jGMA: A lightweight implementation of the Grid Monitoring Architecture

Mark Baker and **Matthew Grove**

Distributed Systems Group, University of Portsmouth, UK
[mark.baker@computer.org, matthew.grove@port.ac.uk]

Abstract

Wide-area distributed systems require scalable mechanisms that can be used to gather and distribute system information to a variety of endpoints. In this paper we report on jGMA, a Java-based implementation of the GGFs Grid Monitoring Architecture (GMA), that we have specifically designed to be compliant, standard-based and fulfil the needs of an in-house grid monitoring system. In the first half of paper we introduce GMA, outline current implementations, and provide the reasons that motivated us to design and then implement jGMA. In second part of the paper we discuss the implementation, it scalability and general performance, and finally conclude the paper and outline future work on jGMA.

1. Introduction

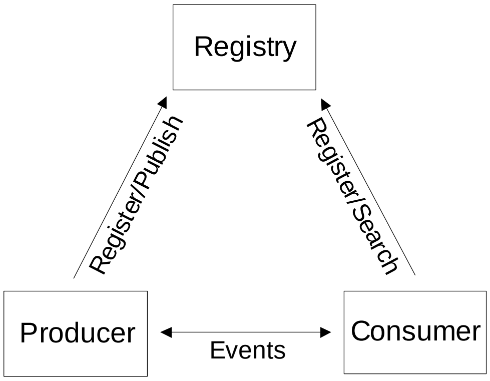


Figure 1: The Architectural View of GMA

The Distributed Systems Group at the University of Portsmouth has for the last few years, been developing a resource monitoring system for the Grid that can gather data from endpoints, filter and fuse this data for subsequent use by a variety of clients. The monitoring system, known as GridRM [1], needs to distribute information over the wide area between, so called, GridRM gateways. The software for distributing this information around GridRM needs to be lightweight, modular, fast, and efficient. There are obviously numerous ways to do this, we have, however, decided to use the Grid Monitoring Architecture (GMA) [2], which is the mechanism recommended by Global Grid Forum (GGF) [3]. The GMA specification sets out the requirements and constraints of any implementation. The GMA is based on a simple consumer/producer architecture with an integrated system registry, see Figure 1.

The GMA is an abstraction of the characteristics required for a scalable monitoring infrastructure over the Grid. The GMA supports a publish/subscribe and query/response model. In this model, producers or consumers that accept connections publish their existence in a directory service (registry). Producers and consumers can then both use the directory service to locate parties, which will act as a source or destination for the data they are interested in. It should be noted that monitoring data is sent from a producer to a consumer; however either the producer or consumer may initiate a subscription or query.

The GGF argue that the requirements of GMA cannot be met by existing event-based services, as the data requirements for monitoring information are different. The GGF list several desirable features for GMA:

- Low latency,
- Capable of a high data rate,
- Minimal system impact,
- Secure,
- Scalable.

2. Similar Work

In this section we briefly discuss the various GMA implementations currently available in the spring of 2004.

2.1 Standalone Implementations

2.1.1 R-GMA (Relational Grid Monitoring Architecture)

R-GMA [4] was developed within the European DataGrid Project [5] as a Grid information and monitoring system. R-GMA is being used both for information about the Grid (primarily to find out about what services are available at any one time) and for application monitoring. A special strength of this implementation comes from the use of the use of a relational model to search and describe the monitoring information. R-GMA is based on Java Servlet technology and uses an SQL-like API. R-GMA can be used in conjunction with C++, C, Python and Perl consumers and/or producers, as well as obviously with Java.

R-GMA is the most ambitious and significant variant of the current GMA implementations that was initiated in September 2000. Since then the software has continuously evolved. Currently R-GMA is being used for an "in-house" testbed [6].

2.1.2 pyGMA (Python GMA)

pyGMA [7] from LBNL [8] is an implementation of the GMA using Python. The developers have used the object-orientated nature of Python to provide a simple inheritance-based GMA-like API. While the features of pyGMA are not comprehensive, it is easy to install and use. pyGMA is supplied with a simple registry, which is designed for testing but is not meant to be deployed. Some sample producers and consumers are provided as a starting point for developing more comprehensive services.

2.2 Other GMA Implementations

There are several other systems, which either exhibit GMA like behaviour or have a GMA implementation embedded within them.

The Metadata Discovery Service (MDS) [9], which is part of Globus Toolkit is based on the emerging Open Grid Services Architecture (OGSA). MDS provides a broad framework within GT3, which can be used to collect, index and expose data about the state of grid resources and services. MDS3 is tailored to work with the OGSA-based Grid Services, it is, itself a distributed Grid Service. While MDS3 is an influential component within GT3, it is not suitable in its current state to use with GridRM as it requires the installation of GT3, which is rather heavyweight for our purposes.

The Network Weather Service (NWS) [10] allows the collection of resource monitoring data from a variety of sources, which can then be used to forecast future trends. NWS purports to have an architecture based on GMA, and components that exhibit GMA-like functionality. However, even though this may be the case, the GMA parts of NWS appear tightly integrated and it would be difficult to break these out of the release.

Autopilot [11] from the University of Illinois Pablo Research Group [12] is a library that can be called from an application to allow monitoring and remote control. Autopilot sensors and actuators (akin to the GMA producers/consumers) report back to a directory service called the AutopilotManager, which allows clients to discover each other. Autopilot can be used to create standalone GMA enabled components in C++, but it requires and builds on functionality provided by the Globus Toolkit (version 2).

A DSG technical report on jGMA [13] contains a matrix comparing the features and functionally of various versions of GMA. As it can be seen R-GMA has great potential, but there are a number of drawbacks, not least of these are the large number of system dependencies required for installation and use. In addition, there are some architectural features, which may limit its scalability and flexibility. Alternatively pyGMA appeared promising, but there are some issues with using it with a Java application, and there are some considerations with regards it having a very simple registry. Finally, MDS3 had the potential to fulfil our requirements for GridRM. Unfortunately the current implementation is embedded in the Globus release, which meant that it would potentially require some reengineering to meet our needs.

2.3 Summary

Currently the embedded versions of GMA do not easily lend themselves to standalone GMA purposes; consequently they cannot be used in their existing form with GridRM. This leaves three alternative GMA implementations pyGMA, Autopilot and R-GMA.

Calling Python (pyGMA) from Java, which is a requirement of GridRM, is not straightforward. While the Jython project [14] allows the use of Java from within Python, there is no simple mechanism for invoking Python from within Java without creating a customised and potentially complex JNI bridge.

R-GMA does provide a native Java API, and initially it was thought that R-GMA would be a suitable implementation for GridRM. However, there are a number of drawbacks with using R-GMA. It can be seen form the technical report

[13] that there are a significant number of dependencies to build R-GMA from source. Also, R-GMA is aimed at one specific version and distribution of Linux (Redhat 7.3). The developers have used a build process, which relies on files and libraries being in non-standard places and the use of a non-portable mechanism for compilation (shell scripts). There is a binary release of R-GMA, however, this is via RPMs, which again limits the platforms on which the system can be automatically installed. Another problem that was encountered is the rapid development and changing nature of R-GMA. This can create problems for a developer trying to work with such a large code base, because it is constantly evolving to keep up with the latest trends and needs of the large number of developers and potential users.

A requirement of GridRM is that it is easy to install and configure across multiple platforms. A complicated set of prerequisites would make its deployment a lengthy and potentially complex task. GridRM requires a GMA implementation that has a lightweight Java API, which is functional, easy to use, and extensible.

3. jGMA Design

The global layer of GridRM requires a wide-area event-based system for passing control and monitoring information between the local GridRM gateways. Ideally, from our point of view, we would have preferred to integrate a third-party GMA implementation into GridRM; this is for obvious reasons, such as reduced development time and minimal support requirements. However, as stated in Section 2, none of the existing GMA implementations met our requirements, and consequently we have developed our own version.

3.1 jGMA Design Criteria

The first steps in our design were to layout a set of general criteria that we felt were necessary and/or desirable. These criteria were based on our experiences whilst investigating the other GMA implementations, the needs of GridRM, and some overarching principals:

- Compliant to the GMA specification,
- Lightweight, with a small and simple API,
- Minimal number of other installation dependencies,
- Simple to install and configure,
- Uses Java technologies, and fulfil GridRM’s needs,
- Supports both blocking and non-blocking-based events,
- Designed to work locally over a LAN or over a wide area such as the Internet,
- Fast and have a minimal impact on its hosts,
- Choice of registry service, from a lightweight one, such as text-based files, to an XML-based one like Xindice [15], or something else, such as a database or MDS,
- Able to work through firewalls,
- Capable of taking advantage of TLS or the GSI,
- Easy to use.

To provide the functionality and features that we desire it was decided to write jGMA in pure Java. This allows us to take advantage of a range of Java-based technologies, as well as providing portability via bytecode that should execute on any compliant Java Virtual Machine.

3.1 jGMA Development and Implementation Issues

jGMA consists of four entities:

- A registry to allow producers and consumers discover each other,
- A Producer/Consumer servlet to allow remote communications,
- Consumers,
- Producers.

jGMA has one dependency, Apache Tomcat [18], which provides a servlet container and a gateway that uses HTTP for inter-gateway communications. It was felt that this dependency did not compromise our design criteria, as Tomcat has become familiar to most Java developers. In addition, GridRM itself requires Tomcat.

3.2 jGMA Communication

jGMA supports both blocking and non-blocking I/O; this provides the flexibility and functionality that will be required in most circumstances. jGMA has two modes of event passing; local, where communications are within one administration domain, i.e. behind a firewall, and global, when traversing more than one administrative domain, i.e. via one or more firewall(s).

Originally Java RMI was used for local communication, as it was a way to rapidly prototype the communication system, this was altered to Java Sockets during the first stage of optimization as RMI imposed significant communication overheads. Currently jGMA uses TCP Sockets and non-blocking communications are simulated. Communications over the wide area use HTTP. Using a gateway Producer/Consumer servlet also allows wide-area connectivity for machines, which do not have direct access to the Internet, which is common with nodes in standard cluster network topologies.

The main difference between local and wide-area communications is that inter-domain messages are sent via the Producer/Consumer servlets rather than directly between consumers and producers.

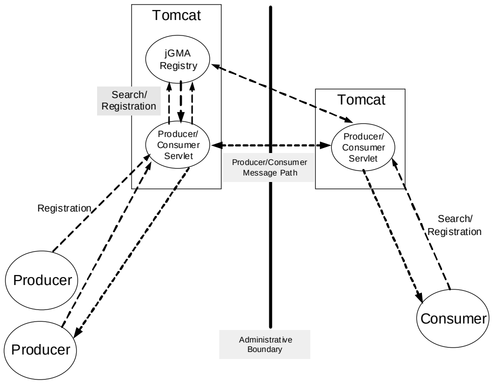


Figure 2: Wide-area jGMA Communications

Figure 2 shows an example of wide-area communications. In this example the registry is located on the same network as the producers, it could just as easily be on the consumer side or on a completely different network. Note that inter-domain messages between a producer and consumer are routed via the Producer/Consumer Servlet, which communicated between each other via HTTP, thus avoiding firewall issues.

3.2.1 Addressing within jGMA

The pseudo unique name is made up of several components which jGMA uses to decide how to route a message, this is most easily explained by way of an example:

http://dsg.port.ac.uk:8080/jGMA/PC?c0a8064_localhost:123_producer_foo_1

- Here **http://dsg.port.ac.uk:8080/jGMA/PC** is the public URL of the Producer/Consumer servlet which can be used to contact the network via the Internet,
- **c0a8064** is the IP address of the Producer/Consumer servlet in hexadecimal,
- **localhost: 123** is the hostname and port which the client is listening on,
- **producer** indicates the type of client, it will be either a producer or consumer,
- **foo** is the preferred name of the client, in a human readable form,
- The number, **1**, in this case is an incremental number, which ensures this name is unique to the Producer/Consumer Servlet.

For the purposes of the development and testing of jGMA the scheme used for naming is more than adequate.

3.3 The jGMA Registry

The overall purpose of the registry in GMA is to match consumers with one or more producers. This is achieved by producers publishing information about themselves and then consumers searching through the registry until they find the relevant match and thereafter the two communicating directly with each other.

The information published in the registry typically includes the unique address, and potentially the attributes and capabilities of the producer. In addition, to limit the retention of stale information, some sort of Time To Live (TTL) tag should be associated with the registration.

An implementation of the GMA should be capable of scaling to global proportions. This implies that there should multiple registries and the registry information should be replicated for fault tolerance purposes.

GridRM uses jGMA to provide a messaging infrastructure between its gateways. GridRM gateways hold detailed information about its producers and what data they can provide. A GridRM client could search through one or more gateways for the producers that it is interested in. However, as the number of producers and gateways becomes large this would produce an unacceptable load on the overall system and also means that a query could take a significant amount of time. It is clear that a meta-level registry service is necessary. Such a service would hold high-level information about producers and gateways that could be interrogated first by a client before doing a low-level and detailed search on individual gateways.

jGMA registries can provide this meta-level registry service for GridRM. Ideally, jGMA registries should be able to “slot” together to form a virtual registry. Such a registry, from a client’s point of view, would appear as one large shared entity. To create the virtual registry requires that the physical registries are distributed and the information they hold is replicated.

jGMA originally used a volatile registry, which stored the names of producers and consumers in memory. This registry was designed to provide the limited functionality required to build and test the rest of jGMA. The jGMA registry contains a parser that allows queries based on a simplified SQL syntax. Maintaining a high-level of abstraction via the registry API (and SQL syntax) has made it possible to create a jGMA registry interface that can plug-in to a variety of potential repositories, including XML and relational databases, as well as other registries such as MDS or R-GMA.

Originally, producers or consumers in jGMA registered just their address (see section 3.2.1) in the registry. The new registry API now not only permits consumers or producers to register, but it also allows an associated XML document be uploaded, which describes features and capabilities of the registered entities. This additional feature means that the developers using jGMA can publish as much information as they wish, and consequently have control over the granularity of the virtual registry service.

For the purposes of GridRM, the jGMA registry service will remain lightweight, we do not intend to store anything other than the information required to provide the GMA-like functionality and use of the extra XML information will be limited.

The objective of this stage of jGMA has been to produce an abstract registry API that can interact with a number of persistent data stores, include relational and XML databases, or a simple flat ASCII file. The current jGMA registry service has been prototyped using the XML Xindice database [15]. Later we plan to test Berkeley DB Java Edition [16] and an ASCII text file as registry components.

The revised jGMA registry addresses the issues of scalability, though a hierarchy of registries, and fault tolerance with partial information replication. In addition, registered entities will be associated with leases, akin to those used by Jini [17]. This will ensure that registries maintain consistent and fresh information.

3.4 The jGMA API

The jGMA API is relatively small; currently there are only 17 method calls in the API. Building on the current basic API and utilising other Java features, such as threads, can achieve the higher-level producer/consumer functionality. For example, it is possible to do simultaneous blocking I/O calls by creating two consumers instead of one, or a more complicated client may create both a consumer and a producer.

4. jGMA Implementation

4.1 Overview

While conceptually the producer, consumer and Producer/Consumer servlet are different; the jGMA implementation reuses the same code for each. This is possible because although they have different logic for processing jGMA messages a large part of their functionality is focused on exchanging messages (events).

Figure 3 shows the internal software structure of the producer, consumer, and Producer/Consumer servlet. It attempts to highlight the importance of the socket `send()` and `receive()` methods. If these methods are slow or poorly implemented, it will affect the whole system. Our initial analysis of the program flow showed that the majority of the execution time was spent manipulating, copying and sending the jGMA messages through the system.

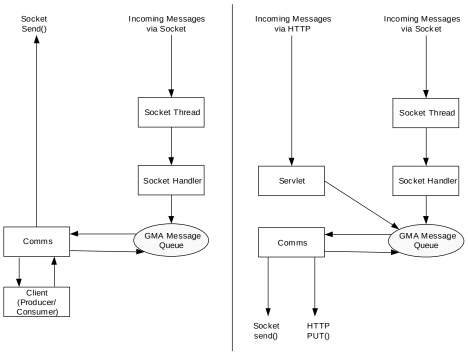


Figure 3: The internal structure of a jGMA client and Producer/Consumer servlet

4.2 Reducing Communications Overheads

In order to reduce message latency we needed an efficient way of passing data between jGMA components. The normal Java programming practice of using objects was replaced with static methods, which manipulated byte arrays. Our objective here was to reduce the number of times Java copied the internal data structures and limit the use of expensive high-level Java API calls. This alteration halved the number of internal copies from four to two.

4.3 Wide-Area Communications

When jGMA stored handled messages as Java objects, wide-area communication were achieved by encoding the object as a string and sending it between Producer/Consumer servlets using HTTP GET; here the data is part of the URL. As discussed in Section 4.2, the jGMA message format was altered from objects to packed binary arrays; there was a need to change the wide-area communication to efficiently send this binary data.

4.3.1 HTTP POST

The HTTP protocol [19] allows binary data to be sent in the body of a HTTP POST request to allow uploading through HTML forms [20]. Using 'multipart/form-data' and 'multipart/mixed' it is possible to send more than one binary message in one request, or mix ASCI and binary in one request, this extensibility and flexibility is desirable, since the GMA specification is not fully defined. For now, jGMA only sends one message per request, but it may take advantage of the ability to send multiple messages in future versions.

1.1 4.4 Summary

jGMA was incrementally adapted and evolved in light of our experiences. Once the software was stable (version 0.3.2) it was tested to measure its performance, this process and the results are presented in Section 5.

5. Testing

5.1 Introduction

This section reports on the initial testing the performance and functionality of jGMA over a local area. The overall aim of this stage was to optimise the communication overheads, assess impact, and confirm overall functionally. We provide an overview of the tests with some key findings highlighted, a more thorough analysis can be found in the technical report [13].

5.2 jGMA Benchmarking

A Java implementation of the traditional Ping-Pong network test was used to measure point-to-point performance. We were careful to omit extra overheads, such as internal processing of GMA messages. By comparing the measurements taken when timing jGMA, to the raw performance, it was possible to analyse performance overheads.

Test 1: Non-blocking I/O – A Ping-Pong between a single producer and consumer. This test involved executing both consumer and producer on the same host, and then with the producer and consumers on different hosts connected via Fast Ethernet.

Test 2: Blocking I/O – A Ping-Pong between a single producer and consumer. These tests were run both over the network and on the same machine in the same way as the non-blocking tests.

Test 3: Scalability Tests – The cluster head node runs the Producer/Consumer servlets, Registry servlet, and one consumer. The producers are distributed over the cluster so that each node can run up to two. N producers are started and the consumer instructs them to send messages to it over Fast Ethernet, the number of messages the consumer handles per second is recorded. Non-blocking I/O was used as this was felt a more realistic simulation than blocking I/O for large numbers of events.

Test 4: Wide-Area Communications – A wide-area environment was simulated on the DSG cluster by running two Producer/Consumer servlets – one for the consumer and one for the producer. The test machines communicate via Fast Ethernet. The test measures the latency and bandwidth of sending jGMA messages over HTTP via a Producer/Consumer servlet for a range of message sizes. For an explanation of the steps in wide-area communications see Section 3.2.2 and Figure 2.

5.3 Benchmarking Results Overview

The performance tests (for details see [13]) showed that for blocking communications there is an extra 8-millisecond overhead compared to raw sockets for Ethernet messages under <256 Kbytes. This overhead is due to processing a blocking message, which we are continuing to investigate. There is the possibility of changing to an eager-reader paradigm here, but this may produce an excessive impact on the host. The overhead currently limits the peak bandwidth, which is 33% of the raw socket bandwidth. For non-blocking communication using messages <256 Kbytes, jGMA produces an overhead of 1.4 milliseconds compared to raw sockets for Ethernet, and the peak bandwidth is 67% of the raw socket performance.

It requires between 7 to 9 producers (depending on the message size) to saturate a jGMA consumer. When more producers are added, the number of messages the consumer can handle does not fall – which indicates that even under a heavy load (1100 x 32 Kbytes messages per second) jGMA is stable. When the number of messages being received by a consumer reaches its peak, new messages begin to queue up in the send buffers of the producers, eventually a producer will consume all available resources and will not be able to add any new messages to its send buffer. It is unlikely that GridRM will generate messages at this rate, but it indicates the throughput that GridRM can expect jGMA to be able to handle reliably.

It became evident that some kind of throttling was needed for the sending mechanisms of jGMA. Currently a producer can generate as many messages as the memory allocated to it by the Java VM can contain. This was desirable when testing the scalability because it allowed stress testing of the jGMA implementation. This, however, does have a side effect, a consumer can only process a certain number of messages per second. When multiple producers are sending at the same time, the consumer will reach a maximum throughput and messages will start to accumulate in the send buffers at each producer.

6. Summary and Conclusion

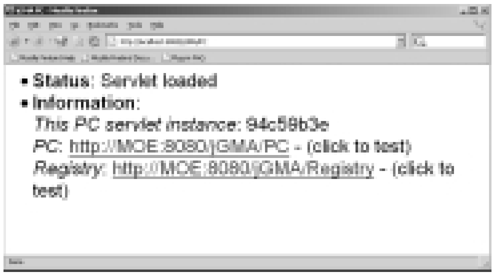
In this paper we have described and discussed jGMA, a lightweight GMA implementation written in Java. We were motivated to produce jGMA due the lack of a viable alternative to use with our grid monitoring system.

Several outstanding issues with jGMA are currently being addressed. These include more efficient event and queue handling to reduce the overheads described in section 5.3, and a means of throttling message generation and delivery. One solution would be to allow the developer to set the buffer size used. A more interesting solution would be to make the sending functions more intelligent, maybe using a sliding-window-based protocol [22] or perhaps using rate feedback [23].

jGMA is fully functional, but is still evolving. In particular the registry service is currently being developed. jGMA has been integrated within GridRM and is being further investigated via its global testbed. jGMA [21] is currently available as a binary release to developers interested in investigating and further enhancing its capabilities.

6.1 Future Work

This initial benchmarking has shown us that jGMA is functional with acceptable performance. Deploying jGMA alongside GridRM over the wide-area has allowed us to test the system more fully. There are several other areas that we are currently exploring, including:



- **Status**: Servlet loaded
- **Information**:
  This PC servlet instance: 94c59b3e
  PC: http://MOE:8080/jGMA/PC - (click to test)
  Registry: http://MOE:8080/jGMA/Registry - (click to test)

Figure 4: The Happy jGMA web interface

- An interactive debugging and monitoring interface. At the moment a simple web interface for testing the installation of jGMA (Figure 4). We intend to extend this interface so that we can dynamically monitor and track jGMA events and queues.
- Registries, as described in the paper, we are moving from a simple volatile registry to a distributed virtual registry.
- Security, so far security has not been addressed in jGMA. This is obviously an important feature for GMA compliance, we are exploring how best to incorporate the GSI.

References

[1] GridRM, http: //gridrm.org/

[2] GMA, http://www-didc.lbl.gov/GGF-PERF/GMA-WG/

[3] Global Grid Forum, http://www.ggf.org

[4] R-GMA, http: //www.r-gma.org/

[5] DataGrid, http: //www.eu-datagrid.org/

[6] R-GMA testbed, http://hepunx.rl.ac.uk/edg/wp3/testbed.html

[7] pyGMA, http: //www-didc.lbl.gov/pyGMA/

[8] LBNL, http: //www-didc.lbl.gov/

[9] Globus MDS, http://www.globus.org/mds/

[10] Network Weather Service, http://nws.npaci.edu/NWS/

[11] AutoPilot, http://www-pablo.cs.uiuc.edu/Project/Autopilot/AutopilotOverview.htm

[12] Pablo Research Group, http: //www-pablo.cs.uiuc.edu/

[13] jGMA: A lightweight implementation of the Grid Monitoring Architecture, DSG Technical Report, http://dsg.port.ac.uk/~mjeg/jGMA/jgma_report2004.pdf

[14] Jython, http://www.jython.org/

[15] Xindice, http://xml.apache.org/xindice/

[16] Berkeley DB Java Edition, http://www.sleepycat.com/products/je.shtml

[17] Jini, http://www.jini.org/

[18] Apache Tomcat, http://jakarta.apache.org/tomcat/

[19] RFC2616, Hypertext Transfer Protocol, HTTP/1.1, http://www.w3.org/Protocols/rfc2616/rfc2616.html

[20] RFC1867, Form-based File Upload, HTML, http://www.ietf.org/rfc/rfc1867.txt

[21] jGMA, http://dsg.port.ac.uk/projects/jGMA/

[22] Understanding the Performance of TCP Pacing, http://netlab.caltech.edu/FAST/references/ Infocom2000pacing.pdf

[23] RFC3448, TCP Friendly Rate Control, http://www.ietf.org/rfc/rfc3448.txt