NetLogger: A Toolkit for Distributed System Performance Analysis

Dan Gunter, Brian Tierney, Brian Crowley, Mason Holding, Jason Lee
Lawrence Berkeley National Laboratory

Abstract

Diagnosis and debugging of performance problems on complex distributed systems requires end-to-end performance information at both the application and system level. We describe a methodology, called NetLogger, that enables real-time diagnosis of performance problems in such systems. The methodology includes tools for generating precision event logs, an interface to a system event-monitoring framework, and tools for visualizing the log data and real-time state of the distributed system. Low overhead is an important requirement for such tools, therefore we evaluate efficiency of the monitoring itself. The approach is novel in that it combines network, host, and application-level monitoring, providing a complete view of the entire system.

1. 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.

Usually the interactions between components are not known ahead of time, and may be difficult to replicate. Therefore, it is important to capture as much of the system behavior as possible while the application is running. It is also important to respond to performance problems as soon as possible; while post-hoc diagnosis of the data is valuable for systemic problems, for operational problems users will have already suffered through a period of degraded performance.

We have developed a methodology, called NetLogger, for monitoring, under realistic operating conditions, the behavior of all elements of the application-to-application communication path in order to determine exactly what is happening within a complex system.

Distributed application components are modified to produce timestamped logs of "interesting" events at all the critical points of 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.

NetLogger has demonstrated its usefulness in several contexts, including the Distributed Parallel Storage System (DPSS)[5], and Radiance[15]. Both of these are loosely-coupled client-server architectures. In principle, however, the approach is adaptable to any distributed system architecture. The way in which NetLogger is integrated into a distributed system will vary, but NetLogger's behavior and utility are independent of any particular system design.

2. NetLogger Toolkit Components

All the tools in the NetLogger Toolkit share a common log format, and assume the existence of accurate and synchronized system clocks. The NetLogger Toolkit itself consists of three 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, and a tool for visualization and analysis of the log files.

2.1. Common log format

NetLogger uses the IETF draft standard 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 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 of accuracy, allowing for microsecond precision in the timestamp. Here is a sample NetLogger ULM event:

```
DATE=20000330112320.95794
3 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.

0Clock synchronization

In order to analyze a network-based system using absolute timestamps, the clocks of all relevant hosts must be synchronized. This can be achieved using a tool which supports the Network Time Protocol (NTP) [8], such as the xntpd [9] daemon. By installing a GPS-based NTP server on each subnet of the distributed system and running xntpd on each host, all the hosts' clocks can be synchronized to within about 0.25ms. If the closest time source is several IP router hops away, accuracy may decrease somewhat. However, it has been our experience that synchronization within 1 ms is accurate enough for many types of analysis. The NTP web site (http://www.eecis.udel.edu/~ntp/) has a list of public NTP servers that one can connect to and synchronize with.

2.2. 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 six languages: Java, C, C++, Perl, Python, and Fortran. 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.

Here is a sample of the Java API usage:

```
NetLogger eventLog = new
NetLogger("testprog");
eventLog.open("dolly.lbl.
gov", 14830 );
...
eventLog.write("WriteIt",
"SEND.SZ=" + sz );
...
eventLog.close();
```

If the value for sz is 49332, and the program is running on the host *dpss1.lbl.gov*, the *write()* statement above will produce the sample log entry provided in the description of ULM, above. In this case, the data will be sent to port 14830 on the host *dolly.lbl.gov*.

2.3. Event log collection and sorting

NetLogger facilitates the collection of event logs from an application which runs across a wide-area network by providing automatic logging to a chosen 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.

Another tool, the Real-Time Collector, has been developed in order to receive events from both applications and system event monitoring services. The jobs of the real-time collector are to coordinate the aggregation of event trace data from applications with monitoring data, and to provide an easy-to-use gateway to the monitoring management system for one or more graphical front-ends. In order to do this, the real-time collector must perform three functions: receive application data, process remote requests, and possibly translate between different log formats.

2.4. 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. These are shown in Figure 1.

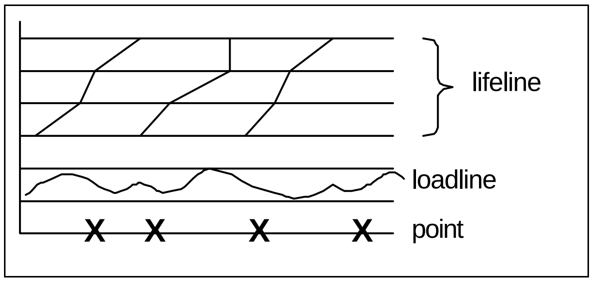

Figure 1: NLV Graph Primitives

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. With time shown on the x-axis, and ordered events shown on the y-axis, the slope of the lifeline gives a clear visual indication of latencies in the distributed system. Each object is given a unique identifier by placing a unique combination of values in one or more of its ULM fields. These values are used for all events along the path. In a client-server system, one such event path might include: a request's dispatch from the client, the request's arrival at the server, the begin and end of server processing of the request, the response's dispatch from the server, and the response's arrival at the client.

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 in a scatterplot. For example, the size of the data passed up from the operating system from individual *read()* calls was instrumented in a distributed file server, producing the graph in Figure 2.

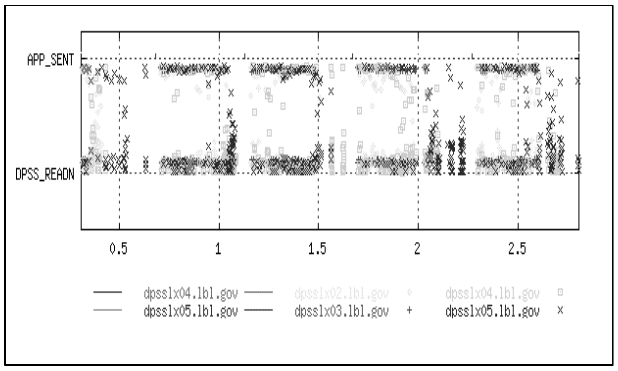

Figure 2: Scatterplot with point primitive

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. The program switches between these two modes at the press of a button. A portion of an *nlv* graph, showing lifelines from a client stacked below events from a server, is shown in Figure 3.

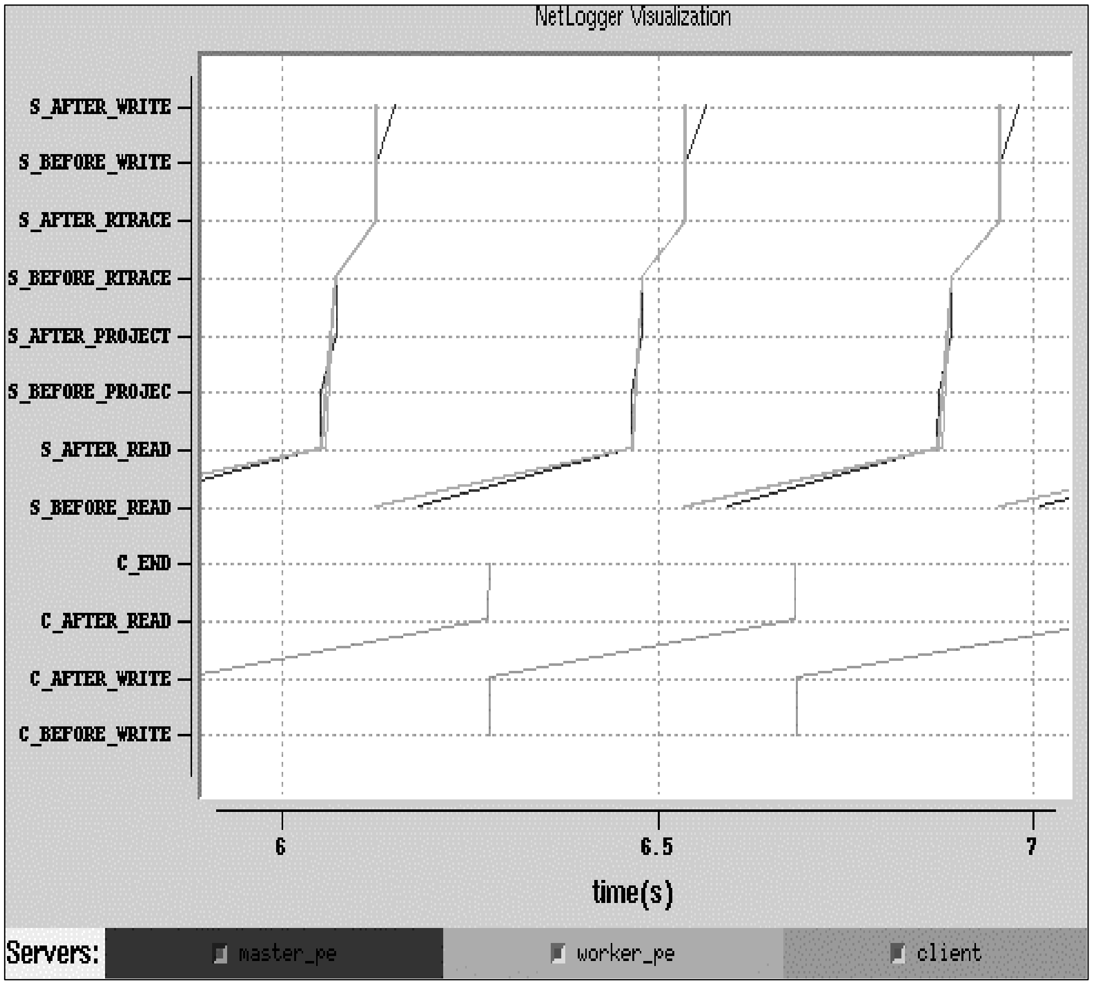

Figure 3: Screenshot of NetLogger Visualization tool (cropped)

The data represented in the screenshot above was generated while debugging a problem with an actual client- server application called Radiance [15], which is a distributed scene rendering engine. A researcher who helped develop the application was experiencing an unusual latency at the beginning of each rendered frame. When he instrumented Radiance with NetLogger, the exact location of this latency became obvious. Between the end of a the client's read operation (C_END) and the start of the server's read (S_BEFORE_READ), a 0.2 second gap, more than expected, appeared on the graph. In response to this observation, the application was modified and the latency was cut in half [12].

3. Analysis of NetLogger Efficiency

Logging can perturb the application or its environment if it occurs too frequently, and in the worst case the monitoring tool can end by monitoring itself.

Therefore, it is important to examine the efficienty of the logging itself. In this section we will analyze performance results gathered from the C, C++, and Java NetLogger APIs.

3.1. Experimental setup

The experiment itself was intentionally kept trivial: 50,000 log lines were written to file, LAN, WAN, and memory destinations. The pseudocode for the entire program is:

```
FOR each destination DO
   FOR N = 1 to 50000 DO
      NetLoggerWrite()
   END
END
```

Two systems were used for testing: a Sun Ultra-60 running Solaris 2.7, and a Pentium II 500MHz box running Linux 2.2.5. Both machines had a 100 Mb/s ethernet connection to the local network. As they were on the same subnet, both machines had a similar connection to the Internet for the wide-area tests, which were all done to the same remote machine.

Four different destinations for NetLogger data were chosen: a host on a wide-area network (physically located near Chicago, Illinois), a host on the local area network (located in Berkeley, CA), and the local disk (using */tmp* to avoid logging over a network disk). The bandwidth in the local area network was around 85 Mb/s -- typical for fast Ethernet -- and the connection to the remote WAN was measured by *netest* to be around 7 Mb/s.

Each test was performed using the C and Java APIs, both with and without buffering the data in memory as described in Section 2.3. The Java version was 1.2, using Sun's HotSpot JVM on Solaris and the Classic VM on Linux. The C programs were compiled with gcc with default compiler optimizations (the –O flag) turned on.

3.2. Results and analysis

The results shown in Figure 4 demonstrate, first, that the choice of Java Virtual Machine (JVM) is critical for good Java performance. On Solaris, which has a "just-in-time" compiler, the Java performance was comparable to the C performance. On Linux, which had a "Classic VM" with no compilation of frequently reused code, the C test ran about 2 to 5 times faster than the Java test.

Second, the Java results showed a remarkable consistency, for both Linux and Solaris, across WAN, LAN, file, and memory tests. This observation leads to the hypothesis that the I/O is not the dominant factor in the Java API's performance. Profiling the test code on both Solaris and Linux confirmed this hypothesis, as it showed that the Java API spent less than 2% of its total time in the I/O routines.

Finally, in the C API, it is notable that writing to memory, while faster than writing to WAN and LAN, was consistently slower than writing to a local file. This is partially explained by the buffering of all disk writes to memory by both the Solaris and Linux kernels.

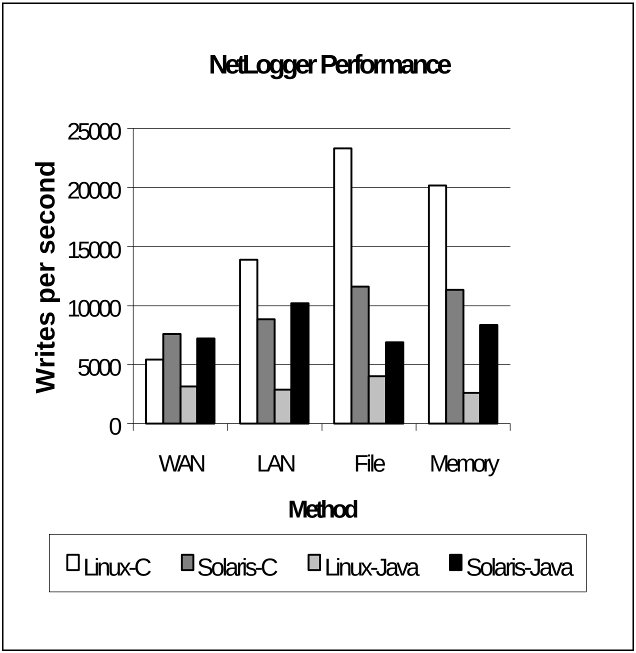

Figure 4: NetLogger Performance Results

3.3. Summary

The results lead to some important lessons on the use of NetLogger:

- in Java it is not worth the trouble to buffer in memory unless writing to a very slow network connection
- however, in Java it is well worth using a just-in-time compiler
- in C, memory buffering will be effective when writing to a remote or local network
- however, in C writing to a local file (*not* an NFS mounted disk) directly is preferable to buffering in memory

The performance which we have reported here are adequate for many types of applications, but of course some applications would benefit from instrumentation at even higher frequencies. Even when it has sole possession of the CPU, NetLogger is unable to log more than 24,000 of lines per second; extrapolating from this result, it would be difficult to log more than a few hundred lines per second with 1% perturbation of the system. Thus, NetLogger is clearly not the right tool for kernel monitoring. We believe that the verbosity of ULM is an important factor in NetLogger’s performance; therefore, use of an internal binary format, such as Pablo's Self-Defining Data Format (SDDF)[2] is being actively researched as a way to extend NetLogger to handle an even wider range of application requirements.

4. Related projects

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 [4]. However, we believe the NetLogger approach is unique in combining application, system, and network monitoring.

In addition, there are quite a few packages, such as Pablo[10], Upshot [14], and others [3], which are designed to do distributed process monitoring. Unlike NetLogger, these packages are aimed at the analysis of MPI/PVM-style parallel processing, and provide only coarse-grained measurements of the end-to-end characteristics of the system, with a focus on CPU utilization. Although NetLogger visualization does not include some of the specialized graph types used for such systems, analysis of large distributed systems is not intractable: we have used NetLogger with some success to analyze a distributed database application with around 200 nodes used in the high-energy physics BaBAR experiment [13].

5. Future work

Our goal is to allow the NetLogger Toolkit to be as useful and unobtrusive as possible, so that it may become a ubiquitous service for distributed applications. We intend to extend the evaluation of NetLogger’s overhead which was discussed here to all the current languages (C, C++, Java, Perl, Python, Fortran) in which it has an API, and to attempt analyses with larger test applications. In addition, we are continuing the integration of NetLogger monitoring with various monitoring management services, and developing more sophisticated analysis tools. Finally, we are committed to adding support for an output format using the emerging standard of XML, to facilitate interoperability with other monitoring services and tools.

6. Conclusion

We have demonstrated that the NetLogger application can be useful for debugging distributed applications, and can log events at a raw speed of 24,000 lines per second. Therefore, we believe that NetLogger could form an integral part of many high-reliability, high-performance distributed systems.

7. Acknowledgements

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-AC3-76SF00098. This is report no. LBNL-46269.

References

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

[2] Ruth Aydt. *The Pablo Self-Defining Data Format*. http://vibes.cs.uiuc.edu/Publications/Documents/documents.htm#SDDF, 1992 latest revision 2000.

[3] S. Browne, J. Dongarra, K. London, *Review of Performance Analysis Tools for MPI Parallel Programs*, http://www.cs.utk.edu/~browne/perftools-review/.

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

[5] Jason R. Lee. *The Image Server System: A High-Speed Parallel Distributed Data Server*, LBL report 36002, http://www-didc.lbl.gov/DPSS/papers/ISS-paper.LBL-report.fm.html

[6] Java Agents for Monitoring and Management. http://www-didc.lbl.gov/JAMM/

[7] Java HotSpot(tm) Technology. http://java.sun.com/products/hotspot/index.html.

[8] D. Mills. *Simple Network Time Protocol (SNTP)*. RFC 1769, March 1995

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

[10] Pablo. http://www-pablo.cs.uiuc.edu/Projects/Pablo/

[11] SIGGRAPH, pp. 459- 472.

[12] D. Robertson. Correspondence, March, 1999.

[13] Tierney, B., Gunter, D., Becla, J., Jacobsen, B., Quarrie, D., *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, LBNL-44828.

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

[15] G.J.Ward. *The RADIANCE Lighting Simulation and Rendering System*. Computer Graphics Proceedings, 1994, ACM