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

January 29, 2004

Abstract:

Wide-area distributed systems require scalable mechanisms that can be used to gather and distribute system information to a variety of endpoints. The emerging Grid infrastructure is rapidly being taken up for technical computing as well as in business and commerce. We are developing a monitoring system, known as GridRM, which needs to distribute information over the wide area between, so called, GridRM gateways. 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 GridRM. The paper is divided into five sections. In the first two sections of the paper we introduce GMA, outline the current implementations, and provide the reasons that motivated us to design jGMA. In section 3 we detail the design and implementation of jGMA. In section 4 we discuss the implementation. In section 5 we specify and execute jGMA benchmarks and then discuss our results. Finally, in section 6, we conclude the paper and outline future work on jGMA.

1 Introduction

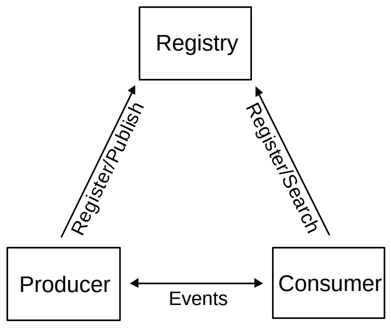

Figure 1: The Architectural View of GMA

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) (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. 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. Table 1, shows a matrix of the features and functionality of various GMA implementations, which we use to highlight the reasons that motivated us to develop jGMA.

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.

	R-GMA	**pyGMA**	**jGMA**	**MDS3**
Languages Supported	Java, C, C++, Python and Perl	Python	Java	C and Java
Implementation Language	Java	Python	Java	C and Java
Installation	Binary – RPMs for RedHat, Source – Python and RedHat like Linux	Uses a Python installe	Binary - one Java .jar file, Source - ANT + Apache Tomcat	GPT package management (included)
Dependencies	Ant, Java 1.4, Bouncy Castle, EDG Java Security, Jakarta Commons, Logging, Jakarta-Axis, Jas, JxUtil, Log4j, MySQL Client/Server, MySQL Java Driver, Netlogger, Prevayler, Python2, Regexp, Swig, Tomcat 4, Xerces C and Java	Python 2, Python SOAP (ZSI), Python-xml, Fpconst	Ant, Apache Tomcat, Java 1.4	Java 1.3 or better, JAAS library, Ant 1.5, Junit, YACC (or Bison), Globus Tookit 3
Transport	HTTP	(HTTP) SOAP	LAN sockets, WAN HTTP	(HTTP) SOAP
I/O Type	Streaming and Blocking	Passive and Active	Blocking and Non-blocking	Query based
Type	Standalone	Standalone	Standalone	Integrated
Registry	RDBMS using MySQL	Simple	Simple/Xindice	Collection of Grid Services
Types of producers	CanonicalProducer, DataBaseProducer, LatestProducer, ResilientStreamProducer	User-based	User-based	User-based
API Size	213 calls (Java API)	46 calls	17 calls	Very large
Security	EDG-security for authentication, SSL for transport	None	None	GSS (SSL and Certs)
Where used	In-house EDG testbed		GridRM	GT3 (many)

Table 1: A comparison of GMA implementations

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] that is part of Globus Toolkit version 3 is based on the emerging Open Grid Services Infrastructure (OGSI). 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 OGSI-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).

Table 1 shows the list of 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 architecture 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 [13] allows the use of Java from within Python, there is no simple mechanism for invoking Python from with 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 in Table 1 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.

3.0.1 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 GridRMs needs,
- Support 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 [14], 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 virtual entities:

- Simple registry to allow clients to discover each other,
- Producer/Consumer Servlet (PC Servlet) to allow remote communications,
- Consumer,
- Producer.

jGMA has one dependency, Apache Tomcat [15], 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 very familiar to 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 by a developer. jGMA has two modes of event passing. The first is local, where communications are within one administration domain, i.e. with a firewall. The second is global, when traversing more than one administrative domains, 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 PC Servlet also allows WAN connectivity for machines, which do not have direct access to the Internet, which is common with nodes in standard cluster network topologies.

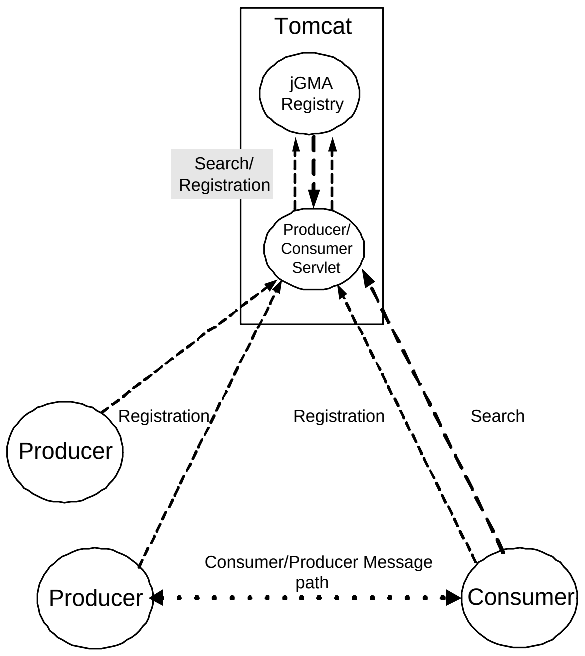

Figure 2: Local jGMA Communications

3.2.1 Local Communications

Figure 2 shows an example of local jGMA communications. The following steps occur when a jGMA consumer attempts to interact with one or more producers:

1. A producer registering with the registry:
   (a) A producer sends a preferred name to the local PC Servlet,
   (b) The PC Servlet creates a pseudo unique name and passes it to the registry,
   (c) The PC Servlet sends the unique name to the producer.
2. A consumer registering with the registry:
   (a) This process is identical to that of the producer, described in Step 1.
3. A consumer queries the registry for a producer service:
   (a) The consumer sends an SQL-formatted query to the PC Servlet,
   (b) The PC Servlet passes the query onto the registry, which replies with any matches,
   (c) The PC Servlet sends the matches back to requesting consumer.
4. Consumer/producer communications:
   (a) The consumer/producer both create a socket connection to the producer/consumer and sends a query/reply.

3.2.2 Wide Area Communication

Figure 3 shows an example of wide-area jGMA 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. The following steps occur when a jGMA consumer attempts to interact with one or more producers.

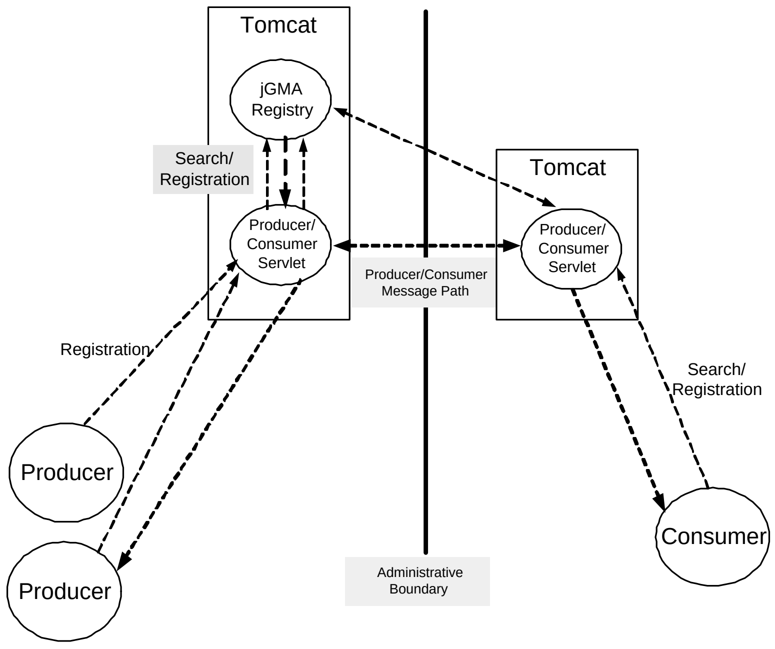

Figure 3: Wide-area jGMA Communications

1. The producer registers with the registry:
   (a) The producer sends a preferred name to the local PC Servlet,
   (b) The PC Servlet creates a pseudo unique name and passes it to the registry,
   (c) The PC Servlet sends the unique name back to the producer.
2. The consumer registers with the registry:
   (a) The consumer sends a preferred name to the local PC Servlet,
   (b) The PC Servlet creates a pseudo unique name and sends it to the remote registry via HTTP,
   (c) The PC Servlet sends the unique name back to the producer.
3. The consumer queries the registry for a producer service:
   (a) The consumer sends a SQL formatted query to PC Servlet,
   (b) The PC Servlet sends the query onto the remote registry servlet via HTTP, which replies with any matches,
   (c) The PC Servlet sends matches back to the consumer.
4. Consumer/producer communications:
   (a) The consumer sends a message via a socket connection to the PC Servlet,
   (b) The PC Servlet sends the message to the remote network via HTTP,
   (c) The remote PC Servlet, on the producer network, sends the message to the producer via a socket connection,
   (d) The producer sends a reply to its local PC Servlet,
   (e) The producers PC Servlet sends the reply via HTTP back to the consumers PC Servlet,
   (f) The consumer PC Servlet sends the reply back via a socket to the consumer.

The main difference between LAN and WAN communications is that inter-domain messages are sent via the PC Servlets rather than directly between consumers and producers.

3.2.3 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 PC Servlet which can be used to contact the network via the Internet,
- **c0a8064** is the IP address of the PC 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 PC Servlet.

While the incremental number guarantees uniqueness of the name within an individual PC Servlet, the name is not guaranteed to be globally unique with a jGMA system. Generating this name within the PC Servlet means that jGMA is more flexible and the implementation is simpler, otherwise it would require a more complex replicated registry to track names and ensure uniqueness.

In order for there to be a name clash the following must take place:

- Two clients must be of the same type (producer or consumer),
- They must register at different PC Servlets,
- They must independently register in the same order,
- When they create their local socket handlers they must get the same local port allocated to them by the OS (the socket handlers use the next available socket),
- The PC Servlets must have the same fully qualified domain name,
- The Tomcat server must be listening on the same port,
- The PC Servlets must have the same IP address (possible with LAN addresses).

So, while the name is not unique, for the purposes of the development and testing of jGMA the scheme used for naming is adequate.

3.3 The jGMA Registry

jGMA currently uses a simple volatile registry, which stores the names of producers and consumers in memory. The registry contains a simple SQL parser that allows queries using a standard SQL syntax. The registry is designed to provide the limited functionality required to build the rest of jGMA, which is ideal for our present purposes; however, we do plan to use alternative registries. In the short term this will be Xindice, which will provide us with XML database functionality. By maintaining a high-level of abstraction via the registry API (and SQL syntax) it will be possible to create a jGMA registry, which plugs into more heavy weight registries as well, such as R-GMA.

We also intend to use XML to describe all events, messages and other information in jGMA. It is important to note that even with this extra functionality the registry will remain a lightweight meta-directory, we do not intend to store anything other than the information required to provide the GMA-like functionality. A revised registry will address the issue of scalability, most probably though a hierarchy of registries with partial information replication.

3.4 The jGMA API

The jGMA API is small and lightweight, its 17 API calls are listed below with a short explanation of what they do.

3.4.1 jGMA Methods

#	Method	Description
1	`incomingGMAMessage(byte[] msg)`	Start communication threads and connect to PC Servlet.

2
3
4
	`registeredName register(suggestedName)`
	`unRegister()`
	`response registryQuery(sql)`
		Queries are sent to the registry using standard SQL syntax.

5
6
	`byte[] blockingSendGMA(String to, byte[] data)`
	`int nonblockingSendGMA(String to, byte[] data)`
		Two forms of jGMA I/O are supported, blocking, and non-blocking. Non- blocking I/O uses sequence numbers so that a reply can be identified when it arrives.

7	`public Message(sequence, command, from, to, data)`	An object is a friendlier representation of the data stored in a jGMA message byte[].
8	`incomingGMAMessage(byte[] msg)`	A callback method, which allows delivery of non blocking messages.

9
10
11
12
	`byte[] createGMAByte(sequence, command, from, to, String data)`
	`byte[] createGMAByte(sequence, command, from, to, byte[] data)`
	`Message unMarshal(byte[] msg)`
	`byte[] marshal(Message msg)`
		Several static methods are provided to interpret and create efficient byte arrays, which are used to store jGMA messages.

13
14
15
16
17
	`String getFrom(byte[] msg)`
	`String getTo(byte[] msg)`
	`String getData(byte[] msg)`
	`int getSequence(byte[] msg)`
	`int getCommand(byte[] msg)`
		Methods to interpret a packed jGMA message.

Table 2: The jGMA API

3.4.2 Summary

The development of more advanced producer/consumer functionality can be achieved by utilising multiple API calls and other Java features, such as threads. 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 PC Servlet are different; the jGMA implementation reuses the same code for each. This is possible because although they have different logic to process jGMA messages a large part of their functionality is focused on exchanging messages (events).

Figure 4 shows the internal software structure of the producer, consumer, and PC 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.

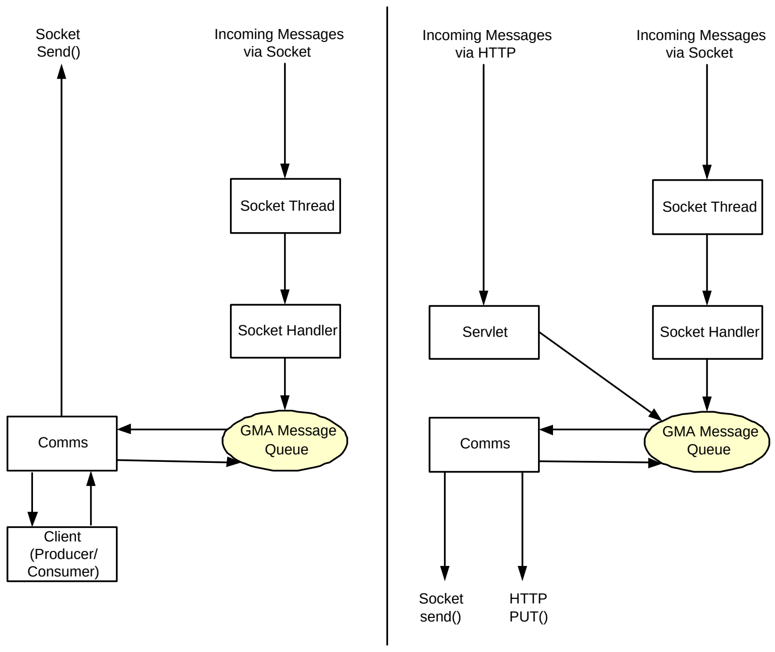

Figure 4: Internal structure of a jGMA client and PC Servlet

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. The flow of data using the byte arrays is shown in Figure 5.

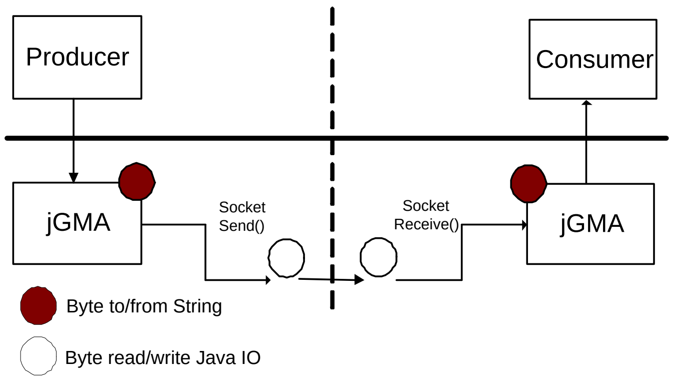

Figure 5: Minimising Internal Copying using Byte Arrays

4.3 Summary

jGMA was incrementally adapted and evolved in light of our experiences. Once the software [16] 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 paper reports on the first stage of testing the performance and functionality of jGMA over a LAN. The overall aim of this stage was to optimise the communication overheads, assess impact, and confirm overall functionally. Two nodes from the DSG cluster were used to perform the tests. Their configuration is shown in Table 3.

Processor Type	Dual Xeon (Prestonia)
Processor Speed	2.8GHz
Processor Cache	512K L2 Cache
Front Side Bus	533MHz
RAM	2 GB ECC
Storage	80 GB EIDE
JVM	Sun Java Version 1.4.2_02
OS	Redhat Linux 9.0, Kernel 2.4.20

Table 3: The DSG Cluster Node Configuration

5.2 A Microsecond Java Clock

The Sun JVM running on Linux has access to a clock with a millisecond resolution. After the initial stages of optimization it was clear that this clock resolution was too coarse for our benchmarking purposes. An alternative microsecond resolution clock was implemented that used JNI to invoke a Linux system call. The implementation has been made publicly available; see [17]. The cost of calling the clock method was measured, so its overhead could be eliminated from the benchmark results; an overhead of approximately 0.7 micoseconds was measured.

5.3 jGMA Benchmarking

Four initial benchmarks were produced to test the blocking and non-blocking communication performance of jGMA I/O, these where run on a single host and over Fast Ethernet. The tests were designed to show the overheads of using jGMA and the affect of message size on system throughput.

5.3.1 Generating baseline performance

A Java implementation of a traditional pingpong network test was used to measure 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 the overheads.

5.3.2 Benchmark 1 & 2 – Non Blocking I/O

Test 1: Non-blocking I/O - A pingpong 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. The pseudo code for the non-blocking consumer and producer is shown in Figure 6.

```
//Pseudo code: Non-blocking
Producer{
  startProducer(testProducer);
  IncomingGMAMessage{
    replyGMAMessage();
  }
}

Consumer{
  startConsumer(testConsumer);
  registrySearch(testProducer);
  RecursiveTest();

  RecursiveTest{
    messageSize++;
    byte[] payload = byte[messageSize];
    startClock();
    NonBlockingGMASend(payload);
  }

  IncomingGMAMessage{
    time = stopClock();
    System.out(messageSize:time/2);
    if(messageSize<max){
      RecursiveTest();
    }
  }
}
```

Figure 6: Non-blocking I/O Pingpong

```
//Pseudo code: Blocking
Producer{
  startProducer(testProducer);
  IncomingGMAMessage{
    replyGMAMessage();
  }
}

Consumer{
  startConsumer(testConsumer);
  registrySearch(testProducer);
  RecursiveTest();

  RecursiveTest{
    messageSize++;
    byte[] payload = byte[messageSize];
    startClock();
    returnedMessage[] = BlockingGMASend(payload);
    time = stopClock();
    System.out(messageSize:time/2);
    if(messageSize<max){
      RecursiveTest();
    }
  }
}
```

Figure 7: Blocking I/O Pingpong

5.3.3 Benchmark 3 & 4 - Blocking I/O

Test 2: Blocking I/O - A pingpong between a single producer and consumer. This test is run both over the network and on the same machine in the same way as the non-blocking I/O. As the pseudo code shows, blocking calls within jGMA are easier for the programmer to use as program flow stops while the I/O is performed. The pseudo code for the blocking consumer and producer is shown in Figure 7.

5.3.4 Benchmark Results

5.3.5 Blocking I/O

Both the localhost and Ethernet performance is exactly the same. It can be seen for messages less than 256 Kbytes, that Ethernet has a fixed latency of about 8 milliseconds. This effect is due to jGMA using two jGMA non-blocking messages to simulate a blocking call, the cost of tracking the messages, matching replies and notifying jGMA threads to wakeup, creates this fixed overhead. This fixed latency smooths out the effect of the transmission time until the time to send is greater than the 8 milliseconds. This means that the faster speeds of localhost I/O do not affect the time to send via a blocking call. We investigated this latency with the aim of eliminating it, but found that the only simple alternative was to poll continuously, which had a negative impact on the jGMA host.

The bandwidth of a blocking call on both the localhost and over Ethernet is severely restricted by the approximately 8 millisecond queuing overhead. The peak bandwidth 3.7 Mbytes/s for both, which is 33% of the raw Java Sockets performance compared to Ethernet. There is a sharp change in the jGMA message transmission times at 256 Kbytes, this is discussed in Section 5.3.7.

5.3.6 Non-Blocking I/O

Both the localhost and Ethernet non-blocking curves mirror each others shape, with an almost constant offset up until 256 Kbytes. Thereafter, the curves merge. Both curves exhibit, what appears to be, a small

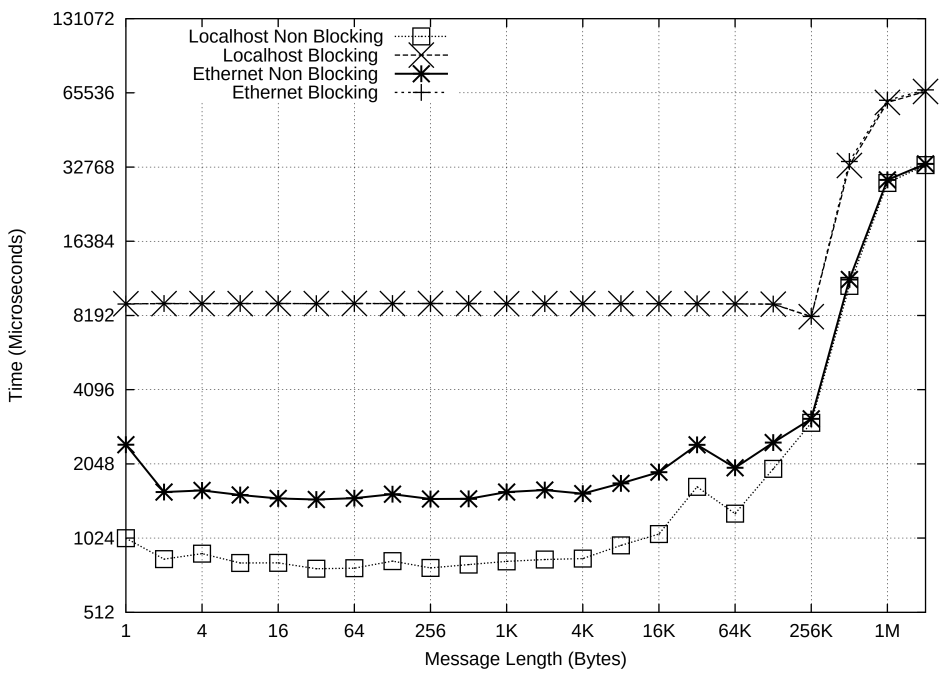

Figure 8: Latency versus Message Length

additional startup latency for 1 byte messages. Also, both curves exhibit a small jump at 32 Kbytes; which we assume is related to some internal JVM buffering.

Localhost non-blocking I/O has an approximate message latency of around 900 microseconds, whereas Ethernet has a latency of 2000 microseconds, between message lengths of 2 bytes and 64 Kbytes. The difference between the two curves are what we would expect for local communication, as opposed to those over Ethernet. The peak bandwidth of non-blocking calls on both the localhost and over Ethernet is about 7.4 Mbytes/s. This peak is 67% of the raw Java Sockets performance over Ethernet.

For messages of less than 256 Kbytes, localhost I/O can process an average of 1000 messages per second, while on Ethernet this slows to approximately 660 messages per second. The bandwidth graph (Figure 9) shows that Ethernet and localhost I/O both achieve the same throughput. A value of 7.4 Mbytes/s for Ethernet performance is good, but we would be expected that localhost I/O would achieve a rate much higher than this.

5.3.7 The 256 Kbyte Phenomenon

In both Figures 8 and 9 at 256 Kbytes there is a steep increase in the time to send a message. We believe this is the effect of the TCP socket buffers in Linux. The nodes of our cluster have their default buffer size set to 256 Kbytes (the Linux default is 16 Kbytes). If the buffer size is reduced, the step in the curves moved to smaller messages sizes, and vise versa.

6 Summary and Conclusion

In this paper we have 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. jGMA, whilst being fully functional, is at an early stage of development. Consequently we are optimizing

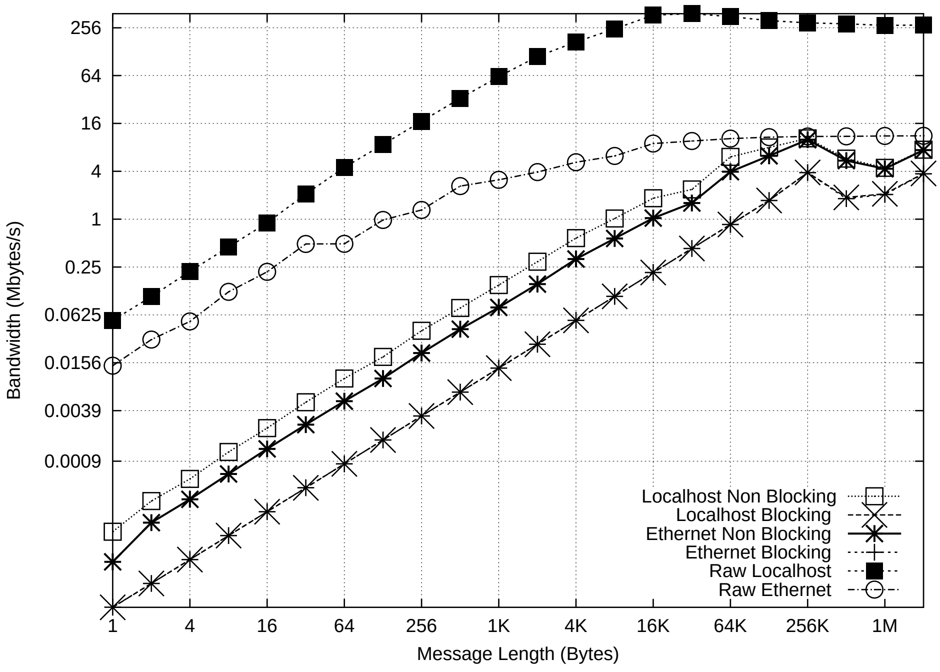

Figure 9: Bandwidth versus Message Length

its performance. In this paper we have presented the results of simplistic benchmarks that give us some idea of the blocking and non-blocking performance we would expect between consumer(s) and producer(s) using point-to-point communications.

The benchmarks have shown that for blocking communications there is an 8 millisecond additional overhead compared to raw sockets for Ethernet messages under 256 Kbytes. This overhead is due to the overhead of processing a blocking message, which we are continuing to investigate. There are possibilities of changing to an eager-reader paradigm here, but this may cause an excessive impact on the host. The overhead currently severely limited the peak bandwidth, which is 33% of the raw socket bandwidth. For non-blocking communication using messages under 256 Kbytes, jGMA produces an overhead of 1.4 milliseconds compared to raw sockets for Ethernet. The peak bandwidth is 67% of the raw socket performance.

6.1 Future Work

This initial stage of benchmarking has shown us that jGMA is functional but has a bottleneck for LAN blocking I/O. We will investigate the means of reducing this overhead without impacting the host systems. We are also startinng to look at optimising WAN communications; these use HTTP. We will design a number of benchmarks similar to the ones we used over a LAN. These are several other issues that we need to explore, including:

- Scalability, here we will look at having multiple end-points (consumers and/or producers) distributed over a LAN and WAN.
- Registries, we are interested in incorporating more sophisticated registries such as Xindice,
- Security, So far security in jGMA has been not been addressed. This is obviously an important feature for GMA compliance, which will be added.

Further results and findings will be reported during the talk at the conference.

References

[1] GridM, 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] Data Grid, 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] University of Illinois Pablo Research Group, http://www-pablo.cs.uiuc.edu/

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

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

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

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

[17] DSG JNI Microsecond Clock, http://dsg.port.ac.uk/projects/javaclock/