R-GMA
Relational Information Monitoring and Management System
User Guide

Version 2.1.1

The WP3 relational team

October 15, 2001

Contents

1 Introduction **2**

1.1 Background 2

1.2 APIs 3

2 Installation **5**

2.1 Java 5

2.2 MySQL 5

2.3 Tomcat 5

2.4 Ant 5

2.5 libwww 5

2.6 R-GMA 6

2.7 Configuring R-GMA for a virtual organization 6

3 Basic R-GMA functionality **8**

3.1 Producing and consuming information 8

3.2 Joint consumer/producer 9

4 Available Sensors **16**

4.1 MDS Producer Sensor 16

A Java API **18**

A.1 Classes 20

B C++ API **42**

B.1 Consumer Class Reference 42

B.2 Producer Class Reference 44

B.3 ProducerConnection Class Reference 47

B.4 Exception Class Reference 48

Chapter 1

Introduction

This is the second, very hurried release of this document to accompany a second release of the product. Please make use of the bugs page linked from http://hepunx.rl.ac.uk/grid/wp3 to complain about it's myriad deficiencies and we will do our best to put them right.

R-GMA is a monitoring and information management service for distributed resources. It exposes a relational model with SQL support to provide static as well as dynamic information about Grid resources. This guide presents instructions on how to install and use R-GMA, it does not go into details of the implementation, but concentrates on the interface that R-GMA presents to the outside world: the application program interface (API) for producers and consumers.

1.1 Background

The R-GMA architecture is based on that of the Grid Monitoring Architecture (GMA) [2] of the Global Grid Forum (GGF) [1]. The GMA as shown in Figure 1.1 consists of three components: consumers, producers and a directory service, which we refer to as a registry as it avoids any implied structure.

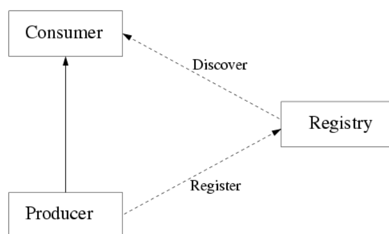


Figure 1.1: Grid Monitoring Architecture

In the GMA producers of information register themselves with the registry when they are instantiated. The registry which may be distributed, describes the type and structure of information the producers want to make available to the Grid. Potential consumers of information can query the registry to find out what type of information is available and locate producers that provide such information. Once a consumer has this information it can contact the producer directly to obtain the relevant data. Note that the architecture also supports joint consumer/producer components as illustrated by the component at the centre of Figure 1.2. This could gather data from several producers and make digested information available to other consumers.

A common API is used to access data, whether it is *fresh* monitoring data or data from an archive. This API allows the registry and schema to be hidden.

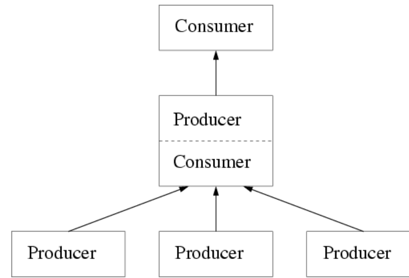


Figure 1.2: Joint Consumer Producer

1.2 APIs

Here we describe the Java API; the C++ API is very similar. The API objects are rather small, most of the real work is done by servlets running on servlet engines. This allows the API objects to be easily replaced by objects in a different programming language.

Two principal services are offered to the other middleware and to end users: a Producer service(1.2.1) to producers of information (sensors) and a Consumer service(1.2.2) to consumers of information. If your program has data it wishes to publish it should instantiate a Producer. The application might be a user program or it could be a piece of middleware code finding the CPU load. A Consumer is used by a program to locate and to request information from a Producer.

Almost all (if not all) Grid services will also act as Producers (of information) and will be able to register the various kinds of information they have on offer. For example a ComputingElement will register as a producer of information on such things as queue size, available batch capacity and suitability for interactive work. Information Consumers will be able to locate suitable Producers to answer their questions.

1.2.1 Producer

A producer is instantiated with the description of the information it has to offer. The first argument is the table name. The second is a character string describing the columns which are fixed and their values **** This is not yet implemented ****. This is a comma separated list of columnName = value pairs. The third argument is an integer for holding a number of flags **** This is not yet implemented ****. The last argument to the constructor is a String describing the tables to be published. This string takes the form of an SQL `CREATE TABLE` statement. This last argument is optional - if the table is already known within the schema.

To publish data, the insert method is invoked. This takes a normal SQL `INSERT` statement. If the timestamp is not set then the producer will derive a time stamp from the system time **** This is not yet implemented ****. The localBufferSize property controls how many tuples of data are cached by the Producer before a transfer to the producer servlet is made. The timeout property **** This is not yet implemented ****allows the person publishing information via a producer to request that tuples are transmitted after a certain time even if the buffer is not full. The remoteBufferSize controls how many rows will be held remotely waiting for consumers. If the buffer is too small or the consumer too slow records will be lost as it makes use of a circular buffer of finite size.

1.2.2 Consumer

The consumer is the means of accessing information made available by various producers using SQL queries against a set of supported tables. The query is analysed making use of the registry to find suitable producers of information. In the general case queries will be sent to different producers and the Consumer servlet acting on behalf of the consumer will process the results **** This is not yet implemented ****.

A consumer handles a single query, expressed as an SQL SELECT statement and supports either a single query/response or it registers for data to be streamed to it from the producer. The consumer talks to the consumer servlet (which is where information is really streamed to) which should be close (in network terms) to the process requesting the information. This servlet locates the best source(s) of information, **** this only works at the moment if there is a single producer of that table and if the SQL SELECT statement mentions only one table **** makes a temporary connection to a producer servlet for a query to be executed once in order to return the last tuple from the producer servlet buffer. This servlet can also make a permanent connection to allow data from the producer servlet buffer to be streamed to its own buffer.

The consumer is created with a String representing the SQL Query you wish to execute. The second constructor allows the specification of the ProducerConnection of the producer to connect to. You can then either execute a query which returns one result or you can set the `bufferSize` to be non-zero which allows the information to be streamed to the consumer servlet. The consumer can then use `count` to see how many records are available and `pop` to get a record.

In order to stop the streaming, the `bufferSize` must be reset to zero.

1.2.3 Archiver

**** This is not yet implemented ****

Chapter 2

Installation

In each case specific versions are mentioned here. Other versions may work but we don't know.

Some of the installation kits may also be picked up from: http://hepunx.rl.ac.uk/grid/wp3/kits. These compenents are more conveniently obtainable via the WP6 web page http://marianne.in2p3.fr/datagrid/testbed1/repositories/pkg-repository.html as RPMS.

2.1 Java

You should use jdk1.3 from Sun or IBM. You need to add java to your PATH variable, otherwise things don't run.

2.2 MySQL

For Linux RedHat 6.1 you need MySQL 3.23.32 and for RedHat 7.1 you need MySQL 3.23.40. Obtain the RPM from http://www.mysql.org

2.3 Tomcat

You will need version 4.0 (jakarta-tomcat-4.0.tar.gz) from http://jakarta.apache.org/tomcat/

2.4 Ant

You will need, to run the demo, version 1.3 (jakarta-ant-1.3-bin.tar.gz) from http://jakarta.apache.org/ant/

2.5 libwww

You will need the libwww libraries to build the C++ API for Redhat6.2 these are w3c-libwww-5.2.8-4.i386.rpm and w3c-libwww-devel-5.2.8-4.i386.rpm and for Redhat7.1 these are w3c-libwww-5.2.8-6.i386.rpm and w3c-libwww-devel-5.2.8-6.i386.rpm.

2.6 R-GMA

Choose a directory in which you would like to deploy R-GMA. Here we denote it as GRIDHOME. In GRIDHOME untar gma.tar.gz . It creates a toplevel directory gma. The next step is to edit the file `gma/release-setup.sh` that sets a number of environment variables. The variable `deploy` holds the directory in which Tomcat is installed. The variable `home` should be set to GRIDHOME. Once release-setup.sh is customized it needs to be executed before one can run the demo. Move the`log4j.props` file into the GRIDHOME directory and customize it. There is one other file that needs to be moved to the right place expected by our software: Create a directory `/opt/edg/info` and move the file `XMLResponse.xsd` located in `gma/API/etc` into it.

The next step is to change to Tomcat's home directory. There exists a subdirectory called `webapps`. Stay in the directory that contains webapps and untar the webapps.tar file. This will put all the servlets in the right place so Tomcat can find them. You are now ready to run the demo on a single machine. Go back to the `GRIDHOME/gma/demo` and run the run script with either `SimpleDemo` or `ClusterLoad` as an argument. The `README` explains briefly what is happening. Have a look at the source to understand in more detail what the different Sensors,Consumers and combined Consumer/Producer units do.

2.7 Configuring R-GMA for a virtual organization

To set up R-GMA for a virtual organization one has to run one RegistryServlet and one SchemaServlet. These make use of a database to store information about producers (RegistryServlet) and tables (SchemaServlet). To be able to produce information one has to run at least one ProducerServlet, however many producers can use the same ProducerServlet to publish data for them. In the same way one has to run at least one ConsumerServlet to be able to consume data that has been published by a producer. For every servlet the web.xml file describing the web-application has to be configured and for Consumer, Producer, DBProduer and Archiver a properties file has to be configured. Each API class needs to know the location of the respective servlet which services it. The properties files are located in `$RGMA_HOME` and currently have to be copied into the home directory of the user running the application code that uses the API class in order to be found. Each servlet has a number of init parameters that are set at the beginning of the servlet life cycle and are now discussed in turn.

2.7.1 SchemaServlet

`schemaDatabaseLocation` is a JDBC URL for the location of the Schema database, see the documentation of your database for more information. The default setting is for a mysql database running on localhost. It probably makes sense to run the database on the same host as the SchemaServlet, but it is not mandatory.
`schemaDatabaseUserName` is the database user name of the schema database. The default is schema.
`schemaDatabasePassword` is the clear text password for the above user. The default is info.

2.7.2 RegistryServlet

`registryDatabaseLocation` is a JDBC URL for the location of the Registry database, see the documentation of your database for more information. The default setting is for a mysql database running on localhost. It probably makes sense to run the database on the same host as the RegistryServlet, but it is not mandatory.
`registryDatabaseUserName` is the database user name of the registry database. The default is registry.
`registryDatabasePassword` is the clear text password for the above user. The default is info.
`schemaServletLocation` is the URL of the SchemaServlet. The default is to run the SchemaServlet on the same host as the RegistryServlet, in which case the same database can hold both the registry and schema database.

2.7.3 ProducerServlet

`registryServletLocation` is the URL of the RegistryServlet. The ProducerServlet has to be able to contact the Registry to register Producers.

2.7.4 DBProducerServlet

`registryServletLocation` is the URL of the RegistryServlet. The DBProducerServlet has to be able to contact the Registry to register DBProducers.

2.7.5 ConsumerServlet

`registryServletLocation` is the URL of the RegistryServlet. The ConsumerServlet has to be able to contact the Registry to find out about Producers.

2.7.6 Tools

To populate the Schema database with a set of known tables and to bring the registry database into a clean state with no registered producers the build file in `$RGMA_HOME/tools/dbases` has to be run. Since soft-state registration is currently not yet implemented so the Registry can get into an inconsistent state. The administration of the registry database will be moved into the RegistryServlet in a future release.

Chapter 3

Basic R-GMA functionality

In this chapter some JAVA code examples of Producers/Consumers are presented. The code can be seen and run from the demo directory. More complicated examples can easily be constructed and adapted to particular situations by following the techniques shown here.

3.1 Producing and consuming information

The examples in SimpleDemo use a CPU load monitoring program for producing information and a variety of consumers (single/query response and streaming). The main classes in these examples are listed below

- **ProdLoadSensor** takes the IP address of the machine (localhost for example) and creates a producer with a table name 'cpuLoad' and a fixed column 'ipaddress'. It then loops a fixed number of times, reads the load information from file "/proc/stat" and publishes it using the producer's INSERT method.
- **TestConsumer** takes an SQL SELECT statement as it's first argument and a producer id as it's second argument. The producer servlet is located on the local machine with a URL of 'http://localhost:8080/ProducerServlet' hard coded in the class. TestConsumer then creates a consumer passing it the SELECT statement and a producerConnection object. Since the producer details are provided for the consumer constructor, there is no need to make use of the registry in this case. TestConsumer then enters a loop where the SELECT query is executed and a ResultSet is retuned and printed to the standard output stream.
- **TestConsumer2** takes an SQL SELECT statement as it's sole argument. Since no information is given about the producer, the registry is contacted in order to provide a vector of all producers capable of providing the required information. In this version of R-GMA, the usual practice is to create a table entry manually in the data base for this to work. The program then enters a loop where the SELECT query is executed and a ResultSet is returned and printed to the standard output stream.
- **TestConsumerStream** takes an SQL SELECT statement as it's first argument and a producer id as it's second argument. The producer servlet is located on the local machine with a URL of 'http://localhost:8080/ProducerServlet' is hard coded in the class. TestConsumer then creates a consumer passing it the SELECT statement and a producerConnection object. The consumer servlet buffer size is then set to 10, in order to start the streaming of the data from the producer via the producer servlet. The program then enter a loop where count is executed to find out if any data is available in the consumer servlet buffer, in which case it is poped (removed) and returned as a ResultSet which is then written to the standard output stream. After a certain number of trying to pop data, the while loop is exited and the streaming from the producer servlet to the consumer servlet is stopped by resetting the buffersize to zero.

Assuming you have installed mySQL, TOMCAT (in directory $TOMCAT_HOME), R-GMA (under directory $HOME/wp3/gma) and compiled the above java code successfully, you can use the following script in order to instantiate a producer of CPU load information. These scripts can easily be adapted if your installation is different from the above assumption. In order for the examples to work with this version of R-GMA you should stop TOMCAT, rebuild the registry databases and then restart TOMCAT before running the scripts as indicated below:

```
$TOMCAT_HOME/bin/tomcat.sh stop
(cd $HOME/wp3/gma/dbases && ./build)
$TOMCAT_HOME/bin/tomcat.sh start
sleep 5
./etc/run ProdLoadSensor  sensor1 &
```

This should create a producer with and id of 1 with ipaddress sensor1 (this should really be an ip address but we are using it here as a placeholder since everything runs on localhost). The sleep statement is used to allow TOMCAT to load before any attempt to contact the servlets is made. Note that run in the etc subdirectoru of SimpleDemo, is another script which is shown in figure 3.7. A consumer of CPU load information can then be started either by giving the producer details (id) to the constructor(better in a separate window)

```
./run TestConsumer 'select * from cpuLoad' 1
```

or by simply specifying a SELECT statement and letting the Registry find a suitable producer for you as in the following

```
./run TestConsumer2 'select * from cpuLoad'
```

A example streaming consumer is instantiated with the following command

```
./run TestConsumerStream 'select * from cpuLoad' 1
```

3.2 Joint consumer/producer

Creating a joint consumer/producer is also fairly simple. First instantiate a second producer of CPU load information (see above) preferably in a seperate window.

```
./run ProdLoadSensor sensor2 &
```

This should create a producer with and id of 2. Then in another window create a producer/consumer which takes data from both producers and presents the merged data and new information from a new producer.

```
./run ConsumerProducer &
```

This should create a producer with and id of 3. Then in another widow instantiate a consumer specifying an id of 3 for the producer. This should receive the merged data from the consumer/producer.

```
./run
TestConsumer 'select * from cpuLoad' 3 &
```

Have a look at how the ipaddress field changed between sensor1 and sensor2.

```
package org.edg.info;
import java.net.*;
import java.io.*;
import java.util.*;

public class ProdLoadSensor {

    static String tableName = "cpuLoad";
        static long SERVE_FREQUENCY = 2000; // milliseconds
    int abs;
    Producer loadInfo;
    static String ipaddress;

    public static void main(String args[]) {

        ipaddress = args[0];
        // new server for threading
        ProdLoadSensor loadSensor = null;
        loadSensor = new ProdLoadSensor();

        loadSensor.begin();
    }

    ProdLoadSensor() {
        System.out.println("L:New load sensor started");

        // create producer
        try {
            loadInfo = new Producer("cpuLoad","ipaddress="+ipaddress,0);
        } catch (ServletConnectionException e) {
            System.out.println("Could not create Producer" + e.getMessage());
            System.exit(1);
        } catch (RegistrationException e) {
            System.out.println("Could not create Producer" + e.getMessage());
            System.exit(1);
        }

    }

    public void begin() {
        String[] load = new String[5];
        while (true) {
            if ((abs++)>=10000) {
                break;
            }

            // get the load information
            load = fetchLinuxLoad();

            // create insert statement
            String publishme;
            publishme =
                "INSERT INTO cpuLoad " +
                "(ipaddress,one,five,fifteen,a,b,timestamp) " +
                "VALUES ('"+ ipaddress                 + "','"
```

Figure 3.1: ProdLoadSensor.java

```
            + Float.parseFloat(load[0]) + "','"
            + Float.parseFloat(load[1]) + "','"
            + Float.parseFloat(load[2]) + "','"
            + load[3] + "','"
            + load[4] + "','"
            + String.valueOf(System.currentTimeMillis()) +  "')";
        System.out.println("L:" + publishme);

        // publish the information
        loadInfo.insert(publishme);
        try {
            Thread.sleep(SERVE_FREQUENCY);
        } catch (InterruptedException ignored) {}
    }
}

String[] fetchLinuxLoad() {
    // use Steve's one for the mo..
    // Ian's one is a bit complex..
    String line;
    try {
        BufferedReader proc = new BufferedReader
            (new FileReader("/proc/loadavg"));
        line = proc.readLine();
        //              System.out.println("L:from /proc:" + line);
        proc.close();
    } catch (IOException zero ) { String[] nothing={"0.0","0.0","0.0","0/0","0/0"};
                                return nothing;} // machine dead?

    StringTokenizer toks = new StringTokenizer(line, " ");
    String[] something = new String[5];
    for (int i=0;i<5;i++)
        {
                something[i] = toks.nextToken();
                //                      System.out.println("L: tokens:" + something[i]);
        }
    return something;
}
}
```

Figure 3.2: ProdLoadSensor.java - part 2

```
package org.edg.info;
/**
   tests the functionality of the Consumer class
*/
public class TestConsumer
{
    public static void main (String[] args)
    {
        Consumer myConsumer = null;
        String selectst = args[0];
        String producerServlet="http://localhost:8080/R-GMA/ProducerServlet";
        int producerId= Integer.parseInt(args[1]);
        ProducerConnection producerConnection =
            new ProducerConnection(producerServlet, producerId);
        //create a Consumer
        try {
            myConsumer = new Consumer(selectst,producerConnection);
        } catch (ServletConnectionException e) {
            System.out.println("Could not create Consumer" + e.getMessage());
            System.exit(1);
        } catch (SubscriptionException e) {
            System.out.println("Could not create Consumer" + e.getMessage());
            System.exit(1);
        }
        //execute the query
        while(true){
            ResultSet  rs = myConsumer.execute();
            System.out.println(rs.toString());
            try {
                Thread.sleep(1000);
            } catch (InterruptedException ignored) {}
        }
    }
}
```

Figure 3.3: TestConsumer.java

```
package org.edg.info;
/**
   tests the functionality of the Consumer class
*/
public class TestConsumer2
{
    public static void main (String[] args)
    {
        Consumer myConsumer = null;
        String selectst = args[0];

        //create a Consumer
        try {
            myConsumer = new Consumer(selectst);
        } catch (ServletConnectionException e) {
            System.out.println("Could not create Consumer" + e.getMessage());
            System.exit(1);
        } catch (SubscriptionException e) {
            System.out.println("Could not create Consumer" + e.getMessage());
            System.exit(1);
        }
        //execute the query
        while(true){
            ResultSet  rs = myConsumer.execute();
            System.out.println(rs.toString());
            try {
                Thread.sleep(1000);
            } catch (InterruptedException ignored) {}
        }
    }
}
```

Figure 3.4: TestConsumer2.java

```
package org.edg.info;

public class TestConsumerStream
{
    public static void main (String[] args)
    {
        Consumer myConsumer = null;
        String selectst = args[0];
        String producerServlet="http://localhost:8080/R-GMA/ProducerServlet";
        int producerId= Integer.parseInt(args[1]);
        ProducerConnection producerConnection =
            new ProducerConnection(producerServlet, producerId);
        //create a Consumer
        try {
            myConsumer = new Consumer(selectst,producerConnection);

            //setbufferSize
        myConsumer.setBufferSize(10);
        } catch (ServletConnectionException e) {
            System.out.println("Could not create Consumer" + e.getMessage());
            System.exit(1);
        } catch (SubscriptionException e) {
            System.out.println("Could not create Consumer" + e.getMessage());
            System.exit(1);
        } catch (QueryException e) {
            System.out.println("Could not create Consumer" + e.getMessage());
            System.exit(1);
        }
        try {
            for (int i=0; i<10000; i++) {
                if ( myConsumer.count() > 0) {
                    ResultSet  rs = myConsumer.pop();
                    System.out.println(rs.toString());
                }
            }
        }
        catch (Exception e) {
            System.out.println("Error :"+e);
        }
        try {
            //stop streaming
            myConsumer.setBufferSize(0);
        } catch (ServletConnectionException e) {
            System.out.println("Could not create Consumer" + e.getMessage());
            System.exit(1);
        } catch (QueryException e) {
            System.out.println("Could not create Consumer" + e.getMessage());
            System.exit(1);
        }
   }
}
```

Figure 3.5: TestConsumerStream.java

Figure 3.6: TestConsumerStream.java - part 2

```
#!/bin/sh
echo run $1 "$2" $3
java -cp $INFO_JARS/info.jar:$INFO_JARS/log4j.jar:classes:$INFO_JARS/xerces.jar  \
$TOMCAT_OPTS info.$1 "$2" $3
```

Figure 3.7: run

Chapter 4

Available Sensors

In this chapter the available sensors that have been implemented using the R-GMA approach are discussed, with the emphasis upon how the sensors are used.

4.1 MDS Producer Sensor

The purpose of the MDS Producer sensor is to publish all the information available from a Globus GRIS server into the R-GMA and to permit a consumer to access this information via the normal R-GMA approach.

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 Grid information stored in a hierarchical LDAP directory structure. Each piece of information is associated with an attribute, with the permitted attributes being defined and grouped by an LDAP schema or 'object class'. The context of the information is given by its position within the directory structure.

There are currently 6 schema defined in the Globus 1.1.3 release:

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

There is one table in the R-GMA corresponding to each of the schema. 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 MDS Producer is implemented in JAVA, in the class **MDSProducer**. The class is supplied with a properties file which points it to a particular GRIS server, and contains a list of schema to publish. The properties file, **MDSProducer.props**, consists of 5 properties in the standard JAVA key=value format with each property taking a new line:

- **mdsBindHost**
  The host upon which the GRIS server is running.
- **mdsBindPort**
  The integer TCP port number on which the MDS is running.
  If not specified the GRIS default of 2135 is used.

- **mdsBindDn**
  The base DN from which to start the LDAP search.

- **objectClasses**
  A comma separated list of the MDS schema (as given by the LDAP 'Objectclass' attribute) to publish from the GRIS.

- **tableNames**
  A comma separated list of the SQL table names in which to publish the information. The order must correspond to the order given in the objectClasses property. It is suggested, although not required, that the SQL table names are the same as the requested object classes.

The MDS Producer will then request all entries of each of the specified object class types, starting the search at the given base DN. The information from each of the entries found is then published, along with the DN of the entry, in the appropriate table.

The MDS Producer is likely to run nearby the GRIS server that it is polling, possibly upon the same machine, although this does not have to be the case since the GRIS servers are polled using the standard LDAP wire protocol. Currently there will be one MDS Producer for every GRIS server. It would be easy to implement a system to provide aggregate information about one or more sites. This would involve a simple Consumer-Producer model where the Consumer side subscribes to all the site MDS Producers and then publishes the aggregate information in some suitable format.

The MDSProducer class includes a main() method as an example of using the class, simply requiring that the supplied MDSProducer.props file be edited to point to the relevant GRIS server. The main method creates a single Producer, connects to the GRIS server and publishes the information once, before exiting. In general, a user of the class will wish publish the information more than once, using the methods described below:

Constructor: **MDSProducer()**
There is currently one constructor, taking no arguments, which reads the properties file and registers the MDS Producer with the ProducerServlet. This makes us of the Producer API.

A single poll: **void** gris**.pollGRIS()**
This method connects to the GRIS, polls for all the object classes specified in the properties file, and then disconnects from the GRIS. The connection is re-made every time since the class has no information about how long the GRIS will tolerate an idle open connection before timing out.

A Consumer of this information may issue an arbitrary SQL query upon the published tables, e.g.

```
SELECT contact AS 'Globus contact string' FROM GlobusServiceJobManager
    WHERE service=jobmanager-pbs AND osname=linux AND Dn LIKE '%dc=RAL%'

SELECT queue, service, freenodes/totalnodes AS 'Current utilisation' FROM GlobusQueue
    WHERE Dn='queue=default,service=jobmanager,hn=griddev.gla.ac.uk,dc=gla,dc=ukhep,o=grid'

SELECT Dn,queue from GlobusQueue
    WHERE totalnodes > 10 AND maxtotalmemory > 512
```

Currently joins between different object class tables are not implemented since the appropriate object used to make the join to an ancestor table is some subpart of the DN; these are currently simple strings, which are cannot easily be broken into subparts within a single SQL statement as required by the join, and are anyway inefficient to make joins upon. Solutions to allow efficient joining are being investigated for the next release, until then joins can be made directly by the client consumer using two queries.

Extra object classes can be added easily to the MDS Producer by registering the table name and column names (corresponding to the attributes of the objectClass schema) with the SchemaServlet, and by adding the objectClass and tableNames into the properties file.

Appendix A

Java API

Package Contents *Page*

Classes

Archiver 20

Archiver is able to consume, archive and produce data.

Consumer 21

Consumes event by event or a stream of events.

DBProducer 22

DBProducers is similar to a normal producer but uses an RDBMS as a buffer in the servlet.

InfoResultSet 24

...no description...

NoSuchColumnNameException 25

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

NoSuchProducerException 26

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

NoSuchTableNameException 27

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

Producer 28

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

ProducerConnection 30

Identifies a connection to a producer servlet by a producer

ProducerInfo 31

Producer Information

PropertyGetter 32

...no description...

QueryException 32

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

RegistrationException 33

Thrown by Registry API if the register method fails

Registry 34

Registry of producers.

ResultSet 36

...no description...

ResultSetMetaData 36

...no description...

Schema 37

Schema of Tables.

ServletConnectionException 38

Thrown by Schema API if any of the methods of the ServletConection fails

SubscriptionException 39

Thrown by Registry API if the register method fails

XMLConverter 40

...no description...

A.1 Classes

A.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. **It is not part of this release.**

A.1.1.1 Declaration

public class Archiver
extends java.lang.Object

A.1.1.2 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 RDMBS identification
    - `user` - JDBC user name
    - `pasword` - JDBC user password

A.1.1.3 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

A.1.2 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

A.1.2.1 Declaration

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

A.1.2.2 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

A.1.2.3 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.ServletConnectionException` -

- *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

A.1.3 Class DBProducer

DBProducers 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.

A.1.3.1 Declaration

public class DBProducer
extends org.edg.info.BaseProducer

A.1.3.2 Constructors

- *DBProducer*
  `public` **DBProducer**`(` `java.lang.String` **rdbms**, `java.lang.String` **user**, `java.lang.String` **password** `)`
  - **Usage**
    * Create a DBProducer and connect it to the specified RDBMS via JDBC.
  - **Parameters**
    * `rdbms` - JDBC RDMBS identification
    * `user` - JDBC user name
    * `pasword` - JDBC user password

A.1.3.3 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

A.1.3.4 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` **proper-tyName** `)`
  - **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`** `)`

A.1.4 Class **InfoResultSet**

A.1.4.1 Declaration

public class InfoResultSet
extends java.lang.Object

A.1.4.2 Constructors

- *InfoResultSet*
  `public` **`InfoResultSet`**`( java.lang.String []` **`columnHeaders`** `)`

A.1.4.3 Methods

- *getColumnData*
  `public String` **getColumnData**`( java.lang.String` **columnMetaData** `)`
- *getMetaData*
  `public Hashtable` **getMetaData**`( )`
- *getString*
  `public String` **getString**`( int` **columnNumber** `)`
- *next*
  `public boolean` **next**`( )`
- *toString*
  `public String` **toString**`( )`

A.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

A.1.5.1 Declaration

public class NoSuchColumnNameException
extends java.lang.Exception

A.1.5.2 Constructors

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

A.1.5.3 Methods inherited from class `java.lang.Exception`

A.1.5.4 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**`( )`

A.1.6 Class **NoSuchProducerException**

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

A.1.6.1 Declaration

public class NoSuchProducerException
extends java.lang.Exception

A.1.6.2 Constructors

- *NoSuchProducerException*
  `public` **NoSuchProducerException**`( )`
  - **Usage**
    * The exception without description

- *NoSuchProducerException*
  `public` **NoSuchProducerException**`( java.lang.String s )`
  - **Usage**
    * The exception with description

A.1.6.3 Methods inherited from class `java.lang.Exception`

A.1.6.4 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**`( )`

A.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

A.1.7.1 Declaration

public class NoSuchTableNameException
extends java.lang.Exception

A.1.7.2 Constructors

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

A.1.7.3 Methods inherited from class `java.lang.Exception`

A.1.7.4 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**`( )`

A.1.8 Class **Producer**

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

A.1.8.1 Declaration

public class Producer **extends** org.edg.info.BaseProducer

A.1.8.2 Constructors

- *Producer*
  `public` **Producer**`( 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
- *Producer*
  `public` **Producer**`( 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

A.1.8.3 Methods

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

- *setRemoteBufferSize*
  `public void` **setRemoteBufferSize**`( int` **remoteBufferSize** `)`
  - **Usage**
    * Set the producer servlet's buffer size

A.1.8.4 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** `)`

A.1.9 Class **ProducerConnection**

Identifies a connection to a producer servlet by a producer

A.1.9.1 Declaration

public class ProducerConnection

extends java.lang.Object

A.1.9.2 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.

A.1.9.3 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( )`

A.1.10 Class **ProducerInfo**

Producer Information

A.1.10.1 Declaration

public class ProducerInfo
extends java.lang.Object

A.1.10.2 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

A.1.10.3 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( )`

A.1.11 Class **PropertyGetter**

A.1.11.1 Declaration

public class PropertyGetter
extends java.lang.Object

A.1.11.2 Constructors

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

A.1.11.3 Methods

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

A.1.12 Class **QueryException**

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

A.1.12.1 Declaration

public class QueryException
extends java.lang.Exception

A.1.12.2 Constructors

- *QueryException*
  `public` **QueryException**`( )`
  - **Usage**
    * The exception without description

- *QueryException*
  `public` **QueryException**`( java.lang.String` **s** `)`
  - **Usage**
    * The exception with description

A.1.12.3 Methods inherited from class `java.lang.Exception`

A.1.12.4 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**`( )`

A.1.13 Class **RegistrationException**

Thrown by Registry API if the register method fails

A.1.13.1 Declaration

public class RegistrationException
extends java.lang.Exception

A.1.13.2 Constructors

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

A.1.13.3 Methods inherited from class `java.lang.Exception`

A.1.13.4 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**`( )`

A.1.14 Class **Registry**

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

A.1.14.1 Declaration

public class Registry
extends java.lang.Object

A.1.14.2 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.

A.1.14.3 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` **fixed-Columns**`, 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` **fixed-Columns**`, 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

A.1.15 Class **ResultSet**

A.1.15.1 Declaration

public class ResultSet
extends java.lang.Object

A.1.15.2 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()

A.1.16 Class **ResultSetMetaData**

A.1.16.1 Declaration

public class ResultSetMetaData
extends java.lang.Object

A.1.16.2 Methods

- *getColumnCount*
  `public int` **getColumnCount**`( )`

- *getColumnName*
  `public String` **getColumnName**`( int` **columnNumber** `)`

A.1.17 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 seperates the registration and description of tables.

A.1.17.1 Declaration

---

public class Schema

extends java.lang.Object

A.1.17.2 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.

A.1.17.3 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

A.1.18 Class **ServletConnectionException**

Thrown by Schema API if any of the methods of the ServletConection fails

A.1.18.1 Declaration

---

public class ServletConnectionException
extends java.lang.Exception

A.1.18.2 Constructors

- *ServletConnectionException*
`public` **ServletConnectionException**`( )`
  - **Usage**
    * The exception without description

- *ServletConnectionException*
`public` **ServletConnectionException**`( java.lang.String` **s** `)`
  - **Usage**
    * The exception with description

A.1.18.3 Methods inherited from class `java.lang.Exception`

A.1.18.4 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**`( )`

A.1.19 Class **SubscriptionException**

Thrown by Registry API if the register method fails

A.1.19.1 Declaration

public class SubscriptionException
extends java.lang.Exception

A.1.19.2 Constructors

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

A.1.19.3 Methods inherited from class `java.lang.Exception`

A.1.19.4 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**`( )`

A.1.20 Class **XMLConverter**

A.1.20.1 Declaration

public class XMLConverter
extends java.lang.Object
implements org.xml.sax.ErrorHandler

A.1.20.2 Constructors

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

A.1.20.3 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.xml.sax.SAXParseException` **ex** `)`
  - **Usage**
    * Error.

- *fatalError*
  `public void` **fatalError**`( org.xml.sax.SAXParseException` **ex** `)`

- **Usage**
  * Fatal error.

- *warning*
  `public void` **`warning`**`( org.xml.sax.SAXParseException` **`ex`** `)`
  - **Usage**
    * Warning.

Appendix B

C++ API

B.1 Consumer Class Reference

Consumes event by event or a stream of events.

```
#include <Consumer.h>
```

Public Methods

- **Consumer** (const string selectStatement)

  Construct a consumer using a string representing the SQL query.

- **Consumer** (const string selectStatement, **ProducerConnection** &producerConnection)

  Construct a consumer using a string representing the SQL query and a specified **ProducerConnection** (p. 47) *Unlike the other constructor which relies on the registry to find a producer, this constructor takes a* **ProducerConnection** (p. 47) *object reference as the second argument.*

- **ResultSet execute** ()

  execute the Consumer's query to return an **ResultSet** (p. **??**).

- void **setBufferSize** (const int bufferSize1)

  set consumer servlet buffersize.

- int **getBufferSize** ()

  get consumer servlet bufferSize.

- int **count** ()

  Number of available events.

- **ResultSet pop** ()

  Return the oldest piece of information from the consumerservlet queue (buffer) and remove it.

- virtual ~**Consumer** ()

  Destroy query objects associated with this instance and unsubscribe from producer.

- string **getStatus** ()

  Get status information.

B.1.1 Detailed Description

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

B.1.2 Constructor & Destructor Documentation

B.1.2.1 Consumer::Consumer (const string *selectStatement*)

Construct a consumer using a string representing the SQL query.

Parameters:
selectStatement the desired SQL select statement

B.1.2.2 Consumer::Consumer (const string *selectStatement*, ProducerConnection & *producerConnection*)

Construct a consumer using a string representing the SQL query and a specified **ProducerConnection** (p. 47) Unlike the other constructor which relies on the registry to find a producer, this constructor takes a **ProducerConnection** (p. 47) object reference as the second argument.

Parameters:
selectStatement the desired SQL select statement.

ProducerConnection identifies the **Producer** (p. 44) of the information

B.1.2.3 virtual Consumer::~Consumer () `[virtual]`

Destroy query objects associated with this instance and unsubscribe from producer.

B.1.3 Member Function Documentation

B.1.3.1 int Consumer::count ()

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.

B.1.3.2 ResultSet Consumer::execute ()

execute the Consumer's query to return an **ResultSet** (p. **??**).

This method should be used to issue one SQL query to the producer and get the last tuple from the producer servletbuffer. **The schema of the XML will almost certainly be changed.**

B.1.3.3 int Consumer::getBufferSize ()

get consumer servlet bufferSize.

B.1.3.4 string Consumer::getStatus ()

Get status information.

Returns:
XML formatted status information.

B.1.3.5 ResultSet Consumer::pop ()

Return the oldest piece of information from the consumerservlet 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. **This method may be replaced by one which returns a java.sql.ResultSet. The schema of the XML will almost certainly be changed.**

B.1.3.6 void Consumer::setBufferSize (const int *bufferSize1*)

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.

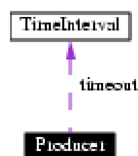
Parameters:
bufferSize1 - desired buffer size measured as number of events

B.2 Producer Class Reference

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

`#include <Producer.h>`

Collaboration diagram for Producer:



Public Methods

- **Producer** (const string tableName, const string fixedColumns, const int flags, const string tableDesc)

  If the table is already known within the schema and is inconsistent with the tableDesc this constructor will fail.

- **Producer** (const string tableName, const string fixedColumns, const int flags)

  For this constructor the tableName must already be known within the schema.

- void **insert** (const string row)

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

- int **getLocalBufferPos** ()

  position in producer's local buffer.

- string **getOne** (const string query)

  returns one entry - this method is here for testing without needing a **Consumer** (p. 42) *and Consumer-Servlet.*

- int **getLocalBufferSize** ()

  returns the producer's local buffer size.

- void **setLocalBufferSize** (const int localBufferSize1)

  sets the producer's local buffer size.

- void **setRemoteBufferSize** (const int remoteBufferSize1)

  Set the producer servlet's buffer size.

- int **getRemoteBufferSize** ()

  Get the producer servlet's buffer size.

- **TimeInterval getTimeout** ()

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

- void **setTimeout** (**TimeInterval** &timeout1)

  set time interval after which local buffer is transmitted even if not full.

- bool **getValidate** ()

  carry out local validation of the row(s) inserted as far as possible.

- void **setValidate** (const bool validate1)

  set value of validate.

- string **getStatus** ()

  Get status information.

B.2.1 Detailed Description

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

B.2.2 Constructor & Destructor Documentation

B.2.2.1 Producer::Producer (const string *tableName*, const string *fixedColumns*, const int *flags*, const string *tableDesc*)

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

B.2.2.2 Producer::Producer (const string *tableName*, const string *fixedColumns*, const int *flags*)

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 andtheir values.

flags is a set of boolean flags encoded in an integer.

B.2.3 Member Function Documentation

B.2.3.1 int Producer::getLocalBufferPos ()

position in producer's local buffer.

This is the index of the last row occupied.

B.2.3.2 int Producer::getLocalBufferSize ()

returns the producer's local buffer size.

B.2.3.3 string Producer::getOne (const string *query*)

returns one entry - this method is here for testing without needing a **Consumer** (p. 42) and Consumer-Servlet.

Parameters:

query as an SQL select statement.

B.2.3.4 int Producer::getRemoteBufferSize ()

Get the producer servlet's buffer size.

B.2.3.5 string Producer::getStatus ()

Get status information.

Returns:

XML formatted status information

B.2.3.6 TimeInterval Producer::getTimeout ()

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

B.2.3.7 bool Producer::getValidate ()

carry out local validation of the row(s) inserted as far as possible.

Primarily this means checking that the columns match. (not implemented yet)

B.2.3.8 void Producer::insert (const string *row*)

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

Parameters:

row as an SQL insert statement.

B.2.3.9 void Producer::setLocalBufferSize (const int *localBufferSize1*)

sets the producer's local buffer size.

Parameters:
localBufferSize1 the desired producer local buffer size.

B.2.3.10 void Producer::setRemoteBufferSize (const int *remoteBufferSize1*)

Set the producer servlet's buffer size.

Parameters:
remoteBufferSize1 the desired producer servlet circular buffer size for this producer.

B.2.3.11 void Producer::setTimeout (TimeInterval & *timeout1*)

set time interval after which local buffer is transmitted even if not full.

(not implemented yet).

B.2.3.12 void Producer::setValidate (const bool *validate1*)

set value of validate.

(not implemented yet)

B.3 ProducerConnection Class Reference

Identifies a connection to a producer servlet by a producer.

`#include <ProducerConnection.h>`

Public Methods

- **ProducerConnection** (string producerServlet1, int connectionId1)

  Construct a ProducerConnection using a string representing the producer URL and an integer representing the producer's unique connection identifier.

- virtual ~**ProducerConnection** ()
- string **getProducerServlet** ()

  get producer servlet URL as a string.

- int **getConnectionId** ()

  get producer connection identifier.

- string **toString** ()

  Produce a string representation of the ProducerConnection.

B.3.1 Detailed Description

Identifies a connection to a producer servlet by a producer.

B.3.2 Constructor & Destructor Documentation

B.3.2.1 ProducerConnection::ProducerConnection (string *producerServlet1*, int *connectionId1*)

Construct a ProducerConnection using a string representing the producer URL and an integer representing the producer's unique connection identifier.

Parameters:

producerServlet1 the URL of the producer servlet.

connectionId1 identifies the specific producer for the desired connection.

B.3.2.2 virtual ProducerConnection::~ProducerConnection () `[virtual]`

B.3.3 Member Function Documentation

B.3.3.1 int ProducerConnection::getConnectionId ()

get producer connection identifier.

B.3.3.2 string ProducerConnection::getProducerServlet ()

get producer servlet URL as a string.

B.3.3.3 string ProducerConnection::toString ()

Produce a string representation of the ProducerConnection.

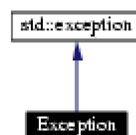
B.4 Exception Class Reference

A simple Exception class for testing.

`#include <Exception.h>`

Inherits **std::exception**.

Collaboration diagram for Exception:



Public Methods

- **Exception::Exception** (const std::string &what_arg)
- const char * **Exception::what** () const

B.4.1 Detailed Description

A simple Exception class for testing.

Will be improved in the future.

B.4.2 Member Function Documentation

B.4.2.1 Exception::Exception::Exception (const std::string & *what_arg*) [inline]

B.4.2.2 const char* Exception::Exception::what () const [inline]

Bibliography

[1] Global grid forum. http://www.gridforum.org/.

[2] 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.