabling the DPSS to perform as a high-speed data server.

The DPSS is implemented using multiple low-cost, “off-the-shelf” medium-speed disk servers. The servers use the network to aggregate multiple outputs for high performance applications. All levels of parallelism are exploited to achieve high performance, including disks, controllers, processors/memory banks, and the network.

A typical DPSS consists of several (e.g., four) Unix workstations, each with several (e.g., four) SCSI III disks on multiple (e.g., two) SCSI host adapters. Each workstation is also equipped with a high-speed network interface. A DPSS configuration such as this can deliver an aggregated data stream to an application of about 400 Mbits/s (50 Mbytes/s), using these components, by exploiting the parallelism provided by approximately four disk servers, 16 disks, eight SCSI host adapters, and four network interfaces.

Other papers describing the DPSS, including one describing the implementation in detail [17], are available at http://www-itg.lbl.gov/DPSS/papers.html.

DPSS Timing Facility

The DPSS and several of its clients have been instrumented to collect time stamps at all important events. A request for a data block takes the following path through the DPSS (see Figure 3). A *request list* (list of data blocks) is sent from the application to the name server (“START”), where the logical block names are translated to physical addresses (server:disk:disk offset). Then the individual requests are forwarded to the appropriate disk servers. At the disk servers, the data is read from disk into local cache, and then sent directly to the application, which has connections to all the relevant servers. Timestamps are gathered before and after each major function, such as name translation, disk read, and network send.

For DPSS applications, the timestamps are sent, with the data block, to the requesting application. The event information is not logged at the time it is collected: rather, it is carried along with the data back to the application and then logged. This post-path logging eliminates the need to correlate information for individual objects from multiple logs. This was done because at the time the NetLogger agents did not exist.

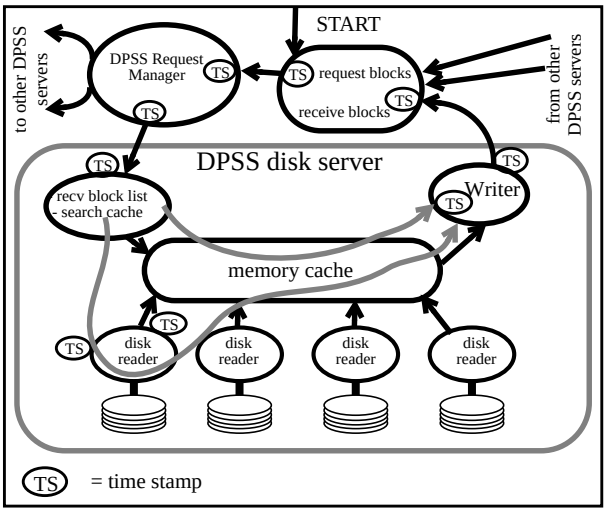

Figure 3: DPSS Performance Monitoring Points

NetLogger Analysis of the DPSS

To date, the most useful analysis technique for NetLogger data has been to construct and examine individual lifelines. Lifelines are characterized by the time of the actions, events, or operations in all of the system components.

A DPSS application call can generate NetLogger events such as this:

```
DATE=19980430133038 HOST=dpss1.lbl.gov
PROG=tv_rcvr LVL=Usage NL.EVNT=APP_RECEIVE
NL.SEC=893968238 NL.USEC=55784
DPSS.SERV=131.243.2.94 DPSS.SID=806
DPSS.BID=160.175.0.0 DPSS.SES=0
DPSS.BSZ=49332
```

This event shows that DPSS application *tv_rcvr* on host *dpss1.lbl.gov* generated this message. The event shows a block of size 49332 with block id 160,175,0,0 was received from a server at address 131.243.2.94 at the precise time given.

Figure 4 illustrates the general operational characteristics of the DPSS with the TerraVision application[7]. DPSS events, together with TCP retransmit events, are plotted on the vertical axis and the corresponding times on the horizontal axis. Each line corresponds to the life of a data request, from application request to application receipt. Each line style in the graphs represents data from a different DPSS disk server. These lifelines have characteristic shapes and relationships that represent how the various algorithms in the system operate and interact in the

7. Results in Figure 4 are from a four year old DPSS server platform (a 60 MHz Sun SS10 with SCSI II disks). Current servers and disks are more than twice as fast; typical throughput is 10 MBytes/sec per DPSS disk server)

face of different environmental conditions (e.g., network congestion).

Referring to Figure 4, TerraVision sends a list of data block requests every 200 ms, as shown by the nearly vertical lines starting at the *app_send* monitor points. The initially single lifelines fan out at the *server_in* monitor point as the request lists are resolved into requests for individual data blocks. Each block request is first represented individually in the read queue (*start_read*).

Using these lifeline graphs it is possible to glean detailed information about individual operations within the disk servers. For example, when two lifelines cross in the area between *start_read* and *end_read*, this indicates that a read from one disk was faster that a read from another disk. (This phenomenon is clearly illustrated for the server represented by the crossing solid lines in Figure 4 at "A".) This faster read might be from a disk with faster seek and read times, or it might be due to two requested blocks being adjacent on disk so that no seek is required for the second block. In Figure 4 we can also see:

- at "B", two different characteristic disk reads (one with an 8 ms read time and one with a 22 ms read time);
- at "C", the average time to cache a block and enter it into the network write queue is about 8.6 ms;
- at "D", the time to parse the incoming request list and see if the block is in the memory cache is about 5 ms;
- at "E", the overall server data read rate (four disks operating in parallel) is about 8 MB/sec;
- at "F", the actual throughput for this server while dealing with a set of real data requests is about 39 Mb/s (this throughput is receiver-limited);
- at "G", there are two cache hits (blocks found in memory) as a result of previously requested, but unsent, data being requested.

WAN Experiment

NetLogger was used to find an ATM switch buffer overflow problem in an experiment over the MAGIC ATM

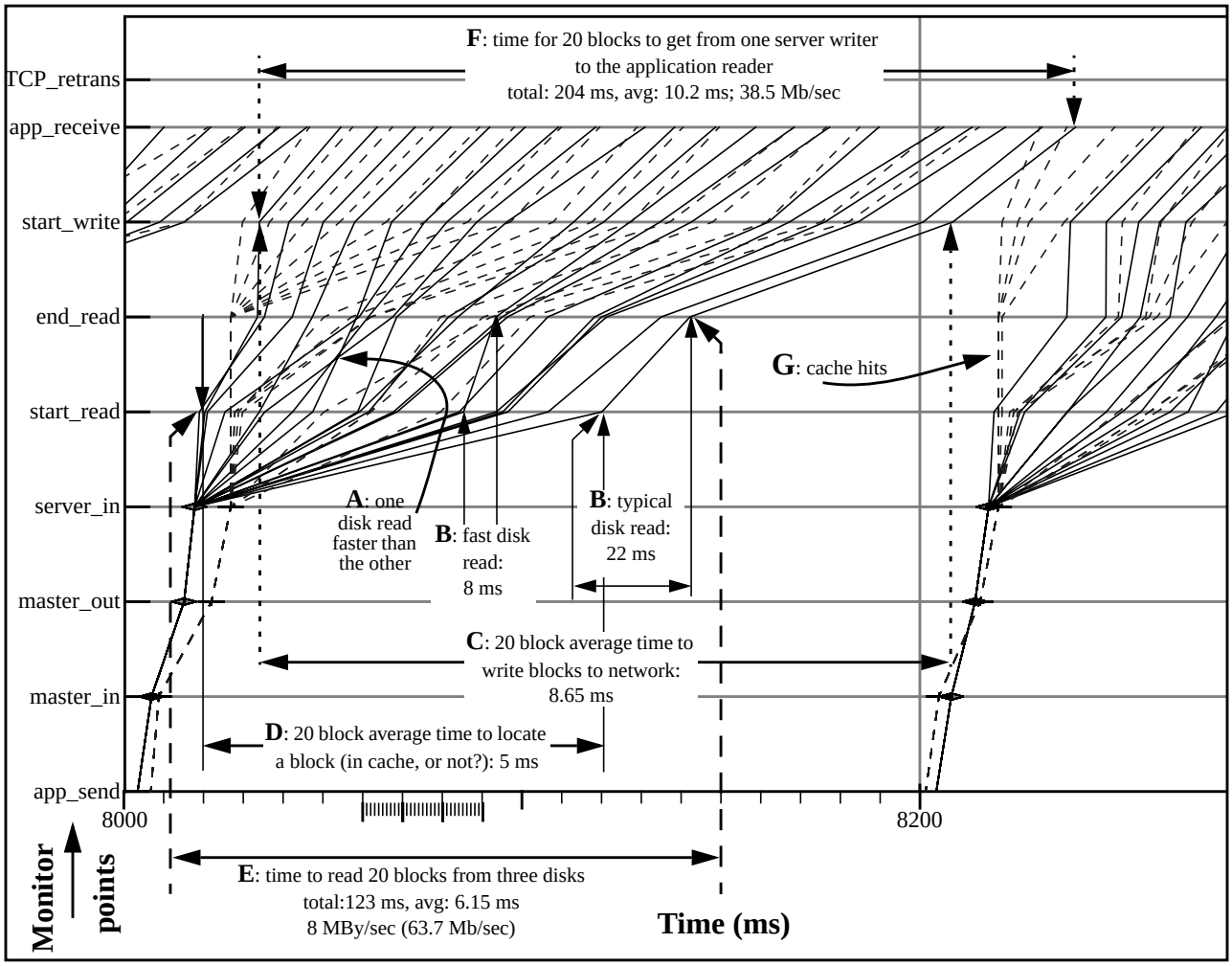

Figure 4: DPSS Performance Detail From a Two Server LAN Experiment

WAN. A detailed analysis of this experiment is in [17]. We were using three DPSS servers sending data to one application, and getting terrible throughput of only 2-3 Mbits/second, while we knew each server to be capable of about 50 Mbits/sec. Using NetLogger, we saw some extraordinarily long delays (up to 5500 ms to send one TCP packet!). These long delays were almost always accompanied by one or more TCP retransmit events. This causes the entire server to stall, because once a block is written to the TCP socket, TCP will re-send the block until transmission is successful. The server continues when the retransmission is successful, letting the next write proceed.

NetLogger analysis led us to closely examine the network topology to try to determine what might cause the TCP retransmits. While the ATM switch at the Sprint TIOC was not reporting buffer overflows, it seemed likely that this was the case. Closer examination found that this switch had small output buffers (about 13K bytes), but the network MTU (minimum transmission unit) is 9180 bytes (as is typical for ATM networks). In this application the three server streams converged at that ATM switch, so that three sets of 9 KByte IP packets were converging on a link with less than half of that amount of buffering available, resulting in most of the packets (roughly 65%) being destroyed by cell loss due to buffer overflows at the switch output port.

Other NetLogger Analysis

NetLogger has also used to successfully analyze several other high performance distributed applications, including a high-energy nuclear physics (HENP) data analysis system called Star Analysis Framework (STAF) [15], designed to process one terabyte of data per day [8]. Using NetLogger we were able to verify correct parallel performance for reading and writing data, and were able to do accurate throughput measurements.

Adding NetLogger to an Application

To add NetLogger event logs to an existing application, it is best to have access to the application source code. However, it is possible to do some monitoring without the source code by writing NetLogger wrappers for various system components. We recommend generating an event log entry for all of the following: before and after disk or network I/O, and before and after any significant CPU computation. Generating a log entry is easy using the NetLogger library. Only three basic calls are required: *NetLoggerOpen*(), *NetLoggerWrite*(), and *NetLoggerClose*() [11].

For example, a call of *NetLoggerWrite* might be:

```
NetLoggerWrite (lp, "EVENT_NAME",
"F1=%d F2=%d F3=%; F4=%.2f",
data1, data2, string, fdata);
```

The first argument is a handle returned by *NetLoggerOpen*; the second is the NetLogger event keyword; the third argument is the format for the user-defined logging data fields, and the remaining arguments are a list of data to fill in the format if argument three.

4.0 NetLogger Overhead

When used carefully, NetLogger adds very little overhead to existing programs. The NetLogger client library call to generate a NetLogger event log takes between 0.2 and 0.5 milliseconds on must current systems, so one should only try to use NetLogger monitoring on events that take at least a few milliseconds. Also, one must be careful not to affect the normal operation of the system being monitored.

For example, to avoid NFS overhead, NetLogger data should be written to local disks only. Also, the logs files themselves should be monitored to ensure they do not consume all available disk space. Also, don't send log messages over a 10 Mbits/sec ethernet network, as NetLogger messages themselves can generate a few megabits per seconds of network traffic.

We have observed that it is also important for it to be easy to turn logging on and off. Otherwise one will need a lot a disk space! For example, a five minute run of TerraVision connected to a two server DPSS generates about 10 MBytes of log files.

5.0 Related Work

There are several research projects addressing network performance analysis. For example, see the list of projects and tools on the Cooperative Association for Internet Data Analysis (CAIDA) Web site [3]. However, we believe that the NetLogger approach is unique in combining application, system, and network -level monitoring together.

There are quite a few packages such as Pablo [13], Paradyn [14], Upshot [18], and others [1] which are designed to do performance analysis of distributed processing. However these systems are aimed at determining the efficiency of PVM / MPI -style distributed-memory parallel processing, and in general do a coarse-grained analysis of the system, with a focus on processor utilization. NetLogger does a fine-grained analysis of the system and its components, and focuses on precise detailed analysis of data movement and messaging. Current NetLogger visualization tools would not work as well on data from a large number of nodes in a PVM/MPI-style system, but could be useful for a detailed analysis of a small number of nodes.

6.0 Future Work

Our goal is to use NetLogger to automatically detect performance problems and, when possible, to use the information from real-time analysis of NetLogger data to

compensate for or correct the problem. It might even be possible to predict potential problems and respond before they arise.

We expect our agent architecture to be extremely useful for meeting this goal. For example, agents can not only provide standardized access to comprehensive monitoring, but also perform tasks such as characterizing event/lifeline patterns based on types of system (mis)behavior, as well as correlating analysis of collections of events in order to detect other types of patterns, thereby providing the basis for adaptive behavior for both the systems and the agents.

The monitoring mechanisms should themselves be adaptive. While detailed analysis of complex problems requires comprehensive data, it is not possible to routinely collect and cache all the data that might be needed. Instead, the monitors must adapt their behavior to the operating state of the system, detecting and/or responding to problems and preserving and making available data at the appropriate scope and granularity.

7.0 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 is proving to be a very useful mechanism for analyzing the performance of distributed applications in high-speed wide-area networks; NetLogger's graphic representation of system performance is especially useful and informative.

We envision that this type of monitoring will be a critical element in building reliable high-performance distributed systems. Experience with event-oriented monitoring leads us to believe that this is a very promising approach, especially when the focus is on the applications and middleware.

Acknowledgments

We gratefully acknowledge the contributions made by many MAGIC project colleagues, and by STAF developer Craig Tull. The work described in this paper is supported by DARPA, Computer Systems Technology Office and by the Director, Office of Energy Research, Office of Basic Energy Sciences, of the U.S. Department of Energy under Contract No. DE-AC03-76SF00098. This is report no. LBNL-41786.

References

[1] S. Browne, J. Dongarra, K. London, "Review of Performance analysis tools for MPI Parallel Programs". http://www.cs.utk.edu/~browne/perftools-review/.

[2] BLT; See: http://www.tcltk.com/blt/.

[3] CAIDA: http://www.caida.org/Tools/taxonomy.html.

[4] T. Finin et. al., DRAFT Specification of the KQML Agent Communication Language, unpublished draft, 1993. http://www.cs.umbc.edu/kqml/.

[5] B. Fuller and I. Richer "The MAGIC Project: From Vision to Reality," IEEE Network, May, 1996, Vol. 10, no. 3.

[6] M. Genersereth and S. Ketchpel, Software Agents, Communication of the ACM, July, 1994.

[7] JATLite: http://java.stanford.edu/java_agent/html.

[8] W.E. Johnston, W. Greiman, G. Hoo, J. Lee, B. Tierney, C. Tull, D. Olson, "High-Speed Distributed Data Handling for On-Line Instrumentation Systems." Proceedings of IEEE/ACM Supercomputing 97. http://www-itg.lbl.gov/DPSS/papers.html.

[9] S. Lau and Y. Leclerc, "TerraVision: a Terrain Visualization System,", Technical Note 540, SRI International, Menlo Park, CA, Mar. 1994. http://www.ai.sri.com/~magic/terravision.html.

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

[11] NetLogger API: http://www-didc.lbl.gov/NetLogger/api.html.

[12] J. Ousterhout, Tcl and the Tk Toolkit, Addison-Wesley, 1994. See: http://www.tcltk.com.

[13] Pablo Scalable Performance Tools, http://vibes.cs.uiuc.edu/.

[14] Paradyn Parallel Performance Tools, http://www.cs.wisc.edu/~paradyn/.

[15] STAF: http://www.rhic.bnl.gov/STAR/html/ssd_l/staf_l/STAF-current/.

[16] B. Tierney, W. Johnston, H. Herzog, G. Hoo, G Jin, and J. Lee, "System Issues in Implementing High Speed Distributed Parallel Storage Systems", Proceedings of the USENIX Symposium on High Speed Networking, Aug. 1994, LBL-35775. http://www-itg.lbl.gov/DPSS/papers.html.

[17] B. Tierney, W. Johnston, G. Hoo, J. Lee, "Performance Analysis in High-Speed Wide-Area ATM Networks: Top-to-Bottom End-to-End Monitoring", IEEE Network, May, 1996, Vol. 10, no. 3. LBL Report 38246, 1996. http://www-itg.lbl.gov/DPSS/papers.html

[18] The Upshot Program Visualization System: http://www-c.mcs.anl.gov/home/lusk/upshot/.