Building a robust distributed system: some lessons from R-GMA

P Bhatti, A Duncan, S M Fisher, M Jiang, A O Kuseju, A Paventhan and A J Wilson

Rutherford Appleton Laboratory, Didcot, OX11 0QX, UK

E-mail: s.m.fisher@rl.ac.uk

Abstract. R-GMA, as deployed by LCG, is a large distributed system. We are currently addressing some design issues to make it highly reliable, and fault tolerant. In validating the new design, there were two classes of problems to consider: one related to the flow of data and the other to the loss of control messages. R-GMA streams data from one place to another; there is a need to consider the behaviour when data is being inserted more rapidly into the system than taken out and more generally how to deal with bottlenecks. In the original R-GMA design the system tried hard to deliver all control messages; those messages that were not delivered quickly were queued for retry later. In the case of badly configured firewalls, network problems or very slow machines this led to long queues of messages, some of which were superseded by later messages that were also queued. In the new design no individual control message is critical; the system just needs to know if each message was received successfully. The system should also avoid single points of failure. However this can require complex code resulting in a system that is actually less reliable. We describe how we have dealt with bottlenecks in the flow of data, loss of control messages and the elimination of single points of failure to produce a robust R-GMA design. The work presented, though in the context of R-GMA, is applicable to any large distributed system.

1. Introduction

The Relational Grid Monitoring Architecture (R-GMA) is being developed as a generic information and monitoring system as part of the EGEE project. It offers powerful capabilities for monitoring in a distributed environment. R-GMA provides a framework to publish and retrieve information using a producer/consumer model defined by the GMA standard [1] from the OGF (or GGF as it was at the time). In this model, applications publish data using a producer, and a consumer is used to retrieve the data. This is done via the R-GMA API using SQL style commands. The properties that define the producer are stored within a registry so that when a consumer is created, it can query the registry to find producers that satisfy its search criteria. Once a list of producers is found, data then flows directly from the producer to the consumer. Queries are of 4 different types and not all producers support all query types. These are:

Continuous queries are subscriptions with producers to receive all new tuples that match the query; tuples are streamed to the consumer service automatically as they are inserted into producer storage. Continuous consumers are also registered in the registry so they can be notified if the list of relevant producers changes during the lifetime of the query.

History queries are evaluated against the current producer data, the results are streamed back to the consumer service, and then they terminate. All versions of any matching tuples are returned.

Latest queries are evaluated against the current producer data, the results are streamed back to the consumer service, and then they terminate. Only those representing the "current state" are returned.

Static queries are one-off database queries and are only supported by On-demand producers. The purpose of On-demand producers is to make external data stores accessible through the R-GMA infrastructure so it's unlikely that an On-demand producer will publish to the same table as Primary and Secondary producers.

R-GMA provides consumer, producer, registry and schema services. The schema holds a list of table definitions and, in the future, the authorization rules. Further details can be found on the R-GMA web site (http://www.r-gma.org) and the JRA1-UK web site (http://hepunx.rl.ac.uk/egee/jra1-uk/) with a full description of the R-GMA specification in the Information and Monitoring Service (R-GMA) System Specification [2]. Our earlier ideas on fault tolerance in R-GMA are described in [3]. However this paper describes rather complex algorithms for registry and schema replication which we have abandoned in place of the much simpler scheme described here.

The system has been deployed by LCG and elsewhere, however it is the LCG deployment that is most interesting because it is the largest.

We refer to three versions of the R-GMA design as indicated below:

EDG corresponding to the version developed within EDG.

EGEE-I for the version deployed in gLite 3.0

EGEE-II for the version that will be rolled out late Summer and Autumn of 2007 as upgrades to gLite 3.1

2. Managing Memory Usage

Tomcat, which is running the R-GMA services, has a certain amount of memory which must be shared between R-GMA services and other services. So even if R-GMA takes good care to minimize memory use, some other service may take it all or the virtual machine configuration may not allow Tomcat to use much memory. Prior to the EGEE-II design JDK1.4 was used. This does not provide a way to control memory use. From JDK 5 it has been possible to monitor memory usage (using an Observer) and take action when it is running low. The action is to set a flag; this is interrogated upon receipt of user calls that may take extra memory such as inserting data into the system or creating new producer or consumer resources and, if the flag is set, an RGMABusyException is sent back to the user.

In general the algorithms we use try to be "fair". As far as possible if you behave reasonably you will not be penalised however in some cases this is not possible because if the problem is caused by a lot of people making perfectly reasonable demands all we can do is reject requests with the RGMABusyException.

3. Avoiding bottlenecks in the data flow

For this we consider a typical case with many producers periodically publishing the status of something of interest to consumers. The specific conditions listed below.

- Data are being published into a single table via four producers.
- Data are being republished via two secondary producers setup up to process latest queries and without a predicate so that they collect all data published into the specified table.

- One consumer making a latest query to get the current status of something. As the two secondary producers are equivalent the query might be directed to either secondary producer.

This is depicted in Figure 1 which also shows the positions of buffers marked with a "B" in a box.

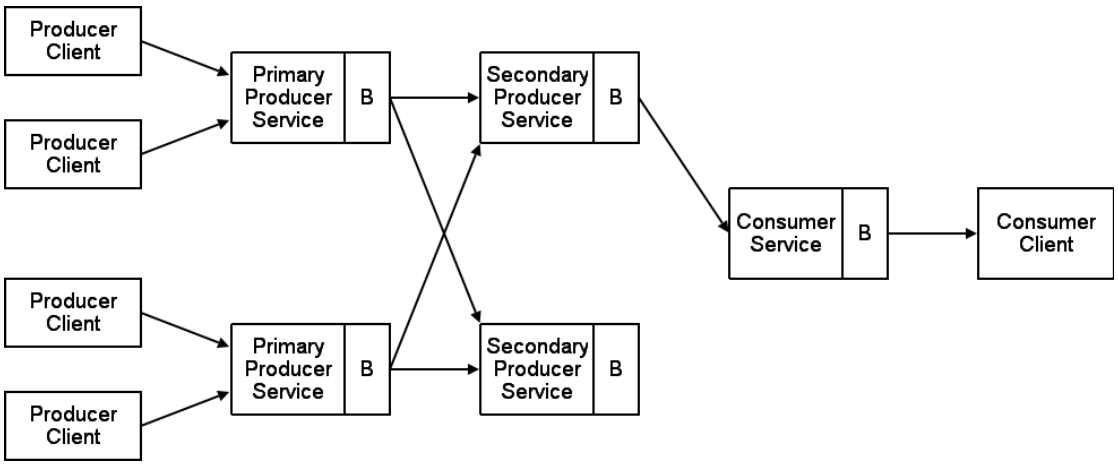

Figure 1. Typical use of R-GMA; the principal buffers are marked with the letter "B"

In the EDG design problems were encountered due to the various buffers becoming full. In some cases a full buffer could affect other components: for example a full consumer buffer could stop a producer streaming data to any other consumer as it repeatedly tried and failed to stream data to the full consumer.

3.1. Consumer Buffering

The consumer has a buffer for each client where the results of the query are stored. If the application is slow to read the data out then these buffers can fill up. We cannot send an RGMABusyException as the only contact will be from the consumer client popping data out of the buffer - which will ease the situation and is to be encouraged. If the client makes no contact with the consumer service it will get timed out. The problem comes when data is being written more rapidly into the consumer buffer than it is being taken out. The solution already adopted for the EGEE-I design is to allocate each consumer instance a certain amount of memory and when this is full data are written to disk. Once the data are all read from disk, the disk file is removed and memory is used again. Even the allocation on disk may run out. In this case we simply close the consumer. The client will notice this when he tries to pop data again as he will receive an RGMAUnknownResourceException. This allows us to cope with peaks of data.

3.2. Primary Producer Buffering

Prior to the EGEE-II design the primary producer used a simple memory buffer which could only respond to very simple queries or an RDBMS giving the full power of SQL; the buffering problem only related to the memory buffer. With the EGEE-II design an RDBMS is always used though it may be only memory resident. Either one or two tuple stores will be created: a history store for history and continuous queries and a latest store to answer latest queries. The history store must hold enough tuples to satisfy the history retention period and must in addition hold tuples for which delivery to existing continuous queries has not been attempted.

The main mechanism to control this is the RGMABufferFullException which is thrown when a producer tries to publish a new tuple and the producer has exceeded a server defined limit. Of course there could also be a very large number of producers each at this limit - in which case we have the RGMABusyException to fall back on as explained in section 2.

3.3. Secondary Producer Buffering

In the EDG and EGEE-I designs the secondary producer was made up of consumers and one producer. In the EGEE-II design this has been streamlined so that the incoming data are stored directly in the tuple store. However a memory based tuple store can grow very large and you cannot send an RGMABusyException because you have no one to send it to. For this reason we do not generally recommend using memory based tuple stores for secondary producers. We will close the secondary producer when it is unable to deal with the memory requirements implied by the requested history retention period. This way the user process that is keeping the secondary producer alive will get an UnknownResourceException when it next makes a showSignOfLife() call. This trigger will be added to the servlet code that would normally be sending the RGMABusyException.

4. Coping with loss of control messages

Figure 2 shows the principal interactions between R-GMA components in the EDG design.

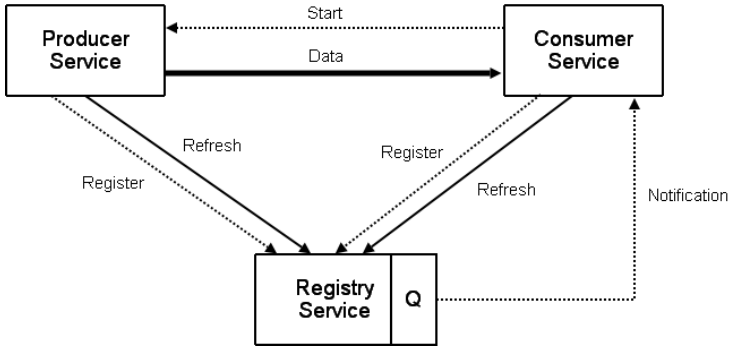

Figure 2. Principal R-GMA interactions in the EDG design. The dotted line indicates that the frequency of the message is relatively low.

In the EDG design the producer service registered a new producer instance and periodically sent refresh messages to the registry. If the producer client did not contact the producer service the refresh message was not sent and the registration of the producer was dropped. Continuous queries were registered in the same way. When the registry saw a match between a producer and consumer a notification message was added to a queue (marked "Q" in Figure 2 to be sent to the consumer to notify it of the producer; the consumer then sent a start message to the producer which responded by starting to stream data. If the message from the registry failed to reach any consumer then that consumer would never find out about the new producer.

The task of matching producers and consumers is carried out partly in the registry and partly in the consumer. This is a compromise between doing the work centrally in the registry which would be a potential bottleneck and asking all consumers to consider all new producers which would be wasteful.

In the EGEE-I design in order to improve robustness, the producers were changed to periodically send out new registration messages rather than just refresh messages. This ensured that in the event of a temporary communication problem consumers would eventually get to

hear about producers and it is no longer essential to deliver each notification message. This led to the registry having to queue a very large number of notification messages. Frequently messages were added to the queue faster than they could be sent because sites were behind firewalls or had other network problems. This blocked the registry for several minutes while the connection timed out. Two extra message queues were introduced to partially address this as shown in Figure 3.

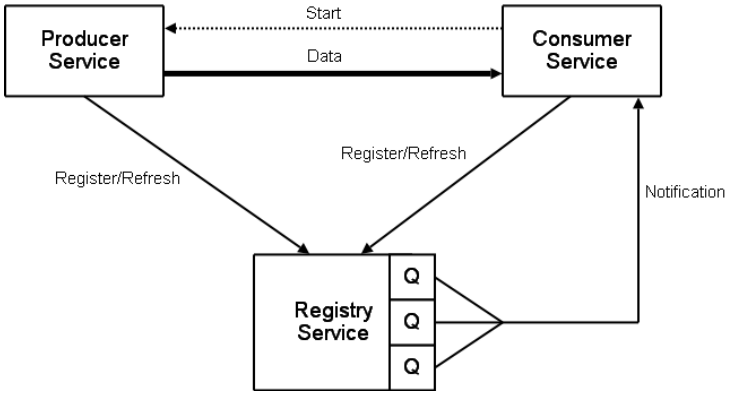

Figure 3. Principal R-GMA interactions in the EGEE-I design. The dotted line indicates that the frequency of the message is relatively low.

These was now a fast, medium and slow queue in the registry for dealing with notification messages. All messages to a site were initially put on the fast queue. If a message to a site failed twice it was moved to the medium queue as were any subsequent messages to that site. Two failures on the medium queue caused messages to that site to be moved to the slow queue. Once messages started flowing again to a site, messages to that site were once again allowed onto the faster queues. However when a problem first occurred on the fast queue then the system was still blocked until the connection timed out. The EGEE-II design addresses this with significant changes as shown in Figure 4.

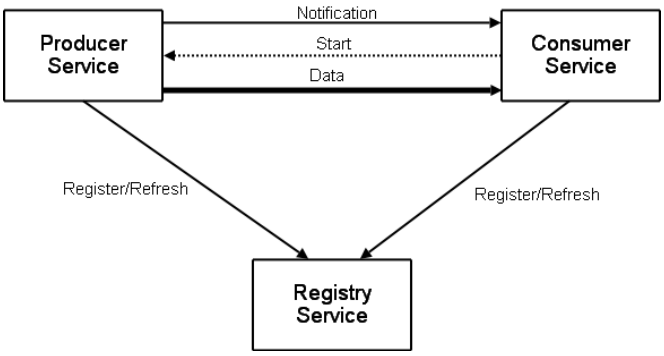

Figure 4. Principal R-GMA interactions in the EGEE-II design. The dotted line indicates that the frequency of the message is relatively low.

The registry no longer sends out notifications. The producers find the consumers and vice versa and then communicate directly. This spreads the load more evenly across the system also if a consumer is currently receiving data from a producer it knows not to bother notifying it

again. For example when a new producer registers, it is returned a set of consumers which may be interested to receive data, it then notifies them and gets a start message back from those that are interested. A producer may now need to inform a large number of consumers - and some of these consumers may be unresponsive for a number of reasons including network problems. Instead of the slow, medium and fast queues we have now implemented a specialized task manager as described below. The task manager is not shown on Figure 4 as it is used for *all* tasks that involve communication between servers except for those where a client is waiting for a response.

4.1. The Task Manager

Tasks which involve remote communication mostly run very quickly but sometimes they take a long time waiting for network timeouts. The task manager has a pool of task invocators to run tasks. We want to avoid having all the task invocators blocked trying to deliver messages to a small set of unresponsive machines. The user of the task manager must identify a key to associate with each task which relates to the potentially unreliable resources used by that task. In the simple case where a task makes remote calls to just one service then the service URL can be used as the key. There is a singleton task manager that learns which keys are associated with successful tasks. The task manager has a queue of tasks to be execute, a set of task invocators - each running as a separate thread - to attempt to run those tasks and a "goodKey" set which holds the good keys. The invocator takes a task from the queue. If the task's key is not in the goodKey set and any other task invocator is using that key the task is re-queued, otherwise it is run and returns a status of:

SUCCESS The task was successful and will not be re-queued

SOFT_ERROR The task failed but it is worth trying it again. So the task is re-queued. Note that the task must be written such that it cannot return SOFT_ERROR indefinitely.

HARD_ERROR The task failed and it is not worth trying again. It is not re-queued.

Only when an invocation is successful on the *first* attempt, the key is added to the goodKey set. If an invocation fails, even with just a SOFT_ERROR its key is removed from the goodKey set. If a task returns a HARD_ERROR then it and all other tasks on the queue with the same key will all be removed.

To ensure that threads are always available to process tasks using resources that are performing well, some invocators will be configured to only take tasks with a key in the goodKey set.

The basic algorithm for a task invocator (omitting synchronization code and other details) is:

```
if queue is empty:  queue.wait()
pop task from queue
if (task.key in goodKey set) or
           (not goodOnly task invocator and task.key not in currentTask map):
    add task.key to currentTask map
    result = task.invoke()
    increment task.attemptNumber
    if result == SUCCESS:
        if task.attemptNumber == 1:  add task.key to goodKey set
        else:  remove task.key from goodKey set
        remove task.key from currentTask map
    else if result == SOFT_ERROR:
        remove task.key from goodKey set
        remove task.key from currentTask map
        re-queue the task
    else:  # HARD_ERROR
        remove task.key from goodKey set
        removeSimilarTasks
        remove task.key from currentTask map
else:
    re-queue task
```

It is the responsibility of the owner of the task to keep a reference to any task it needs to track as the task manager keeps no record of completed tasks. However all components have been designed so that they only need to keep track of a very small number of tasks; thereby avoiding queues building up.

5. Avoiding single points of failure

A virtual Database (VDB) is used by R-GMA to give separate name spaces. The administrator of an R-GMA server chooses which VDBs it will support. This provides an m:n relationship between VDB and R-GMA server which is essential for scalability. It is possible for a query to span VDBs using either normal SQL syntax where instead of just a table name in the query it is pre-pended by the database name and a dot or by an R-GMA specific extension to allow unions over tables with the same name and structure in different VDBs. For example different organizations may choose to store GLUE information in different VDBs but the end-user wants to combine information over some set of VDBs.

Each VDB is represented by a logical schema that holds the table definitions (along with indexes, views and authorization rules) and a logical registry that holds endpoints of all producers and of consumers making continuous queries. Each producer entry is accompanied by sufficient information to allow the consumer to identify the correct set of producers to answer a query.

In the EDG design the logical schema and registry were represented by a single physical schema and registry. In the EGEE-I design we attempted to allow multiple registry instances however the design was too complex and when it was tried it was found that the overall system reliability decreased rather than increasing. In addition during EGEE-I a schema replication mechanism was devised. This was designed with one master schema - but where the master schema was chosen by the participating schemas. The major drawbacks of this design was the difficulty of holding the election and making sure that all schemas were aware of the result.

In the EGEE-II design both the schema and registry replications mechanisms were greatly simplified. The schema replication algorithm works well “in the lab” but yet needs to be deployed. We are confident in the registry replication design because it is very simple.

5.1. Schema Replication

The schema replication makes use of a master schema defined for each VDB. A configuration file for each VDB identifies the master schema. As each server has a copy of the schema for each VDB it supports, all queries are local which makes the schema very reliable. The master schema remains as a single point of failure; however it is only required for updates and we expect this to be infrequent and interactive.

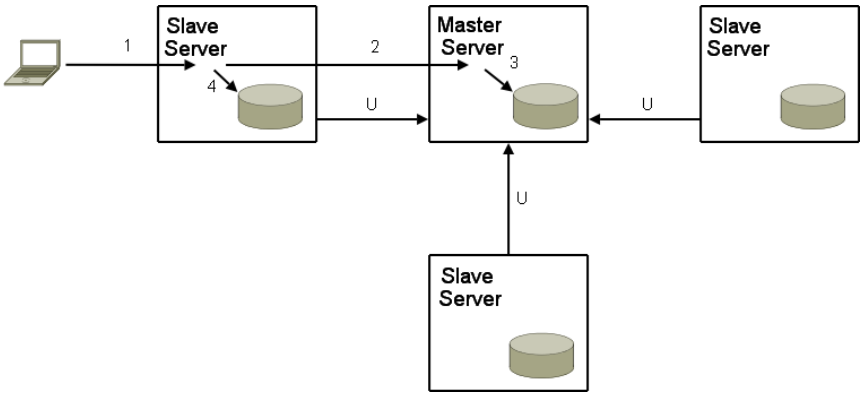

Figure 5. Schema update operations. User updates are shown with sequence numbers 1 to 4 and the regular replication queries are marked "U"

Figure 5 shows the sequence for making a schema update. The slave forwards the request to the master and only if it is successful is the local database updated. Slaves poll the master periodically to receive updates. The master does not know the slaves and does not hold any queues. The slaves ask for changes since a certain time (shown by "U" in Figure 5). If they get a reply the time stamp from the master is stored after updating the local database. Subsequent requests will ask for updates since the new time. This is very simple and should be reliable.

To allow the master to respond correctly, major items in the schema (Table, View and Index) have a time stamp associated with them. This time stamp is set by the master when the tuple is added or updated. Child tuples, such as the columns of a table have implicitly the same stamp. This means that if a column is removed from a table, the table and all its remaining columns will be transmitted upon request.

The replication makes use of the Task Manager (Section 4.1). The algorithm is summarised below:

```
oldMaster = getMaster()
if task not null:
    if the previous task has completed:
        if successful:
            call putSchemaUpdates() on the Schema Database
if oldMaster.name != master.name:
    task = ‘‘getSchemaUpdates(0)on the master Schema’’
else:
    task = ‘‘getSchemaUpdates(timestamp) on the master Schema’’
taskManager.add(task)
```

Schema changes are rather infrequent; the consumer and producer instances poll the schema periodically for each table they are using to see if they have been updated. If an update is

detected the producer or consumer will close itself down as the user probably needs to take some action. Had we used notification it would have been essential for the notification message to get through.

5.2. Registry Replication

The registry, unlike the schema, is highly dynamic. When a new registry entry is created by a direct call to that registry it is stored as a master entry and the request is stored to be transmitted at regular intervals to all other registry instances for that VDB.

The registry is updated either by direct calls or upon receipt of replication messages. A registry "owns" those records that were last changed by direct calls and is responsible for pushing updates of these records to other registries within the same VDB.

Records have an "isMaster" flag that is set by any direct add request and is cleared upon receipt of a replication message and a "lastContactTime" which controls the purging of old records however it is also used to detect a delayed replication message. This is set by the registry receiving a direct call and is subsequently replicated as part of the record.

Direct change requests are always respected, whereas replicated change requests are ignored if they are out of date (i.e. before the lastContactTime associated with the record).

If a registry should be unavailable then direct update requests will get routed to a different registry instance if there is one, and records in the new registry will get the isMaster flag set. The original registry will initially be unaware that another registry has mastership of that record. When it receives replication messages from the new master the isMaster flag will be cleared. The algorithm may result in more than one registry thinking that it is master - however this is not a problem. It also possible to have no master as the master may go down. However in all cases the system will clean up when a registry assumes mastership for the record and replicates its records.

If the clock on a registry is running too far behind time its replication messages will be ignored if the receiving registry already has an entry. If the clock is running fast then the replication message will be received and the lastContactTimeSec will be set in the future which will delay the purging of this record if it is not explicitly deleted. This may result in non-existent resources being returned by the registry.

The algorithm described above assumes that all replication messages are transmitted and processed correctly. Time stamps are associated with the replication messages so that if a message has been lost the recipient will know when the message is received. In this case a full update is transmitted. This is generated by asking the Registry Database for all records with the isMaster flag set. This recovery mechanism is essential for resilience.

A hash table is used to hold the add registration and delete registration requests keyed on the primary key of the entry. This means that if several add registration requests are followed by a delete registration request only the delete will be held. When a request for a resource registration or deletion is received by the Registry Instance, then in addition to passing the request to the Registry Database it is inserted in the hash table.

Each replication cycle a new hash table is created to take new entries and the old one is processed. A message is created from the hash table to be sent asynchronously to all replicas. If at the beginning of the next cycle any messages have not been sent they are purged and a full update will be sent instead to those replicas.

If the registry is restarted it will lose its hash table. When it starts it must send a full update to all replicas. If the registry has been down for some time other registries will have taken mastership and most records will be ignored as they will be too old.

6. Conclusions

We have explained how, by following a few simple principles, we will have a much more reliable system. These principles are:

- Try to think of everything that can go wrong.
- Keep it simple.
- Polling is much simpler though less efficient than notification.
- Make the system self correcting and avoid critical messages.
- Avoid single points of failure.
- Reject incoming requests if a server cannot cope rather than just going slowly or crashing.
- Server code should protect itself against running out of memory.
- External conditions can change at any time: it is not good enough to just check at service startup.

We are confident that this will provide a highly robust and scalable R-GMA to attract new "customers".

Acknowledgments

We wish to thank GridPP and EGEE who have jointly funded this work within the EGEE-II project.

References

[1] Tierney B *et al.* 2002 A grid monitoring architecture. Tech. rep. GGF

[2] JRA1-UK 2004 Information and monitoring service (R-GMA) system specification Tech. Rep. EDMS 490223 EGEE

[3] Byrom R, Coghlan B, Cooke A, Cordenonsi R, Cornwall L, Craig M, Djaoui A, Duncan A, Fisher S, Gray A, Hicks S, Kenny S, Leake J, Lyttleton O, Magowan J, Middleton R, Nutt W, OCallaghan D, Podhorszki N, Taylor P, Walk J and Wilson A 2005 *Advances in Grid Computing - EGC 2005* (*Lecture Notes in Computer Science* vol 3470/2005) (Berlin/Heidelberg: Springer) pp 751–760