EGEE

R-GMA User Guide for C++ Programmers

Document identifier:	**EGEE-JRA1-TEC-503616**
Date:	**June 15, 2010**
Activity:	**JRA1: Middleware Engineering and Integration (UK Cluster)**
Document status:	**FINAL**
Document link:	**https://edms.cern.ch/document/503616/**

Abstract: This document provides the C++ programmer with the information necessary to get started with R-GMA.

CONTENTS

1 INTRODUCTION

1.1 PURPOSE AND STRUCTURE OF THIS DOCUMENT

This document is intended to get people started with R-GMA. It is one of a set, with each member customised for a different programming language.

After this introduction there are sections explaining what should be done to ensure that R-GMA is correctly installed, how to publish information via a *Primary Producer*, how to get information back via a *Consumer*, how to set-up a *Secondary Producer* and how to use the command line and web based tools.

The APIs (in C, C++, Java and Python) are all described in detail in the documentation linked from http://hepunx.rl.ac.uk/egee/jra1-uk/r-gma/ . In addition the documentation is all distributed with the software and may be found as `<rgma_home>/share/doc/rgma-base/<language>.pdf`, where *language* identifies the document. Look at the directory `<rgma_home>/share/doc/rgma-base/` to see the naming scheme.

Some brief release notes from a user's perspective may be found in section 16 which is useful to anyone who is familiar with the previous version of R-GMA.

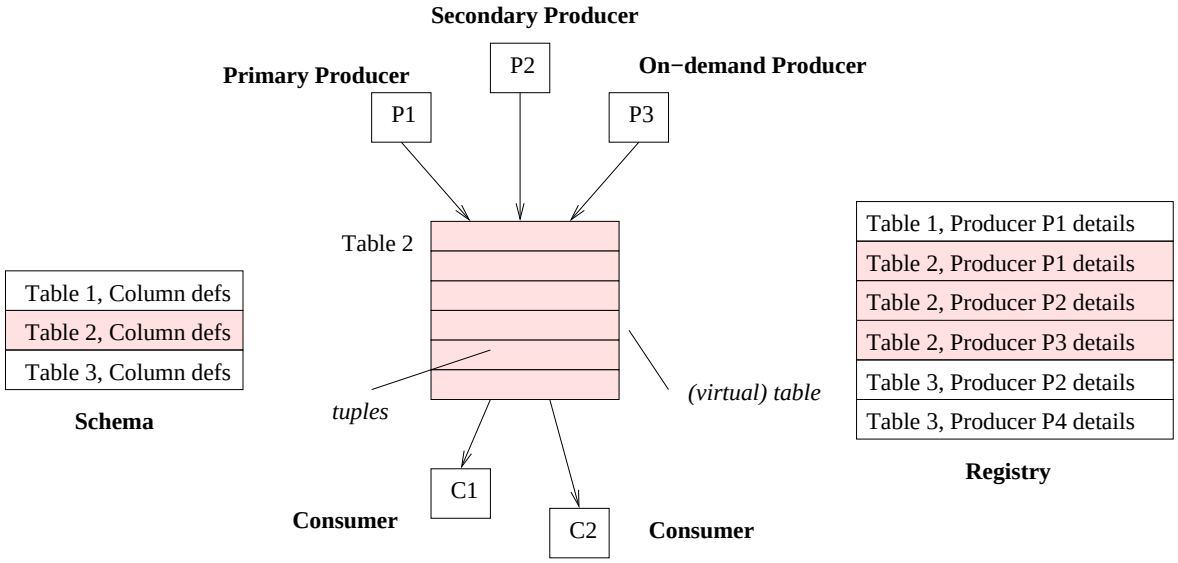
To understand more detail of what R-GMA is meant to do, you may choose to read the specification[1]. However we do not expect the average user to need the specification document.

This User Guide contains a number of code examples. You can find a copy of these in the directory: `<rgma_home>/share/doc/rgma-base/examples/` wherever R-GMA has been installed.

1.2 R-GMA ARCHITECTURE

1.2.1 VIRTUAL DATABASE

R-GMA is an implementation of the Grid Monitoring Architecture (GMA) proposed by the Open Grid Forum (OGF), which models the information infrastructure of a Grid as a set of *Consumers* (who request information), *Producers* (who provide information) and a *Registry* (which mediates the communication between producers and consumers). R-GMA imposes a standard query language (a subset of SQL) on this model – so producers publish *tuples* (database rows) with an SQL insert statement and consumers query them using SQL select statements. R-GMA also ensures that all tuples carry a *time-stamp*, so that monitoring systems (which require time-sequenced data) are inherently supported.



R-GMA presents the data in the form of *virtual databases*, each containing a set of *virtual tables*. As the picture above shows, a single[1] schema contains the name and structure (column names, types and settings) of each virtual table in the system. A single registry contains a list, for each table, of producers who have offered to publish rows for the table. A consumer runs an SQL query against a table, and contacts the registry to select the best producers to answer the query, contacts each producer directly, combines the information, and returns a set of tuples. The mediation process is hidden from the user. Note that there is no central repository holding the contents of the virtual table; it is in this sense, that the database is virtual.

1.2.2 PRODUCERS

There are three classes of producer: *Primary*, *Secondary* and *On-demand*. Each is created by a user application and returns tuples in response to queries from other user applications. The main difference is in where the tuples come from.

For a primary producer, user's code periodically inserts tuples which are then stored internally by the producer service. The producer answers consumer queries from this storage. The secondary producer service also answers queries from its internal storage, but it populates this storage itself by running its own query against the virtual table; the user code only sets the process running; the tuples come from other producers. In the on-demand producer, there is no internal storage; data is provided by the user code in direct response to a query forwarded on to it by the producer service. Examples of the use of the on-demand producer are not currently included in this document.

Producers may be set up to use either database or memory storage, with the data being held in one or two *tuple stores*. One store is used to answer latest queries. The other store is used for both history and continuous queries. If no logical name is specified the data are transient and are removed when the producer is closed down – even if a database is being used to implement the tuple store.

In the case of data base storage, the data can be made persistent by specifying a logical name for the tuple store. A tuple store is identified by the logical name and the DN of the user so several people can use the same logical name. When a producer using a named tuple store closes down, the tuple store and the data in it are preserved. When a new producer is created by that user, specifying the same logical name the tuple store is reused.

The mapping of table names to tables inside the database is carried out by R-GMA and is hidden from the normal user. For those cases where direct access to the database is needed, a facility is provided to the sysadmin to determine the mappings from table name to internal table name for a specified logical name of a tuple store and DN, see Section 15.7.

Each virtual table has a *key* column (or group of columns) declared in the schema. Each tuple also carries a *time-stamp*, added by the primary producer when the tuple is first published into the system, which, together with the key columns, is similar to a primary key for the table. Tuples with the same key, but different values for the time-stamp, can also be thought of as different versions of the same tuple. R-GMA works consistently in UTC[2]. Though the primary producer will add the time-stamp for you, you can set the date and time yourself. This is useful if you want to publish some information from a log file for example. Simply include the *RgmaTimestamp* field in your insert call. The format is the usual "YYYY-MM-DDz hh:mm:ss[.s]" for the time-stamp.

[1] Although there is only one logical schema and registry pair per VDB, replicas are made for scalability and robustness. This is discussed in detail later.

[2] UTC refers to Coordinated Universal Time, which used to be known as GMT.

1.2.3 CONSUMERS

Each consumer represents a single SQL SELECT query on the virtual database. The request is initiated by user code, but the *Consumer Service* carries out all of the work on its behalf. The query is first passed to the registry to identify which producers, for each virtual table in the query, must be contacted to answer it. This process is called *mediation*. The query is then passed by the consumer service to each relevant producer, to obtain the answer tuples directly.

There are four types of query: *continuous*, *latest*, *history* and *static*. The first three types (continuous, latest and history) can optionally take a *TimeInterval* parameter.

A continuous query (which can only act on a single table) causes all new tuples matching the query to be automatically streamed to the consumer when they are inserted to the virtual table by a producer. If a time interval parameter is specified, all existing tuples newer than (*now* - *TimeInterval*) will additionally be returned when the query is started.

A latest query is evaluated on that set of tuples, which for each table and key have the greatest time-stamp value and that have not exceeded their *Latest Retention Period*. In addition, if a time interval is specified, only tuples that are newer than (*now* - *TimeInterval*) will be used. Whether or not you specify a time interval, the query will never use tuples that have exceeded their latest retention period. The latest retention period is described in 1.2.5.

A history query is evaluated over all available versions of tuples. The set can also be restricted by specifying a time interval.

A static query is handled by an on-demand producer like a normal one-off database query. There are no time-stamps or retention periods associated with static queries.

1.2.4 META-DATA

There are four meta-data fields associated with every tuple, these are: *RgmaTimestamp*, *RgmaLRT*, *RgmaOriginalServer* and *RgmaOriginalClient*.

As mentioned above the RgmaTimestamp is normally set by the system to be the time-stamp when the tuple as inserted into R-GMA. If a value is set by the user when INSERTing a tuple into a primary producer this value will be used.

The RgmaLRT holds the *Latest Retention Time.* This field is user readable but not writable. It is derived by adding the RgmaTimestamp to the latest retention period associated with the insert operation. A latest query will never return tuples that have exceeded the RgmaLRT value.

The RgmaOriginalServer is the name of the server where the data were inserted if it can be found in the DNS by reverse look-up. Otherwise it will be a string representation of the numeric IP address.

Finally the RgmaOriginalClient is the name of the user's client machine that communicated with the RgmaOriginalServer to insert the data.

None of the meta-data fields are ever modified once the data are stored.

1.2.5 RETENTION PERIODS

To allow primary and secondary producers to periodically purge "old" tuples, and to give a precise meaning to the "current state" for a latest query, *retention periods* are used.

The *Latest Retention Period* defines how old a tuple can be before it should no longer be considered to be the latest. This time interval is added to the time-stamp and inserted into each tuple published by a primary producer (RgmaLRT), and remains there when a tuple is re-published by a secondary producer.

In addition, primary and secondary producers declare a *History Retention Period* for each table to which they are publishing tuples.

Primary and secondary producers have two logical tuple-stores, one supporting latest-queries and the other supporting continuous and history queries. Producers undertake to retain the *most recent* version of any tuple which has not exceeded its latest retention period, and *all* versions of any tuple which have not exceeded the history retention period.

The history retention period may be longer or shorter than the latest retention period. The history retention period is a (per-table) property of the producer, whereas the latest retention period is a property of the tuple.

1.2.6 Resource Framework and the Termination Interval

Each instance of a producer or consumer in a running R-GMA system exists as a *resource* on a server. Resources have a termination interval associated with them. The value of the termination interval is determined by the server but may be queried by the user.

If resource has not been contacted for a period exceeding its termination interval then the service will close it. This is part of the soft-state registration process used to ensure that non-existent resources are removed within a reasonable length of time. A resource will also be destroyed if the server is restarted. If a user makes a call via the API after the resource has been closed by the service, then the system will automatically create a new resource.

A side effect of the system automatically re-creating a consumer resource is that the query has to be restarted; this may result in duplicate data being returned. If a query is restarted in this manner then a warning will be included in the returned tuple set, see Section 3.

The user should close a resource manually when it is no longer needed to conserve resources on the server.

2 Authorization

2.1 Credentials

R-GMA services permit or deny access to resources (in the general sense) to a user or another service, on the basis of *credentials* held by that user or service. Credentials are extracted from the certificate used for authentication. Each credential has a name such as the Distinguished Name (DN) credential found in all certificates[3]. The credentials found in the *VOMS proxy certificates* used by all EGEE services are:

- Membership of virtual organizations (VO);
- Membership of groups within a virtual organization (GROUP);
- Roles held within a virtual organization (ROLE);

and some, such as GROUP, may take more than one value. All authorization rules are defined in terms of combinations of zero or more credentials.

[3]The DN of host certificates contains the host name.

2.2 Restricting Access to Columns (views)

Views are defined in the schema as a subset of the columns of a table (the *base table* of the view), created using the schema service's `createView` operation. Users can query views just as they would real tables, but they are read-only, so producers cannot publish to them, and they do not appear in the registry. Their purpose is to limit access to only certain columns of the base table: tuples read from a view are actually read from the base table, but the query is only allowed to include columns that form part of the view. Authorization rules for row access to views are defined exactly as for tables (see the next section) and completely replace the rules on the base table.

2.3 Authorizing Read/Write Access to Rows in Tables or Views

All consumers require read authorization to read tuples from views or tables in the virtual database and all producers (primary and secondary) require write authorization to publish tuples into tables in the virtual database.

These authorization rules are stored in the schema; the user who creates the table or view sets the initial rules.

The default rule is that no user is permitted any access to the table or view. Rules are added to grant access only (not to deny it) and they are cumulative. Read and write rules are formatted identically and have the form "*predicate : credentials : action*", where the three clauses are defined as follows:

predicate Defines the subset of rows of the table or view to which this rule grants access. It is an SQL WHERE clause that compares the values in specified columns with constants, other columns or credential parameters (credential name in square brackets, such as [DN]) that are replaced by the corresponding credentials (or set of credentials) extracted from the user's certificate when the rule is evaluated. The predicate expression is restricted to the following operators: AND, OR, NOT, IN, =, LIKE, <>, <, > and IS NULL. This clause may be empty, in which case the rule applies to all rows in the table.

credentials Defines the set of credentials required for a user to be granted access to the subset of rows defined by this rule. It is a boolean combination of equality constraints of the form *[credential]* = *constant*. This clause may be empty, in which case the rule applies to all authenticated users.

action Defines what any matching user is allowed to do to the subset of rows defined by this rule. The valid values are: *R* (read), *W* (publish) or *RW* (read and publish). This clause must not be empty.

Some examples rules are:

```
WHERE Section = 'Marketing':[GROUP] = 'Marketing' OR [GROUP] = 'Management':RW
```

that grants read-write access to any authenticated user with a GROUP credential of 'Marketing' or 'Management', to those rows that contain the value 'Marketing' in the 'Section' column;

```
WHERE Owner = [DN]::R
```

that grants read-only access to any authenticated user, to those rows that contain the value of their DN credential in the 'Owner' column;

```
WHERE Group = [GROUP] OR Public = 'true'::R
```

that grants read-only access to any authenticated user, to those rows that contains one of their GROUP credential values in the 'Group' column, or have a value of 'true' in the 'Public' column;

```
::R
```

that grants read-only access to any authenticated user, to all rows in the table.

2.4 CONTROLLING SCHEMA ACCESS

Also associated with each table are rules defining who can read, write (modify or delete) a table definition. These rules take the form: "*credentials : action*" where credentials and action are defined as above. A call to getAllTables will only show those tables for which you have read access and information about a specific table also requires read access.

In addition there are rules included in the VDB definition that are independent of table. The most useful of these is to say who can create a table. These rules also contain (credential, action) pairs but the action now includes "C" to define who can create a table. The R-GMA setup script defines a local vdb with a name of DEFAULT. This has a rule giving CRW access to the DN of the person running the setup script. When creating a vdb you should consider whether or not to define a rule :CR allowing anyone to create a table or to read a table definition. The DEFAULT vdb has such a rule.

When a table is created, if no schema access rule is provided then a rule of the form:

```
[DN] = <creators DN> : RW
```

is added to the table defintion to give the creator full control of the table defintion.

Permission is granted if any rule in the VDB defintion or a table specific rule permits the action.

2.5 UPDATING THE RULES

The authorization rules may be updated at any time. There will be a short delay while the change is propagated to other servers and while the resources check for schema modifications.

3 EXCEPTIONS

R-GMA APIs may throw one of two kinds of *RGMAException* for reporting when an error occurs in an R-GMA application:

RGMATemporaryException Exception thrown when an R-GMA call fails, but calling the method again after a delay may be successful.

RGMAPermanentException Exception thrown when an R-GMA call fails and repeating the call will not be successful.

These sub-exceptions of RGMAException are implemented as subclasses of RGMAException by those API languages that support the concept, so that clients can just catch RGMAException if they choose to.

Warnings are only issued in R-GMA in response to consumer queries, to indicate possible inadequacies in the results. They are appended to the query's tuple set.

4 RUNNING THE EXAMPLES

In the following chapters there are a number of examples of the use of the R-GMA client code. This assumes that the following environment variables are set:

- `RGMA_HOME` pointing to your installation of R-GMA.
- `GLITE_LOCATION` pointing to your installation of gLite.
- `X509_USER_PROXY` or `TRUSTFILE` is set, pointing to your proxy or trust file respectably.

To compile the example with gcc you might enter:

```
g++ XXXExample.cpp  -o XXXExample \
    -I$RGMA_HOME/include \
    -L$RGMA_HOME/lib -lglite-rgma-cpp -lssl
```

Substitute `XXX` in the above code with the name of the example, i.e. PrimaryProducer or Consumer etc. You will need to add `$RGMA_HOME/lib` to `$LD/LIBRARY/PATH` in order to run the example (to find the shared libraries).

5 MANAGING TABLE DEFINITIONS

Creating and dropping of tables may be done with the command line tool or through the API. We describe here the use of the API.

5.1 CREATE TABLE

The examples in the following sections all use the table "userTable". It is preferable to use a different table name and then clean up when you have finished trying things out by dropping the table (5.3). If you do choose to use a table created by somebody else you will not be allowed to create or drop that table and unless it has been created with authorization rules permitting you to read and write to that table you will not get the behaviour you expect!

5.1.1 CREATE TABLE EXAMPLE

```
0   #include "rgma/Schema.h"
1   #include "rgma/RGMAException.h"

2   #include <string>
3   #include <iostream>

4   using glite::rgma::Schema;
5   using glite::rgma::RGMAException;

6   int main(int argc, char* argsv[])
7   {
8      std::string vdb("default");
```

```
 9      std::string create("create table userTable (userId VARCHAR(255) NOT NULL "
10         "PRIMARY KEY, aString VARCHAR(255), aReal REAL, "
11         "anInt INTEGER)");

12      std::vector<std::string> rules;
13      rules.push_back("::RW");

14      try
15      {
16         Schema schema(vdb);
17         schema.createTable(create, rules);
18      }

19      catch (RGMAException e)
20      {
21         std::cerr << "R-GMA exception: " << e.getMessage() << std::endl;
22      }

23   }
```

Lines 0–3 are the various include statements.

Lines 4–5 are a set of `using` statements to avoid clutter in the code.

Line 8 is the `name` of the virtual database.

Lines 9–11 is the create table statement defining the column names and types.

Lines 12–13 is the authorization rule in the form predicate:credentials:action. This can be a list of rules.

Lines 16–17 contacts the schema and adds the table definition.

Lines 19–22 report any R-GMA exceptions that may occur.

5.2 ALTER TABLE

If the schema is modified resources using it will close down and send a temporary exception so that the client is unaffected. Any resource will then be recreated and if it needs a named tuplestore will find that the schema has changed and will modify the stored table to match.

If a table is dropped and recreated then the application code may find the table missing and get a permanent exception .

Not all possible schema changes can be carried out so an ALTER TABLE call is provided to support only those changes that can be made reliably and to ensure that at all times the table exists and so only a temporary exception will be generated as described above.

If a named tuplestore cannot make the change correctly to the stored data it will simply destroy the old table and create a new one. This will never happen if the ALTER TABLE operation on the schema is used. Changes to the columns of the primary key or the addition of a ”not null” column are not permitted. Types of columns may not be changed except between CHAR and VARCHAR or to increase the size of CHAR or VARCHAR. If a column is removed - it will be removed from the database table. If a column is added - it will be added and set to null for all existing tuples.

The ALTER TABLE and ALTER VIEW facilities are not provided as part of the API but only from the command line tool.

5.3 Drop Table

In order to drop a table replace lines 9–11 from the create table example with:

```
schema.dropTable("userTable");
```

This will remove the table definition from the schema. Within a few minutes any producer and consumer resources still using the table will be destroyed.

6 Primary Producers

6.1 History Retention Period

If the user sends a `close` request the primary producer will no longer be contactable from the user API. However the resource will remain available to consumers until the *HistoryRetentionPeriod* is reached for all of the data. If the termination interval is exceeded, the resource behaves as though it had received a `close` request.

The resource is destroyed immediately when the user issues an explicit `destroy` request.

6.2 Producer Properties

All producers support continuous queries, but you may also specify that you want the producer to also support history and/or latest queries.

The tuple-storage maintained by primary and secondary producers can either be in memory or in a real database table. You should choose whichever is the most appropriate.

For a memory based primary producer, there is a server defined limit on how many tuples can be inserted into each producer. If this limit is exceeded, or if server resources are low, an RGMATemporaryException will be thrown and application code should wait a little while.

A common pattern is to use a memory based producer as the primary one and then to use a secondary producer to collect the information and hold it in an RDBMS. This is the example we consider here.

6.3 Primary Producer Examples

This provides examples of code you might use or adapt.

6.3.1 Simple Primary Producer Example

```
0   #include "rgma/PrimaryProducer.h"
1   #include "rgma/RGMAException.h"
2   #include "rgma/Storage.h"
3   #include "rgma/SupportedQueries.h"
4   #include "rgma/TimeInterval.h"
5   #include "rgma/TimeUnit.h"

6   #include <string>
7   #include <iostream>

8   using glite::rgma::PrimaryProducer;
9   using glite::rgma::RGMAException;
```

```
10  using glite::rgma::Storage;
11  using glite::rgma::SupportedQueries;
12  using glite::rgma::TimeInterval;
13  using glite::rgma::TimeUnit;

14  int main(int argc, char* argv[])
15  {

16     if (argc != 2)
17     {
18        std::cout<< "Usage: PrimaryProducerExample <userId>\n"<< std::endl;
19        exit(1);
20     }

21     std::string args(argv[1]);

22     try
23     {
24        PrimaryProducer pp (Storage::MEMORY, SupportedQueries::C);

25        std::string table("default.userTable");
26        std::string predicate("WHERE userId = '" + args + "'");
27        TimeInterval historyRetentionPeriod(60, TimeUnit::MINUTES);
28        TimeInterval latestRetentionPeriod(60, TimeUnit::MINUTES);
29        pp.declareTable(table, predicate, historyRetentionPeriod,
30              latestRetentionPeriod);

31        std::string insert;
32        insert.append("INSERT INTO default.userTable (userId, aString, ");
33        insert.append("aReal, anInt) VALUES ('");
34        insert.append(args);
35        insert.append("', 'C++ producer', 3.1415926, 42)");
36        pp.insert(insert);

37        pp.close();
38     }

39     catch (RGMAException e)
40     {
41        std::cerr << "R-GMA exception: " << e.getMessage() << std::endl;
42     }

43  }
```

Lines 0–7 are the various `include` statements.

Lines 8–13 are a set of `using` statements to avoid clutter in the code.

Line 24 create a the primary producer which supports continuous queries, with memory based storage. The user can specify that the producer supports history and/or latest queries.

Lines 25–30 define the predicate and a pair of retention periods (60 minutes in each case) and then declare the intention of publishing to a table called `default.userTable`. A producer is able to publish to more than one table.

Lines 31–36 build an SQL insert statement and then call the `insert` method of the primary producer with that SQL statement as an argument.

Line 37 close the primary producer.

Lines 39–42 report any R-GMA exceptions that may occur.

6.3.2 RESILIENT PRIMARY PRODUCER EXAMPLE

This example illustrates how to write a resilient producer according to the recommendations in section 13. The primary producer publishes information periodically every 30 seconds. If there is a temporary problem it retries after one minute. If there is a permanent problem then the program exits.

```
 0  #include "rgma/PrimaryProducer.h"
 1  #include "rgma/RGMAException.h"
 2  #include "rgma/RGMAPermanentException.h"
 3  #include "rgma/RGMATemporaryException.h"
 4  #include "rgma/Storage.h"
 5  #include "rgma/SupportedQueries.h"
 6  #include "rgma/TimeInterval.h"
 7  #include "rgma/TimeUnit.h"

 8  #include <string>
 9  #include <iostream>
10  #include <sstream>

11  using glite::rgma::PrimaryProducer;
12  using glite::rgma::RGMAException;
13  using glite::rgma::RGMAPermanentException;
14  using glite::rgma::RGMATemporaryException;
15  using glite::rgma::Storage;
16  using glite::rgma::SupportedQueries;
17  using glite::rgma::TimeInterval;
18  using glite::rgma::TimeUnit;

19  int main(int argc, char* argv[])
20  {

21     if (argc != 2)
22     {
23        std::cerr << "Usage: " << argv[0] << " <userId>" << std::endl;
24        exit(1);
25     }

26     std::string userId(argv[1]);
27     PrimaryProducer* producer = 0;

28     try
29     {
30        TimeInterval historyRetentionPeriod(50, TimeUnit::MINUTES);
31        TimeInterval latestRetentionPeriod(25, TimeUnit::MINUTES);

32        while(!producer)
33        {
34           try
35           {
36              producer = new PrimaryProducer(Storage::MEMORY, SupportedQueries::C);
37           }
38           catch (RGMATemporaryException te)
39           {
40              std::cerr << "RGMATemporaryException: " << te.getMessage() << std::endl;
41              sleep(60);
42           }
43        }
```

```
44          std::string predicate("WHERE userId = '" + userId + "'");
45          bool tableDeclared = false;

46          while (!tableDeclared)
47          {
48             try
49             {
50                producer->declareTable("default.userTable", predicate,
51                      historyRetentionPeriod,
52                      latestRetentionPeriod);
53                tableDeclared = true;
54             }
55             catch (RGMATemporaryException te)
56             {
57                std::cerr << "RGMATemporaryException: " << te.getMessage() << std::endl;
58                sleep(60);
59             }
60          }

61          int data = 0;
62          std::stringstream insert;

63          while (true)
64          {
65             try
66             {
67                insert.str("");
68                insert << "INSERT INTO default.userTable (userId, aString, "
69                << "aReal, anInt) VALUES('" << userId
70                << "', 'resilient CPP producer', 0.0, " << data << ")";
71                producer->insert(insert.str());
72                std::cout << insert.str() << std::endl;
73                data++;
74                sleep(30);
75             }
76             catch (RGMATemporaryException te)
77             {
78                std::cerr << "RGMATemporaryException: " << te.getMessage() << std::endl;
79                sleep(60);
80             }
81          }
82       }

83       catch (RGMAException pe)
84       {
85          std::cerr << "RGMAPermanentException: " << pe.getMessage() << std::endl;
86          delete producer;
87          producer = 0;
88          exit(1);
89       }

90    }
```

Lines 0–10 are the various `include` statements.

Lines 11–18 are a set of `using` statements to avoid clutter in the code.

Line 27 set up initial state for the producer.

Lines 30–31 set up the history and latest retention periods.

Lines 32–43 create a new primary producer. If there is an RGMATemporaryException then sleep for 60 seconds before retrying to recreate the producer.

Line 44 set up the producer predicate.

Line 45 set up initial state for the table.

Lines 46–60 declare the `default.userTable`. If there is an RGMATemporaryException then sleep for 60 seconds before trying to re-declare the table.

Line 61 initialises the data value that is going to be inserted into each tuple.

Lines 63–81 loops forever - each iteration will insert a new tuple. In our example, tuples contain the userId and the current value of the `data` variable that gets incremented on each successful iteration of the loop. For the purpose of this example a delay of 30 seconds has been introduced between each insert. If there is an RGMATemporaryException then sleep for 60 seconds before retrying to insert the tuple.

Lines 83–89 report an RGMAPermanentException. As this is considered to be a permanent problem there is no point retrying the operation.

7 CONSUMING INFORMATION

7.1 TYPES OF QUERY

There are four types of query: *continuous*, *latest*, *history* and *static* (the latter is only supported by on-demand producers). The set of queries that a particular producer supports is recorded in the registry. All query types except static can take an optional time interval parameter (see below).

A continuous query causes all new tuples that match the query, to be streamed into the consumer's tuple-storage as soon as they are inserted into the virtual table by the producers. Streaming continues until the consumer requests it to stop. If a time interval is specified, the consumer will additionally receive any tuples which are already in the virtual table when the query starts, and which are no older than the time interval. There is no guarantee that tuples are time-ordered. All primary and secondary producers support continuous queries. On-demand producers do not.

Latest and history queries are *one-time* queries: they execute on the current contents of the virtual table, then terminate. In a history query, all versions of any matching tuples are returned; in a latest query, only those representing the "current state" (see 1.2.3) are returned. In both cases, a time interval may be specified with the query, to limit the age of the tuples returned. Primary and secondary producers may optionally support one-time queries. On-demand producers do not.

7.2 CONSUMER EXAMPLES

This provides examples of code you might use or adapt.

7.2.1 SIMPLE CONSUMER EXAMPLE

```
0   #include "rgma/Consumer.h"
1   #include "rgma/QueryTypeWithInterval.h"
2   #include "rgma/RGMAException.h"
3   #include "rgma/TimeInterval.h"
4   #include "rgma/TimeUnit.h"
5   #include "rgma/TupleSet.h"

6   #include <string>
7   #include <iostream>
```

```
 8  using glite::rgma::Consumer;
 9  using glite::rgma::QueryTypeWithInterval;
10  using glite::rgma::RGMAException;
11  using glite::rgma::TimeInterval;
12  using glite::rgma::TimeUnit;
13  using glite::rgma::TupleSet;

14  int main(int argc, char* argsv[])
15  {

16     try
17     {
18        TimeInterval historyPeriod(10, TimeUnit::MINUTES);
19        TimeInterval timeout(5, TimeUnit::MINUTES);
20        std::string select("SELECT userId, aString, aReal, "
21              "anInt, RgmaTimestamp FROM "
22              "default.userTable");
23        Consumer c(select, QueryTypeWithInterval::C, historyPeriod, timeout);

24        bool endOfResults = false;
25        while (!endOfResults)
26        {
27           TupleSet tupleSet;
28           c.pop(2000, tupleSet);
29           if(tupleSet.begin() == tupleSet.end())
30           {
31              sleep(2);
32           }
33           else
34           {

35               TupleSet::const_iterator tuple = tupleSet.begin();
36               while (tuple != tupleSet.end())
37               {
38                   std::cout << "userId=" << tuple->getString(0) << ", ";
39                   std::cout << "aString=" << tuple->getString(1) << ", ";
40                   std::cout << "aReal=" << tuple->getFloat(2) << ", ";
41                   std::cout << "anInt=" << tuple->getInt(3) << std::endl;
42                   ++tuple;
43               }

44           }
45           endOfResults = tupleSet.isEndOfResults();
46        }

47        c.close();
48     }

49     catch (RGMAException e)
50     {
51        std::cerr << "R-GMA application error in Consumer."<< std::endl;
52        std::cout << e.getMessage() << std::endl;
53     }

54  }
```

Lines 0–7 are the various `include` statements.

Lines 8–13 are a set of `using` statements to avoid clutter in the code.

Lines 18–23 create a consumer with a continuous query of `SELECT userId, aString, aReal, anInt, RgmaTimestamp FROM default.userTable`. In this case we have a continuous query which will start with available published tuples from the last 10 minutes and receive any new tuples published. It is possible to leave out the `historyPeriod` and use `QueryType` instead of `QueryTypeWithInterval`, in which case only newly published tuples will be received. The query has a timeout of 5 minutes. With a latest or history query we can expect the query to complete within the timeout. However the continuous query can only be terminated by an explicit abort call or by the timeout being exceeded. So we expect this query to take exactly 5 minutes.

Lines 24–46 retrieve all of the results for the query. While there is data available it will loop without delay. If there is no data currently available it will sleep for 2 seconds before trying `pop` again. The 2000 is the maximum number of tuples to be returned at once.

Lines 3543 extract individual fields from the tuple and print them.

Line 47 close the consumer.

Lines 49–53 report any R-GMA exceptions that may occur.

7.2.2 ONE-OFF QUERIES

For one-off queries, either history or latest, it is preferable to check to see if the query aborted. This can be achieved by looking to see if `c->hasAborted()` is true after either kind of consumer loop. This can be the result of hitting the timeout (5minutes in the above case) or making an explicit `c->abort()` call. Note that continuous queries *only* stop by one of these means so there is no point in checking in that case.

7.2.3 RESILIENT CONSUMER EXAMPLE

This example illustrates how to write a resilient consumer according to the recommendations in section 13. The consumer retrieves information periodically. If there is a temporary problem it retries after one minute.

```
 0   #include "rgma/Consumer.h"
 1   #include "rgma/QueryTypeWithInterval.h"
 2   #include "rgma/RGMAException.h"
 3   #include "rgma/RGMAPermanentException.h"
 4   #include "rgma/RGMATemporaryException.h"
 5   #include "rgma/TimeInterval.h"
 6   #include "rgma/TimeUnit.h"
 7   #include "rgma/TupleSet.h"

 8   #include <string>
 9   #include <iostream>

10   using glite::rgma::Consumer;
11   using glite::rgma::QueryTypeWithInterval;
12   using glite::rgma::RGMAException;
13   using glite::rgma::RGMAPermanentException;
14   using glite::rgma::RGMATemporaryException;
15   using glite::rgma::TimeInterval;
16   using glite::rgma::TimeUnit;
17   using glite::rgma::TupleSet;

18   int main(int argc, char* argsv[])
19   {
```

```
20      Consumer* consumer = 0;
21      std::string select("SELECT userId, aString, aReal, anInt FROM "
22         "default.userTable");

23      try
24      {
25         TimeInterval historyPeriod(30, TimeUnit::SECONDS);

26         while (!consumer)
27         {
28            try
29            {
30               consumer = new Consumer(select, QueryTypeWithInterval::C, historyPeriod);
31            }
32            catch (RGMATemporaryException te)
33            {
34               std::cerr << "RGMATemporaryException: " << te.getMessage() << std::endl;
35               sleep(60);
36            }
37         }

38         while (true)
39         {
40            try
41            {
42               TupleSet tupleSet;
43               consumer->pop(2000, tupleSet);
44               if(tupleSet.begin() == tupleSet.end())
45               {
46                  sleep(2);
47               }
48               else
49               {

50                  TupleSet::const_iterator tuple = tupleSet.begin();
51                  while (tuple != tupleSet.end())
52                  {
53                     std::cout << "userId=" << tuple->getString(0) << ", ";
54                     std::cout << "aString=" << tuple->getString(1) << ", ";
55                     std::cout << "aReal=" << tuple->getFloat(2) << ", ";
56                     std::cout << "anInt=" << tuple->getInt(3) << std::endl;
57                     ++tuple;
58                  }

59               }
60               std::string warning = tupleSet.getWarning();
61               if (warning.length()> 0)
62               {
63                  std::cout << "WARNING: " << tupleSet.getWarning()
64                  << std::endl;
65               }
66            }
67            catch (RGMATemporaryException te)
68            {
69               std::cerr << "RGMATemporaryException: " << te.getMessage() << std::endl;
70               sleep(60);
71            }
72         }
```

```
73      }

74      catch (RGMAException pe)
75      {
76         std::cerr << "RGMAPermanentException: " << pe.getMessage() << std::endl;
77         delete consumer;
78         consumer = 0;
79         exit(1);
80      }

81   }
```

Lines 0–9 are the various `include` statements.

Lines 10–17 are a set of `using` statements to avoid clutter in the code.

Line 20 set up initial state for the consumer.

Lines 21–22 set up the consumers select statement.

Line 25 set up the history period for the query.

Lines 26–37 create a new consumer. This uses a continuous information query starting from 30 seconds in the past so that if the consumer is restarted data should not be lost. N.B. It is possible that this may cause some tuples to be retrieved a second time if the system has to restart the consumer. If there is an RGMATemporaryException then sleep for 60 seconds before retrying to recreate the consumer.

Lines 38–72 loops forever - each iteration will retrieve data and check for warnings. While there is data available it will loop without delay. If there is no data currently available it will sleep for 2 seconds before trying again. See Section 13.4 for further explanation about the pop and sleep. If there is an RGMATemporaryException then sleep for 60 seconds before trying to pop again.

Lines 50–58 extract individual fields from the tuple and print them.

Lines 74–80 report an RGMAPermanentException. As this is considered to be a permanent problem there is no point retrying the operation.

8 Republishing via Secondary Producers

As explained in Section 1.2.2, a secondary producer populates its internal storage by running a query and the user code only sets the process running. This is demonstrated by the code in the examples below.

This code is only shown here for completeness as it is expected that most people will use the `rgma-sp` tool as explained in Section 11 which should do everything that is needed.

If the secondary producer uses a name database storage then it is important that the producer is closed reliably otherwise there may be a delay before the secondary producer can be restarted with the same database storage name.

A secondary producer must keep up with the data being published into various primary producers, and there is no client to report to that resources are running low – in particular there is no point in refusing to accept more tuples as they may then get discarded by the history retention period of the primary producer. To cope with this, a memory based secondary producer will destroy itself when it detects that it cannot function reliably.

8.1 Secondary Producer Examples

8.1.1 Simple Secondary Producer Example

```
0   #include "rgma/RGMAException.h"
```

```
1   #include "rgma/RGMAService.h"
2   #include "rgma/SecondaryProducer.h"
3   #include "rgma/Storage.h"
4   #include "rgma/SupportedQueries.h"
5   #include "rgma/TimeInterval.h"
6   #include "rgma/TimeUnit.h"

7   #include <string>
8   #include <iostream>

9   using glite::rgma::RGMAException;
10  using glite::rgma::RGMAService;
11  using glite::rgma::SecondaryProducer;
12  using glite::rgma::Storage;
13  using glite::rgma::SupportedQueries;
14  using glite::rgma::TimeInterval;
15  using glite::rgma::TimeUnit;

16  int main(int argc, char* argv[])
17  {

18     SecondaryProducer* sp = 0;

19     try
20     {
21        std::string location("cppExample");
22        Storage storage = Storage(location);
23        sp = new SecondaryProducer(storage, SupportedQueries::CL);

24        std::string predicate("");
25        TimeInterval historyRetentionPeriod(2, TimeUnit::HOURS);
26        sp->declareTable("default.userTable", predicate, historyRetentionPeriod);

27        TimeInterval terminationInterval = RGMAService::getTerminationInterval();

28        while(true)
29        {
30           sp->showSignOfLife();
31           sleep(terminationInterval.getValueAs(TimeUnit::SECONDS) / 3);
32        }

33     }
34     catch (RGMAException e)
35     {
36        std::cerr << "ERROR: "<< e.getMessage() << std::endl;
37     }
38     catch (...)
39     {
40        std::cerr << "ERROR: Cuaght unexpected error" << std::endl;
41     }

42     if (sp)
43     {
44        try
45        {
46           sp->close();
47        }
48        catch (RGMAException e)
```

```
49          {
50          }
51      }

52      delete sp;
53      sp = 0;

54  }
```

Lines 0–8 are the various `include` statements.

Lines 9–15 are a set of `using` statements to avoid clutter in the code.

Line 18 set up initial state for the secondary producer.

Lines 21–23 define the parameters and create the secondary producer. In this case we have specified a logical database (cppExample) providing support for latest queries.

Line 24–26 declare a table that the secondary producer will deal with. The predicate (line 24) is an empty string meaning that this secondary producer will collect and republish the whole table. The history retention period is set to 2 hours. This means that tuples will be available until they are 2 hours old - these tuples will be made available to continuous queries. In addition tuples will be available for latest queries. The latest retention period is a property of the individual tuple as defined at the primary producer.

Line 27 obtain the termination interval from the R-GMA server.

Lines 28–32 keep the secondary producer alive. We must issue an `showSignOfLife` call more often than the termination interval. In this case the sleep is for one third of the termination interval. It is possible that the `showSignOfLife` call may fail if it is unable to contact the service, or the service is unable to locate the remote resource. In this case the loop will be exited.

Lines 33–41 report any exceptions that may occur.

Lines 42–51 shuts down the secondary producer.

Lines 52–53 deletes the secondary producer.

N.B. A user can only have one producer using a specific storage location at a time.

Notice that if the secondary producer does fail it could be one third of the termination interval before the program detects this and exits, so you need to set the retention period and sleep parameter according to your needs.

8.1.2 Avoiding a permanent connection to a Secondary Producer

If you don't want to keep the process running continuously you could leave the secondary producer running after disconnecting from the API and then periodically run a job which reconnects. This is what the `rgma-sp-manager` tool does as explained in Section 11.4.

9 The R-GMA Command Line Tool

9.1 Introduction

The R-GMA command line tool offers simple command-based access to the R-GMA virtual database. The interface is intended to be similar to the command-line tools supplied with databases, e.g. MySQL.

9.1.1 STARTING THE R-GMA COMMAND LINE TOOL

To start the R-GMA command line tool, run the command `rgma <vdb>`, where `<vdb>` is the vdb you wish to use.

9.1.2 ENTERING COMMANDS

Commands are entered by typing at the `rgma>` prompt and hitting *Enter* to execute the command. A history of commands executed can be accessed using the *Up* and *Down* arrow keys. Commands can be entered in lower or upper case (but not a mixture of both).

Command auto-completion is supported – hit the *Tab* key when you have partly entered a command and it will either be completed automatically or a list of matching alternatives will be displayed.

9.2 COMMANDS

9.2.1 GETTING HELP

`help`	Displays the list of available commands
`help <command>`	Displays help for a specific command
`help overview`	Provides an overview of the tool
`help example`	Shows some examples

9.2.2 TABLE CREATION

First define the authorization rules you wish to impose on the table. These are in the form predicate:credentials:action see sections 2. It is possible to modify rules associated with a table after you have created it. The following rule grants read and write access to everyone.

```
rgma> rules add ::RW
```

To create a table use the standard SQL `CREATE TABLE` statement, e.g.

```
rgma> CREATE TABLE MyTable (col1 VARCHAR(50) PRIMARY KEY, col2 VARCHAR(50))
```

To modify a table use the `ALTER TABLE` command. For example to add an extra integer column “intcol”:

```
rgma> ALTER TABLE MyTable ADD intcol INTEGER
```

and then get rid of the new column:

```
rgma> ALTER TABLE MyTable DROP intcol
```

To drop a table use the `DROP TABLE` command.

9.2.3 QUERYING DATA

Querying data uses the standard SQL `SELECT` statement, e.g.

```
rgma> SELECT col1 from MyTable
```

The type of query can be changed using the `SET QUERY` command:

```
rgma> SET QUERY L
```

or

```
rgma> SET QUERY C
```

The query interval can also be specified:

```
rgma> SET QUERY C 2 minutes
```

or equivalently as the default units are seconds (other units are minutes, hours and days)

```
rgma> SET QUERY C 120
```

If a query interval is specified for a continuous query, the query will initially return a history of matching tuples up to the specified maximum age. It will then return new tuples as they are inserted.

The query timeout controls how long the query will execute for before exiting automatically.

```
rgma> SET TIMEOUT 3 minutes
```

or equivalently

```
rgma> SET TIMEOUT 180
```

9.2.4 INSERTING DATA

The SQL INSERT statement may be used to add data to the system:

```
rgma> INSERT INTO MyTable (col1, col2) VALUES ('a', 'b')
```

Data are inserted into the system using a producer component. The producers can answer continuous queries, history and latest queries.

A producer may have a predicate associated with it describing the subset of a table it provides. For example, if a table `MyTable` has the column `col` which for your producer will always have the value `me`, you can express this restriction using:

```
rgma> SET PP TABLE MyTable WHERE col1 = 'me'
```

To remove the predicate use:

```
rgma> SET PP TABLE MyTable
```

For a producer that can answer latest and/or history queries, the latest and history retention periods can be controlled using:

```
rgma> SET PP LRP 30 minutes
rgma> SET PP HRP 2 hours
```

9.2.5 SECONDARY PRODUCERS

A secondary producer does not insert new data to the system, but collects data from individual producers and makes it available via its own producer component.

To instruct the secondary producer to consume from the table `MyTable`, use the following command:

```
rgma> SET SP TABLE MyTable
```

It has an associated history retention period that may be controlled:

```
rgma> SET SP HRP 1 days
```

9.2.6 INFORMATION COMMANDS

To show a list of all R-GMA producers that produce the table `MyTable`:

```
rgma> SHOW PRODUCERS OF MyTable
```

To show a list of all table names:

```
rgma> SHOW TABLES
```

To show information about a table `MyTable`:

```
rgma> DESCRIBE MyTable
```

9.2.7 DIRECTED QUERIES

Normally a component of R-GMA called the *mediator* selects which producers are contacted to answer a query. For debugging purposes it may be useful to specify a particular producer to use instead. This is called a *directed query* and can be specified with the `USE PRODUCER` command:

```
rgma> USE PRODUCER <endpoint> <resource id>
```

All future `SELECT` queries will be directed to this producer. Only one producer may be specified. The `<endpoint>` and `<resource id>` should correspond to a valid producer that can answer the type of queries you put to it or no results will be returned. The `SHOW PRODUCERS OF` command displays endpoints and resource IDs of registered producers.

To revert back to using the mediator to select producers, use the command:

```
rgma> USE MEDIATOR
```

10 USING THE WEB TO BROWSE R-GMA INFORMATION

The R-GMA browser can normally be found on R-GMA servers. Its URL takes the form:

`https://host.domain:8443/R-GMA/`

To access the browser, a suitable client certificate needs to be imported into the Web Browser.

The R-GMA browser can be used to:

- View definitions of available tables in the schema.
- View producers that are publishing to a table.
- Pose a mediated query on a table.
- Pose a query to specific producers of a table.
- Publish data to a table.

11 RGMA-SP

The `rgma-sp` tool allows users to setup an R-GMA secondary producer without having to write any code. This is done via a configuration file, which containing a set of user configurable tables. Reasons for archiving tables in this way include:

- To allow the use of latest queries on data that is published by continuous primary producers
- To allow queries that involve joining tables published by different primary producers

11.1 CONFIGURATION

The `rgma-sp` tool reads its configuration from a file using the format:

```
<property> = <value>
```

Lines beginning with `#` are comments and are ignored.

The first "=" is significant in separating the property name and its value. The value may contain '='.

The case of the parameter names is not significant.

The following parameters are recognized in the configuration file:

`Type`	The type of consumer queries the producer supports. This is made up of a sequence of letters from the set `C` for continuous `L` for latest and `H` for history. The case of the letters is not significant. For example `type = CH`
`LogicalName`	Logical name for the database to store archived information. This database will be created, if necessary, by the R-GMA server. This must be specified.
`HRP`	Defines the history retention period for the secondary producer - in minutes.
`Tables`	Table names to archive, defined as a space separated list of table names.
`x_HRP`	HRP for table x, over-rides the value defined by HRP
`x_PREDICATE`	Predicate for table x - if not specified an empty predicate is used

11.2 An Example Configuration File

```
type = L
LOGicalName = fruity
HRP = 60
tables = userTable myTable topBanana
userTable_HRP = 120
userTable_predicate = WHERE anInt = 10
myTable_HRP = 90
topBanana_predicate = WHERE country = 'somewhere'
```

11.3 Running It

To create your secondary producer:

```
rgma-sp start id-file config-file
```

The first parameter `id-file` is the name of a file which will be created. The contents will be used by subsequent calls to identify the secondary producer on the server. The `config-file` should have the format as described above. If successful a number will be returned on stdout. This is the time (expressed in seconds) by which you must contact the secondary producer to keep it alive.

To do this use the ping call:

```
rgma-sp ping id-file
```

The `id-file` is the same file name that was specified when the secondary producer was created. If successful there will be no output. Ping must be called periodically to keep the secondary producer alive.

To terminate the secondary producer:

```
rgma-sp stop id-file
```

Again the `id-file` is the same file name that was specified when the secondary producer was created. If successful there will be no output.

11.4 Managing a Secondary Producer

The easiest way to look after pinging the secondary producer, and if necessary recreating it, is to ask the sysadmin of the R-GMA server to deposit the tested configuration file in a directory `<rgma_home>/etc/rgma-sp-manager` on the server machine. Cron jobs running there will notice new, modified or deleted files in that directory and ensure that there is a secondary producer matching each configuration file. To shut down a secondary producer simply remove its configuration file.

12 SQL

12.1 Examples of SQL Queries

To list selected information about all entries in a table called `userTable`:

```
SELECT userId, aString, FROM userTable
```

To list the entries in this table that have the answer to the meaning of life:

```
SELECT userId, FROM userTable WHERE anInt = 42
```

12.2 Supported SQL

SQL SELECT statements are restricted by 3 components: the R-GMA SQL Parser, limitations on continuous queries, and limitations imposed by external components.

Continuous queries must be of a form which can be evaluated on each tuple in isolation.

Finally the R-GMA SQL parser accepts SQL92 entry level SELECT statements except as listed below:

- Trailing decimal points in a number – e.g. 123.
- The keywords `AS` and `ESCAPE`.
- Nested `SELECT` statement after keyword `SOME`.
- The keyword `HAVING` after a table name or `WHERE` clause.
- ”,” used after `COUNT(*)` – e.g. `SELECT COUNT(*),SUM(NUMKEY) FROM UPUNIQ`
- Lack of space between elements in expression.
- Column names in double quotes.
- Expressions with 2 column name elements inside `MAX()`, `SUM()`, and `AVG()`.
- Every column name in a `SELECT` must be unique. This also implies that queries of the from `SELECT * FROM A,B` are also not allowed as the * would include the R-GMA system columns twice in the join. In the unlikely event that you need two columns in a join with the same name but with a different meaning, the R-GMA team can propose a work-around.

13 ADVICE ON USING R-GMA

This section contains various recommendations to help the user make the best use of R-GMA. The aim is to ensure the most reliable throughput of tuples and reduce unnecessary load on the servers.

13.1 GENERAL ADVICE

- All R-GMA calls raise exceptions - they must all be caught and handled for reliable use.
- Remember to close consumers and producers when you have finished with them. Otherwise they will only be closed when the termination interval has passed.
- If you get a `RGMATemporaryException` exception returned, wait for a period of one minute and retry the operation and repeat every minute. For a `RGMAPermanentException` you should investigate to find the cause.

13.2 PRIMARY PRODUCERS

- Set the latest retention period for tuples (in `declareTable`) to match the life-time for which you think the tuple should be considered to be "latest" information. Typically this will be a little greater than the publication interval.
- Set the history retention period (HRP) to hold at least the last measurement so that a new secondary producer can pick up the latest data. It should not normally be significantly longer than this. If you do make it long you may run out of memory on the service (if using memory storage). In this case you have a problem - if you just leave it, or close it, then it will continue to occupy memory until the HRP expires. The only way to clean it up is to `destroy` it. For this reason HRPs for primary producers using memory storage should normally be short: between 10 and 60 minutes. If the rate of publishing data is very high it may be necessary to have a shorter HRP.

13.3 SECONDARY PRODUCERS

- Make use of the `rgma-sp-manager` and `rgma-sp` tools to do most of the work for you.

13.4 CONSUMERS

- You may get better performance and reduce the load on the server by introducing a delay between successive calls to `pop`. This allows time for the consumer's buffer to fill up a bit rather than getting back very few tuples (or none at all) each time. Smart code would adjust the polling rate to be as low as possible to keep up with the data. A simpler algorithm would be to wait for 2 seconds after any `pop` which returns no tuples.

14 NOTES FOR VDB ADMINISTRATORS

14.1 CREATING A VDB

First identify the servers that will be used – they must all have this new version of the code (or a later one). Two or three of them should be registry instances and one the master schema. Try to choose machines on well run sites with reliable networking for the registry instances and the master schema.

Choose a name for the VDB; ideally this name will be globally unique. A server cannot serve more than one VDB with a given name. Create a VDB configuration file using the **rgma-editvdb** tool. These are the basic commands needed to create a VDB file for three servers, NB all servers in a VDB need to be included:

```
set name myVdb-domain
add host myhost1.domain.xx 8443 RS
add host myhost2.domain.xx 8443 R
add host myhost3.domain.x 8443
add rule CR
write
```

When creating a vdb you should consider what rules to define. In the case above the rule CR allows anyone to create a table or to read a table definition as no credentials have been specified. You may also wish to add a rule:

```
add rule W [DN] = <vdbAdminDN>
```

where `<vdbAdminDN>` is the DN of someone who will be able to modify or delete all table defintions.

This generates the configuration file and guarantees the correctness of the syntax. Identify a URL on a well maintained web server from which this configuration file can be downloaded – and install it there.

To deploy a VDB get sysadmins at the required sites to configure their servers to allow your VDB, see Section 15.8.

14.2 Maintaining a VDB

To modify the VDB subsequently edit the configuration file using **rgma-editvdb** adding, deleting or modifying hosts as required and make it available at the chosen URL. If a change is needed to the set of registries in use it is preferable to have some machines that remain in the set after making the change.

To close down the VDB, stop making the configuration file available via the URL - and at some stage arrange to have any old vdb_url files removed.

Evidently there will be times, in particular when a VDB is first introduced, when some of the servers will not be available, and for a period when changes are made to the VDB definition servers may have a different versions of the configuration. The system has been designed to tolerate this.

Changing master schema is potentially dangerous – the new master must have received all schema updates to avoid losing table definitions. If a freshly installed machine is defined to be the master schema it will have *no* table definitions.

15 Installation, Configuration and Maintenance

There are several configuration files on the server that sysadmin need to be aware of, as well as a number of scripts. It is assumed that `<rgma_home>/sbin/` is in your path.

15.1 Client and Server Configuration

Configuration of both clients and servers is done with the `rgma-setup` script. This includes the production of default `client-acl.txt` and `log4j.properties` files. To see all the options run it with the `--help` option.

To configure a client that is not running on the server use:

```
rgma-setup --hostname=<SERVER HOSTNAME>
```

To configure a server use:

```
rgma-setup --DBAdminPassword=<PASSWORD DB ADMIN ACCOUNT>
```

Two scripts have been provided to test the setup:

```
rgma-server-check
rgma-client-check
```

15.2 CRON JOBS

The setup script places a number of cron jobs in `/etc/cron.d`. These are:

- `rgma-check-tomcat`
  This check for problems with tomcat and restarts it if necessary
- `rgma-create-tomcat-proxy`
  This creates a proxy certificate from the host certificate, for use by tomcat
- `rgma-fetch-vdbs`
  This fetches the VDB configuration files in response to files found in `<rgma_home>/etc/rgma-server-vdb/`, see 15.8
- `rgma-sp-manager`
  This is used to run secondary producer services, see 15.9
- `rgma-sp-manager-remove-lock`
  This is used to run secondary producer services, see 15.9

15.3 LOG FILES

Log files can be found in two locations `/var/log/tomcat5/` and `<rgma_home>/log/rgma/`

15.4 CLIENT-ACL.TXT

The main change for the sysadmin is that there is now a file (`<rgma_home>/etc/rgma-server/client-acl.txt`) that is used to restrict which client machines can connect to a server. It is suggested that the sysadmin will normally restrict to only allow access from within their site. The `rgma-setup` script creates a default `client-acl.txt`, which restricts access to machines from within the same domain. Each line in the file is compared with the client trying to access the server. If a line is found that matches (ignoring case) the end of the hostname of the client then access is allowed. So an entry ".ac.uk" would allow access to clients on machines such as "example.site.ac.uk" but because of the leading dot will not allow access from "anotherexample.pac.uk". Lines beginning with a "#" are ignored. If the file is missing or completely empty then all client access is denied. If however the file contains *any blank lines* then this offers free access, as all strings end with "". This file is re-read every few minutes.

15.5 LOG4J.PROPERTIES

For logging, modify the `log4j.properties` file to meet site requirements. If it is changed, the modifications will be respected within a few minutes. The reading mechanism does not reset the existing logging configuration, so if the logging entry is added then use the keyword INHERITED to make it return the value of its parent.

15.6 GRID CA CERTIFICATES

Since R-GMA authentication is mutual, you will need to install the Certificate Authority files for the CA that signs the server certificate of any server that users wish to use. These are normally located in `/etc/grid-security/`.

15.7 DATABASE MAPPINGS

When users of R-GMA create producers with database storage and a logical name, R-GMA maps their tables onto physical tables within the R-GMA database. The mapping goes from users DN plus users logical database name plus table name to table name in the R-GMA database. It is possible to see this mapping using the script:

```
rgma-show-db --pretty <DBAdminUser>  <DBAdminPassword>
```

15.8 VDB CONFIGURATION

To enable a VDB on a site install a file into `<rgma_home>/etc/rgma-server/vdb` that contains a URL from which the VDB definition file can be downloaded. The name of the file holding the URL is the same as the name of the file to be downloaded with".xml" replaced with ".vdb_url". A cron job downloads the VDB configuration file for each URL file and stores it in `<rgma_home>/var/rgma/vdb` or removes it if the web server reports, via a "404 error" that the VDB definition has been removed.

15.9 RUNNING A USER'S SECONDARY PRODUCER AS A SERVICE

It is possible to run a user's secondary producer as a service with the aid of the `rgma-sp-manager`. Simply deposit the user's configuration file in the directory `<rgma_home>/etc/rgma-sp-manager` on the server machine. The cron jobs of the `rgma-sp-manager` will notice new, modified or deleted files in that directory and ensure that there is a secondary producer matching each configuration file. To shut down a secondary producer simply remove its configuration file.

The `rgma-sp-manager` is controlled by two cron jobs. The first runs the script that handles the starting, stopping and pinging of the secondary producers. The second script if for added resilience. It checks that the first script has not hung, if it has it will kill the process and remove the lock file.

Log files for the `rgma-sp-manager` can be found in `<rgma_home>/log/rgma/`.

16 R-GMA RELEASE NOTES

- The command line tool now supports the `ALTER TABLE` and `ALTER VIEW` commands.
- Schema authorization has been implemented.
- The `rgma-editvdb` tool has been extended to deal with schema authorization rules.

16.1 Features Withdrawn or Modified

- The old style APIs are no longer supported
- There is no special handling for the `DEFAULT` VDB
- Column names must be listed in INSERT and SELECT statements

17 Known Problems and Caveats

17.1 Known Issues

- The query interval is not fully respected for latest and history queries resulting in the return of too much data.
- A view cannot be created in the CLI.
- VDB authorization rules are not accessible to users.
- The CLI reports an error if drop table is called for a table that does not exist. This is not consistent with normal R-GMA behaviour.

17.2 Reporting bugs and getting help

If you think you have found a bug, or if you have suggestions or need help using R-GMA please feel free to send an email to: mailto:rgma-support@physics.gla.ac.uk.

REFERENCES

[1] JRA1-UK. Information and monitoring service (r-gma) system specification. Technical Report EDMS 490223, EGEE, 2004.