A Monitoring Sensor Management System for Grid Environments

Brian Tierney, Brian Crowley, Dan Gunter, Jason Lee, Mary Thompson

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

Abstract

Large distributed systems such as Computational Grids require a large amount of monitoring data be collected for a variety of tasks such as fault detection, performance analysis, performance tuning, performance prediction, and scheduling. Ensuring that all necessary monitoring is turned on and that data is being collected can be a very tedious and error-prone task. We have developed an agent-based system to automate the execution of monitoring sensors and the collection of event data.

1.0 Introduction

The ability to monitor and manage distributed computing components is critical for enabling high-performance distributed computing. Monitoring data is needed to determine the source of performance problems and to tune the system for better performance. Fault detection and recovery mechanisms need monitoring data to determine if a server is down, and whether to restart the server or redirect service requests elsewhere. A performance prediction service might use monitoring data as inputs for a prediction model [26], which would in turn be used by a scheduler to determine which resources to use.

As distributed systems such as Computational Grids [9] become bigger, more complex, and more widely distributed, it becomes important that this monitoring and management be automated. Our approach to this problem is to use monitoring agents [5] in an event management framework that is designed for the requirements of a Grid environment.

1.1 Monitoring Agents

For this discussion, a monitoring *agent* is an autonomous, adaptable entity that is capable of monitoring and managing distributed system components. Our system is based on a collection of such agents, which can be updated or reconfigured on the fly. Agents work with *events*; we use the term event here to mean time-stamped data about the state of any system component, such as current CPU usage, or whether a server is no longer running.

Agents perform a number of tasks. They can securely start monitoring programs on any host, and manage their resulting data. They can provide a standard interface to host monitoring sensors for CPU load, interrupt rate, TCP retransmissions, TCP window size, and so on. Agents can also independently perform various administrative tasks, such as restarting servers or monitoring processes.

This agent-based monitoring architecture is sufficiently general that it could be adapted for use in distributed environments other than the Grid. For example, it could be used in large compute farms or clusters that require constant monitoring to ensure all nodes are running correctly.

1.2 Usage Scenarios

There are many uses for monitoring data and a monitoring data management system within a Grid environment. These include:

- fault detection and analysis: Monitoring data can be used to determine if a job is still running, or has died or hung. Monitoring data can also be used to help determine the cause of the fault.
- job progress monitoring: Users of long running jobs often want to query for the current status of their job. If the job produced the right level of monitoring data, the status could be determined. Job status information could also be graphed, analyzed, or archived.
- detailed performance analysis: Determining the performance bottlenecks in a complex distributed system requires a large amount of precision timestamped monitoring data.

- network-aware applications: With access to the right network monitoring information, applications can be self-tuning or self-configuring, modifying themselves to optimally use the available resources.
- data replication services: Monitoring data is required by data replication services to be able to automatically migrate data to the “best” location.
- scheduling and prediction services: Monitoring data is required by grid scheduling systems to determine the optimal resources to use for a given job.
- auditing systems: Monitoring data is required for accounting purposes, to verify resource utilization, and to verify you are getting the level of service you are paying for.
- configuration monitoring: A monitoring system could be used to automatically maintain databases of current hardware and software resources. Most current resource databases are generated by hand, are almost always out of date. Using real-time monitoring data to update the database ensures that the databases are accurate.

2.0 JAMM Monitoring System

The automated agent-based architecture that we have developed is called Java Agents for Monitoring and Management (JAMM). The agents, whose implementation is based on Java and Java Remote Method Invocation (RMI), can be used to launch a wide range of system and network monitoring tools, and then to extract, summarize, and publish the results. The JAMM system is designed to facilitate the execution of monitoring programs, such as *netstat*, *iostat*, and *vmstat*, by triggering or adapting their execution based on actual client usage. On-demand monitoring reduces the total amount of data collected, which in turn simplifies data management. For example, an FTP client connecting to an FTP server could automatically trigger *netstat* and *vmstat* monitoring on both the client and server for the duration of the connection. Application activity is detected by a *port monitor agent* running on the client and server hosts, which monitors traffic on a configurable set of ports.

We use the JAMM system to generate monitoring data for analysis by NetLogger [24], a toolkit for performance analysis of distributed systems. When performing a NetLogger analysis of a complex distributed system such as a Computational Grid, activating all desired host and network monitoring can be a very tedious task, and may require an account on the server host. The JAMM system greatly facilitates this task.

2.1 JAMM Architecture

The JAMM architecture uses a producer/consumer model, similar to several existing Event Service systems, such as the CORBA Event Service [3]. Both client push (query) and client pull (streaming) modes are supported. In CORBA terms, the “event channel” is embedded in the producer of the data, which is responsible for multiplexing/demultiplexing events. JAMM sensors publish their existence in a directory service. Clients (consumers) look up the sensors in the directory service, and then subscribe to sensor data via an event gateway (producer). The JAMM system consists of the following components, shown in Figure 1, and described in detail below:

- sensors
- sensor managers
- event gateways
- directory service
- event consumers
- event archives

There are many existing systems with an event model similar to the JAMM model. The CORBA Event Service [3] supports independent server and client push/pull of events using standard CORBA object invocation protocols. The specification for the OMG standard CORBA Notification Service adds to the Event Service structured events, differentiated quality-of-service, and most significantly, the ability for the event consumer to send the event supplier a filter. However, as of this writing this standard is new and most major vendors do not have complete implementations. JINI has a “Distributed Event Specification” [13], which is a simple specification for how an object in one Java virtual machine (JVM) registers interest in an event occurring in an object in some other JVM, and in how it receives notification when that event occurs. There are several systems with alternative event models, such as the Common Component Architecture; many of which are summarized in [18].

JAMM’s goals are also similar to some SNMP-based [2] monitoring packages such as HP Openview or SGI’s Performance Co-Pilot [19]. However, SNMP is designed for managing and monitoring network attached devices and hosts, not distributed systems. Integrating application monitoring and arbitrary sensors into an SNMP-based system would be quite difficult.

We believe that none of the previously described systems are a perfect match for a Grid monitoring system. Therefore we have tried to combine the relevant strengths of each.

Other related systems include the Pablo system [4][17], which has had the notion of application *sensors* for several years. Wolski et al. [27] also describe an architecture with similar characteristics to the service described here.

2.2 JAMM Components

sensors

The JAMM system is designed to control the execution of *sensors*. A sensor is any program that generates a time-stamped performance monitoring event. For example, we have sensors that monitor CPU usage, memory usage, and network usage. Sensors are also used to monitor "error" conditions, such as a server process crashing, or CRC errors on a router.

We have implemented the following types of sensors:

- **host sensors:** These sensors perform host monitoring tasks, such as monitoring CPU load, available memory, or TCP retransmissions. The monitoring can be configured to run all the time, or be triggered by detecting network activity on a specified port. Host sensors may be layered on top of SNMP-based tools, and therefore run remotely from the host being monitored.
- **network sensors:** These sensors perform SNMP queries to a network device, typically a router or switch. Information on which device statistics are being monitored is published in the directory service.
- **process sensors**: Process sensors generate events when there is a change in process status (for example, when it starts, dies normally, or dies abnormally). They might also generate an event if some dynamic threshold is reached (for example, if the average number of users over a certain time period exceeds a given threshold).
- **application sensors**: Autonomous sensors can also be embedded inside of applications. These sensors might generate events if a static threshold is reached (for example, if the number of locks taken exceeds a threshold), upon user connect/disconnect or change of password, upon receipt of a UNIX signal, or upon any other user-defined event. Application sensors can also be used to collect detailed monitoring data about the application to be used for performance analysis. These types of sensors would not be directly under JAMM control, but could still feed their results to the JAMM system.

sensor manager

The sensor manager agent is responsible for starting and stopping the sensors, and keeping the sensor directory up to date. Sensors to be run are specified by a configuration file, which may be local or on a remote HTTP server. Sensors can be configured to run always, when requested by a sensor manager GUI, or when requested by the port monitor agent. There is typically one sensor manager per host.

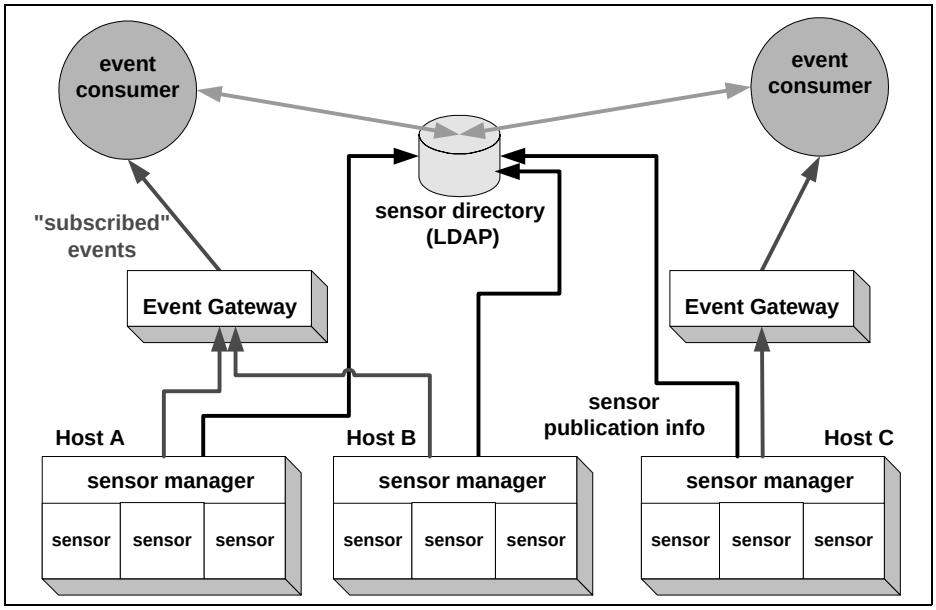

Figure 1: JAMM Components

An important component of the JAMM sensor manager is the *port monitor* agent. This agent monitors traffic on specified ports, and starts sensors only when network traffic on that port is detected. Using the port monitor agent, one is able to customize which sensors are run based on which applications are currently active, assuming that the applications use well-known ports. For example, we can turn on network monitoring when network intensive applications are running, CPU monitoring when CPU intensive applications run, and memory monitoring when memory intensive applications run. This assumes that the applications get triggered via a message from the network, which is usually true for Grid applications.

The port monitor has proven itself to be a very useful component, greatly reducing the total amount of monitoring data that must be collected and managed. It also notably improves the readability of real time analysis monitoring tools by not cluttering them up with uninteresting data.

event gateway

Event gateways are responsible for listening for requests from event consumers. Event gateways can service "streaming" or "query" requests from consumers. In streaming mode the consumer opens an event channel and the events are returned in a stream. In query mode the consumer does not open an event channel, but only requests the most recent event.

Event gateways can accept several types of requests from consumers. The consumer may request all event data, or only to be notified of certain types of events. For example the *netstat* sensor may output the value of the TCP retransmission counter every second, but most consumers only want to be notified when the counter changes, and not every second.

A consumer can also request that an event be sent only if it's value crosses a certain threshold. Examples of such a threshold would be if CPU load becomes greater than 50%, or if load changes by more than 20%. The event gateway can also be configured to compute summary data. For example, it can compute 1, 10, and 60 minute averages of CPU usage, and make this information available to consumers.

The event gateways can also be used to provide access control to the sensors, allowing different access to different classes of users. Some sites may only allow internal access to real-time sensor streams, with only summary data being available off-site. These types of policy would be enforced by the gateways. This mechanism is especially important for monitoring clusters or computer farms, where there may be a large amount of internal monitoring, but only a limited amount of monitoring data accessible to the Grid.

directory service

The directory service is used to publish the location of all sensors and their associated gateway. This allows consumers to discover which sensors are currently active, and which gateway to contact to subscribe to a given sensor's output. Query-optimized directory services such as LDAP, Globus MDS [6], the Legion Information Base, and the Novell NDS, all provide the necessary base functionality for this. We are currently using LDAP, because it is a simple, standard solution. LDAP servers can be hierarchical, with referrals to other LDAP servers which contain the directory service information for each site. LDAP also supports the notion of replicated servers, providing fault tolerance. Replication is critical to JAMM. Otherwise, failure of the sensor directory server could take down the entire system.

Current implementations of LDAP servers are optimized for read access, and do not work well in an environment with many updates.

However, it is possible to use only the naming and communications portions of LDAP, without the underlying database. For example, the Globus [7] system uses its own optimized database underneath the LDAP communications protocol to improve the performance of updates.

We are also interested in exploring the "event notification" service of LDAPv3 [25] as soon as it is available. This service lets a client register interest in an entry (i.e., sensor running) with the LDAP server, and LDAP will notify the client when that entry becomes available or is updated.

event consumers

An event consumer is any program that requests data from a sensor. There are many possible types of consumers. The current JAMM system includes the following:

- **event collector:** This consumer is used to collect monitoring data in real time for use by real-time analysis tools. It checks the directory service to see what data is available, and then "subscribes", via the event gateway, to all the sensors it is interested in. The gateway forwards the event data to the consumer as it is generated. Data from many sensors, as well as streams of data from application sensors, is then merged into a file for use by programs such as *nlv*, the NetLogger real-time event visualization tool [24].

- **archiver agent:** This consumer is used to collect data for an archive service. It subscribes to the logging agents, collects the event data, and places it in the archive. It also creates an archive directory service entry indicating the contents of the archive.
- **process monitor**: This consumer can be used to trigger an action based on an event from a server process. For example, it might run a script to restart the processes, send email to a system administrator, or call a pager.
- **overview monitor**: This consumer collects information from sensors on several hosts, and uses the combined information to make some decision that could not be made on the basis of data from only one host. For example, one may want to trigger a page to a system administrator at 2 A.M. only if both the primary and backup servers are down.

event archives

It is important to archive event data in order to provide the ability to do historical analysis of system performance, and determine when/where changes occurred. While it may not be desirable to archive all monitoring data, it is necessary to archive a good sampling of both “normal” and “abnormal” system operation, so that when problems arise it is possible to compare the current system to a previously working system. Archives might also be used by performance prediction systems, such as the Network Weather Service (NWS) [26].

The JAMM architecture provides a flexible method for selecting what gets archived, because the archive is just another consumer. In some environments very little will be monitored, and in others, it may be desirable to archive everything. With this approach the archive need not get in the way of any real-time monitoring.

2.3 Scalability Issues

One of the biggest issues in defining a monitoring architecture for use in a Grid environment is scalability. It is critical that the act of monitoring does not affect the systems being monitored. In the JAMM model, one can add additional event gateways, and additional sensor directories as needed, reducing the load where necessary. In the case where many consumers are requesting the same event data, the use of an event gateway reduces the amount of work on and the amount of network traffic from the host being monitored. An event gateway would typically be run on a separate host from the grid resources, to ensure that the load from the gateway did not affect what was being monitored.

In the JAMM architecture, event data is not sent anywhere unless it is requested by a consumer. All communication travels directly from the producer of the event data to their consumer. In this way, “impedance matching” between the number of components which must deliver data and the amount of data flowing through the system can be controlled in a precise and localized fashion. Only publication and metadata about the available information are placed in a central directory.

3.0 JAMM Implementation

The JAMM sensor managers, event gateways, and some of the consumers are implemented as Java Activatable Remote Method Invocation (RMI) objects. RMI provides a number of features, including the ability to add, remove, or reconfigure sensors on the fly. RMI also facilitates code development by making the network communication transparent to the programmer.

Activatable RMI objects can be loaded and run simply by invoking one of their methods, and will unload themselves automatically after a period of inactivity. RMI objects can be dynamically downloaded from an HTTP server every time the RMI daemon is restarted, making software updates trivial, and easing code deployment. The use of Java also facilitates porting the agents to a large number of platforms in a heterogeneous Grid environment.

JAMM event data is delivered in ULM (Universal Logger Message) [1] format, a simple ASCII-based format which is used by NetLogger. Extensible markup language (XML) support is also planned, but we are awaiting further progress in standardizing event schemas from the Performance Working Group of the GridForum [8]. We are also looking into adding a binary format option for high throughput event data that can not tolerate the parsing overhead of ASCII formats. We hope that the use of standard formats like XML will make it possible for JAMM to interoperate with other monitoring sensors on the Grid.

We are also considering supplementing RMI with Sun's Java Management Extension (JMX) [14]. JMX has a good communication model and a rich set of SNMP tools, and requires less memory than our current RMI-based implementation. However, the only good implementation of JMX currently is a commercial product from Sun, JDMK [12], which may not be affordable in an academic environment.

4.0 NetLogger and JAMM

JAMM is often used to collect monitoring events for use with the *NetLogger Toolkit* [24]. NetLogger (short for Networked Application Logger) 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.

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.

Using NetLogger, 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.

4.1 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.

4.2 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 by default uses six digits of accuracy, allowing for microsecond precision in the timestamp. Here is a sample NetLogger ULM event:

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

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

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.

4.3 Clock 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) [16], such as the *xntpd* 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.

4.4 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.

4.5 Event log visualization and analysis

We have found exploratory, visual analysis of the log event data to be the most useful means of determining the causes of performance anomalies. The NetLogger Visualization tool, *nlv*, has been developed to provide a flexible and interactive graphical representation of system-level and application-level events. *nlv* uses three types of graph primitives to represent different events. These are shown in Figure 2.

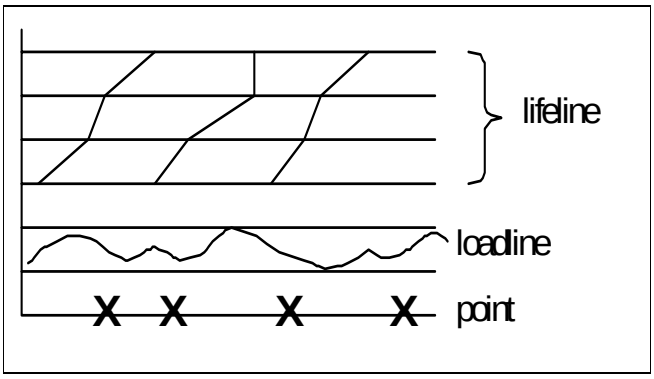

Figure 2: nlv Graph Primitives

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.

The NetLogger client API allows the application to specify an “object ID” for related events, which the NetLogger visualization tools then use to generate an object lifeline. 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 a scatter plot.

Generation of a scatter plot was useful, for instance, to show the distribution of “bytes read” from individual low-level calls to the operating system’s *read()* function. The results, shown in Figure 3, represent each *read()* with a point primitive scaled from zero(0) to the maximum requested bytes on the Y-axis, with time as usual on the X-axis. This graph makes apparent the (unexpected) clustering of the data around two distinct values.

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

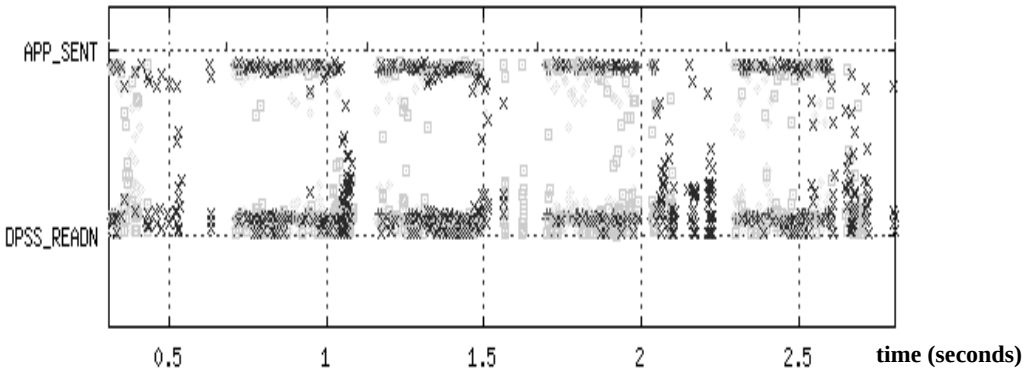

Figure 3: Scatter plot with point primitive

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.

5.0 JAMM Usage

An example use of JAMM is shown in Figure 4. Monitoring data is collected at both the client and server host, and at all network routers between them. All event data is sent to a real-time monitor consumer for real-time visualization and NetLogger analysis. Server and router data is also sent to the archive.

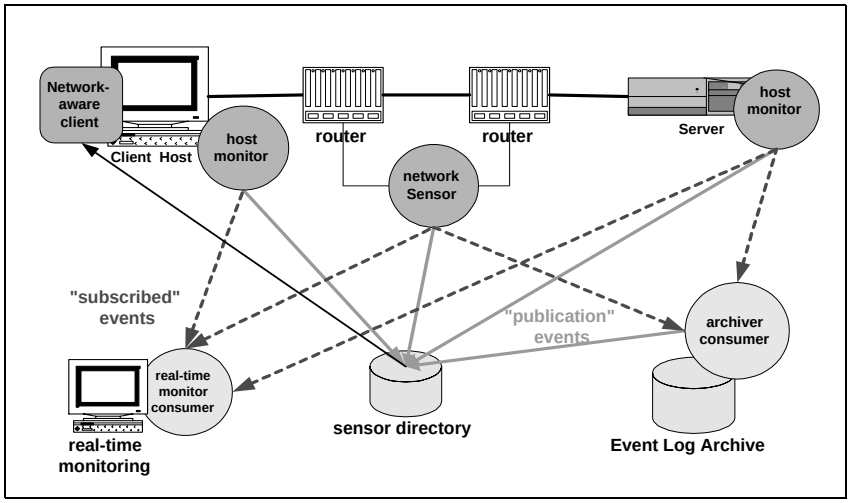

Figure 4: Sample JAMM Usage

Adding new sensors to JAMM is quite simple. If the sensor is an implementation of a JAMM sensor in Java, one just copies the Java class to an HTTP accessible directory and edits the central configuration file. Every few minutes the sensor managers check for updates to the configuration file, and activate new sensors if necessary, publishing them in the sensor directory. If the sensor is not Java-based, it will have to be copied to each host by some other means. Converting existing monitoring tools to JAMM sensors is quite easy using the JAMM Perl sensor modules and the JAMM Java sensor class.

There are various GUI's to facilitate the use of the JAMM system. The JAMM Sensor Data GUI lists all sensors stored in a specific LDAP server, and displays their current status, including such details as frequency, duration, startup time, current number of consumers, and last message. The JAMM Sensor Control GUI facilitates the startup or re-initialization of any available sensors on any JAMM managed hosts. The port monitor also has a GUI client that can do things such as reconfigure the type of monitoring to be done when a port is active, or add a new port of interest to the port monitor. There are also applets that make information produced by JAMM available through a browser by means of tables, charts, and graphs. These are useful in day-to-day system administration, in addition being used to help with performance analysis.

6.0 Results

As an example of how JAMM may be used to monitor a Grid application, we describe how it is used in the DARPA MATISSE project [15], an NGI project whose goal is to enable MEMS (micro-electro-mechanical systems) researchers to efficiently access, manipulate, and view high resolution high frame rate video data of MEMS devices remotely over the DARPA Supernet [20], an OC 48 (2.4 Gbits/second) NGI testbed network.

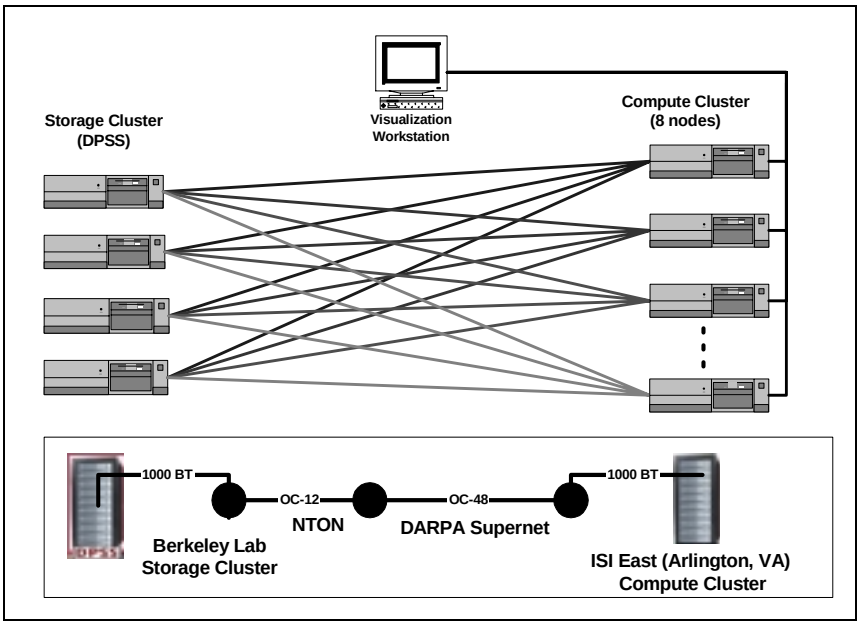

Figure 5: Matisse Application Environment

A demonstration of the Matisse application was performed in May 2000 in Arlington, VA, the configuration of which is shown in Figure 5. Data was stored on a Distributed Parallel Storage System (DPSS) [23] at LBNL in Berkeley, CA. Data was transferred on-demand across Supernet to a Linux compute cluster, which did the data analysis, and then sent the results to a workstation.

JAMM was used to monitor all system components, and verify that all required hardware and software was running properly. JAMM was also used to do real-time performance monitoring and analysis, and proved to be very helpful in tracking down some performance problems.

The following sensors, shown in Figure 6, were used: CPU and memory sensors on every host, DPSS processes monitors, clock synchronization monitors, TCP monitors (retransmits and window size), and network switch / router monitors. The TCP sensor we are using is a version of *tcpdump* modified to generate NetLogger events when it detects a TCP retransmission or a change in window size [21].

Performance from the point of view of the client was quite bursty. Sometimes images arrived at 6 frames/sec, and other times only 1-2 frames/sec. As is typical in this type of environment, it was not clear what the source of the problem was. Was the source of the burstiness the data server, the compute server, the receiving host, the network, or the viewing application? To do the performance analysis, an event collector agent was started, and sensors on all components in use by the application were located in the directory service and subscribed to. This agent collected all the monitoring data, which was then used as input to *nlv*, the NetLogger visualization tool. *nlv* results are shown in Figure 7.

The graph shows time on the X axis, and events on the Y axis. The vertical sloping lines show key events from a trace of a data frame request through the entire system. The less steep the line is, the more time a particular event took. The graph also shows the results of the CPU, memory, and TCP sensors. Note the correlation between the TCP retransmit events and the large gap with no data being received by the application. Also of interest is the high level of system CPU usage on the receiving host, shown by the line for event VMSTAT_SYS_TIME. From this we were able to narrow down the problem to either the network or the receiving host.

The next question was why were there TCP retransmissions? SNMP errors on the end switches and routers were also monitored by JAMM, but no errors were reported. This, along with the high system CPU load, pointed us to the receiving host as the probable source of the problems.

The client was reading data from four DPSS servers, so we next used the *Iperf* network performance test tool [11] to compare TCP performance of a single TCP input stream versus four parallel streams. To our surprise the aggregate

throughput for four streams was only 30 Mbits/sec compared to 140 Mbits/sec for a single stream. This again pointed to the receiving host as the source of the problem. By using a single DPSS server instead of four servers, (and thus one data socket instead of four), we were able to increase the throughput to 140 Mbits/sec. The system CPU load with only one data socket was much lower as well.

We are not yet certain why the performance using four sockets is so much worse than using one socket, yet we believe it has something to do with the amount of load the gigabit ethernet card and device driver place on the system. Interestingly, this behavior is only observed with wide-area transfers; LAN throughput for both one and four data streams are 200 Mbits/second. We plan to perform more experiments to track down the cause of this behavior.

Note that this type of analysis could have been done without the use of JAMM, but would have been much more difficult. One would need to have an account on every system, with superuser privileges (to run the *tcpdump* sensor), and log into every system (13 in this example) and start every sensor by hand, and then copy the results to one place for analysis. Clearly this is more work than most application users are willing to do.

Using JAMM, all that is required is for the application user to start up a consumer and subscribe to the relevant sensor data. JAMM also simplifies the job of a system administrator to configure all the necessary sensors. The NetLogger tools make analysis of this data quite straightforward.

7.0 Current Status and Future Work

At the present time all basic functionality of the main JAMM components (sensor manager, port monitor, event gateway, sensor directory, event collector, and several sensors) is complete. We are currently adding the ability to do filtering and summary data to the event gateway, and working on security, access control, and policy mechanisms as well. We are also developing a ULM to XML filter for the gateway, so a consumer can request either format for event data.

We are also developing a summary data service and client API, as shown in Figure 6. For example, network sensors publish summary throughput and latency data in the directory service, which is used by a “network-aware” client [23] to optimally set its TCP buffer size. The summary data service might be part of the sensor directory, could be a separate LDAP server, or could be built into the gateways. We are exploring the performance and scalability issues of each of these approaches.

7.1 Security Issues

A distributed agent system such as JAMM creates a number of security vulnerabilities which must be analyzed and addressed before such a system can be safely deployed on a production Grid. The users of such a system are likely to be remote from the machines being monitored and to belong to different organizations. Users want to find out what sensors are running and how to subscribe to their event data; users may need to cause sensor programs to be started or to generate a higher level of data collection; and finally users want to subscribe to sensor data via an event gateway. In each case the domain that is being monitored is likely to want to control which users may perform which actions. Discovering what sensors are running and what their gateways are is done via an LDAP lookup. Starting new sensors is

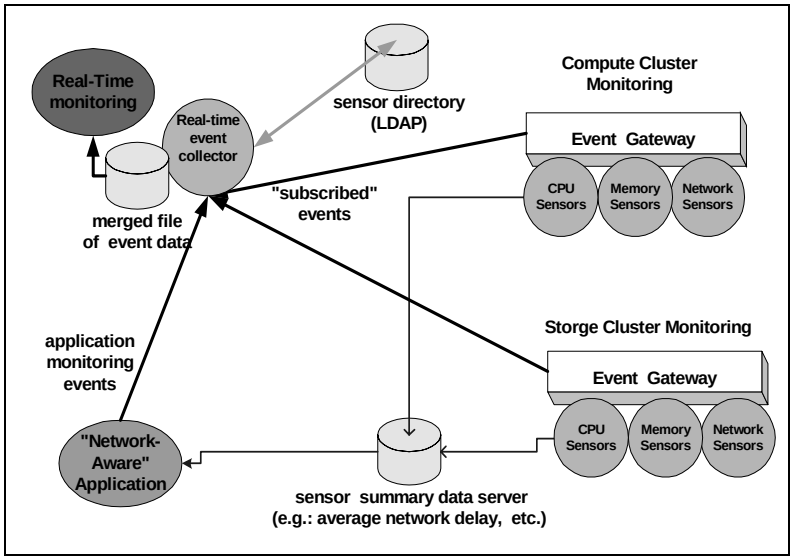

Figure 6: Use of JAMM with the Matisse Application

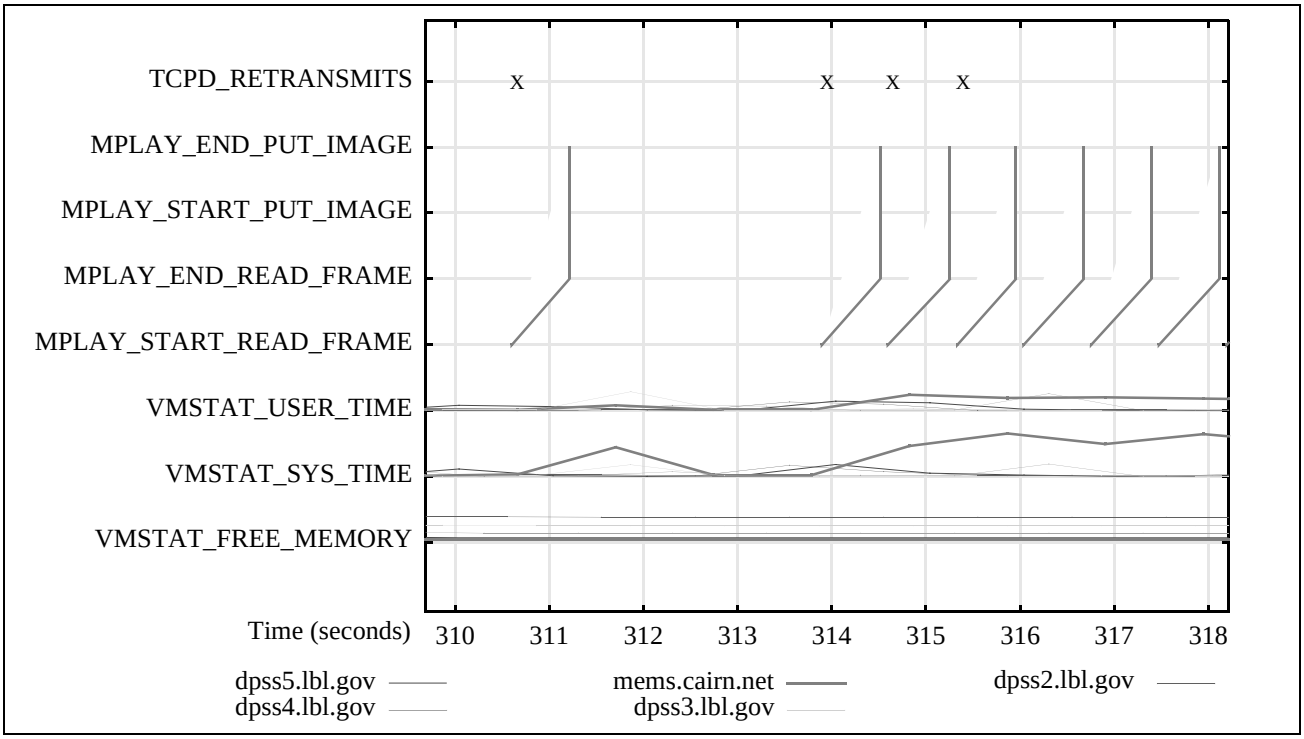

Figure 7: NetLogger real time analysis of JAMM managed Sensor data

done by a request to a gateway, which then contacts a sensor manager. Subscribing to an event stream is done by establishing a network connection to a event gateway. Our goal is to provide a single method for user identification and authorization for each of these steps.

Current LDAP servers provide user/password style protection to regions of the LDAP tree which can be used to protect the lookup and publishing functions. Normally these passwords are sent in clear text, but there are SSL-enabled versions of LDAP, e.g. Netscape, where the LDAP server has a key which can be used to encrypt the connection to the client. Each LDAP server manages its own set of users and passwords. This same user/password scheme could be used to control the other access points. However, this approach becomes awkward if the sensors that a user is interested in are in multiple domains, because each domain must assign user names and passwords.

Public key based X.509 identity certificates [10] are a recognized solution for cross-realm identification of users. When the certificate is presented through a secure protocol such as SSL (Secure Socket Layer), the server side can be assured that the connection is indeed to the legitimate user named in the certificate. SSL libraries exist for HTTP connections, RMI connections, or simply as C or Java libraries to be used with socket code.

There are two existing packages that use identity credentials for client side authentication and authorization for remote resources. One is the Globus implementation of the GSS-API (GSI) [9], which uses a protocol such as SSL, to provide the server side with assurance that the connection is to the legitimate user named in the certificate. A server side map file is used to map the Globus X.509 user identities to local user-ids which can be used by existing access control mechanisms.

The second existing package is Akenti [22], which uses vanilla X.509 identity certificates and SSL protocol to authenticate remote users. In addition, Akenti provides a way for the resource stakeholders to remotely determine the authorization for resource use based on components of the users distinguished name or attribute certificates. These two mechanisms can be combined to allow Globus clients in a Grid environment to present Globus proxy ids, and non-Globus clients to provide standard X.509 identity certificates. At the server side, a domain can decide if they want to use locally maintained access control lists or the more distributed Akenti policy certificates.

User (consumer) access at each of the points mentioned above (LDAP lookup and subscription to a gateway), would require an identity certificate passed though a secure protocol, e.g. SSL. A wrapper to the LDAP server and the gateway could both call the same authorization interface with the user's identity and the name of the resource the user wants to access. This authorization interface could return a list of allowed actions, or simply deny access if the user is unauthorized.

Communication between the gateway and the sensor managers also needs to be controlled, so that a malicious user can't communicate directly with the sensor manager. This is a simpler problem since a sensor manager only needs to

communicate with a small known set of gateway agents and thus can just have a list of the Identity Certificates for each agent to which it will allow a connection.

We plan to add credential based security to the JAMM system in the near future.

8.0 Conclusions

Monitoring is critical to providing a robust, high-performance Grid environment. We have presented a flexible, scalable architecture for managing monitoring sensors for all components of the Grid, including hosts, networks, and applications. The ability to access the event data collected by the monitoring sensors will enhance or enable a wide range of other Grid services, such as scheduling, network tuning, performance analysis, QoS, and so on.

9.0 Acknowledgments

We are greatly indebted to many members of the Grid Forum (http://www.gridforum.org), from whom many of the ideas in this paper came. In particular, discussions with Rich Wolski and Martin Swany, University of Tennessee, Ruth Aydt, University of Illinois, Ian Foster, Steve Tuecke, and Darcy Quesnel, Argonne National Lab, Dennis Gannon, University of Indiana, and Warren Smith, NASA Ames, all contributed to the architecture described here. We also thank Michael Amabile from the Sarnoff Corporation for helping collect the NetLoggerized MEMS data viewing application.

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-46847.

10.0 References

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

[2] Case, J., R. Mundy, D. Partain, B. Stewart, “Introduction to Version 3 of the Internet-standard Network Management Framework”, IETF RFC 2570, April 1999.

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

[4] DeRose, L., D. Reed, “SvPablo: A Multi-Language Architecture-Independent Performance Analysis System,” Proceedings of the International Conference on Parallel Processing (ICPP'99), Fukushima, Japan, September 1999.

[5] Genersereth, M., S. Ketchpel, “Software Agents”, Communications of the ACM, July, 1994.

[6] Fitzgerald, S., I. Foster, C. Kesselman, G. von Laszewski, W. Smith, and S. Tueke, “A Directory Service for Configuring High-Performance Distributed Computations”. In Proc. 6th IEEE Symp. on High Performance Distributed Computing, August 1997.

[7] Globus: http://www.globus.org

[8] Grid Forum “Grid Performance” Working Group: http://www-didc.lbl.gov/GridPerf/

[9] “The Grid: Blueprint for a New Computing Infrastructure”, edited by Ian Foster and Carl Kesselman. Morgan Kaufmann, Pub. August 1998. ISBN 1-55860-475-8.

[10] Housely, R., W. Ford, W. Polk, D. Solo, “Internet X.509 Public Key Infrastructure”, IETF RFC 2459. Jan. 1999

[11] Iperf: http://dast.nlanr.net/Projects/Iperf/index.html

[12] JDMK: http://www.sun.com/software/java-dynamic/

[13] “Jini Distributed Event Specification”, http://www.sun.com/jini/specs/

[14] JMX: http://java.sun.com/products/JavaManagement/

[15] Matisse: http://www.cnri.net/matisse/

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

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

[18] Peng, X, “Survey on Event Service”, http://www-unix.mcs.anl.gov/~peng/survey.html

[19] Performance Co-Pilot: http://oss.sgi.com/projects/pcp/

[20] Supernet: http://www.ngi-supernet.org/

[21] tcpdump: NetLogger version, http://www.ittc.ukans.edu /projects/enable/tcpdump/

[22] Thompson, M., W. Johnston, S. Mudumbai, G. Hoo, K. Jackson, A. Essiari, “Certificate-based Access Control for Widely Distributed Resources”, Proceedings of the Eighth Usenix Security Symposium, Aug. 1999.

[23] Tierney, B. J. Lee, B. Crowley, M. Holding, J. Hylton, F. Drake, “A Network-Aware Distributed Storage Cache for Data Intensive Environments”, Proceeding of IEEE High Performance Distributed Computing conference (HPDC-8), August 1999, LBNL-42896. http://www-didc.lbl.gov/DPSS/

[24] 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 conference, July 1998, LBNL-42611. http://www-didc.lbl.gov/NetLogger/

[25] Wahl M., T. Howes, S. Kille, “Lightweight Directory Access Protocol (v3)”, IETF RFC 2251, Dec. 1997.

[26] Wolski, R., N. Spring, J. Hayes, “The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing,” Future Generation Computing Systems, 1999. http://nsw.npaci.edu/

[27] Wolski, R., M. Swany, S. Fitzgerald, “White Paper: Developing a Dynamic Performance Information Infrastructure for Grid Systems”, http://dast.nlanr.net/ GridForum/Perf-WG/white.PDF