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

April 16, 2004

Abstract:

Wide-area distributed systems require scalable mechanisms to gather and distribute system information to a variety of endpoints. We are developing a monitoring system, known as GridRM, which needs to gather and distribute information over the wide-area between 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 to fulfil the needs of GridRM. The paper is divided into six sections. In the first two sections 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 describe the jGMA benchmarks and then discuss our results. Finally, in section 6, we conclude the paper and outline future work.

1 Introduction

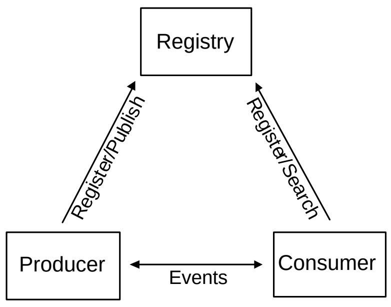

Figure 1: The Architectural View of GMA

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 end-points, and then 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, and 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. Since it was initiated in September 2000 the software has continually 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] that is part of Globus Toolkit version 3 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 core, 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

	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	Currently Integrated
Registry	RDBMS using MySQL	Simple	Simple/Xindice	Collection of Grid Services
Types of producers	CanonicalProducer, DataBaseProducer, LatestProducer, ResilientStream-Producer	User-based	User-based	User-based
API Size	213 calls (Java API)	46 calls	17 calls	Many calls
Security	EDG-security for authentication, SSL for transport	None	None	GSS (SSL and Certs)
Where used	In-house EDG testbed	DMF	GridRM	GT3 (many)

Table 1: A comparison of GMA implementations

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 architectural features, which may limit its scalability and flexibility. Alternatively pyGMA appears 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 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 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 GridRM's 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:

- A simple registry to allow producers and consumers to discover each other,
- A 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 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 by a developer. jGMA has two modes of event passing. The first is local, where communications are within one administration domain, i.e. behind a firewall. The second is global, when traversing more than one administrative domain, i.e. via one or more firewall(s).

Communications over the wide area use HTTP. Using a gateway PC 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.

3.2.1 Wide Area Communication

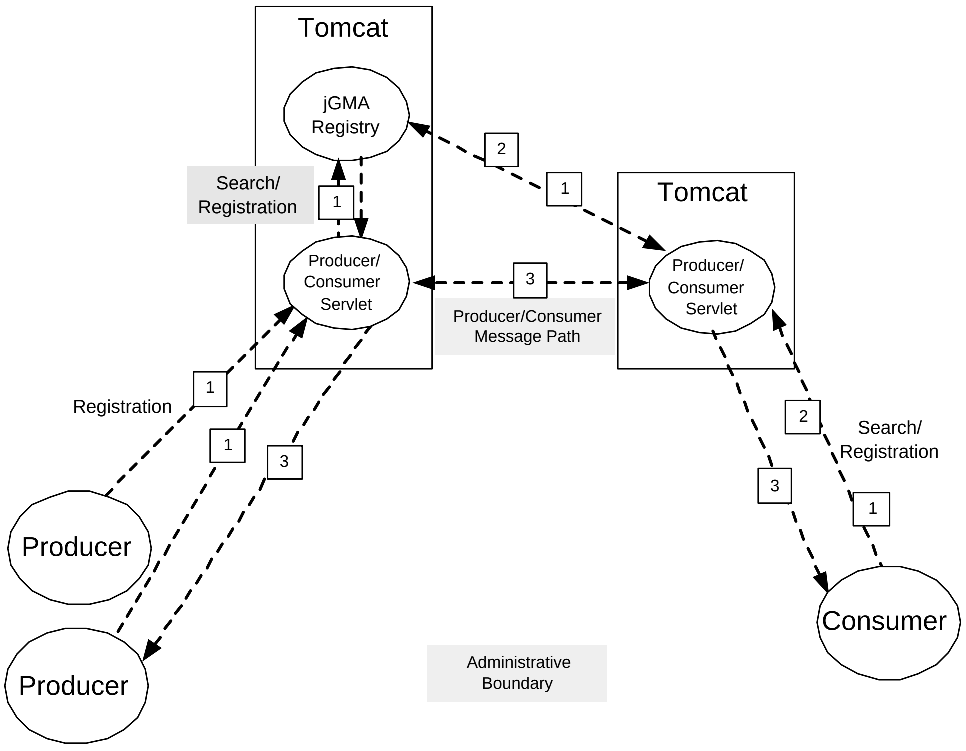

Figure 2: Wide-area jGMA Communications

Figure 2 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.

- [1] The producer and consumer register with the registry,
- [2] The consumer queries the registry for a producer service,
- [3] Consumer/producer communications.

3.2.2 Local Communications

If a consumer and a producer are registered at the same PC Servlet they will communicate directly rather than going via the servlet, there is no need to encapsulate the message in HTTP as the communication will be over the LAN.

3.2.3 Addressing within jGMA

jGMA uses a pseudo unique naming scheme to identify consumers and producers, it is made up of several components which are used route messages. While there is no guarantee that the name will be unique, the naming scheme makes it very unlikely that there will be a clash [16].

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 an XML database. 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 API has 17 calls, which are listed in Table 2.

3.4.1 jGMA Methods

3.4.2 Summary

Building on the current basic API and utilising other Java features, such as threads, can achieve 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.

#	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, cmd, 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[] newGMAByte(sequence, cmd, from, to, String data)`
	`byte[] newGMAByte(sequence, cmd, 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

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 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 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.

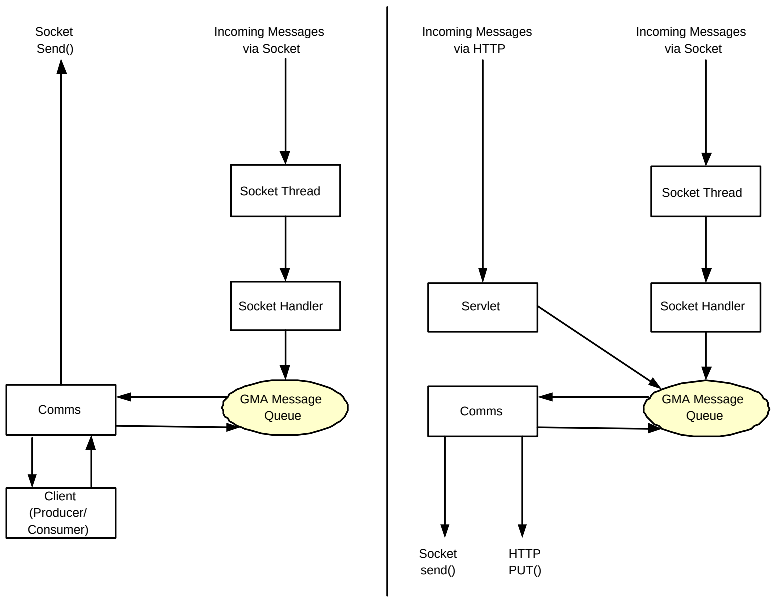

Figure 3: Internal structure of a jGMA client (left) and PC Servlet (right)

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 Communication

When jGMA stored handled messages as Java objects, wide-area communication were achieved by encoding the object as a string and sending it between PC 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 [17] allows binary data to be sent in the body of a HTTP POST request to allow uploading through HTML forms [18]. Using 'multipart/form-data' and 'multipart/mixed' it is possible to send more than one binary message in one request, or mix ASCII 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.

4.4 Summary

jGMA was incrementally adapted and evolved in light of our experiences. Once the software [19] 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 of the performance and functionality of jGMA over a local area. The overall aim of this stage was to optimise the code to reduce the communication overheads, assess impact, and confirm overall functionally. Nine nodes from the DSG cluster (their configuration is shown in Table 3) were used to perform the tests.

5.2 jGMA Benchmarking

The benchmarking was broken down into three stages:

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

1. Baseline performance: Four initial benchmarks were produced to assess blocking and non-blocking I/O performance; these where run on a single host and over Fast Ethernet. These tests were designed to show the overheads of using jGMA and the affect of message size on system throughput.
2. Scalability: These tests were used to measure scalability by determining how many messages per second a consumer can handle. Varying the number of producers sending messages to a consumer helps assess scalability; this test simulated a jGMA consumer under heavy load.
3. Wide-area communication: This benchmark was designed to test jGMA wide-area communications, in particular I/O between a producer and consumer via HTTP.

5.2.1 Generating baseline performance

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.

5.2.2 Benchmark 1 & 2 – Blocking I/O

Test 1: 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.

5.2.3 Benchmark 3 & 4 – Non-Blocking I/O

Test 2: Non-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 blocking tests.

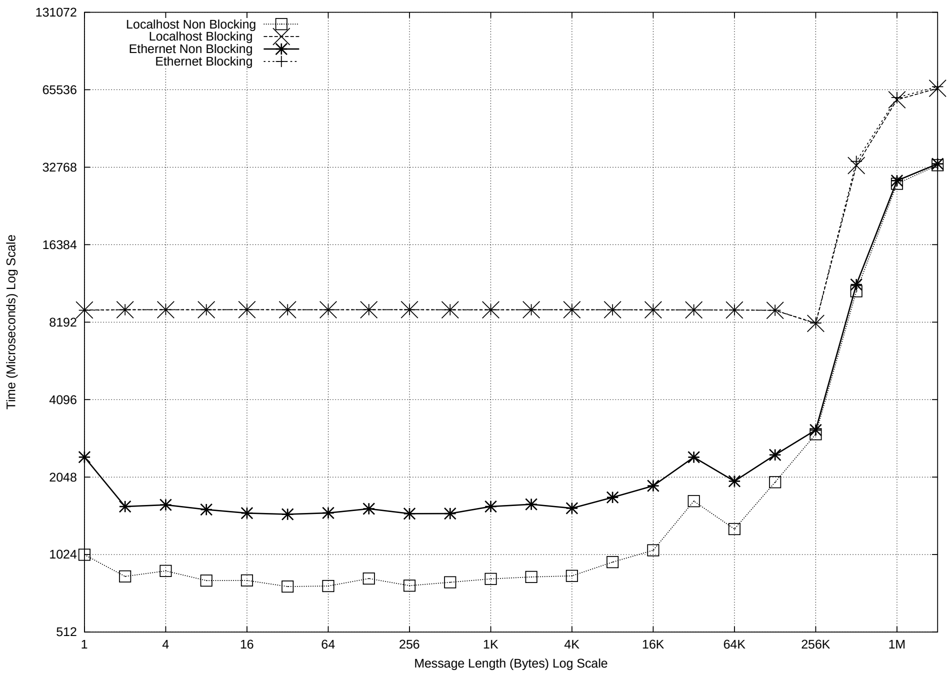

Figure 4: Latency versus Message Length

5.2.4 Blocking I/O Analysis

Both 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

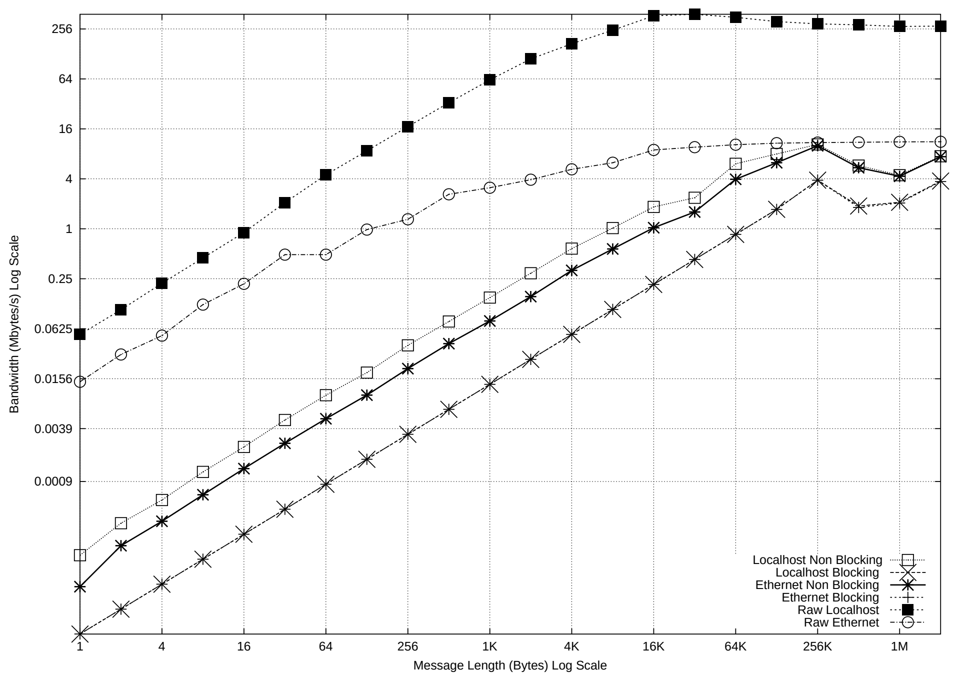

Figure 5: Bandwidth versus Message Length

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 smoothes out the effect of the transmission time until the time to send is greater than the 8 milliseconds. This means that the faster speed of localhost I/O does 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 limited by the approximately 8 millisecond queuing overhead. The peak bandwidth was 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 5.2.6.

5.2.5 Non-Blocking I/O Analysis

Both localhost and Ethernet non-blocking performance curves mirror each other's shape, with an almost constant offset up until 256 Kbytes. Thereafter, the curves merge. Both curves exhibit, what appears to be a small 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 a message latency of around 900 microseconds, whereas Ethernet has a latency of 2000 microseconds, between message lengths of 2 bytes and 64 Kbytes. The differences 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 5) shows that Ethernet and localhost I/O both achieve the same throughput. A value of 7.4 Mbytes/s for Ethernet performance is reasonable, but we would have expected that localhost I/O would achieve a much higher rate.

5.2.6 The 256 Kbyte Event

In both Figures 4 and 5 at 256 Kbytes there is a steep increase in the time to send a message. We believe that this is the effect of the TCP socket buffers used in Linux. The nodes of our cluster have their default buffer size set to 256 Kbytes (the Linux 2.4 default is 16 Kbytes). If the buffer size is reduced, the step in the curves moved to smaller messages sizes, and vise versa.

5.3 Scalability Tests

The cluster head node runs the PC servlet, 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.

5.3.1 Benchmark 5 - Scalability

The test is repeated for a range of message sizes; 8, 16 and 32 Kbyte messages were chosen as they reflect the size that GridRM is expected to produce. The hierarchical layout of GridRM's architecture means that is unlikely that more than ten producers will ever communicate directly with a single producer, testing sixteen producers should indicate whether jGMA scales to meet our needs. It is unrealistic to anticipate thousands of messages per second being produced by GridRM, a slower data rate is expected, but this test is designed to find the limitations of jGMA.

5.3.2 Scalability Analysis

Figure 6 shows that 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 out buffer, this problem is discussed in Section 6. It is

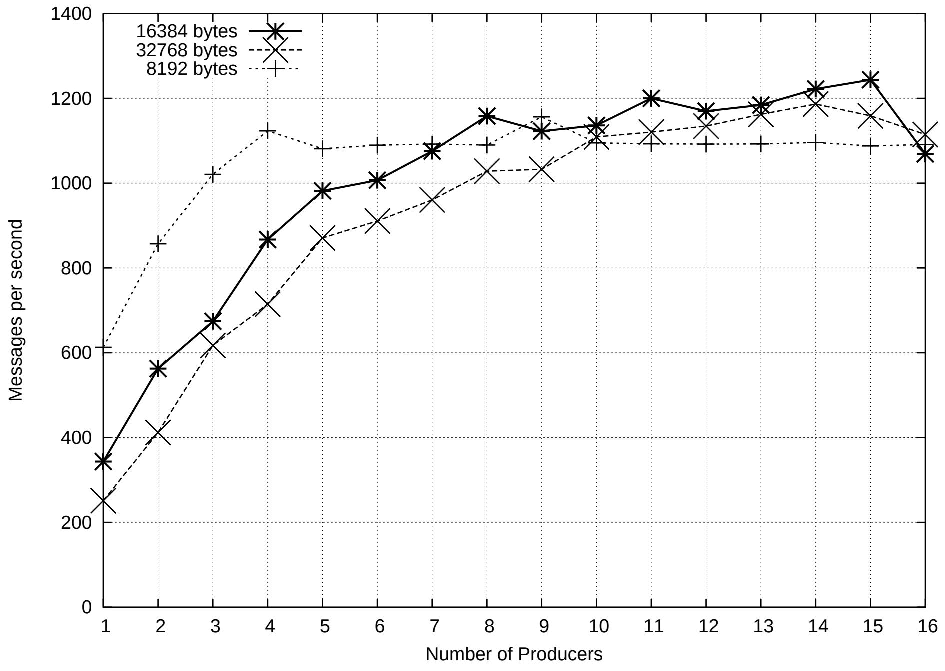

Figure 6: The number of messages per second that can handled by a single consumer, which is being sent messages from between 1 and 16 producers.

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.

5.4 Wide-Area Communications

A wide-area environment was simulated on the DSG cluster by running two PC 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 PC servlet for a range of message sizes. For an explanation of the steps in wide-area communications see Section 3.2 and Figure 2.

5.4.1 Benchmark 6 – Wide-area Communications

The benchmark made use of the same program that was used to test non-blocking I/O (Section 5.2.3) but with the addition of a second PC servlet that forces the producer and consumer to pass events via the PC servlets rather than directly via sockets.

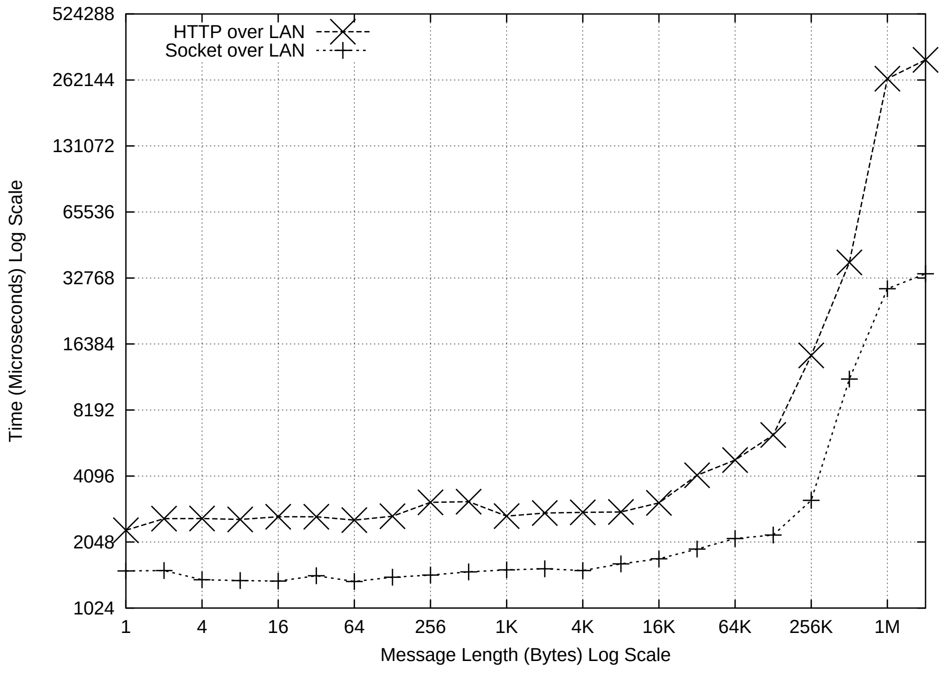

Figure 7: The latency of sending jGMA messages over Fast Ethernet via sockets and HTTP POST.

5.4.2 Wide-area Test Analysis

The HTTP POST plot (Figure 7) reflects the shape of the socket send plot up to messages of about 32 Kbytes, after this point the difference increases as the message size increases. For messages $\leq$ 32 Kbytes, HTTP POST has an added one millisecond latency, which is due to the additional sends between the clients and the PC servlets, and the overheads of processing the inter-PC servlet HTTP encapsulated message. The overhead of using HTTP via the servlets is fixed until around 64k, beyond this the multiple socket sends and HTTP encapsulation have a much larger impact on performance. The maximum throughput for jGMA over HTTP using Fast Ethernet is approximately 2 Mbytes/s.

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. jGMA, whilst being fully functional, is at an early stage of development. Consequently we are still optimizing its performance. In this paper we have presented the results of simple benchmarks that provide us with some insight into the expected performance and capabilities of jGMA.

The performance tests have shown 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 the overhead of 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 over raw sockets for Ethernet, and the peak bandwidth is 67% of the raw socket performance.

During our tests 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. However, this 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.

Without some kind of throttling back or limit-control on the buffer usage, a producer can consume all of the resources available to it and start to generate out of memory errors. While this does not crash the producer, it can mean that it does not have the resources available to receive events from a consumer, for instance instructing it to stop streaming events.

6.1 Future Work

This initial benchmarking has shown us that jGMA is functional but has a bottleneck for blocking I/O. We will investigate the means of reducing this overhead without impacting the host system. There are several other areas that we are starting to explore, including:

- Registries, we are currently adding functionality to interact with more sophisticated registries in the first instance with Xindice,
- Security, so far security in jGMA has not been addressed. This is obviously an important feature for GMA compliance, which will be added.
- Improved queue handling.

Profiling GridRM would help us choose an optimal rate limit for the production of messages. GridRM generates reasonably small messages (<32 Kbytes) and if it does not generate thousands of message per second a simple limit on the buffer sizes would not impact performance.

However this is not necessarily the case for other applications that may use jGMA, consequently a more generic solution must be found. 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 [20] or perhaps using rate feedback [21]. However, we still wish to keep jGMA lightweight and simple so the throttling control must not be over engineered or intrusive to the overall system.

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] 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] Addressing In jGMA, http://dsg.port.ac.uk/projects/jGMA/impl/

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

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

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

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

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