EGEE

R-GMA User Guide for C Programmers

Document identifier:	**EGEE-JRA1-TEC-503615**
Date:	**February 17, 2007**
Activity:	**JRA1: Middleware Engineering and Integration (UK Cluster)**
Document status:	**FINAL**
Document link:	**https://edms.cern.ch/document/503615/**

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

Document Change Log

Issue	Date	Comment	Author
1.0	31 March 2005	First release	JRA1-UK
1.1	18 April 2006	First release, first revision	JRA1-UK

Document Change Record

Item	Reason for Change
Updated consumer example Added explanation to allowed SQL on duplicate columns	To avoid hanging on isExecuting() Needed to support chunking.
Added resilient consumer and producer examples	To aid users

CONTENTS

1 INTRODUCTION **6**

1.1 PURPOSE AND STRUCTURE OF THIS DOCUMENT 6

1.2 R-GMA ARCHITECTURE 6

1.2.1 VIRTUAL DATABASE 6

1.2.2 WEB SERVICES 7

1.2.3 PRODUCERS 7

1.2.4 CONSUMERS 7

1.2.5 RETENTION PERIODS 8

1.2.6 RESOURCE FRAMEWORK AND THE TERMINATION INTERVAL 8

2 GETTING STARTED WITH R-GMA **9**

2.1 PREREQUISITES 9

2.2 SETTING UP FOR USAGE 9

2.3 SIMPLE INTERACTION USING THE COMMAND LINE TOOL 10

2.4 SIMPLE PRODUCING AND CONSUMING INFO WITH THE COMMAND LINE TOOL 11

3 R-GMA INSTALLATION **12**

3.1 INSTALLED COMPONENTS 12

3.2 CHECK YOUR INSTALLATION 12

3.3 SECURITY 13

4 PRIMARY PRODUCERS **13**

4.1 TERMINATION INTERVAL 13

4.2 PRODUCER PROPERTIES 13

4.3 PRIMARY PRODUCER EXAMPLES 14

4.3.1 SIMPLE PRIMARY PRODUCER EXAMPLE 14

4.3.2 COMPILING AND RUNNING THE EXAMPLE 15

4.3.3 RESILIENT PRIMARY PRODUCER EXAMPLE 15

5 CONSUMING INFORMATION **18**

5.1 TYPES OF QUERY 18

5.2 CONSUMER EXAMPLES 19

5.2.1 SIMPLE CONSUMER EXAMPLE 19

5.2.2 CONSUMING CONTINUOUS PLUS OLD INFORMATION 20

5.2.3 ONE-OFF QUERIES 20

5.2.4 CONSUMER EXTRACTING INFORMATION FROM RESULT SET 20

5.2.5 RESILIENT CONSUMER EXAMPLE 21

6 REPUBLISHING VIA SECONDARY PRODUCERS **23**
6.1 SECONDARY PRODUCER EXAMPLES . . . 23
6.1.1 SIMPLE SECONDARY PRODUCER EXAMPLE . . . 23
6.1.2 AVOIDING A PERMANENT CONNECTION TO A SECONDARY PRODUCER 25

7 THE RGMA COMMAND LINE TOOL **26**
7.1 INTRODUCTION . . . 26
7.1.1 STARTING THE R-GMA COMMAND LINE TOOL . . . 26
7.1.2 ENTERING COMMANDS . . . 27
7.2 COMMANDS . . . 27
7.2.1 GENERAL COMMANDS . . . 27
7.2.2 QUERYING DATA . . . 27
7.2.3 INSERTING DATA . . . 28
7.2.4 SECONDARY PRODUCERS . . . 28
7.2.5 INFORMATION COMMANDS . . . 29
7.2.6 DIRECTED QUERIES . . . 29

8 USING THE WEB TO BROWSE R-GMA INFORMATION **29**
8.1 SECURITY . . . 29

9 ADMINISTRATION **30**
9.1 TABLE CREATION . . . 30
9.2 RECOVERY FOLLOWING RESTART . . . 30

10 SQL **30**
10.1 CREATING A DATABASE AS STORAGE FOR A PRODUCER . . . 30
10.2 EXAMPLES OF SQL QUERIES . . . 30
10.3 SUPPORTED SQL . . . 31

11 ADVICE ON USING R-GMA **31**
11.1 GENERAL ADVICE . . . 32
11.2 PRIMARY PRODUCERS . . . 32
11.3 SECONDARY PRODUCERS . . . 32
11.4 CONSUMERS . . . 33

12 RELEASE NOTES FROM A USER PERSPECTIVE **33**
12.1 OVERVIEW . . . 33
12.2 NEW FEATURES . . . 34
12.3 DEPRECATED FEATURES . . . 35
12.4 FEATURES WITHDRAWN . . . 35

13 KNOWN PROBLEMS AND CAVEATS **35**

13.1 FUNCTIONALITY NOT YET IMPLEMENTED 35

13.2 OTHER KNOWN ISSUES 36

13.3 REPORTING BUGS AND GETTING HELP 36

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/glite-r1.5/ . In addition the documentation is all distributed with the software and may be found as `$RGMA_HOME/share/doc/<module>/manual.pdf`, where "module" identifies the document. Look at the directory `$RGMA_HOME/share/doc` to see the naming scheme.

Some brief release notes from a user's perspective may be found in section 12 which is useful to anyone who is familiar with the previous version of R-GMA.

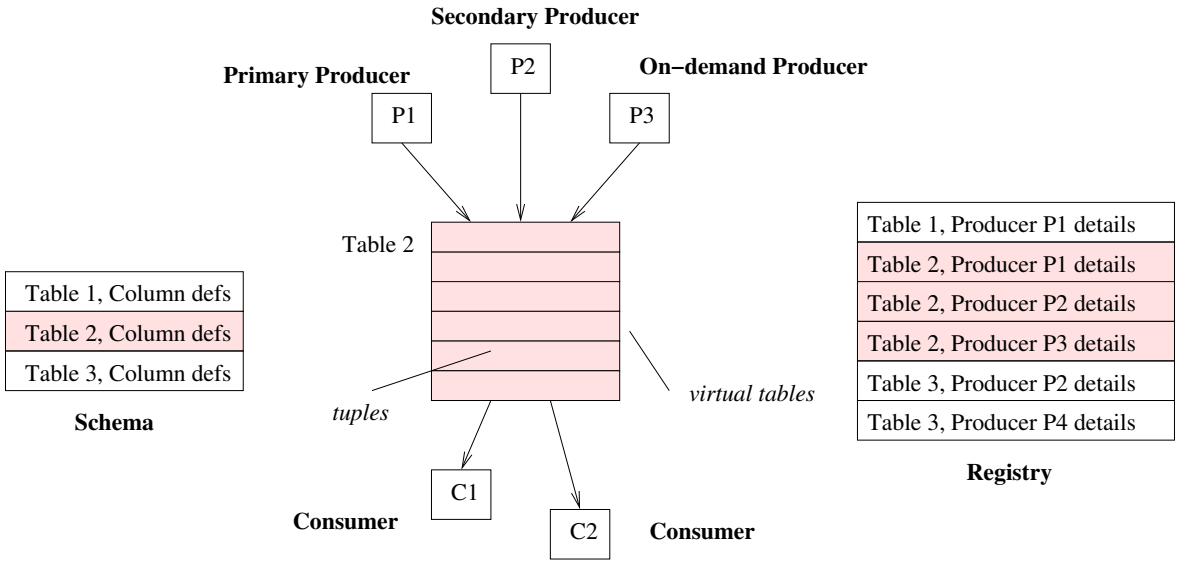
To understand more detail of what R-GMA is meant to do, you may choose to read the specification[1]. However we do not expect the average user to need the specification document.

This document 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 Global Grid Forum (GGF), which models the information infrastructure of a Grid as a set of *Consumers* (who request information), *Producers* (who provide information) and a single *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 information resources of a Virtual Organisation (VO)[1] as a single *virtual database* containing a set of *virtual tables*. As the picture above shows, a single[2] 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* (provide data for) rows for the table. A Consumer runs an SQL query against a table, and the registry selects the best Producers to answer the query in a process called *mediation*. The Consumer then 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 WEB SERVICES

R-GMA will conform to the Web Services Architecture [4]. The version currently released uses servlet technology to run the services rather than WSDL-defined Web Services. When web services are introduced the APIs will be unchanged. Each service has a well defined set of *operations* that it can carry out, as specified in a machine-readable XML document conforming to the Web Services Description Language (WSDL [5]). All operations are requested by applications through an exchange of messages with the service. We provide APIs to offer a convenient way to interact with the services.

1.2.3 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. 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. No examples of the use of the On-demand Producer are included in this document.

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[3].

1.2.4 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.

[1] Currently all VOs share the one namespace.

[2] Although there is only one logical schema and registry pair per VO, identical replicas wil be made for scalability and robustness. This is discussed in detail later. Currently this is not implemented.

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

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 TimeInterval parameter is specified, all existing tuples newer than (*now* - *TimeInterval*) will additionally be returned when the query is first 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 TimeInterval 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 LatestRetention 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 *TimeInterval*.

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.

1.2.5 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 *LatestRetentionPeriod* 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, and remains there when a tuple is re-published by a Secondary Producer. In addition, Primary and Secondary Producers declare a *HistoryRetentionPeriod* for each table to which they are publishing tuples.

Primary and Secondary Producers therefore 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 LatestRetentionPeriod, and *all* versions of any tuple which have not exceeded the HistoryRetentionPeriod.

Primary and Secondary Producers all support continuous queries, however they may not support history and latest queries.

The HistoryRetentionPeriod may be longer or shorter than the LatestRetentionPeriod. The HistoryRetentionPeriod is a (per-table) property of the producer, whereas the LatestRetentionPeriod 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. A resource contains the private data or threads associated with that particular instance (such as the tuple-storage and tuple-streamer in a Primary Producer), and it is created by an R-GMA service, when a user sends a "create" request. It resides on the server with the service, and is given an identifier which is passed back to the client. The client then includes the resource identifier with all subsequent requests relating to that instance (the API takes care of this). A resource is normally destroyed at the explicit request of the user, but in order to protect itself from an accumulation of redundant resources, an R-GMA service requires the user to specify a termination interval (in a restricted range) when it creates the resource. We plan to remove this from user control. To allow for easy migration the termination interval should be set to one hour. If the service doesn't hear from the user for any period exceeding the

termination interval, the resource is destroyed. This is the concept of *soft-state registration*. It puts the onus on the user to keep the resource alive, by making periodic contact with the service, but any contact will do. A *showSignOfLife* call is provided which does nothing but establish this periodic contact. The registry protects itself in the same way, against Producers and Consumers which register then disappear, so a periodic keep-registered message has also to be sent to the registry, by the Consumer and Producer services (this is hidden from the user).

2 GETTING STARTED WITH R-GMA

Here we present one of the simplest forms of interaction with R-GMA, where no coding is required, looking at information already in the tables and adding some very simple information using the command line tool. More information on the command line tool is in section 7 of this document, and in the command line tool document, available from the JRA1-UK web page http://hepunx.rl.ac.uk/egee/jra1-uk/glite-r1.5/ To carry out the simplest form of interaction, publishing and consuming information using the chosen language, see sections 4 and 5.

2.1 PREREQUISITES

In order to do anything in the EGEE grid, a user will need to

1. Obtain a Certificate from one of the Certificate Authorities in the EGEE PMA (see http://www.eugridpma.org/)
2. Register with a Virtual Organisation.
3. Obtain an account on a user interface machine.

Note that at present, it is possible to interact with R-GMA without being a member of a virtual organisation, as there is no authorization in R-GMA, but you will still need a certificate for authentication purposes.

2.2 SETTING UP FOR USAGE

You need to set up a few environment variables in order to use R-GMA client applications. On a gLite User Interface machine, these are defined for you by the `glite_setenv.sh` bash script. If you have not already done so, run this script by typing:

```
bash
```

followed by:

```
. /etc/glite/profile.d/glite_setenv.sh
```

at the shell's command prompt. If this is not found, try executing the `glite-env.sh` script, whose location depends on the installation. For example, enter:

```
. /afs/cern.ch/project/egee/jra1/prototype/ui/glite-env.sh
```

If this doesn't work either, see your system administrator.

You can now generate a grid proxy (for authentication) by typing:

```
grid-proxy-init
```

Note that `voms-proxy-init` can be used instead: `grid-proxy-init` will work until authorization is added to R-GMA after which `voms-proxy-init` must be used. Either way, this will generate a proxy file based on your uid and write it to `/tmp`. Check this file is present, owned by you and read/writable by you, by typing:

```
ls -l /tmp/x509*
```

You also need to tell R-GMA where to find the proxy file by defining the `X509_USER_PROXY` environment variable. The `glite_setenv.sh` script may have done this for you, but if not, type:

```
export X509_USER_PROXY=/tmp/x509up_uxxxxx
```

where xxxxx is replaced by your Unix uid.

To check the setup, type:

```
rgma-client-check
```

This should produce something like:

```
*** Running R-GMA client tests on lxplus063.cern.ch ***

Checking C API: Success
Checking C++ API: Success
Checking CommandLine API: Success
Checking Python API: Success
Checking Java API: Success

*** R-GMA client test successful ***
```

If not, see section 3 on installation or ask your system administrator for help.

2.3 SIMPLE INTERACTION USING THE COMMAND LINE TOOL

Enter:

```
rgma
```

This starts a mini shell and subsequent commands are typed into this. (Note that when using some commands table and column names are case sensitive, so it is best to assume everything is case sensitive.)

```
show tables
```

This should list the tables currently available. It is simple to show the column definitions in a table, try for example:

```
describe GlueSite
```

To see the sites enter:

```
select * from GlueSite
```

To see the services available enter:

```
select * from GlueService
```

To see the status of the various services enter:

```
select * from GlueServiceStatus
```

To see the values for specific fields enter

```
select Name,Description from GlueSite
```

To get a list of valid commands enter

```
help
```

To find out about the select command enter

```
help select
```

To exit the rgma command line interface enter:

```
exit
```

Note that you may also use ‘q’ or CTRL D to exit

2.4 SIMPLE PRODUCING AND CONSUMING INFO WITH THE COMMAND LINE TOOL

This shows how to place very simple information into one table, making it remain for at least 10 minutes and examining it. We use the pre-defined `userTable` which is available for people to use for tests. Enter the following to place the information into the table:

```
rgma
show tables
describe userTable
set producer latest
set producer LRP 10 minutes
insert into userTable values ('JoeBloggs', 'Hello', 1.23, 4)
```

Note that the values should be of the same type and column order as those in the table. Enter the following to read the data back from the table:

```
select * from userTable
```

You might see entries from other users who are also using this table. To select only your own entries you could change the query to:

```
select * from userTable where userId='JoeBloggs'
```

To exit the rgma command line interface enter:

```
exit
```

Some more examples are available within the command line tool by entering:

```
rgma
help examples
```

For more information on the R-GMA command line tool see section 7.

3 R-GMA INSTALLATION

Though this is a user guide, it is worth understanding some aspects of the installation.

3.1 INSTALLED COMPONENTS

Clients communicate with an R-GMA server. For operations using a producer or a consumer this is the local R-GMA server on the site, however some operations of the command line tool contact a registry and/or the schema. A client installation needs to be configured to know its local R-GMA site server, the set of servers running a registry and the server running a schema. Each R-GMA server has the ability to run all services, but most servers only run the producer and consumer services. An R-GMA installation is defined by its schema and registries and sites should *not* configure an additional registry except with the agreement of those having planning authority for that grid. A number of secondary producers, republishing to a database, will normally be set up to answer latest and history queries over aggregated information. An R-GMA server can also be configured to run the browser service to offer web access to R-GMA.

3.2 CHECK YOUR INSTALLATION

You should check your installation by typing "rgma-client-check". If it returns with an OK message then the system is probably installed correctly, if not you have some kind of problem and you should either check the installation as described in the installation guide linked from http://hepunx.rl.ac.uk/egee/jra1-uk/glite-r1.5/ , or ask your sysadmin to install the system correctly. [4]

The file `$RGMA_HOME/etc/rgma/rgma.conf` must contain the URLs of the R-GMA services you want the API to connect to: the standard R-GMA installation creates this for you. If you need to use an HTTP proxy server, set the environment variable `http_proxy` to contain the URL (including port number) of the proxy server.

[4] This test checks that all the four APIs are installed. If you have not installed all of them then you should expect it to report a different number.

3.3 SECURITY

R-GMA client applications require a suitable grid certificate or proxy to access secure R-GMA services (on `https://...:8443/R-GMA`). The location of the security credentials can be specified in one of two ways:

1. Setting environment variable `X509_USER_PROXY` to the location of a grid proxy certificate file (e.g. /tmp/x509up_u00000).
2. Setting environment variable `TRUSTFILE` to the location of a security properties file. An example is installed at $RGMA_HOME/etc/rgma/ClientAuthentication.props. This should contain a setting for either a `gridProxyFile` (as above) or `sslCertFile`/`sslKey`/`sslKeyPasswd` combination. You should make your own copy of this file.

Note that Java clients will need to pass in one of these environment variables to the JVM explicitly using the `-D` option.

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

4 PRIMARY PRODUCERS

4.1 TERMINATION INTERVAL

Each Primary Producer resource has a Termination Interval as described in 1.2.6, that is a time interval within which the user must make contact with the producer service, in order to keep the resource alive and maintain its table entries in the registry. The Termination Interval is set by the user when the resource is created, however this option will probably be removed from control in the future. For now please set it to one hour. If the producer's publication interval (the interval between calls to `insert`) is greater than the Termination Interval, then the user should call `showSignOfLife` periodically to keep it alive.

If the user sends a `close` request the Primary Producer will continue to be available to Consumers until all tuples are older than the *HistoryRetentionPeriod* but will no longer be contactable from the user API. 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.

The user should make allowance for the variable time it will take for any messages to reach the service and be processed. This extra time should be short as the system has been designed with the expectation that clients will contact local services. Even then, the user should always be aware that a resource he was using may have timed out when he next tries to contact it. Various recommendations on timing are provided in the "Advice on using R-GMA" section 11 of this guide.

4.2 PRODUCER PROPERTIES

All producers support continuous queries, but you may also specify in the producer properties 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, as part of the producer properties, whichever is the most appropriate. Memory gives the best performance, wheras an RDBMS gives the best performance for complex queries

- especially those requiring joins[5]. A common pattern is to use a memory based producer as the primary one and then to use a more resilient secondary producer by storing in an RDBMS. This is the example we consider here.

4.3 PRIMARY PRODUCER EXAMPLES

Here we show an example of a piece of code and how to run it.

4.3.1 SIMPLE PRIMARY PRODUCER EXAMPLE

```
 0   #include <stdio.h>
 1   #include <stdlib.h>

 2   #include "rgma.h"

 3   #define MAX_SQL 255

 4   main(int argc, char *argv[]) {
 5     RGMAResource *pp;
 6     int          historyRP, latestRP;
 7     char         insert[MAX_SQL], predicate[MAX_SQL];

 8     if (argc != 2) {
 9       fprintf(stderr, "Usage: %s <userId>\n", argv[0]);
10       exit(1);
11     }

12     if (RGMA_createPrimaryProducer(3600, 0,
13                                    RGMAStorageType_MEMORY,
14                                    NULL,
15                                    &pp) != 0) {
16       fprintf(stderr, "Failed to create producer.\n");
17       fprintf(stderr, "<%s>\n", RGMA_getException(pp)->errorMessage);
18       exit(1);
19     }

20     sprintf(predicate, "WHERE userId = '%s'", argv[1]);
21     historyRP = 3600;
22     latestRP = 3600;
23     if (RGMAPrimaryProducer_declareTable(pp,
24                                          "userTable", predicate,
25                                          historyRP,
26                                          latestRP) != 0) {
27       fprintf(stderr, "Failed to declare table.\n");
28       fprintf(stderr, "<%s>\n", RGMA_getException(pp)->errorMessage);
29       exit(1);
30     }

31     sprintf(insert, "INSERT INTO userTable (userId, aString, aReal, anInt) \
32                      VALUES ('%s', 'C producer', 3.1415926, 42)", argv[1]);
33     if (RGMAPrimaryProducer_insert(pp, insert, 0) != 0) {
34       fprintf(stderr, "Failed to insert.\n");
35       fprintf(stderr, "<%s>\n", RGMA_getException(pp)->errorMessage);
36       exit(1);
```

[5]Currently only RDBMS storage supports join operations

```
37      }

38      RGMA_freeResource(pp);
39      exit(0);
40    }
```

Line 2 is the C include file for R-GMA. It must be included to make any use of the provided R-GMA C library.

Line 5 declares a pointer to an `RGMAResource`. An `RGMAResource` is used to represent both Producers and Consumers. In this example it will be used for a Primary Producer.

Lines 12–19 create the Producer with a *teminationInterval* of 1 hour (3600 seconds), which is as the advice in section 11. The producer properties are set to zero (CONTINUOUS only). The storage type is memory and the storage location is always NULL for this type. The final argument is the address of the `resource`. Note that the function returns non-zero if there is an error. In this case a pseudo-resource is returned to allow extraction of the error message. All functions return something which can be used to check the success of an operation. The `RGMA_getException` call returns the last error which occurred.

Lines 20–30 declare the intention of publishing to the specified table. The first argument identifies the resource — this is true for all calls except the `RGMA_createPrimaryProducer`. The next pair of arguments specifies the table name and the predicate. The table must already be known in the schema and the predicate may be used to specify that you are publishing a specific subset of the complete table. The final arguments are the retention periods (see 1.2.5: the history one stating how long tuples will be kept for *continuous* and *history* queries and the latest one for *latest* queries. The error checking looks just the same as for the createPrimaryProducer. A producer is able to publish to more than one table.

Lines 31–36 construct and insert a tuple. The error handling is again the same as before.

Line 37 frees the resource. Each call to an RGMA_createXXXX should be paired with an RGMA_freeResource. In fact in this case it is not essential as the operating system will clean up for you as the process exits. Be careful when taking non-standard flows through your code to make sure that RGMA_freeResource is called for each Producer or Consumer you create, otherwise your program will suffer a memory leak.

4.3.2 Compiling and Running the example

To compile the program with gcc you might enter:

```
gcc program.c -I$RGMA_HOME/include -L$RGMA_HOME/lib -lglite-rgma-c -lssl
-o PrimaryProducerExample
```

On some operating systems you may need to explicitly link with the ”sockets” library (e.g. libwsock32.a or wsock32.lib on Win32 systems).

To run the PrimaryProducer simply enter

```
PrimaryProducerExample <UserID>
```

4.3.3 Resilient Primary Producer Example

This example illustrates how to write a resilient producer according to the recommendations in section 11. The Primary Producer publishes information periodically every 30 seconds. If there is a network problem or the buffer is full it retries after one minute, if the producer resource no longer exists a new one is created.

```
0   #include <stdio.h>
```

```
#include <stdlib.h>
#include <unistd.h>

#include "rgma.h"

#define MAX_SQL 255

int main(int argc, char *argv[])
{
  RGMAResource  *producer;
  int           historyRPSec, latestRPSec, terminationIntervalSec;
  char          insert[MAX_SQL], predicate[MAX_SQL];
  int           data;
  int           producerCreated, tableDeclared;
  const RGMAException *exception;

  if (argc != 2) {
    fprintf(stderr, "Usage: %s <userId>\n", argv[0]);
    exit(1);
  }

  producer = NULL;
  data = 0;
  sprintf(predicate, "WHERE userId = '%s'", argv[1]);
  terminationIntervalSec = 60*60;
  historyRPSec = 50*60;
  latestRPSec = 25*60;

  producerCreated = 0;
  tableDeclared = 0;

  while (1) {

    while(!producerCreated) {
      if (RGMA_createPrimaryProducer(3600, 0,
                                     RGMAStorageType_MEMORY, NULL,
                                     &producer) != 0) {
        exception = RGMA_getException(producer);
        if (exception->errorType == RGMA_REMOTEEXCEPTION) {
          fprintf(stderr, "Failed to contact R-GMA server: %s - will retry in 60s\n",
                  exception->errorMessage);
          RGMA_freeResource(producer);
          sleep(60);
        }
        else {
          fprintf(stderr, "Unexpected R-GMA error: %s - exiting\n",
                  exception->errorMessage);
          exit(1);
        }
      }
      else {
        producerCreated = 1;
      }
    }

    while (!tableDeclared) {
      if (RGMAPrimaryProducer_declareTable(producer,
                                           "userTable", predicate,
```

```
50                                              historyRPSec,
51                                              latestRPSec) != 0) {
52          exception = RGMA_getException(producer);
53          if (exception->errorType == RGMA_REMOTEEXCEPTION) {
54            fprintf(stderr, "Failed to contact R-GMA server: %s - will retry in 60s\n",
55                    exception->errorMessage);
56            sleep(60);
57          }
58          else if (exception->errorType == RGMA_UNKNOWNRESOURCEEXCEPTION) {
59            fprintf(stderr, "Producer has died - will try to create a new one\n");
60            RGMA_freeResource(producer);
61            producerCreated = 0;
62            tableDeclared = 0;
63            break;
64          }
65          else {
66            fprintf(stderr, "Unexpected R-GMA error: - exiting\n");
67            fprintf(stderr, "%s\n", exception->errorMessage);
68            if (producer != NULL) RGMA_close(producer);
69            exit(1);
70          }
71        }
72        else {
73          tableDeclared = 1;
74        }
75      }

76      if(tableDeclared) {
77        while (1) {
78          sprintf(insert, "INSERT INTO userTable (userId, aString, aReal, anInt) \
79                           VALUES ('%s', '', 0.0 , %d)",
80                  argv[1], data);

81          if (RGMAPrimaryProducer_insert(producer, insert, 0) != 0) {
82            exception = RGMA_getException(producer);
83            if (exception->errorType == RGMA_REMOTEEXCEPTION) {
84              fprintf(stderr, "Failed to contact R-GMA server: %s - will retry in 60s\n",
85                      exception->errorMessage);
86              sleep(60);
87            }
88            else if (exception->errorType == RGMA_UNKNOWNRESOURCEEXCEPTION) {
89              fprintf(stderr, "Producer has died - will try to create a new one\n");
90              RGMA_freeResource(producer);
91              producerCreated = 0;
92              tableDeclared = 0;
93              break;
94            }
95            else if (exception->errorNumber == RGMA_BUFFERFULLEXCEPTION) {
96              fprintf(stderr, "Producer buffer is full - will retry insert in 60s\n");
97              sleep(60);
98            }
99            else if (exception->errorNumber == RGMA_USEREXCEPTION) {
100             fprintf(stderr, "WARNING: Invalid tuple %s (%s)\n", insert,
101                     exception->errorMessage);
102           }
103           else {
104             fprintf(stderr, "Unexpected R-GMA error: - exiting\n");
105             fprintf(stderr, "%s\n", exception->errorMessage);
```

```
106               if (producer != NULL) RGMA_close(producer);
107               exit(1);
108             }
109           }
110           else {
111             printf("%s\n", insert);
112             data++;
113             sleep(30);
114           }
115         }
116       }
117     }
118   }
```

Line 18 initialises the data value that is going to be inserted into each tuple.

Lines 19–22 define a predicate for the producer, a termination interval of 1 hour, a history retention period of 50 minutes and a latest retention period for its tuples of 25 minutes.

Lines 25–117 loop continuously until a fatal error occurs.

Lines 26–46 create a new Primary Producer if one has not already been created. If a RemoteException occurs, the call is retried after a 60s pause. Any other exception is considered to be a fatal error and the program exits. This block loops until either the producer is successfully created or a fatal error occurs.

Lines 47–75 attempt to declare the table "userTable". This loops continuously until either the call is successful, a fatal error occurs, or the producer is found to have died (an UnknownResourceException is caught). In this case the loop exits and the program returns to line 26 to create a new producer. Again, if a RemoteException occurs, the program pauses for 60s then tries again.

Lines 78–80 create the SQL INSERT statement for this tuple. Our simple example tuples just contain the userId and the current value of the `data` variable that gets incremented on each successful iteration of the main loop.

Lines 81–115 attempt to publish the tuple. If an RGMABufferFullException is detected, we wait for 60s before trying to insert the same tuple again. If an RGMAUserException is caught, this means there was an error with the tuple we are trying to insert. In this case the program prints a warning and discards the tuple. Any other exception are handled in the same way as described above. When the insert is successful, the tuple is printed to standard output, the data variable is incremented, and the program pauses for 30s before attempting to insert the next tuple

5 CONSUMING INFORMATION

5.1 TYPES OF QUERY

There are four types of query: *continuous*, *latest*, *history* and *static* (the latter are 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.

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 1.2.4) 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 producer do not.

5.2 CONSUMER EXAMPLES

This provides examples of code you might use or adapt.

5.2.1 SIMPLE CONSUMER EXAMPLE

```
 0  #include <stdio.h>
 1  #include <stdlib.h>
 2  #include <unistd.h>

 3  #include "rgma.h"

 4  main(int argc, char *argv[]) {
 5    RGMAResource  *c;
 6    RGMAResultSet *rs;
 7    int           maxcount, endOfResults;

 8    if (RGMA_createConsumer(3600,
 9                            "SELECT * FROM userTable",
10                            RGMAQueryType_CONTINUOUS, 0,
11                            &c) != 0) {
12      fprintf(stderr, "Failed to create consumer.\n");
13      fprintf(stderr, "<%s>\n", RGMA_getException(c)->errorMessage);
14      exit(1);
15    }

16    if (RGMAConsumer_start(c, 300) != 0) {
17      fprintf(stderr, "Failed to start query.\n");
18      fprintf(stderr, "<%s>\n", RGMA_getException(c)->errorMessage);
19      exit(1);
20    }

21    maxcount=2000;

22    endOfResults = 0;
23    while (!endOfResults) {
24      sleep(5);
25      rs = RGMAConsumer_pop(c, maxcount);
26      if (rs == NULL) {
27        fprintf(stderr, "Failed to pop.\n");
28        fprintf(stderr, "<%s>\n", RGMA_getException(c)->errorMessage);
29        exit(1);
30      } else {
31        RGMA_printResultSet(stdout, rs);
32        endOfResults = rs->endOfResults;
33        RGMA_freeResultSet(rs);
34      }
```

```
35      }

36      RGMA_freeResource(c);
37      exit(0);
38    }
```

Line 3 is the include file for the R-GMA C API.

Lines 8–14 create a Consumer with a *teminationInterval* of 60 minutes (3600 seconds.) This is as recommended in section 11. The query is specified by the next 3 parameters: the SQL query string, the type of query, and how far back in time to start. In this case we have a continuous query over the userTable, and as the option RGMAQueryType_INTERVAL has not been specified, the next argument, 0 is ignored, so the query will start with the next tuples published. However, it is possible to specify how far back in time to start, see example in section 5.2.2.

The error checking follows the same pattern we have seen with the Primary Producer.

Lines 16–19 start the query running for 5 minutes (300 seconds). In fact, as this is a continuous query it should not complete in less than the specified time.

Line 21 specifies the max number of rows that are returned in one call

Lines 23–35 loop until all of the results have been retrieved, with a 5 second sleep between each call to `pop`.

Line 36 frees the resource.

5.2.2 CONSUMING CONTINUOUS PLUS OLD INFORMATION

.

If you want a continuous query starting from 10 minutes ago, you need to say it is continuous with a time interval. Replace line 10 by: `RGMAQueryType_CONTINUOUS, 600`. This is what you would need to pick up the tuple already published by the Primary Producer example.

5.2.3 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 `RGMAConsumer_hasAborted()` is true after either kind of consumer loop.

This can be the result of hitting the timeout (300 seconds in this case) or making an explicit `RGMAConsumer_abort()` call. Note that continuous queries *only* stop by one of these means so there is no point in checking in that case.

5.2.4 CONSUMER EXTRACTING INFORMATION FROM RESULT SET

You can extract specific columns from the result set either by traversing the arrays in the RGMAResultSet structure, or using the RGMA_getResultSetValue() function as shown in the fragment below. In a real application, you should beware that this function will return NULL if the requested column does not exist. Other information such as column types can be extracted directly from the RGMAResultSet structure.

```
0   for (i = 0; i < rs->numRows; ++i) {
1      userId = RGMA_getResultSetValue(rs, i, 0, "userId");
2      aString = RGMA_getResultSetValue(rs, i, 0, "aString");
3      aReal = atof(RGMA_getResultSetValue(rs, i, 0, "aReal"));
4      anInt = atoi(RGMA_getResultSetValue(rs, i, 0, "anInt"));
5      tstamp = RGMA_getResultSetValue(rs, i, 0, "MeasurementTime");
```

```
6       printf("Read: userId = %s, aString = %s, aReal = %.3f, " \
7              "anInt = %d, Timstamp = %s\n",
8              userId, aString, aReal, anInt, tstamp);
9     }
```

5.2.5 RESILIENT CONSUMER EXAMPLE

This example illustrates how to write a resilient consumer according to the recommendations in section 11. The consumer retrieves information periodically every five seconds. If there is a network problem it retries after one minutes, if the consumer resource no longer exists a new one is created.

```
0   #include <stdio.h>
1   #include <stdlib.h>
2   #include <unistd.h>

3   #include "rgma.h"

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

5     RGMAResource  *consumer;
6     RGMAResultSet *rs;
7     const RGMAException *exception;
8     int terminationIntervalSec = 60 * 60;
9     int timeoutSec = 365 * 24 * 60 * 60;
10    int oldDataSec = 30;
11    const char *select = "SELECT * FROM userTable";

12    int consumerCreated = 0;
13    int consumerStarted = 0;
14    int endOfResults = 0;

15    while(1) {

16      while (!consumerCreated) {

17        if (RGMA_createConsumer(terminationIntervalSec, select,
18                                RGMAQueryType_CONTINUOUS | RGMAQueryType_INTERVAL,
19                                oldDataSec, &consumer) != 0) {
20          exception = RGMA_getException(consumer);
21          if (exception->errorType == RGMA_REMOTEEXCEPTION) {
22            fprintf(stderr, "Failed to contact R-GMA server: %s - will retry in 60s\n",
23                    exception->errorMessage);
24            RGMA_freeResource(consumer);
25            sleep(60);
26          }
27          else {
28            fprintf(stderr, "Unexpected R-GMA error: %s - exiting\n",
29                    exception->errorMessage);
30            RGMA_freeResource(consumer);
31            exit(1);
32          }
33        }
34        else {
35          consumerCreated=1;
36        }
37      }
```

```
38      while(!consumerStarted) {

39        if (RGMAConsumer_start(consumer, timeoutSec) != 0) {
40          exception = RGMA_getException(consumer);
41          if (exception->errorType == RGMA_REMOTEEXCEPTION) {
42            fprintf(stderr, "Failed to contact R-GMA server: %s - will retry in 60s\n",
43                    exception->errorMessage);
44            sleep(60);
45          }
46          else if (exception->errorType == RGMA_UNKNOWNRESOURCEEXCEPTION) {
47            fprintf(stderr, "Consumer has died - will try to create a new one\n");
48            RGMA_freeResource(consumer);
49            consumerCreated = 0;
50            consumerStarted = 0;
51            endOfResults = 0;
52            break;
53          }
54          else {
55            fprintf(stderr, "Unexpected R-GMA error: %s - exiting\n",
56                    exception->errorMessage);
57            RGMA_close(consumer);
58            RGMA_freeResource(consumer);
59            exit(1);
60          }
61        }
62        else {
63          consumerStarted = 1;
64        }
65      }

66      if (consumerStarted) {
67        while (!endOfResults) {
68          sleep(5);
69          rs = RGMAConsumer_pop(consumer, 2000);
70          if (rs == NULL) {
71            exception = RGMA_getException(consumer);
72            if (exception->errorType == RGMA_REMOTEEXCEPTION) {
73              fprintf(stderr, "Failed to contact R-GMA server: %s - will retry in 60s\n",
74                      exception->errorMessage);
75              sleep(60);
76            }
77            else if (exception->errorType == RGMA_UNKNOWNRESOURCEEXCEPTION) {
78              fprintf(stderr, "Consumer has died - will try to create a new one\n");
79              RGMA_freeResource(consumer);
80              consumerCreated = 0;
81              consumerStarted = 0;
82              endOfResults = 0;
83              break;
84            }
85            else {
86              fprintf(stderr, "Unexpected R-GMA error: %s - exiting\n",
87                      exception->errorMessage);
88              RGMA_close(consumer);
89              RGMA_freeResource(consumer);
90              exit(1);
91            }
92          }
93          else {
```

```
 94              if (rs->numRows > 0) {
 95                RGMA_printResultSet(stdout, rs);
 96              }
 97              endOfResults = rs->endOfResults;
 98              RGMA_freeResultSet(rs);
 99            }
100          }
101        }

102        if (endOfResults) {
103          RGMA_close(consumer);
104          RGMA_freeResource(consumer);
105          consumerCreated = 0;
106          consumerStarted = 0;
107          endOfResults = 0;
108        }
109      }
110    }
```

Lines 8–11 set properties for a consumer with a one hour termination interval and a query timeout of one year. The query is specified and a 30 second interval for old data is defined so that if the consumer is restarted data should not be lost. N.B. It is possible that this may cause the some tuples to be retrieved a second time if the consumer restarts.

Lines 15–109 create an infinite loop within which a consumer is created and continuously polled for new information.

Lines 16–37 attempt to create a new continuous consumer, detecting any errors that occur. A RemoteException results in a 60 second delay followed by a retry. An RGMAException is considered to be a fatal error, and the program exits. This loops continuously until the consumer is successfully created or a fatal error occurs.

Lines 38–65 attempt to start the continuous query. Again, this section loops continuously until the command is successful or a fatal error occurs. If the consumer has died, an UnknownResourceException is detected. In this case the loop exits, the following section is skipped and the code loops back to line 16 to create a new consumer.

Lines 66–101 loop continuously while the query is running, popping data from the consumer every 5 seconds. Results are printed out as they are received. If the consumer dies or the query ends, the loop exits and a new consumer is created.

Lines 101–108 are executed if the query exits (e.g. it times out, or is aborted). It closes down the consumer and sets flags to indicate that a new consumer needs to be created.

6 REPUBLISHING VIA SECONDARY PRODUCERS

As explained in section 1.2.3, 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. Note that as this example stores information in the database, unlike the primary producer and simple consumer above, this example cannot be used without editing to specify your own local database.

6.1 SECONDARY PRODUCER EXAMPLES

6.1.1 SIMPLE SECONDARY PRODUCER EXAMPLE

```
0   #include <stdio.h>
```

```
1   #include <stdlib.h>
2   #include <unistd.h>

3   #include "rgma.h"

4   main(int argc, char *argv[]) {
5     RGMAResource       *sp;
6     RGMAStorageLocation sl;
7     int                 historyRP;

8     sl.location = "jdbc:mysql://localhost:3306/somedatabase";
9     sl.username = "fred";
10    sl.password = "bloggs";
11    sl.logicalName = NULL;
12    if (RGMA_createSecondaryProducer(3600,
13                                     RGMAProducerProps_LATEST,
14                                     RGMAStorageType_DATABASE,
15                                     &sl,
16                                     &sp) != 0) {
17      fprintf(stderr, "Failed to create producer.\n");
18      fprintf(stderr, "<%s>\n", RGMA_getException(sp)->errorMessage);
19      exit(1);
20    }

21    historyRP = 7200;
22    if (RGMASecondaryProducer_declareTable(sp,
23                                           "userTable", "",
24                                           historyRP) != 0) {
25      fprintf(stderr, "Failed to declare table.\n");
26      fprintf(stderr, "<%s>\n", RGMA_getException(sp)->errorMessage);
27      exit(1);
28    }

29    while (RGMA_showSignOfLife(sp) == 0) {
30      sleep(3300);
31    }

32    fprintf(stderr, "showSignOfLife failed.\n");
33    fprintf(stderr, "<%s>\n", RGMA_getException(sp)->errorMessage);
34    exit(1);
35  }
```

Line 3 is the usual R-GMA include file for C.

Lines 8–20 create the secondary producer. Lines 8–11 define the storage location parameters (you MUST set the logical name to NULL if you want to specify the database connection parameters instead) and lines 12–16 actually do the creation. In this case we have specified a termination interval of 3600 seconds (one hour) with a database providing support for latest queries. It is currently the user's responsibility to ensure that the specified database exists, as explained in 10.1.

Lines 21–28 declare a table that the secondary producer will deal with. Line 23 defines the table and predicate. The predicate is an empty string meaning that this secondary producer will collect and republish the whole table. The history retention period is set to 7200. This means that tuples will be available until they are 2 hours old - these tuples will be made available to both continuous and history queries. In addition tuples will be available for latest queries. In this case however the latest retention period is a property of the individual tuple as defined at the primary producer.

Lines 29–31 keep the secondary producer alive. On line 12 it was given a termination interval of 3600 seconds so here we must issue an RGMA_showSignOfLife call slightly more often than once an hour. In

this case the sleep is for 3300 seconds (55 minutes). It is possible that the `RGMA_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 32–34 print out an error message and exit. This program will continue to run for ever unless there is a problem.

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

6.1.2 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. To do this, insert a declaration:

```
RGMAResourceEndpoint *rep;
```

and replace line 29 to the end by:

```
 0     rep = RGMA_getEndpoint(sp);
 1     if (rep == NULL) {
 2       fprintf(stderr, "Failed to get Endpoint.\n");
 3       fprintf(stderr, "<%s>\n", RGMA_getException(sp)->errorMessage);
 4       exit(1);
 5     }

 6     printf("%s\n", rep->url);
 7     printf("%d\n", rep->resourceId);

 8     RGMA_freeEndpoint(rep);
 9     RGMA_freeResource(sp);
10     exit(0);
11   }
```

Lines 0–5 obtain the ResourceEndpoint. This holds everything which is needed to reestablish a connection to a remote resource.

Lines 6–7 write the URL and the resourceId to stdout. It is assumed that you will capture this output. You then need to run the following code at least every 55 minutes:

```
 0   #include <stdio.h>
 1   #include <stdlib.h>

 2   #include "rgma.h"

 3   main(int argc, char *argv[]) {
 4     RGMAResource         *sp;
 5     RGMAResourceEndpoint rep;

 6     if (argc != 3) {
 7       fprintf(stderr, "Usage: %s <url> <resourceId>\n", argv[0]);
 8       exit(1);
 9     }

10     rep.url=argv[1];
11     rep.resourceId=atoi(argv[2]);
```

```
12      if (RGMA_reconnect(&rep, RGMAResourceType_SECONDARYPRODUCER, &sp) != 0) {
13        fprintf(stderr, "Failed to reconnect.\n");
14        fprintf(stderr, "<%s>\n", RGMA_getException(sp)->errorMessage);
15        exit(1);
16      }

17      RGMA_showSignOfLife(sp);

18      RGMA_freeResource(sp);
19      printf("Done.\n");
20      exit(0);
21    }
```

Line 2 is the normal include file.

Lines 6–9 check that two arguments have been specified. These are the URL and resourceId output by the first program.

Lines 10–11 set up the ResourceEndpoint structure

Lines 12–17 reconnect using the ResourceEndpoint and send a `RGMA_showSignOfLife` call to tell the service to keep the remote resource alive.

Line 18 frees the local resource as usual.

7 THE RGMA COMMAND LINE TOOL

7.1 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.

A complete description of the command line tool is available in the users manual linked from the R-GMA documentation web page: http://hepunx.rl.ac.uk/egee/jra1-uk/glite-r1.5/ .

7.1.1 STARTING THE R-GMA COMMAND LINE TOOL

To start the R-GMA command line tool, run the command `rgma`.

On startup you should receive the following message:

```
Welcome to the R-GMA virtual database for Virtual Organisations.
================================================================

Your local R-GMA server is:

  https://yourlocal.server.machine:8443/R-GMA

You are connected to the following R-GMA Registry services:

  https://amachine.somehwhere:8443/R-GMA/RegistryServlet

You are connected to the following R-GMA Schema service:

  https://amachine.somewhere:8443/R-GMA/SchemaServlet
```

```
Type "help" for a list of commands.

rgma>
```

7.1.2 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 autocompletion 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.

7.2 COMMANDS

7.2.1 GENERAL COMMANDS

`help`	Displays general help information
`help <command>`	Displays help for a specific command
`exit`	Exit the R-GMA command line
`quit`	Same as “exit”

7.2.2 QUERYING DATA

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

```
rgma> SELECT * FROM ServiceStatus
```

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

```
rgma> SET QUERY LATEST
rgma> SET QUERY CONTINUOUS
```

The maximum age of tuples to return can also be controlled using the `SET MAXAGE` command which takes a value and a time unit (seconds, minutes, hours, days - default is seconds).

```
rgma> SET MAXAGE 2 minutes
rgma> SET MAXAGE 120
```

If a maximum age 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.[6]

To disable the maximum age, set it to `none`:

```
rgma> SET MAXAGE none
```

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

```
rgma> SET TIMEOUT 3 minutes
rgma> SET TIMEOUT 180
```

[6] Currently the use of maxage with a continuous query is not fully implemented. If any value is specified for maxage, the query will initially return a history of all matching tuples that have not expired.

7.2.3 INSERTING DATA

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

```
rgma> INSERT INTO Table (col1, col2, col3, col4) VALUES ('a', 'b', 'c', 'd')
```

Data is inserted into the system using a Producer component. All producers can answer continuous queries, but ability to answer latest and history queries is optional and is changed using the `SET PRODUCER` command[7]:

```
rgma> SET PRODUCER latest
rgma> SET PRODUCER latest history
rgma> SET PRODUCER continuous
```

The default setting is for a producer that only answers continuous queries.

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

```
rgma> SET PRODUCER PREDICATE userTable WHERE userId = 'me'
```

To remove the predicate use:

```
rgma> SET PRODUCER PREDICATE <table name> none
```

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

```
rgma> SET PRODUCER latestretentionperiod 30 minutes
rgma> SET PRODUCER historyretentionperiod 2 hours
```

7.2.4 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> SECONDARYPRODUCER MyTable
```

Like the producer, the secondary producer may be configured to answer latest and/or history queries, e.g.

```
rgma> SET SECONDARYPRODUCER latest
```

If the secondary producer can answer history queries, it has an associated history retention period, as for a producer. This is controlled in the same way:

```
rgma> SET SECONDARYPRODUCER historyretentionperiod 1 day
```

[7] Primary and Secondary producers able to answer latest and history queries have not yet been implemented

7.2.5 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
```

7.2.6 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 <url> <resource id>
```

All future `SELECT`queries will be directed to this Producer. Only one producer may be specified. The `<url>`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` command displays urls and resource IDs of registered Producers.

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

```
rgma> USE MEDIATOR
```

8 USING THE WEB TO BROWSE R-GMA INFORMATION

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.

8.1 SECURITY

To access the R-GMA Browser when it is hosted on a secure (HTTPS) server, a suitable client certificate needs to be imported into the Web Browser.

9 ADMINISTRATION

9.1 TABLE CREATION

The new `createTable` API method should now be used for adding tables to an R-GMA schema.

9.2 RECOVERY FOLLOWING RESTART

R-GMA instances (Web Service Resources) are always destroyed when the corresponding service restarts. All calls to services in R-GMA (either from the user or another service) must therefore be prepared to discover that a resource no longer exists, or is no longer registered in the registry, and to handle the error gracefully. The use of time-outs (soft-state registration) on the registry and in the producer and consumer services, ensures entries to non-existent resources are automatically removed within a reasonable length of time.

Those R-GMA services which use permanent storage (registry service, schema service and primary and secondary producers with user-specified tuple stores) do have some degree of resilience, because new resources can connect to existing storage, provided the storage itself is recoverable. In the case of a producer's file or database tuple-store, existing tuples will be automatically available, and will remain in the store until they exceed their retention period, in the usual way.

10 SQL

10.1 CREATING A DATABASE AS STORAGE FOR A PRODUCER

To create the database in the examples (somedatabase with user/password of fred/bloggs) you will need sufficient privileges to be able to create a database. You (or your sysadmin) should connect to the DBMS with the command `mysql` e.g. by entering:-

```
mysql -u root -p
```

and enter:

```
CREATE DATABASE somedatabase;
GRANT ALL PRIVILEGES ON somedatabase.* TO fred@localhost IDENTIFIED BY 'bloggs';
GRANT ALL PRIVILEGES ON somedatabase.* TO fred@"%" IDENTIFIED BY 'bloggs';
```

The first line creates the database. The other two lines create user/password combinations if they do not already exist and give "fred", logged in from any machine, full privileges on "somedatabase"

10.2 EXAMPLES OF SQL QUERIES

There are many good SQL tutorials on the web[3, 2]. Here are listed a few simple examples of working queries.

To list all rows in a table called `GlueSite`:

```
SELECT * FROM GlueSite
```

To list selected information about all sites in this table:

```
SELECT SysAdminContact, UserSupportContact FROM GlueSite
```

To list the names of all sites in this table in the Northern Hemisphere:

```
SELECT Name FROM GlueSite WHERE Latitude > 0
```

To list the endpoints and types of services in this table hosted by sites in the Northern Hemisphere:

```
SELECT Endpoint, Type FROM GlueSite, GlueService WHERE GlueSite.UniqueId = GlueService.GlueSite
```

10.3 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.

Currently the only external component which limits the SQL is the MySQL RDBMS when it is used by a producer for managing storage. As R-GMA is ported to other RDBMS systems in the future, users will be obliged to use only that dialect of SQL which is common to those RDBMSs able to participate in answering a query.

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 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.

11 ADVICE ON USING R-GMA

This section contains various recommendations, many of which relate to timing, 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

11.1 GENERAL ADVICE

- All R-GMA calls raise exceptions - they must all be caught and handled for reliable use.
- If you try to contact a producer or consumer and the server returns a `UnkownResourceException` exception the resource no longer exists so you must recreate the producer or comsumer.
- If the server cannot be contacted a `RemoteException` exception is returned. Wait for a period of one minute and retry the operation and repeat every minute.
- The termination interval should be set to 60 minutes for all producers and consumers. In future, it will probably not be possible for the user to set this parameter.
- Remember to close consumers and producers when you have finished with them. Otherwise they will only be closed, at best, an hour later when the termination interval has passed.
- Always list the columns in INSERTs and SELECTs to provide some protection from schema changes.

11.2 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.
- The primary producer service has a buffer which can fill up. If this happens, the server returns an `RMGABufferFull` exception. On receipt of this exception, the user should wait for one minute before trying to insert the tuple(s) again.
- If data are not being published at least every half an hour create a Primary Producer when it is needed rather than keep it alive. If data are published at less than half hour intervals it is better to keep the producer running.
- In general, it is not necessary to send `showSignOfLife` messages, and we recommend that you do not send them, because `insert` messages will keep a producer resource alive. If a problem should occur you can always create a new producer.
- 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 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, however this will lead to some tuples being lost if a Secondary Producer fails.

11.3 SECONDARY PRODUCERS

- Call `showSignOfLife` for a Secondary Producer to keep it alive. The interval should be 5 minutes less than the shortest History Retention Period (HRP) of the contributing Primary Producers. This way if a Secondary Producer dies it can be restarted without loss of data. If the shortest HRP is more than 60 minutes then the `showSignOfLife` should be sent every 55 minutes (i.e 5 minutes less than the termination interval).

- The HRP can be set to be large with database storage, however with memory consider how much space will be taken up by the tuples.
- gLite 1.4 includes a flexy archiver which will do most of the work for you.

11.4 CONSUMERS

- You may get better performance and will 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 5 seconds after any `pop` which returns no tuples; otherwise not include a delay.
- Do not use `showSignOfLife` but rely on the `pop` calls to keep it alive.

12 RELEASE NOTES FROM A USER PERSPECTIVE

This section describes the changes that a user will see when going from gLite 1.4 to 1.5.

12.1 OVERVIEW

This release of R-GMA in gLite 1.5 fixes more than 40 bugs. Our most serious bug number 8099: "Archivers are Inconsistent" has not been fixed completely but this release should show much better behaviour.

The version is finally free of the old EDG API layer. The APIs communicate directly with the services and are no longer interdependent. One consequence of this is much improved error messages.

APIs and services must be matched for the transition from 1.4 to 1.5 - so a site should upgrade in one go.

For C we provide a 1.4 compatibility library so that code linked against a shareable library will still work. This was not feasible for the C++, so C++ application code must be recompiled. An extra compiler switch is needed for both C and C++ compilation.

Error handling is much improved. You can now expect to receive a clear error message if there is a problem. If you don't - please submit a bug. There are three new exception types:

- RGMASecurityException
- RGMAUserException
- RGMABufferFullException

We are now checking that published data are consistent with the schema. Previously a primary producer would accept almost anything, but then a secondary producer might modify the fields - e.g. truncating strings or it could just reject the tuple. Now we make the check when you try to insert the data into R-GMA this allows us to reject the tuple at source and inform the guilty party.

In future R-GMA will support multiple Virtual Databases (VDBs). We imagine that a VO will "own" and administer more than one VDB. We had begun to introduce the framework for this but realised that we had not done it in the best way. In future the VDB will simply be a prefix to the table name, currently however, the prefix may not be specified. Users should start think about how they with to use the VDB when it is introduced.

The `reconnect` operation no longer sends a `showSignOfLife`. This is an important change of semantics. Typically if you reconnect, you will immediately use the service so you will be unaware of the change.

Finally, there used to be some non-advertised facilities for determining service status. We now offer `getProperty` operations to find out what is happening. These are of limited interest to the normal user. A `setProperty` call is also provided.

12.2 NEW FEATURES

- (LRP on insert) You can now specify a latest retention period for each tuple in the Primary Producer's `insert` call, overriding the one specified in `declareTable`. This is useful, for example, for the R-GMA Service Tool as different LRPs are appropriate for data from different services about which information is being published.
- (Logically named tuple stores) You can now access the database used by a Primary or Secondary producer just by giving it a logical name: R-GMA will map it to a physical database for you. Two new API calls `listTupleStores` and `dropTupleStores` are provided to manage them.
- (Tuple checking) Attempts to insert tuples that are inconsistent with the table definition in the schema will now be rejected.
- (numSuccessfulOps) The Primary Producer's `insertList` call may fail after inserting a number of tuples. You can now determine how far it got with the new `numSuccessfulOps` field in the `RGMAException`. It will contain the number of successfully inserted tuples.
- (endOfResults) A new method called `endOfResults` has been added to the result set returned by the Consumer's `pop` method to allow a much cleaner consumer loop to be constructed. A (Python) example looks like this:

```
consumer.start(timeout)
while 1:
    results = consumer.pop(1000)
    for result in results:
        print result

    if results.endOfResults:
        break

    time.sleep(5)
```

  The calls for the old looping constructs `isExecuting` and `count` are deprecated.
- (Registry and Schema calls) Calls to access the Registry and Schema have been added to the APIs. The most important new calls are `getAllProducersForTable` on the Registry and `createTable`, `dropTable`, `getAllTables` and `getTableDefinition` on the Schema. The temporary `create-table` command line utility has been withdrawn. Note that users should be very wary of calling `dropTable` because table definitions are shared by everyone using the Schema.

See the next section for known limitations of the current R-GMA services.

12.3 DEPRECATED FEATURES

The following features are now deprecated:

- VO-names list in `createPrimaryProducer`, `createSecondaryProducer` and `createOnDemandProducer`.
- `IgnoreSlowConsumers` property in `createPrimaryProducer` and `createSecondaryProducer`: the flag will be ignored and tuples dropped when the History Retention Period (HRP) is exceeded.
- `FILE` storage type in `createPrimaryProducer` and `createSecondaryProducer`: it was never implemented so nobody should miss it.
- HTTP protocol in URI in `createOnDemandProducer`: this was unused.
- `isExecuting` and `count` Consumer calls: use the new `endOfResults` call as described above.

12.4 FEATURES WITHDRAWN

- `setTerminationInterval` for all producers and consumers: this was deprecated in previous releases.
- `popAll` on a consumer: this was deprecated in previous releases.
- `setMetaData`, `setEndOfResults`, `addRow` and `setWarning` in the C++ API's ResultSet class: these were not deprecated previously, but should never have been made public.

13 KNOWN PROBLEMS AND CAVEATS

13.1 FUNCTIONALITY NOT YET IMPLEMENTED

Some API calls and options are not yet implemented by the R-GMA services. These are:

- LATEST or HISTORY producers with MEMORY storage.
- CONTINUOUS Secondary Producers.
- Combined LATEST and HISTORY producers.
- HISTORY and LATEST retention periods (functionality is approximated).
- Primary Producer getHistoryRetentionPeriod() method (functionality is approximated).
- Primary Producer getLatestRetentionPeriod() method (functionality is approximated).
- Secondary Producer getHistoryRetentionPeriod() method (functionality is approximated).
- The getVersion() method on all services.
- The following registry methods: createRegistry(), destroyRegistry().
- The following schema methods: createSchema(), destroySchema(), getTableIndex(), setAuthorizationRules(), getAuthorizationRules(), createIndex(), dropIndex(), createView(), dropView().
- The VDB name parameter on all registry and schema calls is currently ignored.

Schema replication is also not yet implemented.

13.2 OTHER KNOWN ISSUES

There are a number of other known bugs and limitations in the current release.

- Table and column names are currently case sensitive in R-GMA. To get consistent results always respect case.
- Continuous queries are case sensitive but latest and history queries ignore case with a default installation of MySQL.
- Only a subset of SQL92 is currently supported by R-GMA, and continuous queries are even more restricted. Time functions in SQL should not be used.
- Some operations of the command line tool try to access the schema and registry directly rather than just the local R-GMA server. This access may be blocked by a firewall.
- Use Java version 1.4.2_08 or later. There is a known issue with Java 1.4.2 running with the Scientific Linux 3 SMP kernel on dual processor machines, where the Java Virtual Machine can crash without warning. The problem does not seem to occur in RH7.3 or RHEL4.
- The Python API and the command line tool (that uses it) make a new connection to the server for each API operation. When connecting to a secure server this adds a substantial overhead, so multiple API calls (e.g. lots of inserts) will run slowly. The other three APIs are not affected.
- All VDBs share the same name space and information. Though the VDBName parameter is ignored, users are recommended to set it to an empty (zero length) string. This will facilitate the introduction of VDBs. Table names must not contain ”.” This will avoid confusion with VDBNames when they become active.
- If authentication is enabled, to use the R-GMA Browser you must load your certificate into the web browser.
- When creating a table, integer columns must be specified as type INTEGER; the abbreviation INT does not work.

13.3 REPORTING BUGS AND GETTING HELP

If you think you have found a bug, please use Savannah at: https://savannah.cern.ch/projects/jra1mdw/. Check first to see if a bug report has already been submitted.

If you want to consult a human first, or if you have suggestions or need help using R-GMA please feel free to send an email to: mailto:jra1-uk@physics.gla.ac.uk.

REFERENCES

[1] JRA1-UK. Information and Monitoring Service (R-GMA) System Specification. Technical Report EDMS 490223, EGEE, 2004.

[2] Sql tutorial. http://www.w3schools.com/sql/default.asp.

[3] Sql92 tutorial. http://www.firstsql.com/tutor.htm.

[4] Web services architecture working group note. http://www.w3.org/TR/ws-arch/.

[5] Web services description language. http://www.w3.org/TR/wsdl/.