Fault Tolerance in the R-GMA Information and Monitoring System

Rob Byrom[4], Brian Coghlan[6], Andy Cooke[1], Roney Cordenonsi[3], Linda Cornwall[4], Martin Craig[4], Abdeslem Djaoui[4], Alastair Duncan[4], Steve Fisher[4], Alasdair Gray[1], Steve Hicks[4], Stuart Kenny[6], Jason Leake[7], Oliver Lyttleton[6], James Magowan[2], Robin Middleton[4], Werner Nutt[1], David O'Callaghan[6], Norbert Podhorszki[5], Paul Taylor[2], John Walk[4], and Antony Wilson[4]

[1] Heriot-Watt University, Edinburgh
[2] IBM-UK
[3] Queen Mary, University of London
[4] Rutherford Appleton Laboratory,
[5] SZTAKI, Hungary
[6] Trinity College Dublin
[7] Objective Engineering Ltd.

Abstract. R-GMA (Relational Grid Monitoring Architecture) [1] is a grid monitoring and information system that provides a global view of data distributed across a grid system. R-GMA creates the impression of a single centralised repository of information, but in reality the information can be stored at many different locations on the grid. The Registry and Schema are key components of R-GMA. The Registry matches queries for information to data sources that provide the appropriate information. The Schema defines the tables that can be queried. Without the combined availability of these components, R-GMA ceases to operate as a useful service. This paper presents an overview of R-GMA and describes the Registry replication design and implementation. A replication algorithm for the Schema has also been designed.

1 Introduction

R-GMA is a monitoring and information application for grid environments. It is an implementation of the Grid Monitoring Architecture [2] proposed by the Global Grid Forum [3]. R-GMA was developed within the European DataGrid [4] project. . It is currently being re-engineered as web services, within the EGEE (Enabling Grids for E-Science in Europe) project.

It has been used as the information and monitoring service within the DataGrid project, where it provided information on Computing and Storage Elements which was then used by the Resource Broker to decide where to submit jobs. Network monitoring data were also published using R-GMA [5]. A description of the use of R-GMA within DataGrid, along with detailed performance measurements (including indications of the performance gain with replicated registries) is about to be published [6].

P.M.A. Sloot et al. (Eds.): EGC 2005, LNCS 3470, pp. 751 – 760, 2005.

The LHC Computing (LCG) project [7] aims to meet its computing needs by deploying a grid composed of computing resources distributed across many different countries. R-GMA is being used to implement an accounting service on LCG.

Santa-G (Grid-enabled System Area Networks Trace Analysis) was used to develop a network traffic monitoring application within the CrossGrid [8] project. This NetTracer application publishes information using R-GMA that can be used for the validation and calibration of intrusive monitoring systems, and also for analysing the performance of a site in a network.

2 Grid Monitoring Architecture

The Grid Monitoring Architecture models the information infrastructure of the grid as a set of Producers that provide information, and a set of Consumers that request information. A Registry contains details of Producers and Consumers. Producers contact the Registry to announce the structure of the data they provide. Consumers contact the Registry to find Producers that publish information they require. The Registry returns to the Consumer a list of Producers that provide the desired information, and the Consumer then contacts the appropriate Producer or Producers to obtain this information.

3 Virtual Database

R-GMA presents a global view of the information produced by the components of a grid system and by applications running on the grid. It presents this information as a virtual database, introducing the relational data model to the Grid Monitoring Architecture. Information can be inserted into and retrieved from the R-GMA virtual database using a subset of the statements specified in the SQL92 Entry Level standard. To a user who queries this virtual database, it seems as though they are querying a single, centralised repository of information. In actual fact, the information obtained from this virtual database can be stored at a number of different locations.

Each table has a *key* column (or group of columns). In addition, each tuple inserted by a Producer has a timestamp added to it by the Producer. The combination of this timestamp and the key columns is similar to a primary key for the table. This enables R-GMA to provide time-sequenced data to monitoring applications where users may wish to request data published within a particular time interval. Of course for this to be reliable, all systems in an R-GMA installation must synchronize their system clocks using a tool suc as NTP.

4 Producer Service

The Producer service is used to publish data to the virtual database. The R-GMA schema contains the definitions of all tables in the virtual database, and Producers declare their intention to publish to a table by registering themselves with the Registry. The Producer service obtains the table structure from the Schema and

creates a table with identical structure in its own storage. Data in the Producer's storage has a retention period associated with it. When data has been in the storage for longer than this retention period, it is deleted. The length of this retention period is dependent upon how long users require access to tuples after they have been published. There are three kinds of Producers. The difference between them is where the data they publish is coming from.

- *Primary Producer:* The user's code inserts tuples directly into the Producer's storage.
- *Secondary Producer:* The Secondary Producer republishes tuples that have already been published by another Producer. It queries Primary Producers and other Secondary Producers and inserts the tuples returned from these queries into its own storage. This can be useful for creating a service that archives the data from multiple Producers, or for answering queries that require joins by combining tables from multiple Producers.
- *OnDemand Producer:* It may not be practical to transfer a large volume of data to a Producer with SQL INSERT statements. The OnDemand Producer allows a user to request such information when it is required. Only then is the information encoded in tabular form and sent to the user. The OnDemand producer has no tuple storage facility. It sends all queries it receives to additional user code which sends tuples back to the consumer.

5 Consumer Service

Consumers allow users to execute SQL queries on the R-GMA virtual database. The Consumer contacts the Registry to find the Producers required to answer the query. It then sends the query to these Producers and collects the tuples returned into an internal store for subsequent retrieval by the user. Consumers allow four types of query:

- *Continuous:* All tuples matching a query are to be automatically streamed to the Consumer when they are inserted into the table. All Primary and Secondary Producers support continuous queries, but OnDemand Producers do not. An example of where receiving a continuous stream of data from a producer is of use would be a real-time job monitoring application, which must update the status of jobs on the grid as they are executing.
- *Latest:* The most recent version of a tuple matching a query is returned to the user. A resource broker which decides where jobs should be executed on a grid may use a Latest producer. The resource broker needs up-to-date information about the resources on the grid. It is only interested in the most recent state of the resources, not their state over a period of time.
- *History:* All available tuples matching a query are returned. This is useful when the Producer is to be used as part of an archiving application.
- *Static:* These are specific to OnDemand Producers, and are handled like a normal database query (no timestamps or intervals associated with these queries).

6 Registry Service

The Registry enables R-GMA to match Consumers to Producers that provide the information they require. Producers advertise what tables and parts of tables in the virtual database they produce rows for. Consumers can consult the registry to find Producers that can answer their queries. The process by which the Registry finds Producer(s) that are capable of providing information requested by a Consumer is called mediation[9]. Mediation allows a Consumer to have a global view of information from sources distributed across the grid.

7 Schema Service

R-GMA uses a Schema service to define the tables that comprise the virtual database. It also defines the authorization rights for these tables. The Schema service allows operations such as adding or removing tables to/from the virtual database, or getting the name, type and attributes of all columns in a particular table.

8 Namespaces

The term "Virtual Organisation" (VO) [10] has been formulated to represent a set of resources that are shared by a number of users, and rules that specify users' access rights to resources. Individuals and institutions working on the same problems (for example, a community of researchers working in a particular area, such as High Energy Physics or Earth Observation), benefit from pooling their resources and making them available to all members of the community. It is possible that a user can be a member of more than one VO.

R-GMA allows users to issue queries to multiple VOs. A set of VOs may contain semantically different tables with identical names. Clearly a global namespace defined by just the table names is not scalable. The R-GMA solution is to form a virtual global namespace by giving each VO its own disjoint subset of the namespace. Information particular to each VO is contained in a virtual database, the name of the virtual database being that of the VO. Although it is possible to have a single registry and schema, scalability concerns strongly favour that each VO has a separate set of schema and registry services. To issue a query to a particular VO using a consumer, the table name in the SQL query must be prefaced with the VO name. For example, for the LCG VO:

```
SELECT NumberOfJobsRunning FROM LCG.Workload WHERE
Site='RAL'
```

For publishing information we could adopt the same syntactic approach. However as it is common to wish to publish the same information to multiple VOs, the set of VOs that a tuple is published to is included as a separate parameter in the Producer API function call, and is not included in the SQL INSERT statement.

9 Registry Replication

Although there is only one logical Registry per VO, replicas of the Registry are maintained. If a particular Registry fails at any time, an alternative Registry is used instead. This enables the client (i.e. the Consumer or Producer) to carry out a Registry operation in spite of any faults that occur within individual components of the replicated Registry group.

9.1 Selecting a Registry

Clients wishing to connect to a Registry do so by using a selection algorithm, which is carried out internally within the Registry API. A list of available Registry services is provided via a configuration file on the client. During runtime, a simple profile is created by contacting each Registry service identified in the configuration. The quickest response time is then used to select the Registry to handle the client request. This Registry will then be re-used for all further client interaction unless the Registry becomes unavailable - at which point a new one is selected using the same procedure.

A reliable Registry Service can be demonstrated with only a small set of Registries, typically between 2 to 4 per VO. Any overhead incurred from contacting each Registry is therefore acceptable for small groups of Registry peers. A more sophisticated profiling algorithm, perhaps based on the load average or overall reliability of a Registry, might benefit larger installations.

9.2 Registry Replication Design

Registry replication is based on a distributed model where each Registry pushes locally acquired data to its peers. New data is obtained when a Consumer or Producer carries out a registration process. When this occurs, an entry is copied into the local Registry database along with a *replica status* flag indicating the data is fresh. Conversely, the *replica status* flag is set to delete when a Consumer or Producer is closed by the client. A further state referred to as “replicated” is encoded by the *replica status* flag and represents rows that were copied in a previous replica update. An additional *origin* tag is added to each entry that identifies the Registry where the new data was added.

A dedicated thread that runs periodically on each Registry replica initiates the replication process. It checks the existence of newly arrived registrations by reading the replica status flag of entries in the Registry. An additional check is made to ensure the origin matches the current Registry so only local data is considered.

Once a list of candidate data is identified for replication, a replica message is constructed which encodes all the records within an XML format. XML was chosen due to open source API’s which provide XML parsing functionality. Although XML imposes additional padding to a replica message, the schema adopted has been carefully designed to help minimize the message size. On average, XML encoding increases the replica message by a factor of 1.5.

The Registry reads from a configuration file the URLs of all Registry and Schema Servers that are members of the same VO. When the XML message is constructed, a checksum is then computed based on the state of the Registry and the final replica

message is then sent to each Registry peer. A list of available Registries is read-in from a configuration file each time the replication cycle is scheduled. When a replica message is received, the message is passed through a filter that works out in which table each record should be stored or removed. Once each record in the replica update is processed, the checksum is re-calculated to validate the consistency of the databases between the sender and recipient. If the checksums match up, a successful response message is returned to the sender. At that point, the *replica status* flag of all the copied records on the sending registry are set to "replicated" to prevent further replication in future.

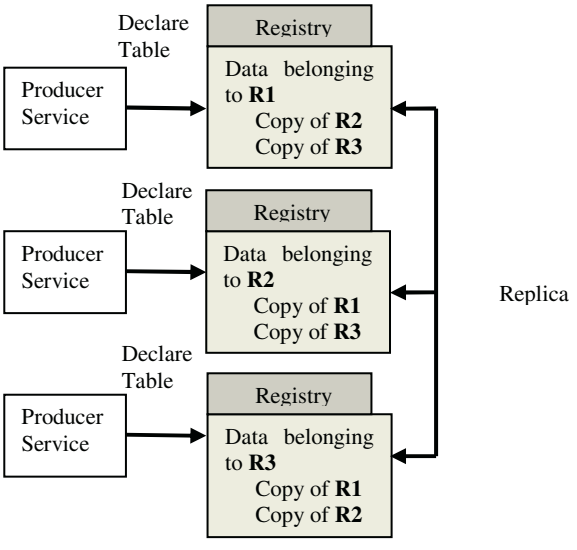

Fig. 1. Only data local to a Registry is replicated

If the checksums do not match, the replication attempt has failed, indicating that the data stored within the sender's database is inconsistent with that of its peer. In order to resolve this conflict, the original sender will copy all records that match the Registry's origin irrespective of whether the record has been replicated or not. The recipient Registry will then delete all records that match the sender's origin and then perform a clean install using the new replica message. This therefore ensures a high level of consistency is preserved, and that errors cannot go unnoticed.

A checksum failure means a Registry may have provided inaccurate mediation prior to correction. In the worst case, a Producer or Consumer may receive false information about a resource that no longer exists. In this case, the Producer and Consumer Service are robust enough to quietly dispose of the false information.

When a replica update is successfully stored (whether it is a complete or partial update) the receiving registry will attempt to mediate newly replicated Consumers with any locally stored Producers which do not match the sender's origin. If a complete update is made, then it is possible a Consumer may receive duplicate calls

for the same producer. If this occurs, the Consumer Service will simply ignore the duplicate. The mediation phase ensures the query plans used by the Consumers and Secondary Producers are therefore complete.

9.3 Batching Updates

A Registry replication update is executed as one large batch. While this approach helps to reduce the overall load on each Registry, some inconsistency is introduced between successive batch updates. However, the Registry copes with any potential inconsistencies by performing an extra mediation phase once a batch update is received. As mentioned previously, this mediation phase ensures that any Consumers that may have missed a Producer are then notified. In the worst case, a Consumer may not receive data until the next batch update. By default, this is set to an interval of 30 seconds but can be re-configured. This approach seems to work, but scalability may require the replication interval to be automatically adjusted, based on Registry usage.

10 Schema Replication Design

We have considered several possible designs for a schema replication system, mostly based around master-slave or atomic commitment protocols. All have struggled to balance the requirements of global consistency, fault tolerance and scalability. In fact we can relax the first of these. Schema replicas don't have to be identical for R-GMA to operate safely, it is only necessary that individual table definitions (column names and types) be uniquely defined. We do permit table names to be re-used (dropped and re-created), but tables are identified internally by a *table number* which is unique for all time, and we don't permit table definitions to be modified. Since mediation is done on table numbers, not names, there is no possibility of the mediator matching up producers and consumers that have different ideas about a table's definition. Of course permitting schema replicas to get temporarily out of step (which could result in consumers not being matched up with producers, and vice versa) is not ideal, but it is safe, and in practice it is very unlikely to occur because dropping and re-creating tables is a rare event, especially on tables that are still being used elsewhere in the VO. With these observations, the replication problem reduces to designing an allocation system for unique table numbers for *new* tables, along with a simple scheme to re-synchronize replicas in slower time, following any changes.

10.1 Schema Write Operations

Changes to the schema are, of course, visible across the whole VO. We can therefore assume that schema write operations will require special privileges, will be relatively rare, and will probably be initiated by hand. We'll see that these assumptions influence some of the design choices we make in the replication design. The write operations are *createTable*, *dropTable* and *alterTableAuthorization*. It turns out that it is convenient, for re-synchronization, to allocate a unique version number to the schema itself which is changed following any write operation. It can also be used as the new table number for a *createTable* operation. Thus the algorithm for all schema write operations is broadly the same:

1. Obtain a write lock on the schema database.
2. Agree a new schema version number with the other replicas, as described below.
3. Commit the schema changes and the new version number to the database, as a single transaction (so schema readers see an atomic change).
4. Release the database lock.
5. Schedule an immediate re-synchronization with the other replicas.

Read operations on the schema are allowed to continue throughout this process.

10.2 Schema Version Number Allocation

Each schema replica has a *version* property and a *nextVersion* property stored in its database. These are both integers, initialised to zero when the replica is first created. Each replica also has access to a list of URLs of all other replicas. Finally each has a *newVersion(proposedVersion)* operation called by other replicas when they want it to agree a new version number, defined as follows:

1. Obtain a write lock on the local schema database.
2. Get the current value of *nextVersion*.
3. If (*nextVersion* < *proposedVersion*)
   Set *nextVersion* to *proposedVersion* and commit it to the database.
   Else
   Set *nextVersion* to (*nextVersion* + *1*) and commit it to the database.
   End If.
4. Release the database lock.
5. Return *nextVersion* to the caller.

When a replica wants to agree a new schema version number, it first increments its own *nextVersion* number and commits it to its database, then proposes it to all other replicas by calling *newVersion()*, repeating the process until agreement is reached. Although a synchronous call to each replica is potentially slow, it is a reasonable approach for a manually initiated operation, given the relatively small number of replicas a VO is likely to require. A time-out is used to make sure the call to each replica completes or fails in a reasonable length of time, and only a majority of replicas are required to reply for the process to succeed. The exact procedure is as follows:

1. Increment the *nextVersion* property to (*nextVersion* + *1*) and commit it to the database.
2. Loop through the list of replicas, sending a *newVersion(nextVersion)* request to each of them.
3. If (fewer than (N + 2) / 2 reply, for N replicas)
   Abandon the operation.
   Else if (all replicas accept *nextVersion*)
   Use *nextVersion* as the new schema version number.
   Else if (this is the second attempt to obtain a new version number)
   Abandon the operation.
   Else
   Set *nextVersion* to the highest version number returned by any replica.
   Commit *nextVersion* to the database, and try again.
   End If.

It is essential that a replica always increments its *nextVersion* property before proposing it to the other replicas, so that concurrent instances of this algorithm don't end up with the same number. There is still a possible race condition where two simultaneous instances contend with each other for a new version number and the process never terminates, but again, since schema write operations are manually initiated, it is sufficient just to make a limited number of tries before giving up with a suitable error message.

10.3 Synchronizing Replicas

A replica synchronizes itself with another replica by sending a synchronization request containing its own *version* number to the remote replica. The remote replica returns its *version* number in reply. If the remote replica is at a higher version number, it also returns a copy of itself with a checksum, much as in the registry replication. If, it is at a lower version number, it schedules its own synchronization request. A replica sends synchronization requests to all other replicas when it first starts up (selecting the most up to date reply) and when it completes a write operation. It also periodically sends re-synchronization requests to any replicas which it believes to be out of date (based on an in-memory list), to prompt them to re-synchronize themselves.

10.4 Notifying Producers and Consumers of Schema Changes

Normally, producer and consumer instances don't need to be notified about schema changes, because they retain their own copy of table definitions for any tables on which they are currently processing queries, and these table definitions are immutable. Table authorization can change, however, and since producer and consumer services are responsible for imposing table authorization, these changes may need to be propagated immediately.

11 Conclusions

R-GMA provides a global view of information produced by applications distributed over a grid. The Registry and Schema are key components of R-GMA. If these components are unavailable, R-GMA ceases to operate as a useful information and monitoring service. Replication algorithms for both the Registry and Schema have been designed. If one of these components fails at any time, an alternative working Registry/Schema can be used instead. When replicas of the Registry and Schema are created, these components are no longer single points of failure within R-GMA. This improves the scalability and fault tolerance of R-GMA.

References

1. A. Cooke, A. Gray et al., *R-GMA: An Information Integration System for Grid Monitoring*, in Proceedings of the Tenth International Conference on Cooperative Information Systems (2003).

2. B. Tierney, R. Aydt, D. Gunter, W. Smith, V. Taylor, R. Wolski and M. Swany, *A Grid Monitoring Architecture*, GGF 2001. http://www-didc.lbl.gov/ggf-perf/gma-wg/papers/gwd-gp-16-3.pdf.
3. Global Grid Forum: http://www.ggf.org/
4. DataGrid project: http://www.edg.org/
5. F. Bonnassieux, *Final Report on Network Infrastructure and Services*, deliverable for DataGrid Project, 2003.
6. A.Cooke, A.Gray et al., *The Relational Grid Monitoring Architecture: Mediating Information about the Grid*, to be published in Journal of Grid Computing
7. LHC Computing Grid Project: http://lcg.web.cern.ch/
8. CrossGrid project: http://www.crossgrid.org/
9. A. Cooke, A.J.G. Gray and W. Nutt, *Stream Integration Techniques for Grid Monitoring,* Journal on Data Semantics, 2, 2004. LNCS 3360
10. I. Foster, C. Kesselman, S. Tuecke, *The Anatomy of the Grid: Enabling ScalableVirtual Organizations*, International Journal of High Performance Computing Applications. 15(3), 2001