



DataGrid

Information and Monitoring (WP3) Architecture Report

Design, Requirements and Evaluation Criteria

WP3: Information and Monitoring Services

Document identifier:	DataGrid-03-D3.2-334453-4-0
Date:	31 Jan 2002
Workpackage:	WP3: Information and Monitoring Services
Partners:	IBM, PPARC, SZTAKI
Lead Partners:	PPARC
Document status:	APPROVED
Deliverable identifier:	DataGrid-D3.2

Abstract: This document presents the WP3 architecture, some Use Cases and requirements along with the design, evaluation criteria and current APIs.

Delivery Slip

	Name	Partner	Date	Signature
From	Steve Fisher	PPARC	31 Jan 2002	
Verified by	Peter Kunszt	CERN	1 Feb 2002	
Approved by	PTB		4 Feb 2002	

Document Log

Issue	Date	Comment	Author
0-0	24 Oct 2001	Submitted for review	WP3
1-0	1 Dec 2001	Major revision - submitted for review	WP3
2	15 Jan 2002	Minor revision - submitted to moderator	WP3
3	19 Jan 2002	Minor revision - submitted to PTB	WP3
3-1	31 Jan 2002	Minor revision - following PTB	WP3
4-0	31 Jan 2002	Final release	WP3

Document Change Record

Issue	Item	Reason for Change

Files

Software Products	User files
L ^A T _E X	http://edms.cern.ch/document/334453/4-0

Contents

1	Introduction	5
1.1	Objectives of this document	5
1.2	Application area	5
1.3	Applicable documents and referenced documents	5
1.4	Document Amendment Procedure	7
1.5	Terminology	7
1.6	Document Structure	8
1.7	Acknowledgements	8
2	Executive Summary	9
3	Architectural Overview	10
3.1	What we mean by Information and Monitoring System	10
3.2	Desired features of an Information and Monitoring System	10
3.3	Comparison with LDAP based approaches	13
3.4	Not an RDBMS	13
3.5	Component types	14
3.6	Security	17
3.7	Variation from the technical proposal	17
3.8	Innovation and Cooperation	17
4	Use Cases	18
4.1	Network performance - WP7	18
4.2	Logging and Bookkeeping - WP1	19
5	Requirements	21
5.1	General	21
5.2	Producers	21
5.3	Consumers	22
5.4	Robustness	22
5.5	Extra facilities	22
5.6	Ease of use and performance	22
5.7	Security	23



6	Design	24
6.1	Servlets	24
6.2	HTTP/XML based protocols	25
6.3	Soft state registration	27
6.4	Security	28
6.5	Timestamps	29
6.6	Configuration files	29
6.7	API code	30
7	Sensors and Displays	32
7.1	MDS Producer	32
7.2	NetLogger	33
7.3	GRM and PROVE	33
8	Evaluation Criteria	37
8.1	Functionality	37
8.2	Performance	37
A	XML based protocols	38
A.1	CircularBufferProducerServlet	38
A.2	DataBaseProducerServlet	42
A.3	RegistryServlet	44
A.4	SchemaServlet	46
A.5	ConsumerServlet	47
A.6	ArchiverServlet	50
B	Java API	52
B.1	Classes	53

1 Introduction

1.1 Objectives of this document

The technical annex explains in relation to WP3 that:

The aim of this work package is to specify, develop, integrate and test tools and infrastructure to enable end-user and administrator access to status and error information in a Grid environment and to provide an environment in which application monitoring can be carried out. This will permit both job performance optimisation as well as allowing for problem tracing and is crucial to facilitating high performance Grid computing.

The annex defines Task 3.1 to be:

Task 3.1: A full requirements analysis and architectural specification will be performed. Interfaces to other sub-systems will be defined and needs for instrumentation of components will be identified. Message format standards will be set up.

The principal output of this task is deliverable D3.2:

D3.2 (Report) Month 9: Detailed architectural design report and evaluation criteria.

This document, which is deliverable D3.2 therefore sets out to document the architecture of a system able to provide the necessary infrastructure and tools for an Information and Monitoring system for the DataGrid.

There has been some adjustment to the set of issues that we consider to be most important from that envisaged in the technical annex. This is explained in Section 3.7

1.2 Application area

Localised monitoring mechanisms will be developed to collect information with minimal overhead and to *publish* the availability of this in a suitable directory service. We will gather information from computing fabrics, networks and mass storage sub-systems as well as from instrumented end-user applications. Interfaces and gateways to monitoring information in these areas will be established and APIs will be developed for use by end-user applications.

The information and monitoring system can be seen to be a fundamental Grid technology. All work packages of the DataGrid will make use of it either to find information or to make information available to others.

The full applicability of this work should be apparent from the Use Cases in Chapter 4

1.3 Applicable documents and referenced documents

- [1] Zoltán Balaton, Peter Kacsuk, Norbert Podhorszki, and Ferenc Vajda. Comparison of representative grid monitoring tools. Technical Report LPDS-2/2000, Laboratory of Parallel and Distributed System, Hungary, 2000.

-
- [2] German Cancio, Steve M Fisher, Tim Folkes, Francesco Giacomini, Wolfgang Hoschek, Dave Kelsey, and Brian L Tierney. The datagrid architecture. Version 2, Jul 2001.
 - [3] Brian Coghlan. A case for relational GIS/GMA using relaxed consistency. Technical Report GWD-Perf-11-1, GGF, 2001.
 - [4] Brian Coghlan, Abdeslem Djaoui, Steve Fisher, James Magowan, and Manfred Oevers. Time, information services and the grid. In K D O'Neill and B J Read, editors, *BNCOD 2001 - Advances in Database Systems*, number RAL-CONF-2001-003 in RAL-CONF. BNCOD, 2001.
 - [5] Karl Czajkowski, Steven Fitzgerald, Ian Foster, and Carl Kesselman. Grid information services for distributed resource sharing. In *Proc. 10th IEEE International Symposium on High-Performance Distributed Computing (HPDC-10)*. IEEE Press, 2001.
 - [6] Peter Dinda and Beth Plale. A unified relational approach to grid information services. Technical Report GWD-GIS-012-1, GGF, 2001.
 - [7] E.F.Codd. A relational model of data for large shared data banks. *CACM*, 13(6), jun 1970.
 - [8] E.F.Codd. Recent investigations into relational data base systems. In *Proc. IFIP Congress 1974*, 1974.
 - [9] B. Tierney et al. The NetLogger methodology for high performance distributed systems performance analysis. IEEE HPDC-7, Jul 1998.
 - [10] D. Gunter et al. NetLogger: a toolkit for distributed systems performance analysis. IEEE Mascots 2000, Aug 2000.
 - [11] Steve Fisher. Relational model for information and monitoring. Technical Report GWD-Perf-7-1, GGF, 2001.
 - [12] Ian Foster, Carl Kesselman, and Steven Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. Technical report, GGF, 2001.
 - [13] Global grid forum. <http://www.gridforum.org/>.
 - [14] S. Gullapalli, K. Czajkowski, C. Kesselman, and S. Fitzgerald. Grid notification framework. Technical Report GWD-GIS-019-01, GGF, 2001.
 - [15] Dan Gunter, Brian Tierney, and Ruth Aydt. Timestamp model for grid computing. Technical Report GWD-PERF-014-1, GGF, 2001.
 - [16] MySQL. <http://www.mysql.com>.
 - [17] SWIG. <http://www.swig.org>.
 - [18] Brian Tierney, Ruth Aydt, Dan Gunter, Warren Smith, Valerie Taylor, Rich Wolski, and Martin Swany. A grid monitoring architecture. Technical Report GWD-Perf-16-1, GGF, 2001.
 - [19] Definition of architecture, technical plan and evaluation criteria for scheduling, resource management, security and job description. Technical Report DataGrid-01-D1.2-0112-0-3, DataGrid, Sep 2001.
 - [20] DataGrid user requirements and specifications for the DataGrid project. Technical Report DataGrid/08/D8.1a/0104-1, DataGrid, 2001.

1.4 Document Amendment Procedure

It is important to note that this document describes a work in progress and not a fully developed architecture. Because we have had to work very quickly to get code ready for the first test-bed, we have adopted the ideas of “Extreme Programming”. Features of this approach are:

- Comprehensive unit tests,
- Short release cycles,
- Adding only what’s needed for the current task,
- Collective code ownership,
- Continual improvement, and
- Adding features in the order of importance.

A document such as this would only be completable at the end of the project. We now have a good body of code and have defined protocols *sufficient for our current implementation*. These protocol definitions and the API are included as appendices - they are incomplete but accurately reflect where we are today. The architectural features described in the body of the document come into various categories:

- those implemented in code, and which work to our satisfaction,
- those implemented in code but which could be improved,
- those we believe we know how to implement in code and
- those we need to design.

We have tried to make it clear in the style of text which category is being explained - and to point out when there are alternatives to consider.

This document will be maintained and updated regularly by WP3. An up to date version will accompany each release as part of the documentation.

1.5 Terminology

For more detail, please see the index.

API Application Program Interface

CE ComputingElement: a Grid-enabled computing resource

GGF Global Grid Forum: <http://www.gridforum.org/>

GMA Grid Monitoring Architecture: monitoring architecture defined by GGF

GRIS Grid Resource Information Server

GRRP Grid Resource Registration Protocol

GSI Grid Security Infrastructure: the Globus Security mechanism

HTML HyperText Markup Language

IMS Information and Monitoring System

JDBC Java Data Base Connectivity: a Java API to an RDBMS

JNDI Java Naming and Directory Interface: a Java interface to LDAP and other directory services

LDAP Lightweight Directory Access Protocol

MDS Meta-computing Directory Service

RDBMS Relational Data Base Management System

R-GMA Relational Grid Monitoring Architecture

SE StorageElement: a Grid-enabled storage system

SQL Structured Query Language, used for querying relational databases

SWIG Simplified Wrapper and Interface Generator: a tool that allows a developer to wrap C/C++ functions for use with scripting languages

UDDI Universal Description, Discovery and Integration: a platform-independent, open framework for describing services, discovering businesses, and integrating business services

VO Virtual Organization: A set of individuals defined by certain sharing rules - e.g. members of a collaboration.

WWW World Wide Web

1.6 Document Structure

Chapter 1: this chapter: introduction, terminology and references.

Chapter 2: executive summary.

Chapter 3: introduces people to what we mean by “Information and Monitoring” and explains our architecture and how it fits in to the DataGrid.

Chapter 4: lists some Use Cases we consider the system to be suited for.

Chapter 5: is an initial list of requirements.

Chapter 6: describes some of the more significant design decisions we have made and our approach to protocols.

Chapter 7: describes some of the sensors and displays we will provide.

Chapter 8: shows the criteria to be used to assess the final product.

Appendix A: explains the protocols used.

Appendix B: shows the current Java API. There is also a C++ API which is not shown here.

1.7 Acknowledgements

We have received significant and valuable input from an informal collaboration with Trinity College, Dublin.

We thank all of our DataGrid colleagues whose ideas we have appropriated.

We are especially grateful to our GGF friends from both the *Performance* and the *Grid Information Services* areas who have either encouraged us or given us new ideas.

2 Executive Summary

The purpose of this work is to produce a flexible infrastructure providing easy access to current and archived information about: the Grid itself (this includes information about resources such as ComputingElements, StorageElements and the Network) and Grid applications (information published by a users job which will typically be used for performance monitoring). We will also provide some sensors able to publish resource information and generic visualisation tools to help interpret that information. Brokering services will make use of the information to determine the optimal resources to use for a given job. Some information is relatively static, such as the site location; while other information, such as CPU load is more dynamic. The architectural and data model of such a system should be simple but flexible.

Our architecture is based on the The Grid Monitoring Architecture (GMA) of the Global Grid Forum (GGF). This consists of three components: Consumers, Producers and a directory service which we prefer to call a Registry.

Producers (of information) register themselves with the Registry. Consumers query the Registry to find out which Producers produce the information they require. A Consumer can then contact the producer directly to obtain the relevant data. The GMA does not constrain any of the protocols nor the underlying data model.

We have chosen a relational model, which is simple and powerful and have constructed not only a monitoring system but an information system with a single spanning architecture. Our GMA implementation, known as R-GMA (Relational GMA) makes information from Producers available to Consumers as relations (tables).

The R-GMA implementation uses HTTP Servlet technology. Communication with the servlets is achieved via an API. This API has been implemented in Java and C++ but C and other languages will be provided soon. The response from a servlet is in the form of an XML document that corresponds to an XML schema definition.

Globus MDS is a widely deployed grid information system but we think that it has some problems: primarily an inflexible data model and a poor query language. For compatibility with MDS we have implemented an MDS Producer sensor. This re-publishes all the information available from a Globus GRIS server or in fact from any LDAP server into the R-GMA format and the information can be accessed using the R-GMA approach.

A number of Use Cases of the system are listed in this document showing how R-GMA can be used by the other DataGrid work packages. A number of requirements for the system are then itemised. Some of these requirements are inferred from the Use Cases, some from discussions and some from our own experience.

Evaluation criteria are defined based on functionality and on performance.

3 Architectural Overview

This chapter is meant to provide an overview of this document, to introduce people to what we mean by “Information and Monitoring” and to explain our architecture and how it fits in to the DataGrid.

3.1 What we mean by Information and Monitoring System

The purpose of this work is to provide a flexible infrastructure to provide easy access to current and archived information about:

The Grid itself This includes information about resources (ComputingElements, StorageElements and the Network), for which the Globus MDS is a common solution and job status (as currently implemented by WP1’s Logging and Bookkeeping System).

Grid applications This is information published by a users job. This will often be used for performance monitoring and is typified by making calls to NetLogger.

We will also provide some sensors able to publish resource information and generic visualisation tools to help interpret the information.

Information produced by the Grid middleware includes very slowly changing data such as the version of operating system to more frequently changing quantities such as the number of running jobs.

Information will in general be organised by *Virtual Organisation* (VO) a term defined by Foster, Kesselman and Tuecke in their paper, the Anatomy of the Grid[12] as a set of individuals and/or institutions defined by a set of sharing rules.

We expect our system to be suitable for monitoring within a fabric and to publish the relatively small amount of information which is visible outside the fabric such as that concerning the StorageElement and the ComputingElement.

In addition to the regular flow of information between the middleware components the system has to cope with potentially high rates of monitoring information produced by applications. For example a user with a job which is distributed over many machines may wish to have the parts running on different machines publish their status information very frequently and then maybe correlate this with disk activity within a StorageElement.

The system we define here is fully consistent with the second version of the DataGrid architecture document [2]

3.2 Desired features of an Information and Monitoring System

3.2.1 The importance of time

Information is introduced to the grid in a temporal order. It may be of value even without retaining this ordering, but its value will be greatly enhanced if the ordering is maintained, for example by attaching a timestamp to it as we plan to do. It is our intention that the timestamp will give the time of the original measurement.

We plan to have a common interface to access data, whether it is *fresh* monitoring data or data from an archive. It is preferable to associate all information with a time stamp - then if there are any delays

in delivering information there is no ambiguity. If information is archived for the purpose of making a model to predict future behaviour, then clearly all the information must carry a time stamp. Consider three use cases of an Information and Monitoring system.

Case 1: A scheduler needs information on the availability of usable resources worldwide. The scheduler cannot afford to wait for information which is not available in a reasonable time. It does not need to find the best solution, but merely one near the optimum. Some information will be out of date, but as Coghlan[3] has explained we should expect that. There is a need to consider queries with a time-out - this is both from the point of view of expressing the query and implementing it - see Plale and Dinda[6]. Jobs will of course run in the future, not when they are submitted, so predictions are important. This implies accumulating old data and having a model. Time stamps are vital on all the information if such a model is to be built.

Case 2: A computing fabric needs to have all recent information held reliably for post mortem analysis when part of the system dies. Again the time stamp is vital, in this case for reconstruction of the sequence of events leading to the failure. A reliable archival mechanism is also needed. In carrying out the analysis it is undesirable to lock the state of the fabric or to stop collecting logging information. One would typically start by looking at events occurring just before the failure.

Case 3: For application monitoring there is a need to correlate fabric information with information produced by a job. The results may be used by sophisticated jobs to optimize their own behaviour according to their environment, i.e. the conditions on the fabric. Time may be used as an index to the sequence of events, and as an input to application-level models of the fabric and job. The event sequence is also useful for application-level debugging.

To summarize: we would like the information system to associate each measurement with the time when it was made; and we would like to handle fresh and archive data in a uniform manner.

3.2.2 Grid Monitoring Architecture

We find the Grid Monitoring Architecture (GMA) [18] of the GGF to be very simple and attractive. The GMA, as shown in Figure 3.1, consists of three components: *consumers*, *producers* and a directory service, which we prefer to call a *registry* as it avoids the implied tree structure of a directory service such as the Lightweight Directory Access Protocol (LDAP) or the Java Naming and Directory Interface (JNDI).

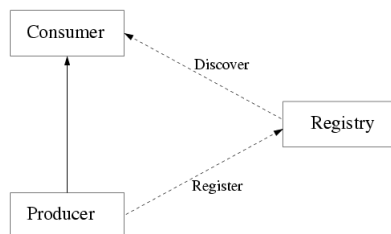


Figure 3.1: Grid Monitoring Architecture

In the GMA producers register themselves with the registry and describe the type and structure of information they want to make available to the Grid. Consumers can query the registry to find out what type of information is available and locate producers that provide such information. Once this information is known the consumer can contact the producer directly to obtain the relevant data. By specifying the consumer/producer protocol and the interfaces to the registry one can build inter-operable services.

The GMA architecture has some similarities with the JINI architecture which also makes a clear distinction between the mechanism for describing the available services from the mechanism used by a client to communicate with that service once it has been found.

The current GMA definition also describes the registration of consumers, so that a producer can find a consumer. We do not like this feature because although it appears to give symmetry this is not really the case, since the system is essentially asymmetric with the information flowing in one direction from producer to consumer. The main reason to register the existence of consumers is so that the registry can notify them about changes in the set of producers that interests them. The same effect can be achieved by other methods as explained in Section 6.3.1. It is sometimes claimed that it is necessary to support archiving of information but we have an alternative solution. This functionality is provided by the DataBaseProducer which is explained in Section 3.5.1. If it should turn out that there is a need for registration of consumers, this can easily be added later.

The GMA architecture also supports joint consumer/producer components as illustrated by the component at the centre of Figure 3.2. This could gather data from several producers and make processed information available to other consumers. For example the three producers might publish information from three different ComputingElements. The joint consumer/producer could then average over a 10 minute period and produce combined information for the three contributing ComputingElements.

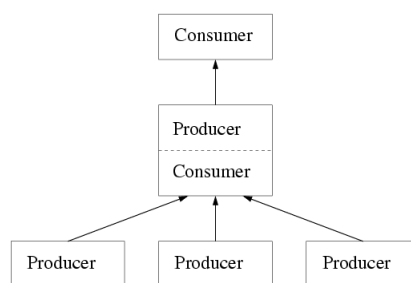


Figure 3.2: Joint Consumer Producer

The GMA architecture was of course devised for monitoring but we think it makes an excellent basis for an information system.

We wish to use a simple but flexible model - the GMA. As the GMA is only an architecture there are at least three different implementations - of which ours is one. The known implementors meet regularly at GGF and we discuss by e-mail to consider the relative merits of the different solutions. Our solution is currently the furthest developed but we are very interested in working with the other implementors to produce the best solution.

As the GMA definition does not constrain any of the protocols nor the underlying data model, we were free when producing our implementation to adopt a data model which would allow the formulation of powerful queries over the data.

The information system must be based on a data model sufficiently flexible to cope with the queries we wish to make.

3.2.3 The relational model

The relational model appeared some thirty years ago [7, 8] to overcome the drawbacks of the hierarchical and network data bases. The resulting Relational Data Base Management Systems (RDBMS) have almost eliminated all competition. There are a few Object Oriented Data Base Management Systems now but the Object Relational Hybrid is more common. The point of this section is however not to discuss the RDBMS, but the relational model, which is responsible for the success of the RDBMS.

The relational model organises data in tables (which can be thought of as a set of objects of the *same type* and having only public data) Each row of the table is known as a tuple.

One ways of accessing the information is to use relational algebra. Here one has to define operations on tables: select (rows from a table), project (columns from a table) and join (combine two tables). In this case the sequence of processing is clearly defined. Alternatively a query language such as SQL (based upon relational calculus) may be used. This allows a user to express a query and the system will decide how best to implement it and in what order to carry out the various operations – the sequence of which could be expressed in relational algebra. A single query can *combine information scattered over many different tables*.

3.2.4 Summary of desirable features

1. We would like the information system to associate each measurement with the time when it was made.
2. We would like to handle fresh and archive data in a uniform manner.
3. The information system must be based on a data model sufficiently flexible to cope with the queries we wish to make.

3.3 Comparison with LDAP based approaches

The Globus MDS is a common Grid Information Service. Unfortunately it is based on LDAP. This has the benefits of being easy to distribute and having well defined protocols *but* the data model is inflexible and the query language is poor.

An SQL query is capable of extracting information from a large number of different tables. LDAP, in common with other hierarchical structures is fine *if you know all the queries in advance* as you can build your database to answer that question very rapidly. Unfortunately if you fail to anticipate the question getting an answer could be very expensive. The LDAP query language cannot give results based on computation on two different objects in the structure – or, expressed in relational language, there is no *join* operation.

Our GMA implementation, known as R-GMA (Relational GMA) which was proposed to the GGF by Fisher [11] makes information from producers available to consumers as relations (tables) and also uses relations to handle the registration of producers. R-GMA is consistent with the principles of the GMA.

Section 4 shows various Use Cases and serves to highlight the power of the relational model, of SQL and of the R-GMA.

3.4 Not an RDBMS

An RDBMS offers a number of features which are ill-suited to the Grid. Most systems offer transactions so that you can make a series of changes and then commit or rollback the whole lot. It does not seem sensible to consider trying to lock a world wide distributed replicated database.

For our system, R-GMA, you should consider a VO's information to be organised logically as one huge relational database - but the implementation is based on a number of loosely coupled components. We expect the components to fail from time to time (not our software of course - but the hardware) and we expect parts of the network to break periodically. The huge data base is partitioned, with the *description of the partitioning held in the registry* as described in Section 3.5.2.

The components are Producers and Consumers, where the Producers register themselves in the Registry. The information stored in the Registry by each Producer is an identifier for each type of table it can

produce and for that table a *predicate* (or logical condition) expressed as an SQL WHERE clause which defines the part of that table within the VO which is held by that Producer.

Consider an example of a very much simplified ComputingElement (where the contents of Time/Date field are omitted to save space):

ComputingElement

Country	Site	OpSys	FreeCPUs	Time/Date
UK	RALTB1	Linux	47	...

The producer at RALTB1 in the UK will always generate information such that:

```
WHERE Site = 'RALTB1' AND Country = 'UK'
```

is true. So this WHERE clause gets stored in the registry. A combined consumer/producer collecting information for the UK and republishing it would register:

```
WHERE Country = 'UK'
```

A consumer/producer component called an *archiver* is available to do this kind of job. See Section 3.5.5 Note that we use relational techniques to handle both the registry and the consumer/producer interactions.

Normally small integers are used as *surrogate keys* when designing data base schemas. A small integer is used as the primary key rather than some more natural string so that it can be referenced more compactly by other tables holding this integer. The allocation of these small integers would be difficult and it is suggested that this practice should not be used in the R-GMA system.

3.5 Component types

3.5.1 Producers

We have so far defined not just a single Producer but three different types: a CircularBufferProducer, a DataBaseProducer and a PluggableProducer. All appear to be Producers as seen by a Consumer - but round the back they are rather different. The PluggableProducer is perhaps the most general - but it has not yet been implemented. It is intended to accept a PlugIn which will collect the desired information and make it available as a Producer. The CircularBufferProducer and the DataBaseProducer have been coded and they both implement a Publisher interface. The Publisher interface allows the user to publish information via a Producer. So to summarize all 3 Producers implement the Producer interface but only the CircularBufferProducer and the DataBaseProducer implement the Publisher interface.

A Publisher (i.e. a CircularBufferProducer or a DataBaseProducer) is instantiated with the description of the information it has to offer specified by an SQL CREATE TABLE statement and a character string describing the columns which are fixed and their values. This will soon be changed to accept a general predicate in the form of an SQL WHERE clause.

To publish data, a method is invoked which takes the form of a normal SQL INSERT statement.

The CircularBufferProducer writes information to a circular buffer where it can be picked up by a Consumer whereas the DataBaseProducer writes the information to an RDBMS. The CircularBufferProducer never blocks - but the information sent can be lost if a Consumer is too slow. The DataBaseProducer is slower as it writes each record to an RDBMS - but no information will be lost. An intermediate solution which we will also consider is to use a memory-mapped file to handle the circular buffer.

The DataBaseProducer will normally need cleaning from time to time. We plan to implement a policy, possibly specified partially by an SQL `WHERE` clause. For example it might delete records more than a week old from some table or it may only hold the newest one hundred rows, or it might just keep one record from each day. The cleaning policy also forms a part of the specification of the DataBaseProducer and needs to be stored in the registry. This has not been designed yet.

We do not expect all information to have a time to live attribute. However for those tables where this information is provided it is easy to implement a policy of only keeping “unexpired” information. Of course the information may be being saved as a historical record in which case deleting all “expired” information would not be a sensible policy.

3.5.2 Registry and Schema

GMA is rather similar to web services - specifically the UDDI (Universal Description, Discovery and Integration) which separates the description of a service from the list of providers of that service. From our consideration of this mechanism we decided to implement GMA by separating the description of the table – the Schema, from the list of providers of that table – the Registry.

The Registry holds a table identifier for each logical table and a representation of the `WHERE` clause which defines the partition of the logical table which the Producer has to offer. The detailed information about the tables - what columns they have and of what type - is held in the Schema. Each VO will have one logical Registry linked to one logical Schema. We need to devise a way to be able to distribute and replicate the Registry and Schema for performance and robustness. We have not yet considered the best way to do this. When a Producer terminates it should remove its entry from the Registry. The Schema holds descriptions of some predefined tables and other descriptions of other tables are added by the Registry whenever it encounters a table it does not know. When a Producer de-registers its tables, the Registry will in turn delete any redundant entries from the Schema except for the predefined set of tables. Dynamic Schema modification is not yet implemented.

We are working with WP2 on the distribution of the registry and schema as this has much in common with their work on the Replica Catalog.

It is important to note that *the normal user will not make use of the Registry and Schema directly*. A Producer will look after registering itself and a Consumer will look up what it needs in the Registry on behalf of the user.

3.5.3 Consumers

The Consumer is the means of accessing information made available by various Producers. A Consumer handles a single query, expressed as an SQL `SELECT` statement. A Consumer may be told which Producer to connect to, otherwise it consults the Registry to find the “best” source or sources of information. In the simplest case, where all the information is one one table and there is only one registered Producer of that table, the Consumer can identify the best producer by consulting the registry. In the general case we will use a Mediator as explained in the next subsection. There are two modes of transfer of information from the Producer to the Consumer. The user can ask for the Consumer to execute its query, in which case it asks the Producer to process it once. The user may of course repeat this process - if there are no new data the same, current, data will be returned. This is the pull model. Alternatively the user can request that the SQL query be sent to the Producer and whenever new data are available which satisfy the query they are transmitted. This is a push model, *but the streaming of data from the Producer to the Consumer is initiated by the Consumer* This definition of streaming - i.e. transmit when new data are available is applicable when the Consumer is communicating with a CircularBufferProducer where each slot in the buffer only holds a few rows from one table. When the Producer moves the write pointer on in the buffer data are sent to each Consumer which has requested the streaming of data. Streaming from a DataBaseProducer can only be carried out when the query is upon one table. In this case each `INSERT` into the table will make a new tuple available – this is not yet implemented.

It might be possible to allow *different* definitions of a table within a schema - so that one producer has a table with more columns than another producer of a table with the same name. This will facilitate schema evolution but will be more complex to manage; so we are initially making the assumption that there is logically one registry, with one schema, known to the whole VO.

3.5.4 Mediators

The Mediator is to support the Consumer when it is not told which Producer to get information from and there is not a single source of information known to the Registry. We expect work on this component to start soon (around January 2002). We currently have a rather clear idea of what it should do but not how it should do it. If the Consumer cannot find just one Producer the Mediator will do three things:

1. consult the Registry to find all the Producers which can contribute information
2. take the original SQL query and construct queries each of which can be dealt with by a single Producer
3. for each of the new queries construct a consumer to perform that query and then execute it
4. combine the results.

It does not appear feasible for a Mediator to support streaming. The task of combining the results can be very simple and involve no more than concatenating the results from the different sub-queries but it may involve some relational algebra.

A Mediator may choose to create an Archiver (see Section 3.5.5) to rapidly carry out popular complex queries.

It has not yet been decided whether the Mediator will be implemented as another service accessed by the Consumer servlet or as code which goes directly into the Consumer servlet to provide this extra functionality.

3.5.5 Archivers

The Archiver is a joint consumer/producer. It is not a fundamental service but just a convenience for the users. It is perhaps easiest to understand it in terms of its implementation. When you create an Archiver it creates a DataBaseProducer. You then tell the Archiver which tables you want, along with a predicate to indicate that you only want some of the available information. For example:

```
WHERE Country = 'UK'
```

For each of these tables the Archiver has been asked to collect, it creates a Consumer and requests that the information be streamed to it and then publishes it in the DataBaseProducer.

The Archiver can be seen to have three purposes:

1. It collects and maintains information in one place. This avoids the delay of going each time to the many sources of information.
2. It stores historical information - i.e. acting as an archiver
3. It offers an alternative to the mediator when you have to do joins over tables held in different places.

This last point is very important today as the Mediator is not yet implemented. It will however remain an interesting alternative, if repeated queries are made, to use an archiver which is working in the background keeping all the information it needs up to date to answer the query next time it comes. The Mediator appears to be the ideal solution for a complex one-off query.

3.6 Security

So far we have not implemented any security features. We will use the security module developed by WP2 for Spitfire, suitable extended and/or adapted in collaboration with them.

3.7 Variation from the technical proposal

At the time of writing the technical annex we had indicated that the key issues included:

- Scalability to large numbers of resources and users operating in the Grid,
- Architectural optimisations to minimise resource usage for queries,
- Applicability of agent technology,
- Data discovery via directories of “published” information, and
- Validity of information describing time dependent and non-repeatable situations.

Scalability is still a key issue - we must find an effective way to distribute the Registry and Schema.

We have a very flexible architecture which allows a large number of topologies to be constructed. We will shortly be considering the best way to achieve good performance

Mobile Agents were thought to be a suitable technology - the agents would go around looking for information. However this is no longer considered to offer any obvious advantages. The registry has all the information it needs to locate any piece of information within a VO. This is easy and efficient.

Data discovery in the sense mentioned above is covered by the Registry.

Concerning the validity of information: all tuples have a time stamp so it is clear when a measurement was made. It may or may not be the latest information; however the notion of latest makes little sense in the grid. See Coghlan [3]

3.8 Innovation and Cooperation

We have decided to adopt the Grid Monitoring Architecture [18] (GMA) of the Global Grid Forum [13] (GGF). However, as has been explained, we are implementing that architecture in a novel way making full use of the relational data model [7, 8]. Our system is therefore referred to as R-GMA. Our ideas have been presented and discussed at the GGF [3, 11]. Many of our members are very active within that body and are also in contact with the database community [4]. As has been indicated above, we are working closely with WP2 on various common issues to ensure common solutions.

4 Use Cases

This chapter lists Use Cases for an information and monitoring system within the DataGrid. These have been constructed from consideration of the other work packages within the DataGrid.

We re-iterate that we are trying to design a system which will be flexible enough to address all these Use Cases and cope with future developments.

4.1 Network performance - WP7

WP7 has a number of requirements that can exploit a relational approach. The measurements of network metrics that WP7 provides to other work-packages, primarily the resource broker of WP1 are best expressed as relations. The typical situation is exemplified by the following table.

RTTMeasurement

SiteA	SiteB	Tool	Measurement
RAL	Lyon	Pinger	1500
RAL	CERN	Pinger	800
RAL

This table shows measurements of Round Trip Time (RTT) time made by a specific tool from RAL to other sites in the Grid. This information is published by a Producer located in RAL. Lyon and CERN will run Producers that contribute further views of this relation where SiteA is Lyon or CERN respectively. Conceptually all the RTT information is held in one single table expressing a relation between pairs of sites. For illustration we show two further tables describing some aspects of a StorageElement

StorageElement

SEId	Site	Diskfree
RAL_S1	RAL	100
Lyon_S1	Lyon	200
CERN_S1

and a ComputingElement

ComputingElement

CEId	Site	LRMS
RAL_Q1	RAL	PBS
CERN_Q1	CERN	LSF
Lyon_Q1	Lyon	...

Each table is made available by a Producer, that publishes information about all the available SEs and CEs respectively. One of the interesting questions one can ask is the following:

Are there ComputingElements from which the RTT to a Storage Element is less than 1000 and if yes I want to know the Id's and the RTT for all of them.

Using the relational approach this is compactly expressed as the following SQL query:

```
SELECT C.CEId, S.SEId, R.Measurement FROM ComputingElement C, StorageElement S,  
RTTMeasurement R WHERE C.Site=R.SiteA AND S.Site=R.SiteB AND R.Measurement  
< 1000
```

The **SELECT** part specifies which columns to select for the resulting table, the **FROM** part specifies the tables from which information is combined and the **WHERE** part contains the join conditions to arrive at the final table (N.B.: Without a **WHERE** clause the table resulting from a select from two tables with N and M rows respectively is $N * M$).

Execution of this select statement against the above set of tables would result in the following table:

ResultSet

CEId	SEId	Measurement
RAL_Q1	CERN_S1	800

The information that WP7 provides would only be the **RTTMeasurement** relation, but combined with other tables that express information about other resources on the DataGrid one can form powerful queries. It is apparent that a tree-structured data model would hide the fact that a network measurement is essentially a relation.

Another requirement of WP7 is that summarised information like **RTT** needs to be published for use by other work packages. How this measurement was obtained should be irrelevant to the user. Nevertheless detailed information should be published as well for internal WP7 use. This is easily achieved in R-GMA by defining a set of tables for the detailed information as well as one or more tables for the summary information. WP7 could then provide a joint Consumer/Producer which would consume from the detailed tables and republish the summary information in the summary tables. Users of the information WP7 provides would only ever consume from the summary tables and would not have to worry about how these measurements were made.

The information is clearly of value to WP1. An archiver could be set up for WP1 which would collect all relevant information for the selection of suitable resources.

4.2 Logging and Bookkeeping - WP1

The Logging and Bookkeeping (L&B) Service component of the Workload Management System gathers and manages information about jobs and system components [19]. The purpose of the L&B Service is to provide job status monitoring. It must be reliable and fault tolerant. Events generated by Workload Management System components are pushed to the L&B server where they are stored in a database. The status of jobs can then be determined from the stored events.

There are three main components within the system: the Local Logger, Inter Logger and L&B Server.

The Local Logger accepts events from event sources (L&B Producers) and passes them to the Inter Logger. It also uses a checkpoint file to make recovery possible in case of failures.

The Inter Logger then forwards events to appropriate L&B servers

The L&B servers store the events and answers job status queries.

Using the the R-GMA components. The L&B information could be published via a DataBaseProducer (doing the job of the Local Logger) this would ensure that no data were lost.

There appears to be no benefit in using an Inter Logger but rather the L&B servers could instantiate an Archiver to collect data from a group of Inter Loggers. If the connection between the Local Logger and L&B server was broken for a while the Archiver has no mechanism to get back that data it has missed



though it would still be waiting at the Local Logger, so a better solution would be for WP1 to construct a special Producer/Consumer rather than using the provided Archiver which would have the logic to catch up on any missed data.

The task of querying jobs is then very simple. Queries could select on any of the stored fields to find for example all jobs belonging to a certain user. It could happen that one wants to know about jobs running on UK machines but which belong to users affiliated to French sites. This can easily be expressed provided we have a table somewhere giving the users' affiliations.

5 Requirements

Some of these requirements are inferred from the Use Cases described in Section 4 and others from Part A of EU DataGrid Deliverable D8.1 [20] which lists some of the Information and Monitoring Requirements in section 7.3.1. This set of requirements is probably incomplete.

As it is hard to write the requirements in a totally abstract way, we first specify some Constraints which we are obeying:

1. We will use a simple but flexible model - the GGF Grid Monitoring Architecture.
2. The IMS must be based on a data model sufficiently flexible to cope with queries spanning the entire information space.

As these constraints are well satisfied by a relational GMA implementation we assume this solution for the rest of the requirements.

5.1 General

1. Fresh and archive data should be handled in a uniform manner.
2. The IMS should associate each measurement with the time when it was made.
3. The information system should be highly scalable.

We should not impose limits as to the size the grid can become. Over time more resources and users will join the grid.

5.2 Producers

1. A user must be able to publish rows of a table (via a Producer) even if this table is not previously known to the system
2. When registering a Producer it must be possible to specify via an arbitrary predicate expressible as an SQL `WHERE` clause what partition of that table is available from that Producer.

This is known as the partitioning predicate

3. It shall not be possible to publish rows of a table which are not consistent with a table definition already registered within that VO
4. It shall not be possible for a Producer to publish rows of a table which are inconsistent with its partitioning predicate
5. There must be a type of Producer which doesn't get blocked by a slow Consumer.
6. There must be a type of Producer which does not lose information except as defined by the policy of that particular Producer

It will in practice be limited by disk space

5.3 Consumers

1. A user must be able to locate all Producers of information within the VO and determine their partitioning predicates.
2. A Consumer should be able to obtain information from a Producer on a one off basis
3. A Consumer should be able to obtain a stream of new information as and when it becomes available from a Producer.
4. Any query expressible by SQL should be answerable by a Consumer though the information it needs is widely scattered throughout the system.

5.4 Robustness

1. There must be no single points of failure.
2. Partial network failures should have minimal impact. The system should recover once the network is back.

5.5 Extra facilities

1. Access to the system should be available from the command line.
2. Access to the system should be available via an API.
3. Access to the system should be available via the WWW.
4. Additional tools should be available to help with administration of the system.

These allow interaction with the Registry and Schema databases.

5.6 Ease of use and performance

1. User documentation must exist.
2. An installation guide must exist.
3. A system guide must exist.
4. The system should be easy to use.
5. It should be possible for an experienced system manager to install the software and configure it to provide a basic information service on their system within 1 hour.
6. The system should answer a query quickly.

This is the information needed to continue development of the software

This depends upon the nature of the query. If a user poses a new question which requires access to large amounts of data this cannot be quick. We can provide a system which will handle any queries that may be made but infrequent queries will take longer.

5.7 Security

1. It must be possible to restrict knowledge of the existence of Producers of information to specific authorised users.

If the user is not aware of what Producers exist, they do not know there are resources to which they are not allowed access. This makes it much harder for the attacker.

2. A Producer must be able to restrict read access to information to specific authorised users. Access must be controllable at a level of granularity specified by the intersection of a set of rows as expressed by an SQL WHERE clause and a set of columns.

3. A Producer must be able to restrict the publishing of information to specific authorised users.

We need control over who is permitted to place information into the system. This is both to avoid being flooded with information from a faulty program or from a hacker and also to avoid the publishing of misleading information.

6 Design

This chapter sets out some of the design decisions we have made in the R-GMA implementation.

6.1 Servlets

We chose to base our initial implementation upon HTTP Servlet technology. There are servlets which are not HTTP Servlets but in this document we use the term servlet to mean an HTTP Servlet. A servlet runs inside a servlet container. There are many servlet containers to choose from - both stand-alone ones and those designed as plug-ins to existing web servers. We have been using the Apache Tomcat server which can be used either stand-alone or as a plug-in to a number of web servers. The servlet runs inside the Java virtual machine with the servlet container and so has all the usual advantages of Java portability.

There are a number of reasons for selecting servlet technology:

1. It is a very simple way to build a distributed system
2. The bulk of the code is running in the servlets and can be written in Java without any problems using it from any programming language from which an existing HTTP library can be called.
3. The use of the technology continues to grow.

HTTP is a simple stateless protocol. The client makes a request which is of a certain type. Two common types are `POST` and `GET`. A `GET` request is formulated by appending a query string to the end of a URL to provide some keyword-value pairs. For example `?name=Fred`. Some servers limit the length of the string so this is not generally a reliable technique. A `POST` request passes all the parameters directly over the socket connection as part of the HTTP request body. There is no limit to the amount of information which can be sent this way. We have written our servlets to respond to both styles of request. To write a servlet you have to subclass `javax.servlet.http.HttpServlet` and override the `doGet` and `doPut` methods. Communication with the servlets is easy in Java, C, C++, Python and Perl so we can provide APIs in each language which seem natural in that language. We currently support Java and C++. The API code is small and all follows the same pattern.

We have so far defined 6 servlet classes: `CircularBufferProducer`, `DataBaseProducer`, `Consumer`, `Registry`, `Schema` and `Archiver`. The servlet container normally only instantiates one instance of a servlet for each class. This is very good for performance because the servlet object is ready to respond without delay and provides persistence but does require care in coding because a single servlet may be handling many requests simultaneously.

As explained above a single servlet can serve many API instances. This avoids needing a large number of servers, but offers great flexibility as we can always add more servers if necessary to balance the load.

The `Registry`, `Schema` and `DataBaseProducer` all make use of an RDBMS for example MySQL [16] interfaced via JDBC. This offers further flexibility as this also allows the database to be remote from the servlet if so desired as the location of the database is itself specified by a URL.

The architecture is such that it is easy to integrate with existing schemes. Some of the sensors and displays we plan are described below in chapter 7.

Figure 6.1 shows a simple example with three pieces of sensor code each with a single producer object. Two of the `CircularBufferProducers` are making use of one servlet but the other is making use of another producer servlet. The dashed lines show invocation and the solid lines the main flow of information.

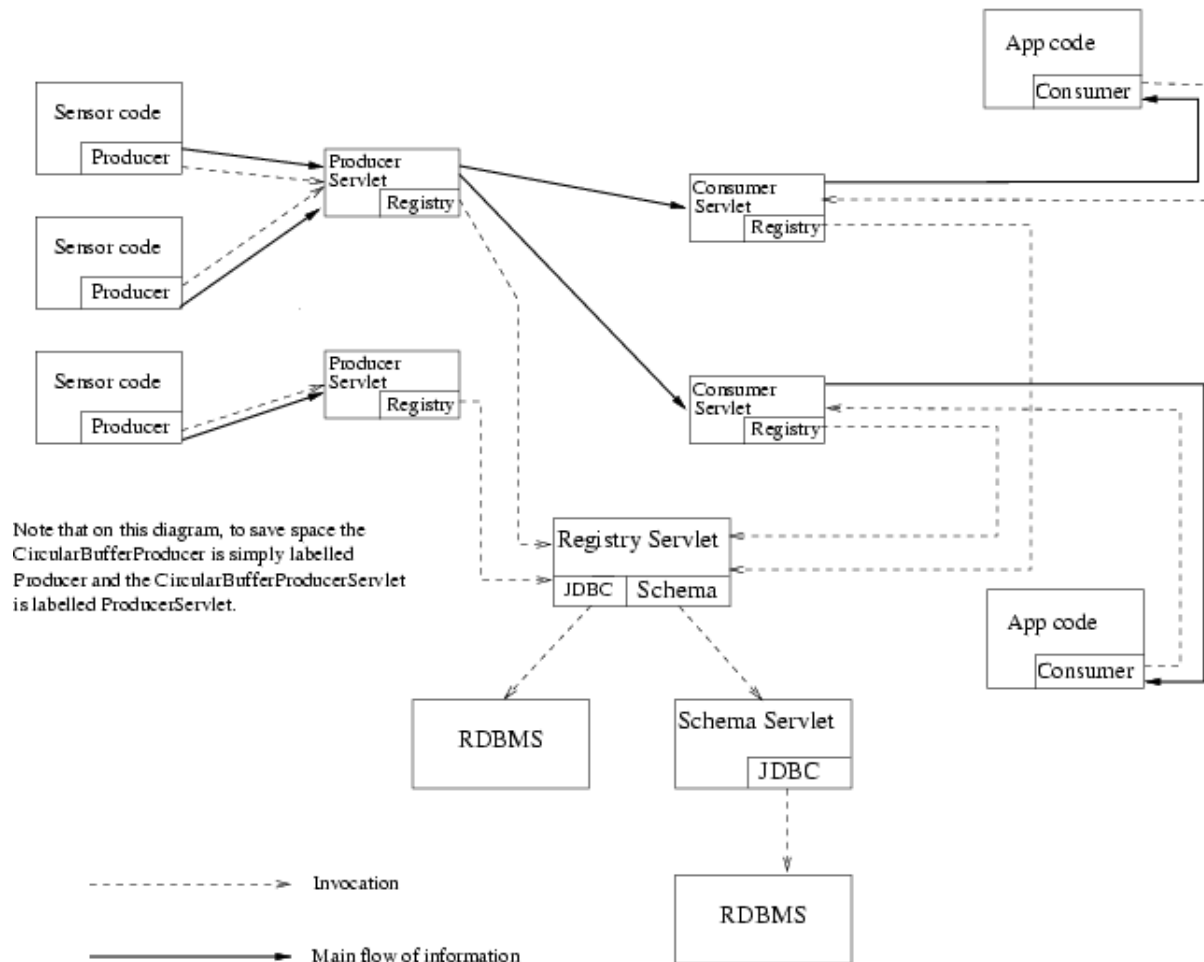


Figure 6.1: Topology with three CircularBufferProducers and two Consumers

There are two pieces of application code each with a consumer object getting data from a consumer servlet. No consumer wants the data from the third producer so there is only the small local flow of data from the producer to the producer servlet. The figure also shows how the consumer and producer servlets make use of the registry which in turn makes use of a schema and an RDBMS. The schema talks to the schema servlet which also makes use of an RDBMS.

6.2 HTTP/XML based protocols

To set the size of the queue into which data are streamed for a consumer with *consumerId* = 1 on a ConsumerServlet running on port 8080 on localhost to 13, one issues the following http request:

```
http://localhost:8080/R-GMA/ConsumerServlet/setBufferSize?consumerId=1&bufferSize=13
```

This request has four components:

- host part: localhost:8080
- context path: /R-GMA/ConsumerServlet

- additional path information: /setBufferSize
- request parameter part: ?consumerId=1&bufferSize=13

The host part identifies host and port on which the servlet container is running. The context path identifies which servlet class the servlet container has to use to respond to this request (ConsumerServlet in this case). The additional path information is used by the selected servlet to identify which method of the servlet class should be executed (setBufferSize in this case). The doGet method of each servlet contains a switch which selects the method based on the additional path information. The request parameter part is used to pass parameters to the selected method (set the buffer size to 13 for Consumer 1), these parameters are accessed via the getParameter() method of the HttpServletRequest object. When specifying the protocol it is therefore sufficient to list the method name and the request parameters that need to be passed. When we list the HTTP GET/POST request parameters for each service that any of the 6 servlets we have described above offer, we list the types and cardinalities of the different parameters. The cardinality ranges are indicated by: `MinimumCardinality..MaximumCardinality` with the following possible values:

- "0..1" is a single-valued optional parameter,
- "0..*" is a multi-valued optional parameter,
- "1..1" is a single-valued mandatory parameter and finally
- "1..*" is a multi-valued mandatory parameter.

The response from a servlet is always in the form of XML which corresponds to an XML schema definition. The XML schema describes a **choice**¹ between a **ResultSet** a **Status** and an **Error**.

A **ResultSet** is the representation of the normal result of an SQL Query - that is a table (of data). As our implementation is based on the relational model, it is rather natural to encode responses as tables. We list the names of the columns of the returned tables as well as their types and the cardinality of the number of rows returned. Some methods do not have a canonical return object, e.g. the setBufferSize method, but in a distributed environment it is not enough that no error is returned, to infer that the remote method invocation completed successfully. In those cases we therefore return an OK status object. If an error occurs during processing of a request the nature of the error is being passed back in an XML encoded Error object, so that the API can take appropriate action. The Error holds sufficient information to allow a clear error message to be generated or an exception to be thrown (according to the language of the API) by the API code. It should now be clear how to interpret the following table:

Request Parameters			
Name	Type	Cardinality	Meaning
consumerId	Int	1..1	The connection ID identifying the consumer to the servlet.
bufferSize	Int	1..1	The size of the queue buffer on the ConsumerServlet, where the streamed data from the ProducerServlet are stored.

Returned Status	
Name	Meaning
OK	The buffer size was successfully set to bufferSize.

Returned Error	
Name	Meaning
BufferError	The buffersize was not successfully set to the required size.

The protocols defined so far in our implementation are tabulated in Appendix A

¹choice is an XML schema tag indicating that the syntax will be chosen from a number of possibilities

6.3 Soft state registration

Within a Grid environment producers will be created and destroyed. They will also fail due to hardware and software problems and they may simply become inaccessible because the infrastructure fails.

It is too difficult and not necessary to have an accurate up to the second view of the producers within the grid. We obviously still need some way of deciding how reliable our view of the grid is, so we need some mechanism to help with this decision. This mechanism will be soft state since unless a fresh registration request is received, the knowledge of the producer will eventually be removed from the registry.

Gullapalli et al., have submitted a draft paper on a Grid Notification Framework [14] to the GGF. This defines a generic notification framework to convey existence information and state properties about grid entities. The Grid Resource Registration Protocol (GRRP) as defined by Czajkowski et al. [5] is an implementation of such a mechanism.

Due to our chosen implementation using a small compact API and servlets for the entities within our system we have several interactions which may require some notification protocol. These interactions fall into two categories, the registering of the existence of entities within the grid and a servlet's awareness of an API which has initialised a connection.

6.3.1 Registering the existence of entities

This information is stored within the registry database and queried via the registry API.

As explained previously, we have decided only to register producers. The main reason to register the existence of consumers is so that the registry can notify them about changes in the set of producers that interests them. However we think that this is not necessary, since a consumer can re-query the registry periodically to establish the appearance and disappearance of producers. We believe this is acceptable since a real-time view of the Grid does not exist. It will be easy to add a facility at a later date, if a need is found, which will allow consumers to register.

We have not yet implemented a registration mechanism, but have examined the Grid Notification Framework and developed some ideas for our implementation. Where possible we try to use the same terminology as the GRRP.

Our registration mechanism will have four parts:

Entity identifier Made from URL of the ProducerServlet and Connection ID within the servlet for the producer.

ValidFrom timestamp This indicates the time at which we were sure the producer existed.

ValidTo timestamp This indicates the time at which we assume the producer is uncontactable unless we receive further notification with a ValidTo timestamp further into the future.

KeepTo timestamp This indicates the time at which we assume the producer no longer exists and any record of it should be removed from the registry.

The timestamps and format for these are discussed in Section 6.5.

6.3.2 Notification between APIs and servlets

The Consumer, Producer and Archiver APIs result in the creation of state at the servlet side. In the case of the Producer and Consumer this is a Connection ID and some storage for the information produced or received. In the case of the Archiver this involves the connection ID, one or more Consumers, a DataBaseProducer and a thread to populate the DataBaseProducer.

This state is created upon initialisation of the Consumer, Producer or Archiver object using the API. Because of this if the servlet were to decide that the object no longer existed, the object could no longer communicate with the servlet - its Connection ID would be invalid. To re-establish this communication the object would in effect have to be re-created. Bearing this in mind we would rather falsely assume the object exists than falsely assume it no longer exists.

The Registry and Schema API to servlet communications do not involve any state creation within their servlets. Each connection can be considered to be independent of any other and so no notification messages are required.

We have two possible solutions in mind for the case where state is stored:

State Creation - Solution 1

We can use our registration protocol with the servlets, keeping the three timestamps mentioned above for each object. This means either the default times must be very large to avoid falsely assuming an object no longer exists or left to discretion at object creation time. The object would then have to periodically send a new notification to renew the three timestamps and ensure that the servlet is aware it still exists.

The problems with this method are the requirement for extra code within our 'compact' API, larger messages between API and servlet and more state within the servlet. Perhaps more importantly if we allow the users to specify the time to live and this is set low then the chance of falsely assuming the object no longer exists will be high.

State Creation - Solution 2

Only use a registration protocol from the ProducerServlet to the Registry as described previously and use a thread within the servlets to decide which objects should be considered to no longer exist.

The API has no added code but whenever it contacts the servlet the servlet makes a note of the time at which this happens. The servlet sends a registration notification to the registry at a regular interval with a list of all objects that have contacted it since the last notification. The timestamps will be appropriate to the frequency of these registration notifications between the servlet and registry.

Finally a thread will run infrequently upon the servlet to flag any objects that have been inactive for longer than a set time interval, if they are still flagged several time intervals later they are considered to no longer exist.

We allow the servlet to control the frequency of registration notification, time to live and period of inactivity for an object.

It is possible to allow the per object specification of time intervals at object creation. The servlet can then base the timestamp calculations upon these intervals.

The time interval for the thread tidying up inactive objects on the servlet can be changed and published as a well understood default. e.g If your consumer/producer is inactive for more than x hours it will be considered to no longer exist. A manual 'no-op' can effectively be used to keep contact with the servlet and ensure its existence does not fall into question if this is important, e.g. issuing a getStatus request of the servlet.

With this solution no special messages need exist between API and servlet, there is less chance of the servlet falsely assuming an object no longer exists and the registry can tidy itself up independently of the servlets, presenting a more accurate view of the Grid to Consumers.

6.4 Security

Currently we have no security implemented.

6.4.1 Servlet Container

The servlet container will have an X509 certificate associated with it, as must the user. When a request is handled by the servlet container, the 2 way authentication process must be carried out: the servlet container must authenticate the user, and the user must authenticate the servlet container.

6.4.2 Registry

The Registry must be able to check which users are allowed to read about which Producers and there may be controls on who is allowed to register new Producers.

6.4.3 Schema

It is not clear that there is any need to prevent people from reading the Schema information however there may be controls on who is allowed to store new table definitions.

6.4.4 Producer

The Producer must be able to control who can read what data. Control may be required at the level of a part of a table where the 'part' may be a projection (a set of columns) or a selection (a set of rows).

6.4.5 DataBase

JDBC connections normally use a clear text password. We should try to do better than this.

6.5 Timestamps

There are many ways to timestamp, and many different representations (formats), e.g. ISO8601 and RFC-1305 (NTP). We plan to make use of the GGF definition of the UTC based time-stamp [15]. These timestamps have three basic components, one representing the date, one the time and one some extra information on such things as precision. These components must retain credibility, and this is only possible if they are acquired from time sources that are synchronized in a known and consistent fashion, for example via NTP; this therefore is a constraint on the eventual solution. Furthermore, the components may need to be formatted in different ways for, say, humans and schedulers. We have to be certain that they are interpreted in the same way everywhere, especially in respect of leap-seconds, irrespective of the format and any translations between formats.

In the context of the R-GMA, time is measured whenever information is acquired. Some information, for example that published via a DataBaseProducer, gets stored in a database. The quantity of information may be very large, so the timestamp should be stored efficiently. The GGF timestamp can be stored in a short string, but mapping onto relational tables is less efficient (DATE and TIME plus the various precision fields). By way of comparison, LDAP would require 2 fields, one for the date and time (combined) and one for the rest. However we expect that in practice the field holding quantities such as precision will not change very often, and so some optimisations may be possible.

6.6 Configuration files

User settings are made by Property files. Currently these must be in the user's home directory but this will be improved to allow a sequence of locations to be searched. One important thing to be specified in

a Property file is a list of locations where a ConsumerServlet may be found. Installation settings for the servlets themselves are made in the `web.xml` files.

6.7 API code

Code currently exists in C++ and Java. The code is currently written by hand for the two languages. A C API will also be provided and the Simplified Wrapper and Interface Generator (SWIG) [17] will be used to generate interfaces for Perl, Python and Tcl.

The communication between the servlets and the API is described below in Appendix A. The Java API of our current code is shown in Appendix B. Some features of the Java API (the C++ one is very similar) and the code implementing the API will be discussed in the following sections.

6.7.1 Producers

As explained in Section 3 the CircularBufferProducer and the DataBaseProducer both implement the Publisher interface.

To publish data, the `insert` method is invoked. This takes a normal SQL `INSERT` statement. If the time-stamp is not set then the producer will derive a time-stamp from the system time. The `localBufferSize` property controls how many tuples (rows) of data are cached before a transfer is made to the servlet. The `timeout` property allows the person publishing information via a producer to request that tuples are transmitted before the buffer is full.

The CircularBufferProducer has a property `remoteBufferSize` which controls the size of the circular buffer. The CircularBufferProducerServlet (and some other components of the system) makes use of a Java SQL library which is able to build a parse tree from the input SQL and carry out some simple operations on tables.

The DataBaseProducer communicates with a relational database (we have tested MySQL and Postgresql) via JDBC. There is currently no mechanism to construct indices in the database used by the DataBaseProducerServlet. However a DataBaseProducer is not in a good position to know what indices to define as it depends upon the Consumers' access patterns. So perhaps the Consumer servlets or the Mediator servlets should control index creation. Indices are purely an optimization issue; as you add indices searching time goes down but insertion time goes up.

6.7.2 Consumer

The consumer is created with a `String` representing the SQL query you wish to execute. An alternative constructor also allows the specification of the URL of the producer to which you wish to connect. You can then either execute a query which returns (in the case of Java) an `org.edg.info.ResultSet` or you can via the Consumer API set the `bufferSize` of the ConsumerServlet to be non-zero which allows the information to be streamed to the ConsumerServlet, after which you can use `count` to see how many records are available and `pop` to get a record - again as an `org.edg.info.ResultSet`. The `org.edg.info.ResultSet` is a subset of a `java.sql.ResultSet`.

The ConsumerServlet makes a temporary connection to a ProducerServlet for a query to be executed once, or makes a permanent connection to allow data to be streamed.

The mechanism of using the Consumer API to count the records available in the ConsumerServlet buffer and then popping them is a little clumsy but does have the advantage that the client API is simple. It should also be possible to call some method which causes the process to sleep until data are available - as is the case with a normal read operation. This will be provided for those API languages which can support it.

6.7.3 Registry

The Registry object talks to a servlet which handles access to an RDBMS via JDBC where the association between producers and the tables they produce is held. The database also holds the values of any fixed columns for a particular producer.

A Producer can register its existence and the table it produces by invoking the `register` method. The consumer can then discover the producer by invoking the `getProducerConnections` method which returns a `Vector` of `org.edg.info.ProducerConnection`.

6.7.4 Schema

The Schema object talks to a servlet which handles access to an RDBMS via JDBC which holds the description of all the known tables. It holds two tables one which describes tables and one which describes the columns of the tables.

6.7.5 Archiver

The archiver collects information and republishes it. The Archiver object holds little code but communicates with a nearby `ArchiverServlet` which makes use of a `DataBaseProducer`.

An Archiver is instantiated with the details of the RDBMS it is to use. It is informed about each "table" it is to publish by being told to add the table. The Archiver simply instantiates a `DataBaseProducer`, and then for each table it adds it to the `DataBaseProducer` and instantiates a `Consumer` to collect that information and sets up a thread to collect the information and re-publish it to the `DataBaseProducer`. It might appear that the task of the Archiver is so simple that it does not warrant a servlet. The advantage of the servlet is that it will be possible for this to be told to continue after the API object which started it has vanished.

7 Sensors and Displays

The R-GMA is a framework. It should be possible to integrate it with a number of existing systems, some of which are listed by Balaton et al. [1].

Existing monitoring and information systems can be broken into two parts: a sensor and a display. We plan to be able to use the various sensors to produce data and displays to consume data. This will generally require small modifications to the existing sensor and display code.

7.1 MDS Producer

The purpose of the MDS Producer sensor is to publish all the information available from a Globus GRIS server or in fact from any LDAP server into the R-GMA and to permit a Consumer to access this information using the R-GMA approach. This is one part of a strategy to allow MDS and R-GMA to interwork; MDS info provider scripts can also be written which will obtain information published via R-GMA and make it available via MDS.

The Globus GRIS server publishes information about the status of the Grid and its components, such as available CPU nodes, available service types and the status of batch queues. The server is implemented using the LDAP protocol, with the information stored in a tree-like LDAP directory structure. Each piece of information is associated with an attribute, with the permitted attributes being defined and grouped by an LDAP schema into 'object classes'. The context of the information is given by its position within the directory structure. Since MDS and R-GMA are based on two very different data models a generic solution to the interworking of the two approaches does not seem to be possible. What is being offered here is a way to make the information offered by a GRIS also available as tables within R-GMA. Currently the MDSProducer simply polls a GRIS and republishes as R-GMA. As a next step we envisage an implementation which uses the information providers that feed the GRIS directly, thereby removing the dependency on LDAP. As application and middleware programmers become more familiar with R-GMA a reorganisation of the tables can be undertaken to use R-GMA efficiently.

There are 6 ObjectClasses defined in the Globus 1.1.3 release:

- **globusBenchmarkInformation**
- **globusNetworkInterface**
- **globusQueue**
- **globusServiceJobManager**
- **globusSoftware**
- **grADSoftware**

Furthermore DataGrid publishes information according to a number of ObjectClasses which are republished into the following set of R-GMA tables:

- **NetMonHostLoss**
- **NetMonHostRTT**
- **NetMonHostThroughput**

- **NetMonLossPacketSize**
- **NetMonRTTPacketSize**
- **NetMonThroughputBufferSize**
- **NetMonTooliperfER**
- **NetMonToolpingER**
- **SiteInfo**
- **StorageElement**
- **StorageElementProtocol**
- **StorageElementStatus**

For the Globus GRIS there is exactly one table in R-GMA for each of the objectclasses. Since each schema consists of a number of attributes, these attributes form the column names of the relational table. An additional column is added to each table, giving the LDAP distinguished name (DN), or the context, of the entry. The way the DataGrid LDAP schemas are used is more complicated especially for the networking information, but there is a correspondence between a table and a certain combination of objectclasses. R-GMA cannot currently republish information about the FileElement objectclass because this information is not permanently held in the LDAP server but dynamically created requiring the knowledge of a local filename. The MDSProducer is completely generic and only assumes knowledge about the names of the objectclasses.

7.2 NetLogger

NetLogger [9, 10] relies upon placing monitoring calls in the application code - these calls are analogous to those made to publish data via an R-GMA Producer. The messages produced are routed to a file or to a `netlogd` daemon (written in Python) according to the setting of an environment variable. We will write a modified `netlogd` which will receive the NetLogger messages and instantiate one or more producers to publish data in R-GMA style. We may also providing a wrapper around our producer code to emulate the NetLogger calls and then not require the NetLogger library nor a special `netlogd`. The third NetLogger style component will be able to read or to `tail` a file of NetLogger data and publish it via R-GMA. To interface to the NetLogger display we will write code which instantiates a consumer and talks directly to the NetLogger display.

7.3 GRM and PROVE

GRM and PROVE are application monitoring and visualisation tools of the P-GRADE graphical parallel programming environment. These tools will be modified for application monitoring in the DataGrid. The instrumentation library of GRM is generalised for a flexible trace event specification. The components of GRM will be connected to the R-GMA using its Producer and Consumer APIs.

7.3.1 Instrumentation for GRM

The application code should be instrumented with calls to produce trace events. GRM provides an instrumentation API and library for tracing. Each trace event can be thought of a a row of a table.

Start and end of process. The application process must call the `GMI_Start()` function to initialise the instrumentation library and prepare for trace generation. The name of the process and a unique identifier should be given as arguments. The last trace event should be generated by calling the `GMI_End()` function.

Block trace events. The basic event types are the `BeginBlock` and `EndBlock` events that can be used for defining the start and the end of code regions in the application process. PROVE shows a colored box representing the program region.

User defined trace event types. To give a flexible trace definition method to the user, new event types can be defined giving a format string and an ID.

```
int GMI_DefineEvent( int eid, char *format, char *descr);
```

The predefined format ensures that all events with the given identifier will have the same format in the trace file. Events can be generated giving the event ID (`eid`) and data values according to the format specification. This is like using `printf` in the code.

```
int GMI_Event( int eid, ... );
```

7.3.2 Connection of GRM and R-GMA

The series of trace data is very similar to a relational table. Different types of events have different fields and so need to be held in different tables. The instrumentation library in the application process can then produce such events and propagate them through R-GMA.

Figure 7.1 shows the R-GMA based monitoring and the relation between the components.

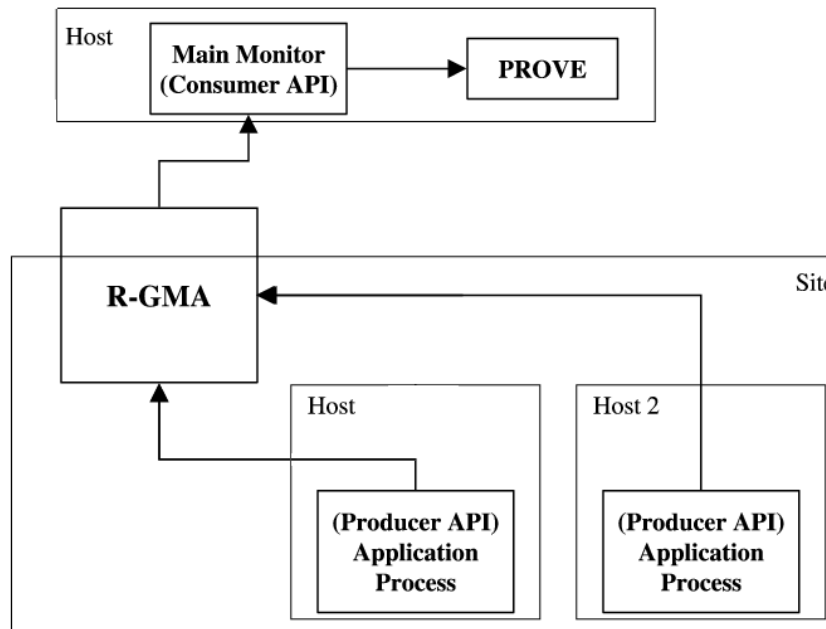


Figure 7.1: Structure of GRM in R-GMA

The R-GMA architecture provides a standard way to deliver performance data from the execution environment to the place of analysis. The application processes generate trace events that should be

transferred to the PROVE visualisation tool. The instrumentation library will use the Producer API of R-GMA to publish the trace data.

Instead of modifying PROVE, the Main Monitor process of GRM will be modified to behave as a Consumer in the R-GMA architecture. It will collect all trace data from the application using the APIs of R-GMA. The generation of the trace file and the visualisation tool will not need to be modified.

The producers of trace data (instrumented application processes) and the consumer (Main Monitor) can be running in different domains behind firewalls. Thus, they may not be able to communicate directly with each other. Fortunately, the connection is not direct in the implementation of R-GMA. The application processes are connected to a Producer Servlet (through the Producer API) while the consumer communicates through a Consumer Servlet (through the Consumer API). These servlets provide a route within the R-GMA from the Producer to the Consumer.

7.3.3 Visualisation with PROVE

PROVE presents the trace of the execution in a time-space diagram (see bottom right window in Figure 7.2 below). The status of the process is shown as a colored box over the time axis. If the user clicks on a box detailed information about the corresponding event is printed in the message window (start time, end time, type of the block). Communication between two processes is represented by arrows.

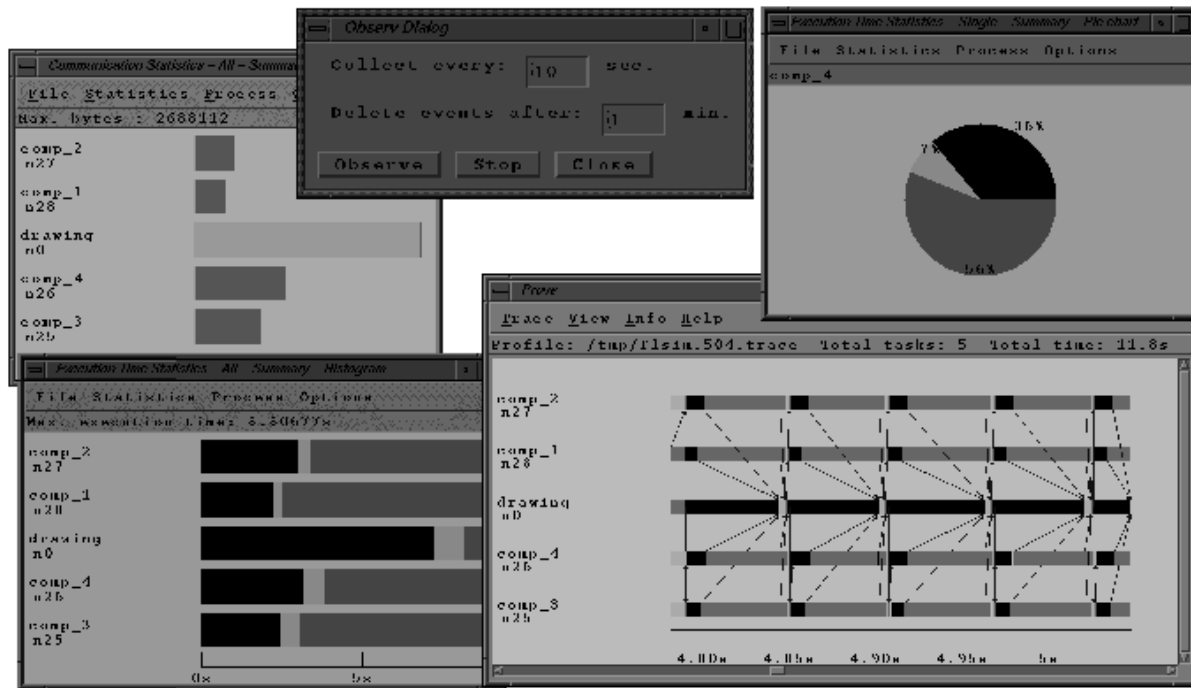


Figure 7.2: PROVE displays about the execution of an application

PROVE has also several statistical displays about the time spent in each code region or about the communication in each (or all) communication region(s) of the code.

PROVE is a semi-on-line tool, that is, it can collect trace data regularly from the monitoring tool and refresh the displays. The top center dialog in Figure 7.2 is the observation dialog in which the user can define the frequency of the collection and the amount of data to be kept in memory (PROVE deletes all events that are older than the specified time value).

In the current version, PROVE can support the visualisation of user-defined trace events with one type of visualisation. A user-defined event is represented as a small standing box on the time axis at the point

when the event had been generated. The color of the box depends on the identifier of the event and can be defined in the configuration file of PROVE. If the user clicks on such a box the full event string is printed in the message window.

Several processes are merged into a trace file and are visualised in PROVE. It must be ensured that different processes have different process identifiers in the trace file (this identifier is defined by the application process when it calls the GMI.Start() function). The task ID of the process in the parallel application is the most appropriate identifier. It is also important that in message events these identifiers should be used to identify the source and target of the communication.

8 Evaluation Criteria

This chapter outlines our evaluation criteria, some of which are easier to assess than others. We have chosen two tests, one based on functionality and one on performance.

We will also make comparisons with MDS (an information service) and with NetLogger, which is primarily a monitoring system.

We hope to demonstrate that the R-GMA system is a good general purpose solution as a Grid Information and Monitoring System and is well matched to the requirements that we have listed in Section 5.

A number of alternatives, MDS, Ftree and R-GMA, are being considered as the basis of the information service. As part of the activities of WP3, these implementations will be evaluated and compared using a set of performance, scalability and reliability criteria to determine which is the most suitable for deployment within the testbeds. These evaluations will be documented and discussed with the other members of the project and authors of the implementations.

8.1 Functionality

The first criterion is that the system should be able to do what is described in the set of Use Cases in Section 4

8.2 Performance

Within a grid environment the derivation of a performance metric in terms of specific values is extremely difficult. However if it takes several minutes for each simple query to return the system is clearly not a usable one. One of our goals will be to achieve a strong user base and we do not believe this will be possible if we suffer from inadequate performance. We will of course work with our users to improve performance immediately in any critical areas.

Precise criteria for acceptable performance for different examples of usage will be derived later after consultation with users. For example, if a user wishes to find a Computing Element on which to run a simple job they will probably expect to find this within a few seconds at most. If they wish to make a complex query interrogating several legacy databases it is likely this will take a few minutes at least.

For the purposes of this document our second criterion is to be able to answer the query:

Give me all computing elements with processor x with software package y within my VO within 5 seconds.

We have designed a system which is extremely flexible and as such we should be able to alter the system ratio of servlets to API connections and the distribution of these servlets and associated databases to improve performance at any point. We have also designed our software using servlets, partially due to the desire to add functionality extremely quickly. This design does allow for the replacement of servlets or re-implementation of components to produce a more performant system. The exact direction of changes and specific changes required depend upon the type and amount of workload to which the system will be subjected. This knowledge won't be fully understood until the system has been used by real users. It is also very important to minimise the impact which acting as a producer of information will have. There will be an impact both upon the application producing the information and upon the rest of the Grid.

A XML based protocols

Here we list the protocols defined so far. To interpret these tables please see the explanation of one of the tables in Section 6.2. These tables represent the current implementation of our protocols. Some details are still in a state of flux and it will be indicated how we expect things to change.

The general principle of the architecture is that every object that a user instantiates talks to a service-entity of the same name (except for the word Servlet at the end which stems from our implementation choice). This service entity contains the “business logic” and does the actual work. The APIs are coded in multiple languages. It is natural to follow this approach also for the interactions of components inside R-GMA such as the storage of table descriptions and the registration of producers. In general every method of an API object corresponds to a method of the associated service-entity. We sometime refer to the methods of the servlets as services that the servlet offers to the associated object. For reasons of efficiency the communication between a Consumer service-entity and a the (CircularBuffer/Database)Producer service-entity is special and there are methods of the latter that are not accessed through an API. In what follows we refer to these service-entities as servlets which is how they are implemented.

Currently the error handling is very rudimentary and none of the servlets return a proper XML-encoded error object. The entries for Returned Error in the tables below is therefore to be understood as what will shortly be returned.

A.1 CircularBufferProducerServlet

For each Producer object that uses the service of this Servlet, the Servlet sets up a circular buffer whose size can be changed by the respective Producer object. If new data are written more quickly by a Producer than data are read by a subscribed Consumer through its associated ConsumerServlet, data can be lost. If this is not acceptable, a DataBaseProducer should be used instead. It is possible to stream data from a CircularBufferProducerServlet to a ConsumerServlet. The stream method is however not accessed through an API and therefore not directly accessible to a Consumer object. When data are streamed the Consumer object accesses the data through a queue from which data are popped and to which entries are added by the CircularBufferServlet.

A.1.1 CircularBufferProducer

This method corresponds to the constructor of the CircularBufferProducer object. The servlet will assign a connectionId to the requester which is returned and registers the requester with the registry. If tableName is known by the Schema, tableDesc can be omitted. Currently a Producer can only produce known tables. The parameter fixedColumns is used to assert values for particular columns, i.e. the Producer publishes a table where these columns are fixed. This information which is stored in the registry can be used to restrict the number of Producers that have to be contacted in order to satisfy a query. This parameter will in future be replaced by a more general predicate about the produced information expressed as an SQL WHERE clause. The case of fixed columns will then be a special case of this assertion.

Request Parameters			
Name	Type	Cardinality	Meaning
tableName	String	1..1	The name of the table to register
fixedColumns	String	0..1	The names of the columns which are fixed and their values, a colon separated list of "=" separated name-value pairs.
flags	Int	0..1	A set of Boolean flags encoded in an integer.
tableDesc	SQL	0..1	An SQL CREATE TABLE string for the table being registered.

Returned ResultSet		
Name	Type	Meaning
connectionId	Int	The ProducerServlet-assigned connection ID.

Returned Error	
Name	Meaning
RegistrationError	The registration of the specified producer did not complete successfully

A.1.2 insert

This method is used to publish data. The connectionId assigned by the Servlet is used in any further requests to identify the Producer object to the Servlet that serves it. The data to be published is sent as if inserting into a table in a database. The insert statement is parsed by the servlet and the data is stored in a way that allows SQL queries to be executed against it. Currently the insert statement inserts one and only one row. This will be modified shortly to allow multiple rows to be inserted.

Request Parameters			
Name	Type	Cardinality	Meaning
connectionId	Int	1..1	The connection ID identifying the producer to the servlet.
insert	SQL	1..1	The SQL insert string for the table this producer has registered

Returned Status	
Name	Meaning
OK	the insert method was successful

Returned Error	
Name	Meaning
InsertError	The specified row(s) was(were) not successfully delivered to the ProducerServlet. A consumer will not be able to consume this information.

A.1.3 execute

This method is not accessed through the CircularBufferProducer API but is one of the two methods that the ConsumerServlet calls to retrieve data that the CircularBufferProducer (identified by connectionId) has inserted into the circular buffer on the Servlet. When called, the method returns the result of the SQL query executed against the most recent row inserted. Note that the ConsumerServlet has a method of the same name which corresponds to a method of the Consumer object. The chain of events is as follows: when the execute method of the Consumer object is executed, it requests the execute service on the ConsumerServlet which in turn contacts the relevant CircularBufferProducerServlet where the query is executed and the result passed back to the ConsumerServlet, which passes it back to the requesting Consumer object.

Request Parameters			
Name	Type	Cardinality	Meaning
connectionId	Int	1..1	The connection ID identifying the producer to the servlet.
select	SQL	1..1	An SQL select string.

Returned ResultSet		
Name	Type	Meaning
ColumnName1	String	Value of the first column corresponding to the result of the query, a string in this example.
ColumnName2	Int	Value of the second column corresponding to the result of the query, an int in this example.
...	...	And so forth

Returned Error	
Name	Meaning
QueryError	An Error occurred when executing the query.

A.1.4 stream

This is the second method that is not accessed through the CircularBufferProducer API but is called by the ConsumerServlet to stream data from the CircularBufferProducerServlet. The CircularBufferProducerServlet internally maintains a pointer to where in the circular buffer data are being inserted by the CircularBufferProducer object and where data are being read from by the thread that streams the data to the ConsumerServlet. The pointers wrap around when the end of the buffer is reached. It is ensured that the read pointer never overtakes the write pointer, hence no duplicate entries are delivered to the Consumer object if the streaming thread reads faster than the Producer object inserts data. However the reverse is not true, i.e. the reading thread does not block the writing thread. This means that when the write pointer overtakes the read pointer data will be lost. To start streaming the method is called with interruptStreaming set to true and to stop the streaming with interruptStreaming set to false.

Request Parameters			
Name	Type	Cardinality	Meaning
producerId	Int	1..1	The connection ID identifying the producer on the ProducerServlet.
select	SQL	1..1	An SQL select statement
consumerId	Int	1..1	The connection ID identifying the consumer on the ConsumerServlet.
interruptStreaming	Boolean	1..1	A Boolean value telling the ProducerServlet, whether or not to interrupt streaming

Returned ResultSet		
Name	Type	Meaning
ColumnName1	String	Value of the first column corresponding to the result of the query, a string in this example.
ColumnName2	Int	Value of the second column corresponding to the result of the query, an int in this example.
...	...	And so forth

Returned Status	
Name	Meaning
OK	Streaming was successfully interrupted

Returned Error	
Name	Meaning
QueryError	An error occurred when executing the query

A.1.5 setBufferSize

The size of the circular buffer can be set with this method. The default value is one. It is important to set the buffer size appropriately since with a buffer size of one, every newly published row overwrites the previously published one, i.e. only the last inserted row is ever available unless a Consumer asks for the data to be streamed. If a relevant set of information consists of more than one row, e.g. networking information from a site A to n other sites, the buffer size should be set to a multiple of n. In future this will not be necessary as we will allow multiple inserts to be carried out in one operation and the set of inserted rows will occupy one slot in the circular buffer.

Request Parameters			
Name	Type	Cardinality	Meaning
connectionId	Int	1..1	The connection ID identifying the producer to the servlet.
bufferSize	Int	1..1	The size of the cyclic buffer on the ProducerServlet that the Producer with the specified connectionId uses to store its data.

Returned Status	
Name	Meaning
OK	The buffer size was successfully set to bufferSize

Returned Error	
Name	Meaning
BufferError	The buffer size was not successfully set to the required size.

A.1.6 getBufferSize

The size of the circular buffer can be queried using this method.

Request Parameters			
Name	Type	Cardinality	Meaning
connectionId	Int	1..1	The connection ID identifying the producer to the servlet.

Returned ResultSet		
Name	Type	Meaning
bufferSize	Int	The currently set buffer size on the ProducerServlet for the Producer with the given connectionId.

Returned Error	
Name	Meaning
BufferError	An error occurred when trying to retrieve the buffer size.

A.1.7 getStatus

This method is used to return status information about the Producer identified by connectionId. The information returned is likely to be more useful in the future.

Request Parameters			
Name	Type	Cardinality	Meaning
connectionId	Int	1..1	The connection ID identifying the producer to the servlet.

Returned ResultSet		
Name	Type	Meaning
connectionId	Int	The connection ID identifying the producer to the servlet.
bufferSize	Int	The size of the Buffer on the ProducerServlet.
columnNames	String	A comma separated list of the names of the columns of the table this producer has registered.

Returned Error	
Name	Meaning
QueryError	The Status of the producer identified by connectionId could not be retrieved.

A.2 DataBaseProducerServlet

We pointed out above, that by its very nature the CircularBufferProducer cannot guarantee the persistence of the data. Writing published data into a database ensures this persistence. The database is therefore only the persistence medium. Hence this is not a general database service and no administrative functions are provided. There are some subtle differences between the CircularBufferProducerServlet and the DataBaseProducerServlet. First of all since data are inserted into a database there is no buffer size and hence no associated methods and one cannot stream data from a DataBaseProducerServlet. Streaming is possible in some simple cases but we have not implemented this yet. For the same reason the execute method does not just return the result of the query executed against the last inserted row, but the result of the query executed against the whole of the table. Furthermore a DataBaseProducer object can act as a Producer of more than one table. Each table is registered individually. Likewise the DataBaseProducerServlet can accept queries against all tables it registered.

A.2.1 DataBaseProducer

This method corresponds to the constructor of the DataBaseProducer object. It assigns a connectionId to the DataBaseProducer object it serves and sets up the connection to the database.

Request Parameters			
Name	Type	Cardinality	Meaning
rdbms	URL	1..1	The RDBMS to be used by the DataBaseProducerServlet to store the data persistently.
user	String	1..1	The username for the RDBMS
password	String	1..1	The password to access the RDBMS

Returned ResultSet		
Name	Type	Meaning
connectionId	Int	A DataBaseProducerServlet assigned and locally unique ID identifying the Archiver to the Servlet.

Returned Error	
Name	Meaning
RegistrationError	The set up of the DataBaseProducer was not successful.

A.2.2 add

This operation adds a table to the (initially empty) set of tables that the DataBaseProducer object publishes. Each added table is registered as a separate producer with the registry. It is (currently) assumed that the table already exists in the database specified in the the DataBaseProducer method. The other parameters have the same meaning as for the CircularBufferProducerServlet.

Request Parameters			
Name	Type	Cardinality	Meaning
tableName	String	1..1	The name of the table to register
fixedColumns	String	0..1	The names of the columns which are fixed and their values, a colon separated list of "=" separated name-value pairs.
flags	Int	0..1	A set of Boolean flags encoded in an integer.

Returned Status	
Name	Meaning
OK	The add operation was successful.

Returned Error	
Name	Meaning
RegistrationError	The add operation was not successful.

A.2.3 insert

Since the DataBaseProducer handles more than one table, the insert service is slightly different. For the CircularBufferProducer there is (currently) exactly one insert statement corresponding to one row of the table. For the DataBaseProducerServlet the insert parameter can appear any number of times. It can take a number of different insert statements for the same table, or an insert statement for each table or any mixture. Since these insert statements are passed on to the database, it takes care that data are inserted in the right table.

Request Parameters			
Name	Type	Cardinality	Meaning
connectionId	Int	1..1	The connection ID identifying the DataBaseProducer to the servlet.
insert	SQL	1..*	The SQL insert string(s) for the table(s) this DataBaseProducer has registered

Returned Status	
Name	Meaning
OK	The insert operation was successful.

Returned Error	
Name	Meaning
InsertError	The specified row(s) was(were) not successfully delivered to the ProducerServlet. A consumer will not be able to consume this information.

A.2.4 execute

Because the DataBaseProducer stores records persistently in a RDBMS, the execute method can return more than one record contrary to the CircularBufferProducerServlet where execute returns the most recent record (or in future a set of records from the most recent slot). As in the case of the CircularBufferProducerServlet this method is not accessed through the DataBaseProducer API but is used by

the ConsumerServlet to pull data from the DataBaseProducerServlet and corresponds ultimately to the execute methods of the Consumer API.

Request Parameters			
Name	Type	Cardinality	Meaning
connectionId	Int	1..1	The connection ID identifying the DataBaseProducer to the servlet.
select	SQL	1..1	An SQL select string.

Returned ResultSet		
Name	Type	Meaning
ColumnName1	String	Value of the first column corresponding to the result of the query, a string in this example.
ColumnName2	Int	Value of the second column corresponding to the result of the query, an int in this example.
...	...	And so forth

Returned Error	
Name	Meaning
QueryError	An Error occurred when executing the query.

A.2.5 getStatus

See the remarks for the same method of the CircularBufferProducerServlet.

Request Parameters			
Name	Type	Cardinality	Meaning
connectionId	Int	1..1	The connection ID identifying the DataBaseProducer to the servlet.

Returned ResultSet		
Name	Type	Meaning
connectionId	Int	The connection ID identifying the DataBaseProducer to the servlet.

Returned Error	
Name	Meaning
QueryError	The Status of the DataBaseProducer identified by connectionId could not be retrieved.

A.3 RegistryServlet

The Registry object is used for two operations. One is the registration of producers requested by ProducerServlets, the other is the discovery of suitable producers to answer queries requested by ConsumerServlets. In a future release soft state registration of producers and a better description of what producers provide (see the discussion about fixedColumns vs. assertions above) will be implemented. This will entail changes in the details of the API, but since the Registry does not need to be accessed directly by a user of R-GMA this is not a great concern. We will nevertheless provide GUIs to browse/administer the registry of producers.

The Registry does not store details about the tables that producers publish. This task is handled by the Schema object. The registration of producers and the descriptions of what these producers provide are two different tasks. This is similar to the way that for web services the UDDI registry only registers the existence of a web service and keeps a pointer to an WSDL document describing what the service offers.

A.3.1 register

This is the method which is invoked to register a producer. A Producer object is identified by the combination of the URL of the ProducerServlet that serves it and the connectionId assigned to it. The combination of the two must be unique. The registration of Producers of tables that are not known by the Schema is not yet supported.

Request Parameters			
Name	Type	Cardinality	Meaning
tableName	String	1..1	The name of the table to register
fixed	String	0..*	The name of a column which is fixed and its value, an "=" separated name-value pair.
flags	Int	0..1	A set of Boolean flags encoded in an integer.
tableDesc	SQL	0..1	An SQL CREATE TABLE string for the table being registered.
producerURL	URL	1..1	The URL of the ProducerServlet serving the producer
connectionId	Int	1..1	The ProducerServlet assigned and locally unique ID if the producer.

Returned Status	
Name	Meaning
OK	The registration of the producer was successful.

Returned Error	
Name	Meaning
RegistrationError	The registration of the producer was not successful.

A.3.2 getProducerConnections

This is the method that ConsumerServlets use to discover producers that can answer a particular query. The simplest discovery is by tableName in which case every producer identified by the URL of the corresponding servlet and the connectionId is returned. Giving more qualifications such as the fixed values of columns restricts the set of returned Producers. It is of course possible that no producer matches the search criteria.

Request Parameters			
Name	Type	Cardinality	Meaning
tableName	String	1..1	The name of the table.
flags	Int	0..1	A set of Boolean flags encoded in an integer.
fixed	String	0..*	The name of a column which is fixed and its value, an "=" separated name-value pair.

Returned ResultSet		
Name	Type	Meaning
producerURL	URL	The URL of the ProducerServlet serving the producer.
connectionId	Int	The ProducerServlet assigned and locally unique ID of the producer.

Returned Error	
Name	Meaning
NoSuchProducerError	The operation failed.

A.3.3 getProducerInfo

This method returns detailed information about a particular producer. The information is so far very simple but will be improved to allow browsing of the registry.

Request Parameters			
Name	Type	Cardinality	Meaning
producerURL	URL	1..1	The URL of the ProducerServlet serving the producer.
connectionId	Int	1..1	The ProducerServlet assigned and locally unique ID of the producer.

Returned ResultSet		
Name	Type	Meaning
columnName	String	The name of a column of the table produced by this producer.

Returned Error	
Name	Meaning
NoSuchProducerError	The operation failed.

A.4 SchemaServlet

The Schema holds the definition of the tables that producers can publish. Currently dynamic schema evolution is not supported but will be in a future release. This relies on an implementation of soft state registration of tables such that a table definition is removed from the Schema once the last producer of the table is gone. We expect a number of predefined tables to exist as required by the middleware. Methods to browse the Schema will also be included.

A.4.1 translateTableName

For efficiency reasons it is faster to use an index rather than the name of the table in the registry. This method provides the translation between tableName and tableId.

Request Parameters			
Name	Type	Cardinality	Meaning
tableName	String	1..1	The name of a table.

Returned ResultSet		
Name	Type	Meaning
tableId	Int	The tableId of the requested table in the Schema.

Returned Error	
Name	Meaning
NoSuchTableError	The tableName could not be translated into a tableId.

A.4.2 translateColumnName

When the Registry stores information about fixedColumns it does so by using an index for the name of the columns. This method provides the translation between columnName and columnId.

Request Parameters			
Name	Type	Cardinality	Meaning
tableId	Int	1..1	The column ID of a table.
columnName	String	1..1	The name of a column of that table.

Returned ResultSet		
Name	Type	Meaning
columnId	Int	The ID of the requested column in the Schema.

Returned Error	
Name	Meaning
NoSuchColumnNameError	For the specified tableId, columnName could not be translated into a columnId.

A.4.3 getTableInfo

Since the Registry does not store information about the tables producers publish, it needs to contact the Schema to retrieve this information when requested through the getProducerInfo method of the Registry.

Request Parameters			
Name	Type	Cardinality	Meaning
tableId	Int	1..1	The ID of a table.

Returned ResultSet		
Name	Type	Meaning
columnName	String	The name of a column of the specified table.

Returned Error	
Name	Meaning
NoSuch TableError	no table info could be retrieved for this tableId.

A.5 ConsumerServlet

The Consumer is the object through which users of R-GMA can retrieve information published by Producers. The information that is required is specified as an SQL **SELECT** statement. This is where the power of the relational model comes in. The **SELECT** statement can join information from different Producers. The Registry and Schema are used to locate relevant Producers, however this is completely transparent to the user.

A.5.1 Consumer

This corresponds to the constructor of the Consumer object. It is possible to subscribe to a particular Producer by specifying both the producerURL and the producerId. If those are not given, the ConsumerServlet contacts the Registry to discover Producers that can answer the query passed by the select parameter. The **SELECT** statement is stored for later reference. Currently the ConsumerServlet can only deal with the situation where the Registry returns exactly one ProducerConnection. How to deal with general queries requiring integration of information from different producers (e.g. joins) is currently under investigation and will be part of a future release.

Request Parameters			
Name	Type	Cardinality	Meaning
select	SQL	1..1	An SQL select statement.
producerURL	URL	0..1	The URL of the producer Servlet.
producerId	Int	0..1	Producer Id identifying the producer against which the select statement is executed.

Returned ResultSet		
Name	Type	Meaning
consumerId	Int	A consumer-servlet-assigned and locally unique ID of the consumer, by which a consumer identifies himself to his consumer servlet in future requests.

Returned Error	
Name	Meaning
SubscriptionError	The set-up of a consumer for the specified Producer failed.

A.5.2 getStatus

This method is not yet fully implemented, it will eventually return relevant information about a Consumer, but currently only returns the connectionId.

Request Parameters			
Name	Type	Cardinality	Meaning
connectionId	Int	1..1	The connection ID identifying the consumer to the servlet.

Returned ResultSet		
Name	Type	Meaning
connectionId	Int	The connection ID identifying the consumer to the servlet.

Returned Error	
Name	Meaning
QueryError	The status of the consumer identified by connectionId could not be retrieved.

A.5.3 setBufferSize

This method sets the size of the queue into which data from a CircularBufferProducerServlet are streamed. When set to a nonzero value, the ConsumerServlet calls the stream method on the ProducerServlet requesting streaming to start. When set back to zero The ConsumerServlet calls the stream method on the CircularBufferProducerServlet again to stop the streaming thread.

Request Parameters			
Name	Type	Cardinality	Meaning
consumerId	Int	1..1	The connection ID identifying the consumer to the servlet.
bufferSize	Int	1..1	The size of the queue buffer on the ConsumerServlet, where the streamed data from the ProducerServlet are stored.

Returned Status	
Name	Meaning
OK	The buffer size was successfully set to bufferSize.

Returned Error	
Name	Meaning
BufferError	The buffersize was not successfully set to the required size.

A.5.4 getBufferSize

This the corresponding getter method to retrieve the current value of the buffer size.

Request Parameters			
Name	Type	Cardinality	Meaning
consumerId	Int	1..1	The connection ID identifying the consumer to the servlet.

Returned ResultSet		
Name	Type	Meaning
bufferSize	Int	The currently set buffer size on the ConsumerServlet for the Consumer with the given connectionId.

Returned Error	
Name	Meaning
BufferError	An error occurred when trying to retrieve the buffer size.

A.5.5 execute

This method executes the SQL query of the Consumer object and retrieves the latest published row (or in the future set of rows) stored on a CircularBufferProducerServlet or all matching rows from a DataBaseProducerServlet. See the explanations about the execute method on the respective servlets for more information. Since the SQL query to be executed is stored on the ConsumerServlet, only the consumerId identifying the Consumer object to the ConsumerServlet needs to be passed.

Request Parameters			
Name	Type	Cardinality	Meaning
consumerId	Int	1..1	The connection ID identifying the consumer to the servlet.

Returned ResultSet		
Name	Type	Meaning
ColumnName1	String	Value of the first column corresponding to the result of the query, a string in this example.
ColumnName2	Int	Value of the second column corresponding to the result of the query, an int in this example.
...	...	And so forth

Returned Error	
Name	Meaning
QueryError	The execution of the select statement failed.

A.5.6 count

The count and pop methods are used to access the queue in which data streamed from a CircularBufferProducerServlet is stored. The count method returns the number of records in the queue.

Request Parameters			
Name	Type	Cardinality	Meaning
consumerId	Int	1..1	The connection ID identifying the consumer to the servlet.

Returned ResultSet		
Name	Type	Meaning
count	Int	The number of items in the consumer servlet's queue.

Returned Error	
Name	Meaning
BufferError	The number of items in the queue could not be determined.

A.5.7 pop

This method pops one record off the queue and returns it to the Consumer object.

Request Parameters			
Name	Type	Cardinality	Meaning
consumerId	Int	1..1	The connection ID identifying the consumer to the servlet.

Returned ResultSet		
Name	Type	Meaning
ColumnName1	String	Value of the first column corresponding to the result of the query, a string in this example.
ColumnName2	Int	Value of the second column corresponding to the result of the query, an int in this example.
...	...	And so forth

Returned Error	
Name	Meaning
QueryError	The execution of the select statement failed.

A.5.8 destroyConsumer

This method releases the resources allocated for the calling Consumer on the ConsumerServlet.

Request Parameters			
Name	Type	Cardinality	Meaning
consumerId	Int	1..1	The connection ID identifying the consumer

Returned Status	
Name	Meaning
OK	The destruction of the Consumer was successful.

Returned Error	
Name	Meaning
SubscriptionError	The destruction of the Consumer was not successful.

A.6 ArchiverServlet

The task of an archiver is to make it easy to consume data from a variety of Producers and republish them conveniently in a single location. The ArchiverServlet uses a DataBaseProducer to republish the data, hence to browse an Archiver one uses the Registry to find out which tables are republished and to read data from this DataBaseProducer one uses a Consumer. The issue of deleting data from the Database used to archive the data has not been addressed yet.

A.6.1 Archiver

This method corresponds to the Constructor of the Archiver object and takes the relevant parameters to set up the DataBaseProducer which republishes the data. Security is currently not implemented, but will be addressed in a future release.

Request Parameters			
Name	Type	Cardinality	Meaning
rdbms	URL	1..1	The RDBMS to be used by the ArchiverServlet to store the data.
user	String	1..1	The username for the RDBMS
password	String	1..1	The password to access the RDBMS

Returned ResultSet		
Name	Type	Meaning
connectionId	Int	An ArchiverServlet assigned and locally unique ID identifying the Archiver to the Servlet.

Returned Error	
Name	Meaning
RegistrationError	The set up of the archiver was not successful.

A.6.2 add

This methods takes parameters that are passed to the add method of the DataBaseProducer object to add a table to be published to the database. See the DataBaseProducer for more information. The producerConnection parameter can be used to identify one or more Producers from which to consume information. If no producerConnection is given, a Consumer with select statement

```
select * from tableName
```

and constraints give by fixedColumns and flags is set up.

Request Parameters			
Name	Type	Cardinality	Meaning
connectionId	Int	1..1	
tableName	String	1..1	The name of the table to archive.
fixedColumns	String	0..1	The names of the columns which are fixed and their values, a colon separated list of "=" separated name-value pairs.
flags	Int	0..1	A set of Boolean flags encoded in an integer.
producerConnection	String	0..*	Identifies a specific producer. A " " separated connectionId and producerURL.

Returned ResultSet		
Name	Type	Meaning
connectionId	Int	The connection ID identifying the archiver.

Returned Error	
Name	Meaning
ArchivingError	The add operation did not complete successfully.



B Java API

Package Contents

Page

Classes

Archiver	53
<i>Archiver is able to consume, archive and produce data.</i>	
CircularBufferProducer	54
<i>CircularBufferProducers is able to register a table when it is created and subsequently to publish information.</i>	
Consumer	56
<i>Consumes event by event or a stream of events.</i>	
DataBaseProducer	58
<i>DataBaseProducers is similar to a normal producer but uses an RDBMS as a buffer in the servlet.</i>	
NoSuchColumnNameException	60
<i>Thrown by Schema API if translateColumnNames is called with a column Name that does not exist in the Schema for the specified table</i>	
NoSuchProducerException	61
<i>Thrown by Registry API if getProducerConnections or getProducerInfo is called with invalid parameters or an error occurs that results in an invalid return of the methods</i>	
NoSuchTableNameException	62
<i>Thrown by Schema API if translateTableName is called with a table that does not exist in the Schema or if getTableInfo is called with an invalid tableId</i>	
ProducerConnection	63
<i>Identifies a connection to a producer servlet by a producer</i>	
ProducerInfo	63
<i>Producer Information</i>	
PropertyGetter	64
<i>Obtains properties from a set of locations.</i>	
QueryException	65
<i>Thrown by Consumer API if the execute method or pop method fail</i>	
RegistrationException	66
<i>Thrown by Registry API if the register method fails</i>	
Registry	67
<i>Registry of producers.</i>	
ResultSet	69
<i>R-GMA implementation of a ResultSet.</i>	
ResultSetMetaData	70
<i>R-GMA implementation of ResultSetMetaData.</i>	
Schema	70
<i>Schema of Tables.</i>	
ServletConnectionException	71
<i>Thrown by Schema API if any of the methods of the ServletConection fails</i>	
SubscriptionException	72
<i>Thrown by Registry API if the register method fails</i>	
XMLConverter	73
<i>Parses XML and constructs ResultSet</i>	

B.1 Classes

B.1.1 CLASS Archiver

Archiver is able to consume, archive and produce data. For each table it is instructed to handle it instantiates a consumer, to "select *" that table and stores it in an RDBMS (via JDBC) and announces itself as a producer of that information.

DECLARATION

```
public class Archiver
extends java.lang.Object
```

CONSTRUCTORS

- *Archiver*

```
public Archiver( java.lang.String rdbms, java.lang.String user, java.lang.String
password )
```

 - Usage
 - * Create an archiver and connect it to the specified RDBMS via JDBC.
 - Parameters
 - * `rdbms` - JDBC RDBMS identification
 - * `user` - JDBC user name
 - * `password` - JDBC user password

METHODS

- *add*

```
public void add( java.lang.String tableName, java.lang.String fixedColumns, int
flags )
```

 - Usage
 - * specify table to consume, archive and, in turn, produce
 - Parameters
 - * `tableName` - is the name of the SQL table
 - * `fixedColumns` - identifies the columns which are fixed and their values
 - * `flags` - is a set of boolean flags encoded in an integer

- *add*

```
public void add( java.lang.String tableName, java.lang.String fixedColumns, int
flags, java.util.Vector producerConnections )
```

 - Usage
 - * specify table to consume, archive and, in turn, produce
 - Parameters

- * `tableName` - is the name of the SQL table
- * `fixedColumns` - identifies the columns which are fixed and their values
- * `flags` - is a set of boolean flags encoded in an integer
- * `Vector` - of `producerConnections` to identify the producers

B.1.2 CLASS `CircularBufferProducer`

`CircularBufferProducers` is able to register a table when it is created and subsequently to publish information.

DECLARATION

```
public class CircularBufferProducer
extends org.edg.info.BaseProducer
```

CONSTRUCTORS

- *CircularBufferProducer*
`public CircularBufferProducer(java.lang.String tableName, java.lang.String fixedColumns, int flags)`
 - Usage
 - * For this constructor the `tableName` must already be known within the schema
 - Parameters
 - * `tableName` - is the name of the SQL table
 - * `fixedColumns` - identifies the columns which are fixed and their values
 - * `flags` - is a set of boolean flags encoded in an integer
- *CircularBufferProducer*
`public CircularBufferProducer(java.lang.String tableName, java.lang.String fixedColumns, int flags, java.lang.String tableDesc)`
 - Usage
 - * If the table is already known within the schema and is inconsistent with the `tableDesc` this constructor will fail.
 - Parameters
 - * `tableDesc` - is the table description. This is the same as the SQL for CREATE TABLE

METHODS

- *getRemoteBufferSize*
`public int getRemoteBufferSize()`
 - Usage
 - * Get the `CircularBufferproducer` servlet's buffer size

- *setRemoteBufferSize*
public void **setRemoteBufferSize**(int remoteBufferSize)

– Usage

- * Set the CircularBufferproducer servlet's buffer size

METHODS INHERITED FROM CLASS org.edg.info.BaseProducer

- *getLocalBufferSize*
public int **getLocalBufferSize**()

– Usage

- * returns the producer's local buffer size

- *getProducerConnection*
public ProducerConnection **getProducerConnection**()

– Usage

- * Get ProducerConnection back.

– Returns - ProducerConnection

- *getProperties*
protected void **getProperties**(java.lang.String className, java.lang.String propertyName)

– Usage

- * get Properties from somewhere.

- *getStatus*
public ResultSet **getStatus**()

– Usage

- * Get status information.

– Returns - ResultSet containing status information for this producer

- *getTimeout*
public TimeInterval **getTimeout**()

– Usage

- * get time interval after which local buffer is transmitted even if not full

- *getValidate*
public boolean **getValidate**()

– Usage

- * carry out local validation of the row(s) inserted as far as possible. Primarily this means checking that the columns match. (not implemented yet)

- *insert*
public void **insert**(java.lang.String row)

– Usage

- * If the timestamp field is null, the BaseProducer will generate a time stamp for you (not implemented yet).

– Parameters

- * row - as an SQL insert statement

- *setLocalBufferSize*
public void **setLocalBufferSize**(int localBufferSize)

– Usage

- * sets the producer's local buffer size

-
- *setTimeout*
`public void setTimeout(org.edg.info.TimeInterval timeout)`
 - Usage
 - * set time interval after which local buffer is transmitted even if not full
 - *setValidate*
`public void setValidate(boolean validate)`
 - Usage
 - * set value of validate. (not implemented yet)
 - *transmit*
`protected void transmit(java.lang.String row)`

B.1.3 CLASS Consumer

Consumes event by event or a stream of events. It is able to find a producer of information and consume it either by requesting individual events or by requesting a stream of events

DECLARATION

```
public class Consumer
extends java.lang.Object
```

CONSTRUCTORS

- *Consumer*
`public Consumer(java.lang.String selectStatement)`
 - Usage
 - * Construct a consumer using a String representing the SQL query.
 - Parameters
 - * `selectStatement` - the desired SQL select statement
- *Consumer*
`public Consumer(java.lang.String selectStatement, org.edg.info.ProducerConnection producerConnection)`
 - Usage
 - * Construct a consumer using a String representing the SQL query and a specified `ProducerConnection` Unlike the other constructor which relies in the registry to find a producer, this constructor takes a `ProducerConnection` as the second argument.
 - Parameters
 - * `selectStatement` - the desired SQL select statement
 - * `ProducerConnection` - identifies the Producer of the information

METHODS

- *count*
public int count()
 - Usage
 - * Number of available events. Returns the number of pieces of information which can be popped from the consumer servlet buffer. This information should have been streamed in when bufferSize was set to a value >0. This will lead to an exception if bufferSize =0.

- *execute*
public ResultSet execute()
 - Usage
 - * execute the Consumer's query to return an ResultSet. This method should be used to issue one SQL query to the producer and get the last tuple from the producer servlet buffer. **The schema of the XML will almost certainly be changed.**

- *finalize*
protected void finalize()
 - Usage
 - * Destroy query objects associated with this instance and unsubscribe from producer.

- *getBufferSize*
public int getBufferSize()
 - Usage
 - * get consumer servlet bufferSize

- *getStatus*
public String getStatus()
 - Usage
 - * Get status information.
 - Returns - XML formatted status information
 - Exceptions
 - * org.edg.info.ServletException -

- *pop*
public ResultSet pop()
 - Usage
 - * Return the oldest piece of information from the consumer servlet queue (buffer) and remove it. This information should have been streamed in when bufferSize was set to a value >0. This will lead to an exception if bufferSize =0. **The schema of the XML will almost certainly be changed.**

- *setBufferSize*
public void setBufferSize(int bufferSize)
 - Usage
 - * set consumer servlet buffersize. If this is greater than zero it allows information to be streamed from the producer servlet buffer to the consumer servlet buffer as it becomes available. In order to stop this streaming, bufferSize should be reset to zero.
 - Parameters
 - * bufferSize - - desired buffer size measured as number of events

B.1.4 CLASS DataBaseProducer

DataBaseProducers is similar to a normal producer but uses an RDBMS as a buffer in the servlet. Unlike a normal producer it can deal with multiple tables.

DECLARATION

```
public class DataBaseProducer
extends org.edg.info.BaseProducer
```

CONSTRUCTORS

- *DataBaseProducer*

```
public DataBaseProducer( java.lang.String rdbms, java.lang.String user, java.lang.String password )
```

 - **Usage**
 - * Create a DataBaseProducer and connect it to the specified RDBMS via JDBC.
 - **Parameters**
 - * **rdbms** - JDBC RDBMS identification
 - * **user** - JDBC user name
 - * **password** - JDBC user password

METHODS

- *add*

```
public void add( java.lang.String tableName, java.lang.String fixedColumns, int flags )
```

 - **Usage**
 - * For this method the tableName must already be known within the schema
 - **Parameters**
 - * **tableName** - is the name of the SQL table
 - * **fixedColumns** - identifies the columns which are fixed and their values
 - * **flags** - is a set of boolean flags encoded in an integer

- *add*

```
public void add( java.lang.String tableName, java.lang.String fixedColumns, int flags, java.lang.String tableDesc )
```

 - **Usage**
 - * If the table is already known within the schema and is inconsistent with the tableDesc this will fail.
 - **Parameters**
 - * **tableDesc** - is the table description. This is the same as the SQL for CREATE TABLE

METHODS INHERITED FROM CLASS `org.edg.info.BaseProducer`

- *getLocalBufferSize*
public int `getLocalBufferSize()`
 - Usage
 - * returns the producer's local buffer size
- *getProducerConnection*
public `ProducerConnection getProducerConnection()`
 - Usage
 - * Get `ProducerConnection` back.
 - Returns - `ProducerConnection`
- *getProperties*
protected void `getProperties(java.lang.String className, java.lang.String propertyName)`
 - Usage
 - * get Properties from somewhere.
- *getStatus*
public `ResultSet getStatus()`
 - Usage
 - * Get status information.
 - Returns - `ResultSet` containing status information for this producer
- *getTimeout*
public `TimeInterval getTimeout()`
 - Usage
 - * get time interval after which local buffer is transmitted even if not full
- *getValidate*
public boolean `getValidate()`
 - Usage
 - * carry out local validation of the row(s) inserted as far as possible. Primarily this means checking that the columns match. (not implemented yet)
- *insert*
public void `insert(java.lang.String row)`
 - Usage
 - * If the timestamp field is null, the `BaseProducer` will generate a time stamp for you (not implemented yet).
 - Parameters
 - * `row` - as an SQL insert statement
- *setLocalBufferSize*
public void `setLocalBufferSize(int localBufferSize)`
 - Usage
 - * sets the producer's local buffer size
- *setTimeout*
public void `setTimeout(org.edg.info.TimeInterval timeout)`
 - Usage
 - * set time interval after which local buffer is transmitted even if not full
- *setValidate*
public void `setValidate(boolean validate)`
 - Usage
 - * set value of validate. (not implemented yet)
- *transmit*
protected void `transmit(java.lang.String row)`

B.1.5 CLASS NoSuchColumnNameException

Thrown by Schema API if translateColumnNames is called with a column Name that does not exist in the Schema for the specified table

DECLARATION

```
public class NoSuchColumnNameException
extends java.lang.Exception
```

CONSTRUCTORS

- *NoSuchColumnNameException*
`public NoSuchColumnNameException()`
 - Usage
 - * The exception without description
- *NoSuchColumnNameException*
`public NoSuchColumnNameException(java.lang.String s)`
 - Usage
 - * The exception with description

METHODS INHERITED FROM CLASS java.lang.Exception

METHODS INHERITED FROM CLASS java.lang.Throwable

- *fillInStackTrace*
`public native Throwable fillInStackTrace()`
- *getLocalizedMessage*
`public String getLocalizedMessage()`
- *getMessage*
`public String getMessage()`
- *printStackTrace*
`public void printStackTrace()`
- *printStackTrace*
`public void printStackTrace(java.io.PrintStream)`
- *printStackTrace*
`public void printStackTrace(java.io.PrintWriter)`
- *toString*
`public String toString()`

B.1.6 CLASS NoSuchProducerException

Thrown by Registry API if `getProducerConnections` or `getProducerInfo` is called with invalid parameters or an error occurs that results in an invalid return of the methods

DECLARATION

```
public class NoSuchProducerException
extends java.lang.Exception
```

CONSTRUCTORS

- *NoSuchProducerException*
`public NoSuchProducerException()`
 - Usage
 - * The exception without description
- *NoSuchProducerException*
`public NoSuchProducerException(java.lang.String s)`
 - Usage
 - * The exception with description

METHODS INHERITED FROM CLASS `java.lang.Exception`

METHODS INHERITED FROM CLASS `java.lang.Throwable`

- *fillInStackTrace*
`public native Throwable fillInStackTrace()`
- *getLocalizedMessage*
`public String getLocalizedMessage()`
- *getMessage*
`public String getMessage()`
- *printStackTrace*
`public void printStackTrace()`
- *printStackTrace*
`public void printStackTrace(java.io.PrintStream)`
- *printStackTrace*
`public void printStackTrace(java.io.PrintWriter)`
- *toString*
`public String toString()`

B.1.7 CLASS NoSuchTableNameException

Thrown by Schema API if translateTableName is called with a table that does not exist in the Schema or if getTableInfo is called with an invalid tableId

DECLARATION

```
public class NoSuchTableNameException
extends java.lang.Exception
```

CONSTRUCTORS

- *NoSuchTableNameException*
`public NoSuchTableNameException()`
 - Usage
 - * The exception without description
- *NoSuchTableNameException*
`public NoSuchTableNameException(java.lang.String s)`
 - Usage
 - * The exception with description

METHODS INHERITED FROM CLASS java.lang.Exception

METHODS INHERITED FROM CLASS java.lang.Throwable

- *fillInStackTrace*
`public native Throwable fillInStackTrace()`
- *getLocalizedMessage*
`public String getLocalizedMessage()`
- *getMessage*
`public String getMessage()`
- *printStackTrace*
`public void printStackTrace()`
- *printStackTrace*
`public void printStackTrace(java.io.PrintStream)`
- *printStackTrace*
`public void printStackTrace(java.io.PrintWriter)`
- *toString*
`public String toString()`

B.1.8 CLASS ProducerConnection

Identifies a connection to a producer servlet by a producer

DECLARATION

```
public class ProducerConnection
extends java.lang.Object
```

CONSTRUCTORS

- *ProducerConnection*
public **ProducerConnection**(java.lang.String producerServlet, int connectionId)

- Usage

- * Construct a **ProducerConnection** using a string representing the producer URL and an integer representing the producer's unique connection identifier. @param producerServlet1 the URL of the producer servlet. @param connectionId1 identifies the specific producer for the desired connection.

METHODS

- *getConnectionId*
public int **getConnectionId**()

- Usage

- * get producer connection identifier.

- *getProducerServlet*
public String **getProducerServlet**()

- Usage

- * get producer servlet URL as a String.

- *toString*
public String **toString**()

B.1.9 CLASS ProducerInfo

Producer Information

DECLARATION

```
public class ProducerInfo
extends java.lang.Object
```

CONSTRUCTORS

- *ProducerInfo*
public **ProducerInfo**(java.lang.String tableName, java.lang.String fixedColumns,
int flags, java.lang.String tableDesc)
 - **Usage**
 - * Producer Information.
 - **Parameters**
 - * **tableName** - is the name of the SQL table
 - * **fixedColumns** - identifies the columns which are fixed and their values
 - * **flags** - is a set of boolean flags encoded in an integer
 - * **tableDesc** - is the table description. This is the same as the SQL for CREATE TABLE

METHODS

- *getFixedColumns*
public String getFixedColumns()
- *getFlags*
public int getFlags()
 - **Usage**
 - * get flags
 - **Parameters**
 - * **flags** - is a set of boolean flags encoded in an integer
 - **Returns** - flags
- *getTableDesc*
public String getTableDesc()
- *getTableName*
public String getTableName()

B.1.10 CLASS PropertyGetter

Obtains properties from a set of locations. Currently the only location considered is \$HOME

DECLARATION

```
public class PropertyGetter
extends java.lang.Object
```

CONSTRUCTORS

- *PropertyGetter*
public **PropertyGetter**(java.lang.String name)

METHODS

- *getProperties*
public Properties **getProperties**()

B.1.11 CLASS QueryException

Thrown by Consumer API if the execute method or pop method fail

DECLARATION

```
public class QueryException
extends java.lang.Exception
```

CONSTRUCTORS

- *QueryException*
public **QueryException**()
 - **Usage**
* The exception without description
- *QueryException*
public **QueryException**(java.lang.String s)
 - **Usage**
* The exception with description

METHODS INHERITED FROM CLASS java.lang.Exception

METHODS INHERITED FROM CLASS `java.lang.Throwable`

- *fillInStackTrace*
`public native Throwable fillInStackTrace()`
- *getLocalizedMessage*
`public String getLocalizedMessage()`
- *getMessage*
`public String getMessage()`
- *printStackTrace*
`public void printStackTrace()`
- *printStackTrace*
`public void printStackTrace(java.io.PrintStream)`
- *printStackTrace*
`public void printStackTrace(java.io.PrintWriter)`
- *toString*
`public String toString()`

B.1.12 CLASS `RegistrationException`

Thrown by Registry API if the register method fails

DECLARATION

```
public class RegistrationException
extends java.lang.Exception
```

CONSTRUCTORS

- *RegistrationException*
`public RegistrationException()`
 - Usage
 - * The exception without description
- *RegistrationException*
`public RegistrationException(java.lang.String s)`
 - Usage
 - * The exception with description

METHODS INHERITED FROM CLASS `java.lang.Exception`

METHODS INHERITED FROM CLASS `java.lang.Throwable`

- *fillInStackTrace*
`public native Throwable fillInStackTrace()`
- *getLocalizedMessage*
`public String getLocalizedMessage()`
- *getMessage*
`public String getMessage()`
- *printStackTrace*
`public void printStackTrace()`
- *printStackTrace*
`public void printStackTrace(java.io.PrintStream)`
- *printStackTrace*
`public void printStackTrace(java.io.PrintWriter)`
- *toString*
`public String toString()`

B.1.13 CLASS Registry

Registry of producers. Currently registry has no state, but when we address multiples VOs this is expected to change.

DECLARATION

```
public class Registry
extends java.lang.Object
```

CONSTRUCTORS

- *Registry*
`public Registry(java.lang.String location)`
 - Usage
 - * “Creates a registry object that is used to register producers.
 - Parameters
 - * `location` - is the URL of the registry servlet that deals with request of this registry object.

METHODS

- *getProducerConnections*
`public Vector getProducerConnections(java.lang.String tableName, java.lang.String fixedColumns, int flags)`
 - Usage

- * Discover producers from registry based on the table name, the value of fixed columns and flags
 - **Parameters**
 - * `tableName` - is the name of the SQL table
 - * `fixedColumns` - identifies the columns which are fixed and their values a colon separated list of "=" separated name value pairs can be null pointer
 - * `flags` - is a set of boolean flags encoded in an integer
 - **Returns** - Vector of `ProducerConnections`

 - *getProducerInfo*
`public String getProducerInfo(org.edg.info.ProducerConnection pc)`
 - **Usage**
 - * Look up `ProducerInfo` of a producer identified by a `ProducerConnections` `pc`. It will return null if the `ProducerConnection` is not known.
 - **Parameters**
 - * `pc` - `ProducerConnection`
 - **Returns** - `ProducerInfo` CURRENTLY STILL A STRING !!!

 - *getRegistryServletLocation*
`public String getRegistryServletLocation()`
 - **Usage**
 - * Getter for the location of the Registry Servlet.

 - *getStatus*
`public String getStatus()`
 - **Usage**
 - * Get status information of the registry CURRENTLY NOT IMPLEMENTED.
 - **Returns** - XML formatted status information as a String

 - *register*
`public String register(java.lang.String tableName, java.lang.String fixedColumns, int flags, org.edg.info.ProducerConnection pc)`
 - **Usage**
 - * Register a producer of a table. For this operation to succeed a table with name `tableName` must already exist in the schema
 - **Parameters**
 - * `tableName` - is the name of the SQL table
 - * `fixedColumns` - identifies the columns which are fixed and their values a colon separated list of "=" separated name value pairs
 - * `flags` - is a set of boolean flags encoded in an integer
 - * `pc` - a `ProducerConnection`
 - **Returns** - a complete `tableDesc` from the Registry if the table is already registered CURRENTLY STILL A STRING
 - **Exceptions**
 - * `java.lang.Exception` - - whatever is needed

 - *register*
`public String register(java.lang.String tableName, java.lang.String fixedColumns, int flags, java.lang.String tableDesc, org.edg.info.ProducerConnection pc)`
 - **Usage**
-

- * Register a producer of a table. CURRENTLY NOT YET IMPLEMENTED If the table is already known but inconsistent with tableDesc an exception will be thrown. tableName must already be known within the schema
- **Parameters**
 - * `tableName` - is the name of the SQL table
 - * `fixedColumns` - identifies the columns which are fixed and their values a colon separated list of "=" separated name value pairs
 - * `flags` - is a set of boolean flags encoded in an integer
 - * `tableDesc` - an SQL CREATE TABLE string
 - * `pc` - a `ProducerConnection`
- **Returns** - a complete tableDesc
- **Exceptions**
 - * `java.lang.Exception` - - whatever is needed

B.1.14 CLASS ResultSet

R-GMA implementation of a `ResultSet`. The Class implements a subset of the methods of the `java.sql.ResultSet` class.

DECLARATION

```
public class ResultSet
extends java.lang.Object
```

METHODS

- *getInt*
public int getInt(int columnNumber)
- *getInt*
public int getInt(java.lang.String columnName)
- *getMetaData*
public ResultSetMetaData getMetaData()
- *getString*
public String getString(int columnNumber)
- *getString*
public String getString(java.lang.String columnName)
- *next*
public boolean next()
- *toString*
public String toString()

- **Usage**

- * can be called at any time to print the `ResultSet` without disturbing the behaviour of `next()`

B.1.15 CLASS ResultSetMetaData

R-GMA implementation of ResultSetMetaData. The Class implements a subset of the methods of the java.sql.ResultSetMetaData class.

DECLARATION

```
public class ResultSetMetaData
extends java.lang.Object
```

METHODS

- *getColumnCount*
public int getColumnCount()
- *columnName*
public String columnName(int columnNumber)

B.1.16 CLASS Schema

Schema of Tables. This object is used to access a Schema Servlet that holds the schemas for tables that can be registered with a registry. A Registry is closely associated with a Schema as the former uses the latter. Having a Registry and Schema separates the registration and description of tables.

DECLARATION

```
public class Schema
extends java.lang.Object
```

CONSTRUCTORS

- *Schema*
public Schema(java.lang.String location)
 - Usage
 - * Creates a schema object that is used to hold descriptions of tables.
 - Parameters
 - * location - is the URL of the schema servlet that deals with requests for this schema.

METHODS

- *getSchemaServletLocation*
`public String getSchemaServletLocation()`
- *getStatus*
`public String getStatus()`
 - **Usage**
 - * Get status information.
 - **Returns** - XML formatted status information
- *getTableInfo*
`public ResultSet getTableInfo(java.lang.String tableId)`
 - **Usage**
 - * get information about a table
 - **Parameters**
 - * **tableId** - the Identifier of the table
- *translateColumnNames*
`public String translateColumnNames(java.lang.String tableId, java.lang.String [] fixedColumns)`
 - **Usage**
 - * Translate the name/value pairs of fixed columns into **columnId**/value pairs.
 - **Parameters**
 - * **a** - **tableId**
 - * **an** - array of Strings each an "=" separated name value pair
 - **Returns** - an array of Strings each an "=" separated **columnId** value pair
- *translateTableName*
`public String translateTableName(java.lang.String tableName)`
 - **Usage**
 - * Translates a **tableName** into a **tableId**.
 - **Parameters**
 - * **a** - **tableName**
 - **Returns** - **tableId**

B.1.17 CLASS `ServletConnectionException`

Thrown by Schema API if any of the methods of the `ServletConnection` fails

DECLARATION

```
public class ServletConnectionException  
extends java.lang.Exception
```

CONSTRUCTORS

- *ServletConnectionException*
`public ServletConnectionException()`
 - Usage
 - * The exception without description
- *ServletConnectionException*
`public ServletConnectionException(java.lang.String s)`
 - Usage
 - * The exception with description

METHODS INHERITED FROM CLASS `java.lang.Exception`

METHODS INHERITED FROM CLASS `java.lang.Throwable`

- *fillInStackTrace*
`public native Throwable fillInStackTrace()`
- *getLocalizedMessage*
`public String getLocalizedMessage()`
- *getMessage*
`public String getMessage()`
- *printStackTrace*
`public void printStackTrace()`
- *printStackTrace*
`public void printStackTrace(java.io.PrintStream)`
- *printStackTrace*
`public void printStackTrace(java.io.PrintWriter)`
- *toString*
`public String toString()`

B.1.18 CLASS `SubscriptionException`

Thrown by Registry API if the register method fails

DECLARATION

```
public class SubscriptionException
extends java.lang.Exception
```

CONSTRUCTORS

- *SubscriptionException*
`public SubscriptionException()`
 - Usage
 - * The exception without description
- *SubscriptionException*
`public SubscriptionException(java.lang.String s)`
 - Usage
 - * The exception with description

METHODS INHERITED FROM CLASS `java.lang.Exception`

METHODS INHERITED FROM CLASS `java.lang.Throwable`

- *fillInStackTrace*
`public native Throwable fillInStackTrace()`
- *getLocalizedMessage*
`public String getLocalizedMessage()`
- *getMessage*
`public String getMessage()`
- *printStackTrace*
`public void printStackTrace()`
- *printStackTrace*
`public void printStackTrace(java.io.PrintStream)`
- *printStackTrace*
`public void printStackTrace(java.io.PrintWriter)`
- *toString*
`public String toString()`

B.1.19 CLASS XMLConverter

Parses XML and constructs ResultSet

DECLARATION

```
public class XMLConverter
```

CONSTRUCTORS

- *XMLConverter*
public **XMLConverter**()
 - **Usage**
 - * Constructor

METHODS

- *convertXMLResponse*
public **ResultSet** **convertXMLResponse**(java.lang.String xml)
 - **Usage**
 - * generate resultSet from xml string. This returns null if the input string is empty
- *error*
public void **error**(org.edg.info.SAXParseException ex)
 - **Usage**
 - * Error.
- *fatalError*
public void **fatalError**(org.edg.info.SAXParseException ex)
 - **Usage**
 - * Fatal error.
- *warning*
public void **warning**(org.edg.info.SAXParseException ex)
 - **Usage**
 - * Warning.

Index

Archiver, 31

CircularBufferProducer, 14
Consumer, 14, 30

DataBaseProducer, 14

Global Grid Forum, 11, 17
GMA, *see* Grid Monitoring Architecture
Grid Monitoring Architecture, 11, 17
Grid Notification Framework, 27
GRM, 33
GRRP, 27

IMS, *see* Information and Monitoring System
Information and Monitoring System, 10

JINI, 12

MDS, 37
Mediator, 16

NetLogger, 33, 37

PluggableProducer, 14
Producer, 14
PROVE, 33
Publisher, 14, 30

RDBMS, *see* Relational Data Base Management
System
Registry, 11, 15, 31
relational algebra, 13
Relational Data Base Management System, 12
relational model, 12
requirements, 21

Schema, 15, 31
security, 28
SQL, 14, 30
SWIG, 30

tuple, 13

Use Case, 18

Virtual Organisation, 10
VO, *see* Virtual Organisation