Monitoring Data Archives for Grid Environments

Jason Lee, Dan Gunter, Martin Stoufer, Brian Tierney
Lawrence Berkeley National Laboratory

Abstract

Developers and users of high-performance distributed systems often observe performance problems such as unexpectedly low throughput or high latency. To determine the source of these performance problems, detailed end-to-end monitoring data from applications, networks, operating systems, and hardware must be correlated across time and space. Researchers need to be able to view and compare this very detailed monitoring data from a variety of angles. To address this problem, we propose a relational monitoring data archive that is designed to efficiently handle high-volume streams of monitoring data. In this paper we present an instrumentation and monitoring event archive service that can be used to collect and aggregate detailed end-to-end monitoring information from distributed applications. This archive service is designed to be scalable and fault tolerant. We also show how the archive is based on the "Grid Monitoring Architecture" defined by the Global Grid Forum.

1.0 Introduction

Developers and users of high-performance distributed systems often observe unexpected performance problems. It can be difficult to track down the cause of these performance problems because of the complex and often indirect interactions between the many distributed system components. Bottlenecks can occur in any of the components through which the data flows: the applications, the operating systems, the device drivers, the network interfaces, and/or in network hardware such as switches and routers.

In previous work we have shown that detailed application monitoring is extremely useful for both performance analysis and application debugging [30][2][29]. Consider the use-case of monitoring some of the High Energy Physics (HEP) Grid projects [21][14][9] in a *Data Grid* environment. These projects, which will handle hundreds of terabytes of data, require detailed instrumentation data to understand and optimize their data transfers. For example, the user of a Grid File Replication service [3][4] notices that generating new replicas is taking much longer than it did last week. The user has no idea why performance has changed -- is it the network, disk, end host, GridFTP server, GridFTP client, or some other Grid middleware such as the authentication or authorization system?

To determine what changed, one needs to analyze monitoring data from hosts (CPU, memory, disk), networks (bandwidth, latency, route), and the FTP client and server programs. Depending upon the application and systems being analyzed, from days to months of historical data may be needed. Sudden changes in performance might be correlated to other recent changes, or may turn out to occur periodically in the past. In order to spot trends and interactions, the user needs to be able to view the entire dataset from many different perspectives.

A relational database that supports SQL [25] is an excellent tool for this type of task. SQL provides a general and powerful language for extracting data. For example, with SQL we can do queries such as:

- find the average throughput for the past 100 runs
- return all events for application runs that coincided with reports of network errors
- return all events for application runs where the throughput dropped below 10 Mbits/sec and CPU load was over 90%
- return all host and network events during application runs that took over 30 minutes
- return all events for application runs that failed (reported an error or never completed) during the last week
- return all events for applications runs where the total execution time was more than 50% from the average time for the past month

0-7695-1524-X/02 $17.00 (c) 2002 IEEE

Over the past two years the Global Grid Forum's Grid Performance Working Group [12] has worked to define the "Grid Monitoring Architecture" (GMA) [28], which describes the major components of a Grid monitoring system and their essential interactions. In this paper we show how the archive uses the GMA "producer" and "consumer" interfaces to allow users to active the monitoring and retrieve the data.

We also address the scalability issues inherent to aggregating monitoring data in a central archiving component. The archive must be able to easily handle high-speed bursts of instrumentation results, in order to avoid becoming a bottleneck precisely when the system is most loaded.

2.0 Related Work

There are several application monitoring systems, such as Pablo [22], AIMS [32], and Paradyn [19]. However these systems do not contain archival components. One of the first papers to discuss the use of relational databases for application monitoring was by Snodgrass [24], who developed a relational approach to monitoring complex systems by storing the information processed by a monitor into a historical database. The Global Grid Forum Relational Database Information Services research group is advocating the use of relational models for storing monitoring data, and this group has produced a number of documents, such as [5][10] and [8].

A current project that includes a monitoring archive is the *Prophesy* performance database [31]. Prophesy collects detailed pre-defined monitoring information at a function call level. The data is summarized in memory and sent to the archive when the program completes. This means that *Prophesy* does not need to be concerned with efficient transfer of the results. In contrast, our system for analyzing distributed applications, called *NetLogger* (and described below), provides a toolkit for user-defined instrumentation at arbitrary granularity. Typically this generates far too much data to hold in memory, so the data must be streamed to a file or socket. This means that our archive architecture must handle much more data that the Prophesy system. In addition, Prophesy includes modeling and prediction components, where our system does not.

The Network Weather Service team is currently adding an archive to their system, and the data model we are using, described below, is derived from the NWS data model described in [26]. There are several other monitoring systems that are based on the Global Grid Forum's GMA, including CODE [23] and R-GMA [11]. R-GMA contains an archive component, but does not appear to be designed to handle large amounts of application monitoring data. Spitfire [16] is a web service front-end to relational databases, which could potentially be used for an event archive.

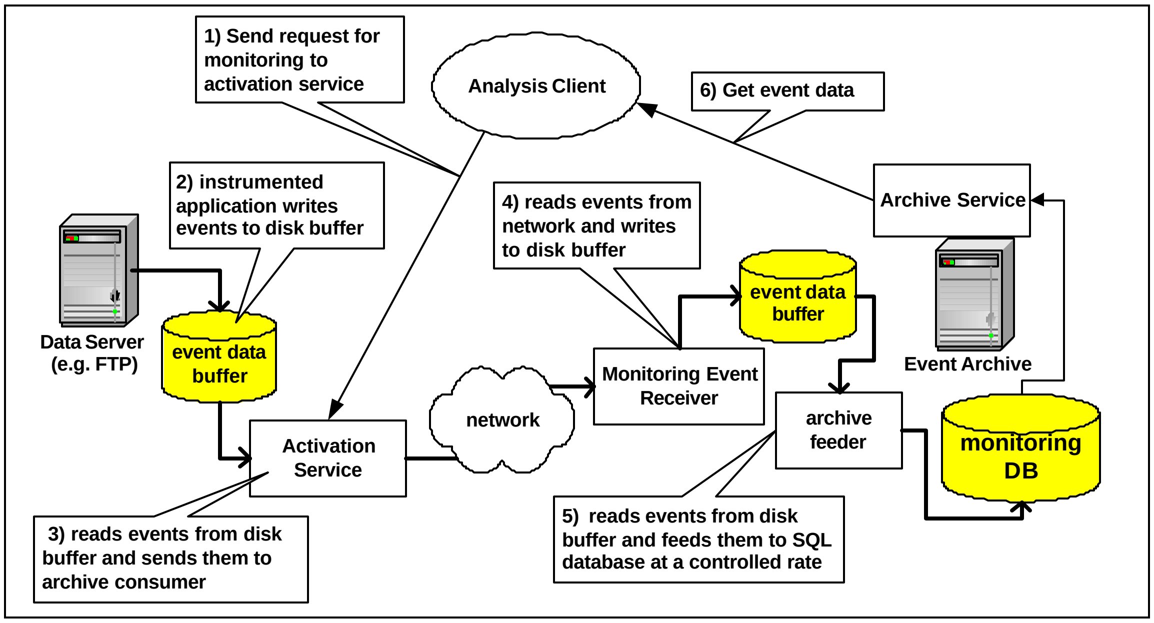
1) Send request for monitoring to activation service
Analysis Client
6) Get event data
2) instrumented application writes events to disk buffer
4) reads events from network and writes to disk buffer
Archive Service
Data Server (e.g. FTP)
event data buffer
event data buffer
Event Archive
Monitoring Event Receiver
network
archive feeder
Activation Service
monitoring DB
3) reads events from disk buffer and sends them to archive consumer
5) reads events from disk buffer and feeds them to SQL database at a controlled rate

Figure 1: Monitoring Archive System Components

3.0 Monitoring Components

The system described in this paper has four main monitoring components: the *application instrumentation*, which produces the monitoring data; the *monitoring activation service*, which triggers instrumentation, collects the events, and sends them to the requested destinations; the *monitoring event receiver*, which consumes the monitoring data and converts the events to SQL records and writes them to a disk buffer; and the *archive feeder*, which loads the SQL records into an event archive database. These components are illustrated in Figure 1. A previous paper focused on the first two components [15], while in this paper, we focus on the last two components (section 5), and on how the archive can be used for distributed system analysis.

In order for a monitoring system to be scalable and capable of handling large amounts of application event data, none of the components can cause the pipeline to “block” while processing the data, as this could cause the application to block while trying to send the monitoring to the next component. For example, instrumenting an FTP server requires the generation of 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.

Depending on the runtime environment, there are several potential bottlenecks points in the flow of event data. For example, a bottleneck might exist on the network when sending monitoring events from the producer to archive host. Another likely bottleneck is the insertion of events into the event archive database. To avoid blocking, the system must impedance-match slow data “sinks” with fast data “sources” by buffering data to disk at all bottleneck locations, as shown in Figure 2. This is similar to the approach taken by the Kangaroo system for copying data files [27].

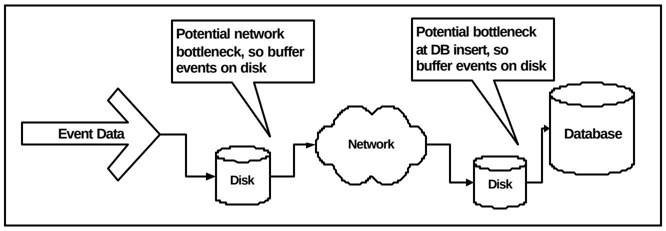
Potential network bottleneck, so buffer events on disk
Potential bottleneck at DB insert, so buffer events on disk
Event Data
Disk
Network
Disk
Database

Figure 2: Bottleneck Points

Therefore this system can handle a higher *burst* data rate than *sustained* data rate. If the sustained data rate is faster than the slowest component then the disk buffers will eventually fill and the pipeline will block. However very detailed monitoring information, such as before and after each I/O operation, is typically only needed occasionally. For example, when a user wants to explore a particular performance issue. Most of the time coarse summary data is sufficient. In other words, you don’t need to dust for fingerprints until a crime (in this case, the “crime” of poor performance) has been committed. Using our monitoring archive architecture, the slower components will not block the pipeline, but only add some latency as the data waits in a disk buffer for processing.

4.0 Previous Work

The system described in this paper is built upon two components: GMA and NetLogger. The GMA provides a high-level framework, characterizing system components as “consumers” or “producers” that can search for each other, and subscribe or query for data. NetLogger provides the plumbing for the system, sending timestamped items of data, or *events*, efficiently and reliably between components. In this section, we describe NetLogger and GMA and their relationship to each other.

4.1 NetLogger Toolkit

At Lawrence Berkeley National Lab we have developed the *NetLogger Toolkit* [30], 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.

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 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, an 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 the critical points in the code, then links the application with the NetLogger library. We have found that for this type of distributed systems analysis, clock synchronization of roughly 1 millisecond is required, and that the NTP [20] tools that ship with most Unix systems (e.g.: *ntpd*) can provide this level of synchronization.

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

Figure3 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. CPU usage and TCP Retransmission data are logged along with application events. Each related set of events, or *lifeline*, represents one block of data, and one can easily see that occasionally a large amount of time is spent between *Server_Send_Start* and *Client_Read_Start*, which is the network data transfer time. From this plot it is easy to see that these delays are due to TCP retransmission errors on the network (see *BytesRetrans* in the figure). 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.

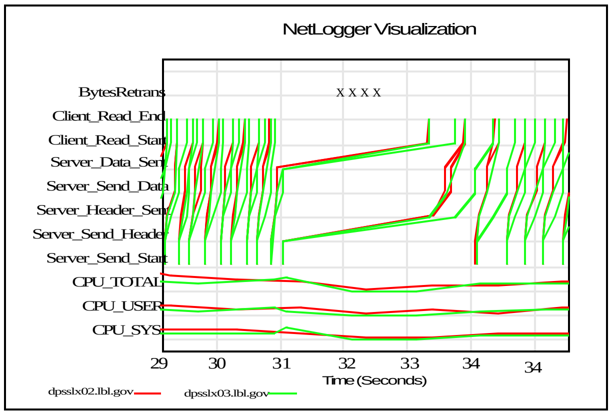

Figure 3: Sample NetLogger Results

We recently added to NetLogger an efficient self-describing binary wire format, capable of handling over 600,000 events per second [15]. This means that we can handle over 6000 events per second with only a one percent CPU perturbation of the application. We have also recently added several other new features to NetLogger, including an activation mechanism, and fault tolerance support.

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 special component called the activation service daemon via HTTP, 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.

Fault tolerance is provided through the NetLogger *reconnect* feature. Using a single API call, the user specifies a "backup" destination for the data, and a reconnect interval. If the primary NetLogger connection fails, the library will transparently begin writing data to the backup location and checking at the given interval to see if the primary connection comes back up. If the primary connection comes back up, NetLogger will start writing data there again, after (optionally) sending the backed-up data first.

These features help make NetLogger's event channel a robust, efficient, and flexible transport layer protocol for GMA, described below.

4.2 Grid Monitoring Architecture (GMA)

We have helped lead a Global Grid Forum (GGF) effort to defined a highly scalable architecture for Grid monitoring, called the *Grid Monitoring Architecture*, or GMA. This work has taken place in the GGF Performance and Information Area, which also has groups working to standardize the protocols and architectures for the management of a wide range of Grid monitoring information, including network monitoring.

The prime motivation of the GMA is the need to scalably handle dynamic performance information. In some models, such as the CORBA Event Service [6], all communication flows through a central component, which represents a potential bottleneck in distributed wide-area environments. In contrast, GMA performance monitoring data travels directly from the producers of the data to the consumers of the data. In this way, individual producer/consumer pairs can do "impedance matching" based on negotiated requirements, and the amount of data flowing through the system can be controlled in a precise and localized fashion. The design also allows for replication and reduction of event data at intermediate components acting as caches or filters.

In the GMA, the basic unit of monitoring data is called an *event*. An event is a named, timestamped, structure that may contain one or more items of data. This data may relate to one or more resources such as memory or network usage, or be application-specific data like the amount of time it took to multiply two matrices. The component that makes the event data available is called a *producer*, and a component that requests or accepts event data is called a *consumer*. A *directory service* is used to publish what event data is available and which producer to contact to request it.

The GMA architecture supports both a subscription model and a request/response model. In the former case, event data is streamed over a persistent "channel" that is established with an initial request. In the latter case, one item of event data is returned per request.

The GMA architecture has only three components: the producer, consumer, and directory service. This means that only three interfaces are needed to provide interoperability, as illustrated in Figure 4.

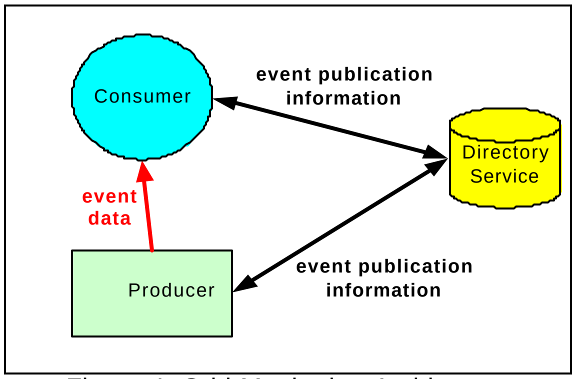

Figure 4: Grid Monitoring Architecture Components

The **directory service** contains only metadata about the performance events, and a mapping to their associated producers or consumers. In order to deliver high volumes of data and scale to many producers and consumers, the directory service is not responsible for the storage of event data itself.

A **consumer** requests and/or receives event data from a producer. In order to find a producer that provides desired events, the consumer can search the directory service. A consumer that passively accepts event data from a producer may register itself, and what events it is willing to accept, in the directory service.

A **producer** responds to consumer requests and/or sends event data to a consumer. A producer that accepts requests for event data will register itself and the events it is willing to provide in the directory service. In order to find a consumer that will accept events that it wishes to send, a producer can search the directory service.

A producer and consumer can be combined to make what is called a *producer/consumer pipe*. This can be used, for example, to filter or aggregate data. For example, a consumer might collect event data from several producers, and then use that data to generate a new derived event data type, which is then made available to other consumers, as shown in Figure5. More elaborate filtering, forwarding, and caching behaviors could be implemented by connecting multiple consumer/producer pipes.

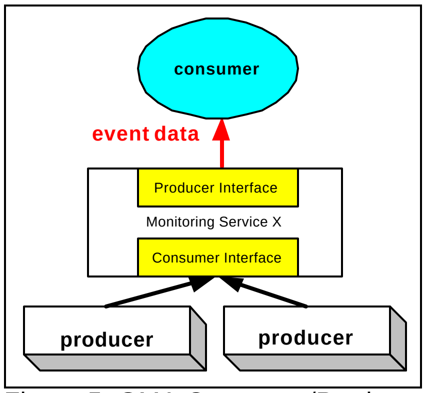

Figure 5: GMA Consumer/Producer Pipes

4.3 Combining NetLogger and GMA

In our implementation of the GMA, we have separated the control and data channels. SOAP messages are used for the control channel, and NetLogger serves as the "data channel" to transfer events between producers and consumers. Separating the data channel and control channel semantics is a standard mechanism used by programs like FTP to provide the ability to set low latency TCP options on the control channel, while setting high throughput TCP options on the data channel. The binary NetLogger wire format described in [15] is very easily parsed, providing an efficient data transport protocol for monitoring events. We use SOAP messages to exchange the subscribe, unsubscribe, or query parameters. SOAP is the emerging standard for exchanging messages in a Web Services environment, is quickly becoming the de-facto standard for transferring structured data, and as such seemed a good candidate for low bandwidth control messages.

4.4 Use of GMA in the Monitoring Archive

The GMA provides a common framework for structuring the interactions between the user and the activation service, event receiver, and archive service components (described in Section 3, and shown in Figure 1). In GMA terms, the activation service is a producer, the event receiver is a consumer, and the archive service is a producer. When the user is requesting activation from the activation service, or querying the archive service, the user is in the role of a GMA consumer.

To illustrate this, consider the process of sending monitoring events for a file transfer to the archive, illustrated in the top half of Figure 6. First the user searches the directory service for the appropriate archive and activation service (or this is configured manually). Then the user *subscribes* to the activation service, indicating that results should be sent to the archive. The user is a *consumer* asking for events from the activation service *producer*, and directing the results to another *consumer*, the archive. The activation service will then send monitoring data directly to the archive. The archive can then *register* in the directory service some metadata about these events.

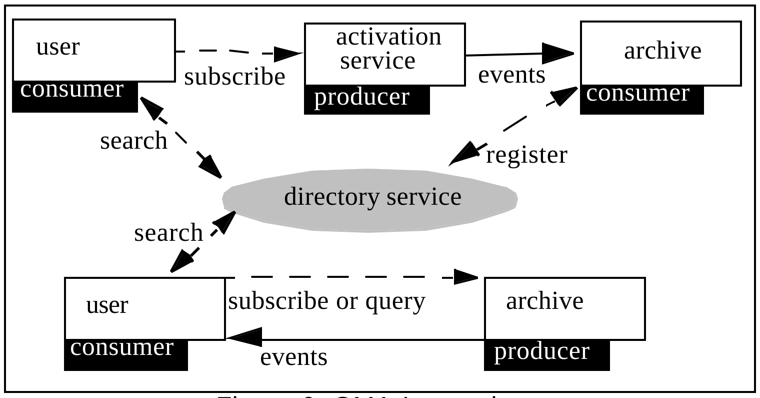

Figure 6: GMA Interactions

The bottom half of Figure6 illustrates the process of retrieving data from the event archive. First the user searches the directory service for an archive containing events of interest. The user is again a consumer, but this time the archive is a producer of events. Then the user does a *query* to the event archive, embedding, for example, some SQL statements into the request, and receives their desired information as a response.

Using the GMA interfaces, we can provide a coherent framework for the series of interactions necessary to activate, archive, and retrieve event data.

5.0 Monitoring Event Receiver and Archive Feeder

The *monitoring event receiver*, shown in Figure 1, reads monitoring events from the network and writes them, unparsed, to disk. The *archive feeder* then asynchronously parses these disk files and loads them into the database. This design has several advantages over a single component that loads the data as it arrives. First, the event receiver is extremely fast and light-weight, as it does little more than copy bytes off the network. Second, a partial or total failure of the database will not affect the event receiver as long as there is enough disk space to buffer the data until it is restarted. Finally, the system is easier to maintain because each component can be separately upgraded and restarted.

For efficiency, data is loaded in batches into the database using the SQL *load* command. In order to maintain good interactivity for database queries, we wanted to keep any individual database load, which can seriously impact query time, to take about one second. We ran a few tests and discovered that for our particular database setup, we need to break up the incoming data stream into disk files of 2500 events each. This value will vary based on the type of database on other type of hardware being used.

This setup allows us to maintain database interactivity regardless of the speed at which the data arrives. It also allows us to control server load, and provide a degree of fault tolerance, as all data will continue to be buffered on disk if the database server is down. This setup also allows scalability by simply adding more disk space as needed for the database and temporary files.

Even with the incoming data reduced to a trickle, the database will not be able to execute SQL queries or loads efficiently if the data model is inappropriate (or poorly implemented in the database tables). Our data model, described next, is simple but also performs well for common types of queries.

6.0 Data Model and Data Archive

Our data model, shown in Figure7, is based on NWS archive work [26]. It is very simple: we describe each *event* with a name, timestamp, “main” value, “target”, program name, and a variable number of “secondary” string or numeric values. The target consists of a source and destination IP address, although the destination address may be NULL. There is a many-to-one relationship from events to event *types*, and events to event *targets.* Therefore these *entities* can be put into separate indexed tables to allow fast mapping and searching of events. Figure8 shows sample events for TCP throughput (from *iperf*) and application monitoring (from a *GridFTP* server) represented in this model.

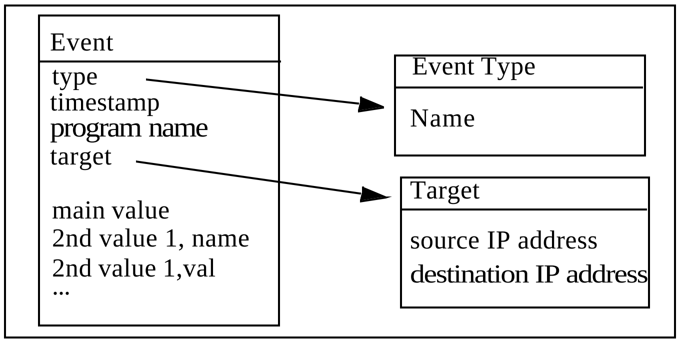

Figure 7: Event Data Model

We have optimized the actual database tables somewhat from the general model described above, both for database size and speed. These optimizations are based on common sense and an intuition of the most frequent types of queries. For example, the “secondary” values are subdivided into two tables, one for strings and one for numbers, because storing both in the same table would waste space and slow down indexing. We use the primary key in the event table to index against the string and number tables. This allows us to quickly reconstruct the original events.

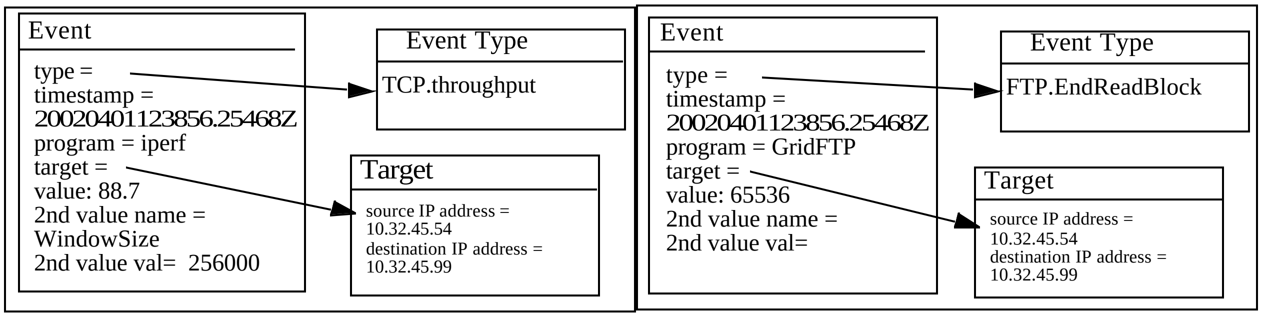

Figure 8: Example events (*Iperf* and *GridFTP*)

The usefulness of this data model hinges on the presence of globally unique, consistent names for each event of interest. It does not require that the events all fit into one unified schema namespace such as the Desktop Management Task Force Common Information Model (CIM) [7], which is a common data model of an implementation-neutral schema for describing overall management information in a network/enterprise environment. In our simple data model, it only matters that the event ‘TCP.throughput’ is not sometimes called ‘throughput-TCP’. There must be a one to one mapping of events to names. On the other hand there is no unified hierarchical which ties events together, which allows all events to be treated in a completely general manner. In other words, the monitoring events CPU.user and TCP.throughput are parsed and stored identically.

The unifying abstraction is the same one used by NetLogger: timestamped name-value pairs. It is no coincidence that the atomic events expected by the database have the same complexity as those generated by NetLogger. Although it is certainly possible to send and parse more complex data structures, our general approach is to break complex structures into “atomic events” of a single timestamp and value, and send and parse these atomic events as efficiently as possible. In monitoring and application instrumentation, the events of interest are generally simple enough that they can be modeled with only a few atomic events. Moreover, in this normalized form, the data can be transferred between databases, visualization tools, and filtering components, efficiently and easily.

7.0 Results

The following simple example demonstrates the power of a relational archive for analyzing monitoring data. Consider the case of unexplained periodic dips in network throughput. To understand the cause, we construct a query to find all events that happened during any time when the network throughput was below 1/5 of the average on that path. This query is simply a matter of extracting links with bandwidth measurements, which are already defined by a type "bandwidth" in the database. Then in SQL, using the *AVG()* command, we compute across these links the average of all the 'bandwidth' events. This now forms a baseline for link bandwidth. SQL can now supply us with all of the dips in the bandwidth over the time period of interest by performing a comparison of bandwidth values against this baseline. Finally, we extract all events within one minute of one of these bandwidth dips on both the sending and receiving hosts.

The results of these queries are graphed with *nlv*, the NetLogger analysis tool, shown in Figure9. Throughput values are from a long *Iperf* [17] test. This graph clearly shows that there was a spike of CPU usage at the same time the network throughput dropped. This indicates that the host was CPU bound, and not able to handle interrupts from the network adaptor fast enough to prevent a drop in throughput.

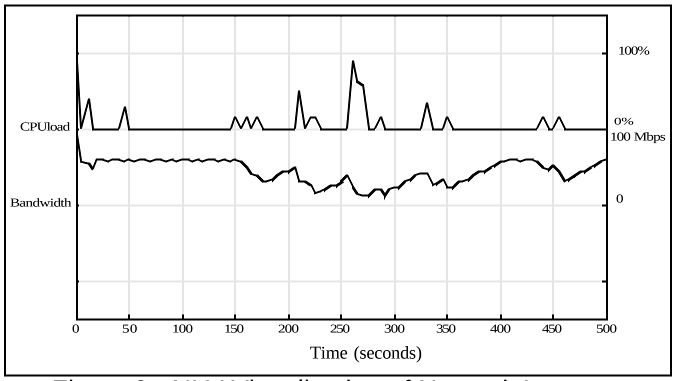

Figure 9: NLV Visualization of Network Loss vs. CPU Load

Another example involves a longer-term examination of *Iperf* throughput between a set of hosts. We used SQL to extract the minimum, maximum, mean, and standard deviation of the throughput over several days. We found a high standard deviation, and on further analysis discovered that the throughput had a bi-modal distribution, as shown in Figure 10. To see if the bi-modal distribution was due to TCP slow start issues on high bandwidth-delay product networks that Floyd describes in [12], we then queried the archive to extract information on all TCP retransmits that occurred during all *Iperf* runs. We then plotted the time of the first TCP retransmit together with the throughput of the run, shown in Figure11. In this plot we see that there appears to be at least some correlation between the time of the first retransmit and the overall throughput of the run. However this correlation does not appear to be strong enough to fully explain why *Iperf* performance is bi-modal, and further analysis is required.

These examples demonstrate some of the power of a relational archive of monitoring results, which allows one to explore issues such as this.

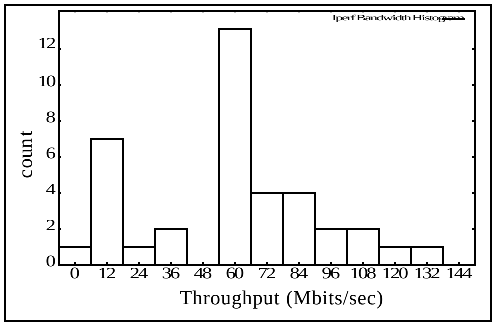

Figure 10: Histogram of *Iperf* throughput

Understanding GridFTP Behavior

Another example is an analysis of GridFTP performance data. We are monitoring and archiving the following information:

- GridFTP server: start and end events for file transfers, including all parameters associated with the transfer, such as file size, TCP buffer size, number of parallel streams, and so on.
- CPU and memory events, sampled every second during the duration of the FTP file transfers, on both the client and server.
- TCP data, from the web100 kernel's TCP Extended Statistics MIB [18], including the number TCP retransmissions, congestion window size, and round trip time. This data is also one-second samples during the duration of a file transfer.

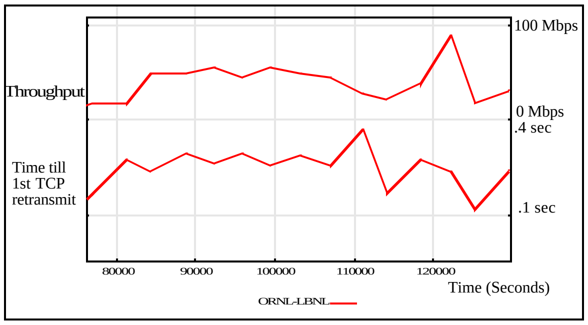

Figure 11: Time till 1st retransmit in *Iperf*

We then queried the archive for all information on FTP transfers between two hosts where the number of parallel streams and TCP buffer size were equivalent. We then looked for transfers that were "unusual", e.g., where the data throughput for the transfer was more than two standard deviations from the mean. We then requested from the archive all CPU, memory, and TCP information during those transfer intervals. Figure12 shows a subset of these results graphed in *nlv*. The upper part of the graph shows CPU idle time and the lower part shows GridFTP throughput. The dip in the graph indicates high CPU utilization. During this unusually high CPU activity on the client host, the FTP transfers had low throughput.

This example also shows the utility of a relational archive of monitoring results, which allows one to find colorations such as this.

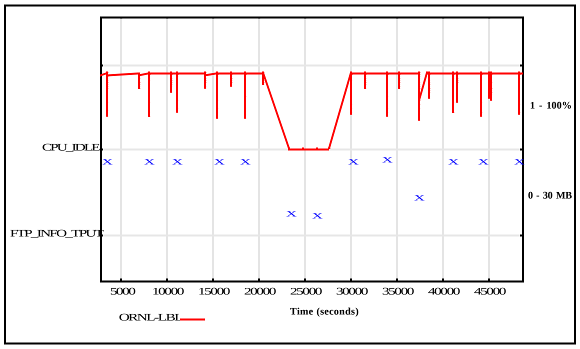

Figure 12: GridFTP and CPU usage

Scalability Tests

In order to test the scalability of the architecture, we ran the following test. We sent 26,000 events/second to the *monitoring event receiver* for one hour. During this time, a total of 4.5 GB (~100 million records) of events were buffered on disk, and these files took a total of 4 hours to load these events into the archive database. (These tests were done using a mySQL database on a 800 MHz Pentium system runing Linux 2.4.) As described above, we are loading only 2500 events at a time to ensure that database queries are not inordinately degraded. Clearly if you had the requirement of archiving 26,000 events per second on a regular basis, you would use a more powerful database system. However this demonstrates that this architecture can handle bursts of events at these rates without blocking and without affecting database query time.

8.0 Conclusion

In this paper we have explained how a relational monitoring event archive is useful for correlating events and understanding performance problems in distributed systems. A relational database with historical data allows for the establishment of a *baseline* of performance in a distributed environment, and finding events that deviate from this baseline is easy with SQL queries. We have also shown that an architecture built around the Grid Monitoring Architecture (GMA) can scale in a Grid environment. Our architecture addresses the scalability issues inherent to aggregating monitoring data in a central archiving component. This archive handles high-speed bursts of instrumentation results without becoming a bottleneck.

9.0 Acknowledgments

We want to thank members of the Global Grid Forum Discovery and Monitoring Event Description Working Group, especially Martin Swany, for their help in defining the data model we are using for monitoring events. 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-50216.

10.0 References

[1] 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.

[2] 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.

[3] Cancio, G., S. Fisher, T. Folkes, F. Giacomini, W. Hoschek, D. Kelsey, B. Tierney. The DataGrid Architecture. http://grid-atf.web.cern.ch/grid-atf/doc/architecture-2001-07-02.pdf

[4] Chervenak, A., et. al., *Giggle: A Framework for Constructing Scalable Replica Location Services*, Proceeding of the IEEE Supercomputing 2002 Conference, Nov. 2002.

[5] Coghlan, B., *A case for Relational GIS/GMA using Relaxed Consistency*, GGF Informational Draft GWD-GP-11-1, http://www.gridforum.org/1_GIS/RDIS.htm

[6] CORBA. Systems Management: Event Management Service. X/Open Document Number: P437, http://www.open-group.org/ onlinepubs/008356299/

[7] Desktop Management Task Force Common Information Model (CIM), http://www.dmtf.org/standards/standard_cim.php

[8] Dinda, P. and B. Plale. *A Unified Relational Approach to Grid Information Services*. Grid Forum Informational Draft GWD-GIS-012-1, http://www.gridforum.org/1_GIS/RDIS.htm

[9] European Data Grid Project http://www.eu-datagrid.org/

[10] Fisher, S., *Relational Model for Information and Monitoring*, GGF Informational Draft GWD-GP-7-1, http://www.gridforum.org/1_GIS/RDIS.htm

[11] Fisher, S. Relational Grid Monitoring Architecture Package, http://hepunx.rl.ac.uk/grid/wp3/releases.html

[12] Floyd, S., *Limited Slow-Start for TCP with Large Congestion Windows*, IETF draft, work in progress, May 2002. URL: http://www.icir.org/floyd/papers/draft-floyd-tcp-slowstart-00b.txt.

[13] Global Grid Forum (GGF): http://www.globalgridforum.org/

[14] GriPhyN Project: http://www.griphyn.org/

[15] 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, July 2002.

[16] Hoschek, W., G. McCance, *Grid Enabled Relational Database Middleware*, Global Grid Forum Informational Draft http://www.gridforum.org/1_GIS/RDIS.htm

[17] Iperf, NLANR, http://dast.nlanr.net/Projects/Iperf/

[18] Mathis, M., R. Reddy, J. Heffner and J. Saperia, TCP Extended Statistics MIB, IETF draft, February, 2002, http://www.ietf.org/internet-drafts/ draft-ietf-tsvwg-tcp-mib-extension-00.txt

[19] Miller, B., Callaghan, M., et al., *The Paradyn parallel performance measurement tools*, IEEE Computer, Vol. 28 (11), Nov. 1995.

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

[21] Particle Physics Data Grid (PPDG): http://www.ppdg.net/

[22] 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.

[23] Smith, W. *A Framework for Control and Observation in Distributed Environments*. NAS Technical Report Number: NAS-01-006, http://www.nas.nasa.gov/~wwsmith/

[24] Snodgrass, R., *A Relational Approach to Monitoring Complex Systems*, ACM Transactions on Computer Systems, Vol. 6, No. 2 (1988), 157-196.

[25] SQL. Database Language SQL. ANSI X3.135-1992

[26] Swany, M. and R. Wolski, *Representing Dynamic Performance Information in Grid Environments with the Network Weather Service*, Proceeding of the 2nd IEEE International Symposium on Cluster Computing and the Grid, Berlin, Germany, May 2002

[27] Thain, D., Jim Basney, Se-Chang Son, Miron Livny. *The Kangaroo Approach to Data Movement on the Grid*. Proceedings of the Tenth IEEE Symposium on High Performance Distributed Computing, San Francisco, California, August 2001

[28] Tierney, B., R. Aydt, D. Gunter, W. Smith, V. Taylor, R. Wolski, M. Swany. *A Grid Monitoring Service Architecture*. Global Grid Forum White Paper. http://www-didc.lbl.gov/GridPerf/

[29] 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.

[30] 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, http://www-didc.lbl.gov/NetLogger/

[31] Wu, X., Taylor, V., et. al., *Design and Development of Prophesy Performance Database for Distributed Scientific Applications*, Proc. the 10th SIAM Conference on Parallel Processing for Scientific Computing, Virginia, March 2001.

[32] Yan, L., Sarukkai, S., and Mehra, P., *Performance measurement, visualization and modeling of parallel and distributed programs using the AIMS toolkit*, Software Practice and Experience, Vol. 25 (4), April 1995.