

White Paper: A Grid Monitoring Service Architecture (DRAFT)

*Brian Tierney, Ruth Ayd, Dan Gunter, Warren Smith,
Valerie Taylor, Rich Wolski, Martin Swany, and the
Grid Performance Working Group
Global Grid Forum*

Abstract

Large distributed systems such as Computational and Data Grids require a substantial amount of monitoring data be collected for a variety of tasks such as fault detection, performance analysis, performance tuning, performance prediction, and scheduling. Some tools are currently available and others are being developed for collecting and forwarding this data. The goal of this paper is to describe a common architecture with all the major components and their essential interactions in just enough detail that Grid Monitoring systems that follow the architecture described can easily devise common APIs and wire protocols. To aid implementation, we also discuss the performance characteristics of a Grid Monitoring system and identify areas that are critical to proper functioning of the system.

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 and application 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 [14][10]. A performance prediction service might use monitoring data as inputs for a prediction model [16], which would in turn be used by a scheduler to determine which resources to use.

There are several groups that are developing Grid monitoring systems to address this problem [11][16][9][14] and these groups have recently seen a need to interoperate. In order to facilitate this, we have developed an architecture of monitoring components. A Grid monitoring system is differentiated from a general monitoring system in that it must be scalable across wide-area networks, and include a wide range of heterogeneous resources. It must also be integrated with other Grid middleware in terms of naming and security issues. We believe the Grid Monitoring Architecture (GMA) described here addresses these concerns and is sufficiently general that it could be adapted for use in distributed environments other than the Grid. For example, it could be used with large compute farms or clusters that require constant monitoring to ensure all nodes are running correctly.

2.0 Design Considerations

With the potential for thousands of resources at geographically different sites and tens-of-thousands of simultaneous Grid users, it is important for the data management and collection facilities to scale while, at the same time, protecting the data from spoiling.

In order to allow scalability in both the administration and performance impact of such a system, the decision-making as to what is monitored, measurement frequency, and how the data is made available to the public must be widely distributed and dynamic. Thus, instead of a centralized management component, multiple independent management components synchronize their state through a directory service, which may itself be distributed. Distributing management in this fashion also helps minimize the effects of host and network failure, making the system more robust under precisely the kinds of conditions it is trying to detect.

In some models, such as the CORBA Event Service, all communication flows through a central component, which represents a potential bottleneck. In contrast, we propose that performance event data, which makes up the majority of the communication traffic, should travel 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 based on current load considerations. The design also allows for replication and reduction of event data at intermediate components acting as consumer/producer caches or filters. Use of these intermediate components lessens the load on producers of event data that is of interest to many consumers, with subsequent reductions in the network traffic, as the intermediaries can be placed “near” the data consumers. The directory service contains only metadata about the performance events and system components and is accessed relatively infrequently, reducing the chance that it would be a bottleneck.

We also considered a purely SNMP-based solution for monitoring, but rejected it because we felt that the SNMP’s simple GET/SET model is not rich enough, as there is no support for subscription. Also, it is not clear that security model maps well to the Grid Security Infrastructure. However, we definitely envision the use of SNMP-based tools as a source of monitoring data.

3.0 Architecture

The GMA architecture supports both a producer/consumer model, similar to several existing Event Service systems such as the CORBA Event Service [1], and a query/response model. For either model, producers or consumers that accept connections publish their existence in a directory service. Consumers use the directory service to locate one or more producers generating the type of event data they are interested in. Each consumer then subscribes to or queries the matching producer(s) directly. Likewise, a producer may query the directory service to locate consumer(s) that accept and process event data in a given manner – for example, a consumer that archives event data for later analysis. Once the appropriate consumer is identified, the producer would connect to it directly and stream the event data – similar in behavior to when a consumer subscribes to a producer, but initiated by the producer.

3.1 Terminology

The monitoring data that the GMA is designed to handle are timestamped *events*. An event is a named collection of data. The data may relate to anything, but common events will be memory usage, network usage, or “error” conditions such as a server process crashing. The *producer* is the component that makes the event data available. A *consumer* is any process that requests or accepts event data. A *directory service* is used to publish what event data is available and which producer to contact to get it. All of these components are described in detail below.

3.2 Components

The architecture consists of the following components, shown in Figure 1:

- consumers
- producers
- directory service

By defining three interfaces: the consumer to producer interface, the consumer to directory service interface, and the producer to directory service interface; we can build “standard” grid monitoring services that will all inter-operate.

Directory Service

To locate, name, and describe the structural characteristics of any data available to the Grid, a distributed directory service for publishing this information must be available. The primary purpose of this directory service is to allow information consumers (users, visualization tools, programs and resource schedulers) to discover and understand the characteristics of the information that is available. In addition, information producers must be

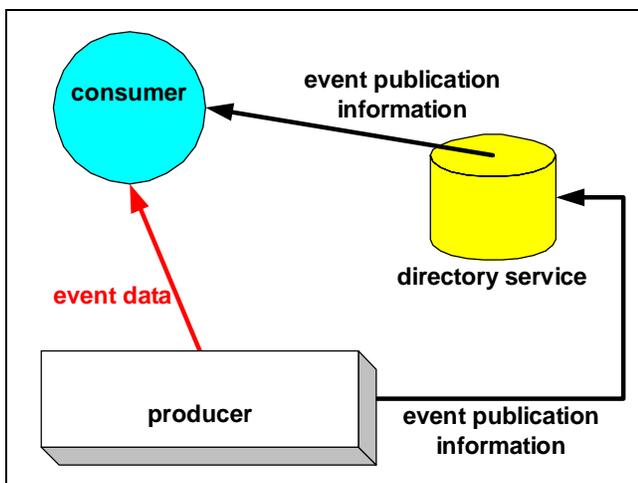


Figure 1: Grid Monitoring Architecture Components

able to update the information to reflect the system state. In the context of common operations for both consumers and producers, they will be collectively referred to as *clients*.

The directory service contains a listing of all available event data and their associated producers. This allows consumers to discover what event data are currently available (through the producer registration), what the characteristics of the data are, and which producer) to contact to receive a given type of event data.

The directory service, however, is not responsible for the storage of performance data itself—only its name and other characteristics. We assume the names and characteristics associated with dynamic performance data change slowly (unlike the performance data itself). That is, the name and structural characteristics of a data set remain relatively constant while the valid contents of the data set may change dramatically over time.

The functions supported by a directory service are:

1. Authorize-consumer – Establish identity of a consumer, which is in turn mapped to access permissions for the next, or possibly several subsequent, transaction(s).
2. Authorize-producer – <same as Authorize-consumer>, although different mechanisms may be used for the two authorization operations.
3. Search – Perform a search for event data. The client should indicate whether only one result, or more than one result, if available, should be returned. An optional extension would allow the client to get multiple results one element at a time using a “get next” query in subsequent searches.
 - Preconditions: The client is authorized to perform the search.
 - Postconditions: The result(s) of the search are returned, which includes a well-defined null value for searches which did not match in the directory.
4. Add – Add a record to the directory.
 - Preconditions: The client is authorized to add the record. The record conforms to the directory’s schema. The record is not a duplicate.
 - Postconditions: The record is in the directory.
5. Remove – Remove a record from the directory.
 - Preconditions: The client is authorized to remove the record. The record matches exactly one record in the directory.
 - Postconditions: The record is not in the directory.
6. Update – Change the state of a record in the directory.
 - Preconditions: The client is authorized to modify the record. The record matches exactly one record in the directory.
 - Postconditions: The record now has the new values.
7. Version request – A client may request the current version of the interface. The version numbering system is TBD.

Query-optimized directory services such as LDAP [15], Globus MDS [3], the Legion Information Base, and the Novell NDS, all provide the necessary base functionality for this service, but only in their fully distributed implementations. Some public-domain implementations of these services do not support distributed implementation.

consumer

A consumer is any program that receives event data from a producer. Consumers that will accept asynchronous requests from producers will publish this information in the directory service. The functions supported by a consumer are:

1. Authorize to producer – The consumer contacts a producer and proves its identity. This may need to be performed once per “session”, or on every request.
2. Authorize from producer - The consumer accepts authorization requests from a producer and verifies its identity. As in *Authorize to producer*, this may be done once per session or on every producer-initiated request.

3. Query – The consumer receives one event or set of events from the producer. Optional extensions are a *filter* to indicate interest in only a subset of events or to perform transformations on event data.
 - Preconditions: The consumer is authorized to receive these event(s). The event data is available.
 - Postconditions: One or more events are returned, together, in the reply.
4. Consumer-initiated Subscribe – The consumer establishes a connection to the producer to receive events in a stream.
 - Preconditions: The consumer is authorized to connect to the producer and receive these event(s). The event data is available.
 - Postconditions: Same as for Query, except that in addition to returning the most recent event, on success the producer will either (a) return events in a stream over the connection used for the request or (b) inform the consumer of the location of a new connection from which it can read the stream of events.
 - Other behaviors: If the consumer closes the established connection, the producer should simply consider the subscription ended (generating no errors). If the underlying source of event data stops producing data, the producer may close the connection without warning, so consumers should be designed to recover gracefully in this instance.
5. Consumer-initiated Unsubscribe – The consumer tells a producer to close the subscription. An optional extension is a “close all” version which closes all subscriptions for this consumer.
 - Preconditions: The subscription exists for the producer/consumer pair. The consumer is authorized to end it.
 - Postconditions: The subscription is removed. No more data should be sent for this subscription after the producer has confirmed.
6. Producer-initiated Subscribe - The consumer accepts subscriptions from producers who wish to send events.
 - Preconditions: The producer is authorized to send events to this consumer.
 - Postconditions: A new subscription is created for this producer/consumer pair.
7. Producer-initiated Unsubscribe - The consumer accepts an unsubscribe request from the producer.
 - Preconditions: The subscription exists. The producer is authorized to end it.
 - Postconditions: The subscription is removed.
8. Authorize to directory – The consumer contacts the directory service and proves its identity. This may need to be performed once per “session” or on every lookup.
9. Lookup – The consumer makes a query to the directory service, of which at least 2 types should be available: (1) producer: get data for a producer associated with an event. (2) event: get the description of the event.
 - Preconditions: Authorization has been performed.
 - Postconditions: The directory service is unchanged (read-only operation).
10. Update - The consumer updates records in the directory service regarding events for which this consumer will accept producer-initiated subscriptions.
 - Preconditions: Authorization has been performed.
 - Postconditions: The directory service has more/less/modified records reflecting the new information.

There are many possible types of consumers. These may include:

- **real-time monitor:** 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” to all the events it is interested in. The producers then send the event data to the consumer as it is generated. Data from many sources can then be used for real-time performance analysis.
- **archiver:** This consumer may be used as to collect data for the archive service. It subscribes to the producers, collects the event data, and places it in the archive. We note that a monitoring architecture needs this component, as 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 desirable 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. In this architecture, the archive is just another consumer.

- **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, call a pager, etc.
- **overview monitor:** This consumer collects events from several sources, 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.

producer

Producers are responsible for providing event data to consumers, either by request or asynchronously. Producers will publish event availability information in the directory service. The functions supported by a producer are:

1. Authorize from consumer – The producer establishes a consumer’s identity and access permissions. Authorization may be combined with subscription or query requests, or performed separately with the results stored in a shared “key” of some sort.
2. Authorize to consumer - The producer contacts a consumer and proves its identity. As for *Authorize to producer*, this may need to be performed once per session or on every new request.
3. Query – The producer returns a single set of event(s) in response to a consumer query.
 - Preconditions: The consumer is authorized to receive data about the event.
 - Postconditions: The event data, if present, is returned.
4. Consumer-initiated Subscribe – Accept consumer requests to establish a stream of event data (subscription). This request should include parameters and filters, etc.
 - Preconditions: Consumer is authorized to subscribe to requested event data.
 - Postconditions: The subscription is added for a consumer, and the producer either (a) returns events in a stream over the connection used for the request or (b) informs the consumer of the location of a new connection from which it can read the stream of events.
5. Consumer-initiated Unsubscribe – This is the normal operation by which a consumer ends its subscription. An optional “unsubscribe all” extension would allow the consumer to cancel all its subscriptions at once. As mentioned in the consumer section, if a consumer summarily closes its connection, the producer should automatically unsubscribe it everywhere.
 - Preconditions: The subscription exists for this producer/consumer pair.
 - Postconditions: The consumer/producer pair has one less subscription.
6. Producer-initiated Subscribe - A producer asynchronously begins a subscription with a consumer.
 - Preconditions: The producer is authorized to send data to the consumer.
 - Postconditions: The subscription is added and the producer may now send data.
7. Producer-initiated Unsubscribe - The producer informs a consumer that the subscription is ending.
 - Preconditions: The subscription exists for this consumer/producer pair. The consumer supports this function, allowing producers to asynchronously unsubscribe.
 - Postconditions: The subscription is removed. Note that even in the case of failure the subscription may be removed by the producer.
8. Version – A consumer may request the current version of the interface. The version numbering system is TBD.

Producers can service “streaming” or “query” requests from consumers. In streaming mode the consumer makes a single request, then receives events in a stream until an explicit action is taken to end the connection. In query mode the consumer makes a single request and receives a single event in reply.

The producers are also used to provide access control to the event data, allowing different access to different classes of users. Since Grids typically have multiple organizations controlling the resources being monitored there may be different access policies (firewalls possibly), support for different frequencies of measurement, and willingness to allow access to different performance details for consumers “inside” or “outside” of the organization running the resource. Some sites may allow internal access to real-time event streams, while providing only summary data off-site. The producers would enforce these types of policy decisions. 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.

There may also be components that are both consumers and producers. 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 Figure2.

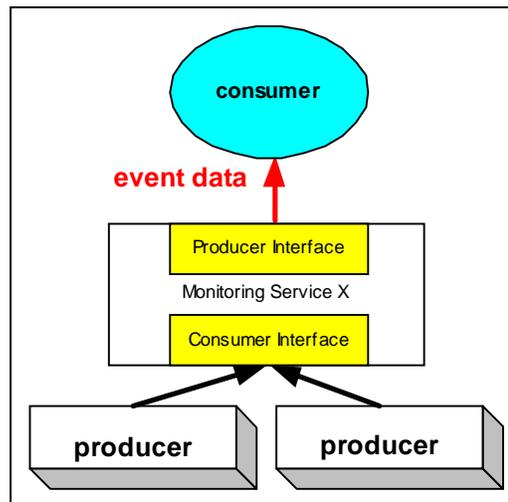


Figure 2: Joint Consumer/Producer

Sources of Event Data

There are many possible sources of event data, including monitoring *sensors*. The following is a summary of common types of sensors:

- **host sensors:** These sensors perform host monitoring tasks, such as monitoring CPU load, available memory, or TCP retransmissions. Host sensors may be layered on top of SNMP-based tools, and therefore run remotely from the host being monitored. Host sensors could also be used to monitor host configuration information, such as what versions of the operating system or other software packages are installed.
- **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 may not register themselves with the directory service, but could still feed their results to the system. A special case of application sensors would be library sensors that would be embedded in library code and compiled into the application.
- **storage or I/O sensors:** These sensors perform any monitoring of storage systems such as disks and tapes, obtaining information on block size, access time, seek time, etc.

- **middleware sensors:** These sensors would gather information about middleware services such as directory and authentication servers. They could report request volume, average service time, number of requests returned due to timeouts, etc. and would be used to discover and repair performance problems in this service layer.

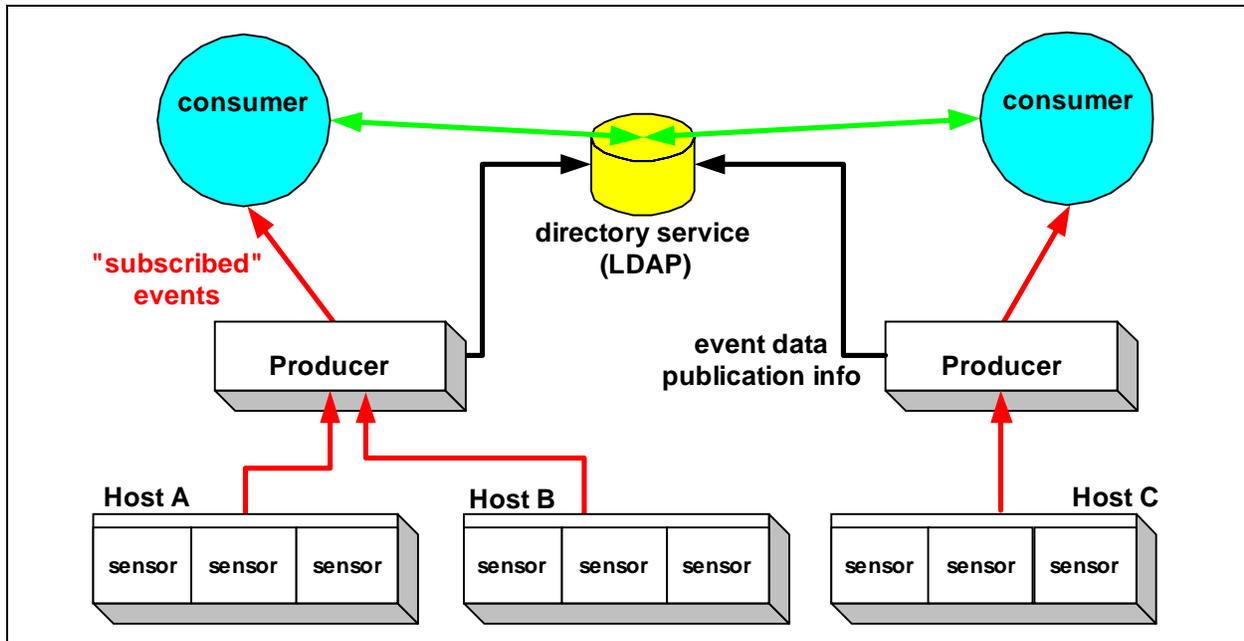


Figure 3: Relationship of Producers and Sensors

A producer may be associated with a single sensor, all sensors on a given host, all sensors on a given subnet, or any arbitrary group of sensors. This is not defined by the architecture, but is left as an implementation decision. Figure 3 shows one example of how this might be implemented. We note that there are scalability and reliability issues with how this is implemented, as described below.

Optional Producer Tasks

There are many other services that producers might provide, such as event filtering and caching. For example, producers could optionally perform any intermediate processing of the data the consumer might require. A consumer might request that a prediction model be applied to a measurement history from a particular sensor, and then be notified only if the predicted performance falls below a specified threshold. The producer might in this case filter the data for the consumer and deliver it according to the schedule the consumer determines. Another example is that a consumer might request that an event be sent only if its 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 producer might 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. Information on which services the producer provides would be published in the directory server, along with the event information.

Protocols

The next step is to define what the protocol for consumer to producer communication, and for consumer and producer to the directory service communication. For example, current proposals include using LDAP for communicating with the directory service, and SOAP for subscribe requests. These issues will be addressed in future Global Grid Forum Performance Working Group documents.

4.0 Sample Use

An example use of the GMA is shown in Figure 4. Event data is collected on each host and at all network routers between them, and aggregated at a producer, which registers the availability of the data in the directory service. A real-time monitoring consumer subscribes to all this event data for real-time visualization and performance analysis. The producer is capable of computing summaries of network throughput and latency data, enabling a “network-aware” client [11] to optimally set its TCP buffer size. A subset of the producer’s data, that from from the “server” and “router” nodes, is also sent to an archive.

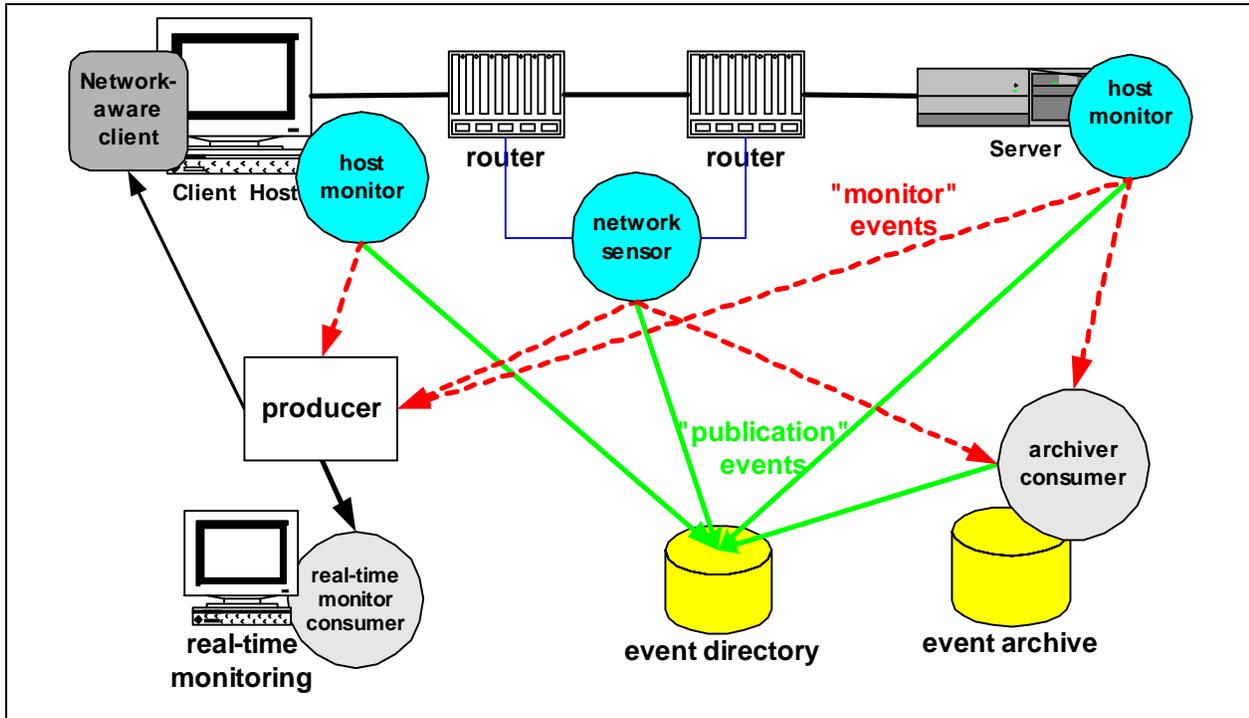


Figure 4: Sample Use of Monitoring System

5.0 Implementation Issues

The purpose of a monitoring system is to reliably deliver timely and accurate information without perturbing the system. Therefore the architecture must consider performance issues explicitly and make recommendations and requirements of implementations with the goal of avoiding services which look good on paper but which fail in practice. We discuss several of these implementation design issues below.

Monitoring service characteristics

The following characteristics distinguish performance monitoring information from other system data, such as files and databases.

Performance information has a fixed, often short lifetime of utility. Most monitoring data may go stale quickly making rapid read access important, but obviating the need for long-term storage. The notable exception to this is data that gets archived for accounting or post-mortem analysis.

- **Updates are frequent.** Unlike the more static forms of “metadata,” dynamic performance information is typically updated more frequently than it is read. Most extant information-base technologies are optimized for query and not update, making them potentially unsuitable for dynamic information storage.
- **Performance information is often stochastic.** It is frequently impossible to characterize the performance of a resource or an application component using a single value. Therefore, dynamic performance information may carry *quality-of-information* metrics quantifying its accuracy, distribution, lifetime, etc., which may need to be calculated from the raw data.

- **Data gathering and delivery mechanisms must be high-performance.** Because dynamic data may grow stale quickly, the data management system must minimize the elapsed time associated with storage and retrieval. Note that this requirement differentiates the problem of dynamic data management from the problem of providing an archival performance record. The elapsed time to read an archive, while important, is often not the driving design characteristic for the archival system. We believe that archival data is useful both for accounting purposes and for long-term trend analysis. It is our belief, however, the separate but complimentary systems for managing and archiving Grid performance data respectively are required, each tailored to meet its own set of unique performance constraints.
- **Performance measurement impact must be minimized.** There must be a way for monitoring facilities to be able to limit their intrusiveness to an acceptable fraction of the available resources. If no mechanism for managing performance monitors is provided, performance measurements may simply measure the load introduced by other performance monitors.

General Implementation Strategies

A number of the authors of this white paper have built various monitoring systems. The following lessons have been learned from this experience, and should be considered when implementing a monitoring system.

- **The data management system must adapt to changing performance conditions dynamically.** Dynamic performance data is often used to determine whether the shared Grid resources are performing well (e.g. fault diagnosis) or whether Grid load will admit a particular application (e.g. resource allocation and scheduling). To make an assessment of dynamic performance fluctuation available, the data management system cannot, itself, be rendered inoperable or inaccessible by the very fluctuations it seeks to capture. As such, the data management system must use the data it gathers to control its own execution and resources in the face of dynamically changing conditions.
- **Dynamic data cannot be managed under centralized control.** Having a single, centralized repository for dynamic data (however short its lifespan) causes two distinct performance problems. The first is that the centralized repository for information and/or control represents a single-point-of-failure for the entire system. If the monitoring system is to be used to detect network failure, and a network failure isolates a centralized controller from separate system components, it will be unable to fulfill its role. All components must be able to function when temporarily disconnected or unreachable due to network or host failure. For example, a producer must still be able to accept connections from consumers even if its connection to sensors or the directory server is down. In addition, once access is restored, producers must be able to reconfigure themselves automatically with respect to the rest of the running service components. A second problem with centralized data management is that it forms a performance bottleneck. For dynamic data, writes often outnumber reads. That is, performance data may be gathered that is never read or accessed since demand for the data cannot be predicted. Experience has shown that a centralized data repository simply cannot handle the load generated by actively monitored resources at Grid scales.
- **All system components must be able to control their intrusiveness on the resources they monitor.** Different resources experience varying amounts of sensitivity to the load introduced by monitoring. A two megabyte disk footprint may be insignificant within a 10 terabyte storage system, but extremely significant if implemented for a palm-top or RAM disk. In general, performance monitors and other system components must have tunable CPU, communication, memory, and storage requirements.
- **Efficient data formats are critical.** In choosing a data format, there are trade offs between ease-of-use and compactness. While the easiest and most portable format may be ASCII text including both event item descriptions and event item data in each transmission, this also the least compact. This format may be suitable for cases where a small amount of data is recorded and transmitted infrequently. However, some sources of event data can generate huge volumes of data in a short amount of time, demanding that a more efficient data format be adopted. Compressed binary representations that can be read on machines with different byte orders is one possibility. Transmitting only the item data values and using a data structure obtained separately to interpret the data is another way to reduce the data volume. XML is an emerging standard that allows the data description to be separated from the data values. The XML schema could be placed in a separate directory server, retrieved, and used in conjunction with the

event data values. Another possibility is to send the data descriptor one time when a consumer subscribes to a producer, and send only the data values for each event transmission. The GMA could support registration of a data format for each event, allowing different events to use the format most appropriate for their needs. Consumers could be provided plug-in modules to convert from one format to another.

Scalability

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 has minimal affect on the systems being monitored. In this model, one can add additional producers and additional directory servers as needed, reducing the load where necessary. In the case where many consumers are requesting the same event data, the use of a producer reduces the amount of work on and the amount of network traffic from the host being monitored. As such, the resources that a producer will use must, themselves, be scheduled. A producer might be run on a separate host from the Grid resources, to ensure that the load from the producer did not affect what was being monitored.

In particular, we believe that the GMA is more scalable than the CORBA Event Service. In the GMA, event data is not sent anywhere unless it is requested by a consumer. Many of the current event service systems, including CORBA, send all event data to a central component, which consumers then contact. In the GMA, only event data subscription information (i.e.: which producer to contact) is sent to a central directory server. Event data goes directly from producer to consumer. We believe this model will scale much better in a Grid environment.

In addition, for the GMA system to scale, performance monitoring consumers (particularly those that require the cooperation between two or more producers) must coordinate their interactions to control intrusiveness. For example, if network performance is to be monitored between all pairs of hosts attached to a single Ethernet segment, the network probes required to generate end-to-end measurements cannot occur simultaneously. If they do, both the quality of the readings that are gathered and the network capacity that is available for other work will suffer. If performance monitors are not coordinated in the Grid, the intrusiveness of performance monitoring may strongly impact available performance, particularly as the system scales. That is, if all performance facilities operate their own monitoring sensors, Grid resources will be consumed by the monitoring facilities alone. Coordinating a Grid-wide collection of sensors is complicated both by the scale of the problem (there are many Grid resource characteristics to monitor) and by the dynamically changing performance and availability of Grid resources that are being used to implement the dynamic data management service.

One recommended producer service that is important for system scalability is that of consumer-specified caching. Often a consumer needs to access only a small subset of the global data pool, and will sacrifice fast access for tight data consistency. An automatic program scheduler, for example, might want the “freshest” data that can be delivered for a specified set of hosts with no more than a one second access delay. To achieve this functionality at Grid scales, producers must cache the data the consumer will want and deliver whatever data is available at the time of request. Experience with dynamic program scheduling indicates that this type of producer is valuable to scalable performance within the Grid [2].

Security Issues

A distributed system such as this 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.

Typical user actions will include queries to the directory service concerning event data availability, subscriptions to producers to receive event data, and requests to instantiate new event monitors or to adjust collection parameters on existing monitors. In each case, the domain that is being monitored is likely to want to control which users may perform which actions.

Public key based X.509 identity certificates [6] 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.

User (consumer) access at each of the points mentioned above (directory lookup and requests to a producer), would require an identity certificate passed through a secure protocol, e.g. SSL. A wrapper to the directory server and the producer 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 producer and the sensors may also need to be controlled, so that a malicious user can not communicate directly with the monitoring process.

6.0 Related Work

There are many existing systems with an event model similar to the one described here. CORBA includes an "event service" [1] that has a rich set of features, including the ability to push or pull events, and the ability for the consumer to pass a filter to the event supplier. JINI also has a "Distributed Event Specification" [7], which is a simple specification for how an object in one Java™ virtual machine (JVM) registers interest in the occurrence an event occurring in an object in some other JVM, and then receives a notification when that event occurs. There are also several other systems with alternative event models, such as the Common Component Architecture; many of which are summarized in [8]. However, we believe that none of the existing systems is a perfect match for a Grid monitoring system; therefore we have tried to combine the relevant strengths of each. Another related system is Autopilot [9], which has had the notion of *sensors* for several years, and which implements a similar publish/lookup/subscribe architecture. Note that this list of systems is not intended to be exhaustive, but only illustrative of the usefulness of the proposed architecture.

7.0 Acknowledgements

Input from many people went into this document, including almost all attendees of the various Grid Forum meetings. The LBNL portion of this paper was supported by the U. S. Dept. of Energy, Office of Science, Office of Computational and Technology Research, Mathematical, Information, and Computational Sciences Division, under contract DE-AC03-76SF00098 with the University of California.

8.0 References

- [1] CORBA, “Systems Management: Event Management Service”, X/Open Document Number: P437, <http://www.opengroup.org/onlinepubs/008356299/>
- [2] Dail, H, G. Obertelli, F. Berman, R. Wolski, and A. Grimshaw “Application-Aware Scheduling of a Magnetohydrodynamics Application in the Legion Metasystem”, Proceedings of the 9th Heterogeneous Computing Workshop, May 2000.
- [3] S. Fitzgerald, I. Foster, C. Kesselman, G. von Laszewski, W. Smith, and S. Tuecke, “A Directory Service for Configuring High-Performance Distributed Computations”. In *Proceedings 6th IEEE Symposium on High Performance Distributed Computing, August 1997*.
- [4] The Globus project: See <http://www.globus.org>
- [5] 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.
- [6] Housely, R., W. Ford, W. Polk, D. Solo, “Internet X.509 Public Key Infrastructure”, IETF RFC 2459. Jan. 1999
- [7] Jini Distributed Event Specification”, <http://www.sun.com/jini/specs/>
- [8] Peng, X, “Survey on Event Service”, <http://www-unix.mcs.anl.gov/~peng/survey.html>
- [9] Randy L. Ribler, Jeffrey S. Vetter, Huseyin Simitci, and Daniel A. Reed, “Autopilot: Adaptive Control of Distributed Applications,” Proceedings of the 7th IEEE Symposium on High-Performance Distributed Computing, Chicago, IL, July 1998.
- [10] W. Smith, “Monitoring and Fault Management,” http://www.nas.nasa.gov/~wsmith/mon_fm
- [11] Tierney, B., B. Crowley, D. Gunter, M. Holding, J. Lee, M. Thompson A Monitoring Sensor Management System for Grid Environments Proceedings of the IEEE High Performance Distributed Computing conference (HPDC-9), August 2000, LBNL-45260.
- [12] Tierney, B. Lee, J., Crowley, B., Holding, M., Hylton, J., Drake, F., “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/>
- [13] 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/>
- [14] A. Waheed, W. Smith, J. George, J. Yan. “An Infrastructure for Monitoring and Management in Computational Grids.” In *Proceedings of the 2000 Conference on Languages, Compilers, and Runtime Systems*.
- [15] Wahl M., Howes, T., Kille S., “Lightweight Directory Access Protocol (v3)”, Available from <ftp://ftp.isi.edu/in-notes/rfc2251.txt>
- [16] Wolski, R., Spring, N., Hayes, J., “The Network Weather Services: A Distributed Resource Performance Forecasting Service for Metacomputing,” Future Generation Computing Systems, 1999. <http://nsw.npaci.edu/>