EGEE

R-GMA Design

6.2.0

July 2, 2010

Abstract: This document is an implementation design for R-GMA

CONTENTS

1 INTRODUCTION

1.1 PURPOSE AND STRUCTURE OF THIS DOCUMENT

This document presents the more important aspects of the design for the Information and Monitoring Services middleware component (R-GMA) where it is not convenient to use comments in the code.

The various sections describe different parts of the system.

It is suggested that you read first about the Primary Producer Service in Section 3.1 and then from the Request Management Subsystem in Section 2.1 down to and including the Remote Call Subsystem in Section 11.

2 SERVICES AND RESOURCES

Package: `org.glite.rgma.server.servlets`

Package: `org.glite.rgma.server.services.Service`

Package: `org.glite.rgma.server.services.resource`

2.1 REQUEST MANAGEMENT SUBSYSTEM

An operation call on an R-GMA Service is handled by an object of class RGMAServlet which extends HttpServlet running within a Servlet Container (Tomcat). The class defines init() to set up all the services and destroy() to get rid of them as well as doGet() and doPut() to react to https POST and GET requests. Tomcat will ensure that init() is called before any call to doPost() or doGet(). Tomcat allocates an HTTP Processor Thread for each incoming request so syncronization must be considered carefully throughout the R-GMA code.

2.2 SERVICE

Each Service has operations of two kinds - those operations which are only invoked by other services (known as the System API) and those which may be invoked via an R-GMA client (the User API)

The way in which calls from one service to a remote service are made is explained in section 11.

The base class of all services is Service.

2.3 RESOURCE MANAGEMENT SUBSYSTEM

The Resource Management Subsystem is responsible for keeping Resources registered, closing them when they exceed their termination interval and managing the delayed-destroy of some resource types. The subsystem consists of two parts, a `Resource` base class for all resources, and a `ResourceManagementService` which extends `Service` (Section 2.2) handles the resource lifecycle.

As an example of its use, the `ResourceManagementService` is extended by the `ConsumerService` (section 4) and the `Resource` by the `ConsumerResource`

2.3.1 RESOURCE

A resource is identified by a ResourceEndpoint (a combination of a URL and an integer resource ID), and has a termination interval (the amount of time it will stay active for without any user contact), a creator DN (from the certificate of the user who created the resource) and a last contact time (the system time when the last user API method was called on the resource).

Resources use 'soft-state' registration and will be automatically closed and destroyed if they are not contacted by a user for a time greater than their termination interval.

A resource may be in one of 4 states:

NEW Just created but not yet ready for use

ACTIVE Ready for use

CLOSED May not be contacted via the User API

DESTROYED May not be contacted via any API

`close` puts the resource in the `CLOSED` state, likewise `destroy` puts it in the `DESTROYED` state. `canDestroy` may be overridden by the concrete implementation to specify that a resource which is closed should not yet be destroyed. For example, a primary producer which has been closed should stay alive and registered until the tuples it holds have all expired. A user may override this behaviour by calling `destroy` directly.

`updateLastContactTime` updates the resource's last contact time to the current time. A resource whose last contact time plus its termination interval is less than the current system time is expired and is liable to be closed. `hasExpired` tests if a resource has expired.

2.3.2 LocalUpdateTask

One of these timer tasks is associated with each resource. The task can be run quite frequently as it is very cheap. The code takes the form:

```
if resource has expired:  close it

if resource is closed

    if resource can be destroyed:
        destroy it
        remove resource from map and cancel task

else if resource is destroyed:  remove resource from map and cancel task
```

2.3.3 RemoteUpdateTask

One of these timer tasks is also associated with each resource. This is more costly as it accesses a registry which may be remote. This has a run method with the following code:

```
if resource not closed and not destroyed:  update the registry passing in
the registry termination interval.
```

Note that the registry termination interval is set to be slightly longer than the interval between running this task.

3 PRODUCER SERVICES

These extend the ResourceManagementService. Their primary function is to manage the producer resources.

3.1 PRIMARY PRODUCER SERVICE

3.2 SECONDARY PRODUCER SERVICE

3.3 ON-DEMAND PRODUCER SERVICE

4 CONSUMER SERVICE

Package: `org.glite.rgma.server.services.consumer`

This extends the ResourceManagementService. Its primary function is to manage the consumer resources. The consumer resource extends the `Resource` described in Section 2.3.1 in addition to the states defined by a resource it has a mode which must be one of:

NEW Not yet started

STARTED Start has been called

RUNNING Plan has been generated and producers started

FINISHED Query completed

ABORTED Query aborted

The transition from `STARTED` to `RUNNING` is made by the GetPlans task for a normal start call and is set directly within a start call for a directed query. The transition from `RUNNING` to `FINISHED` is triggered by the `pop()` call.

The ConsumerResource deals with plans from the mediator. These all require calls to the registry and so make use of the task queue. For any resource only one plan related task should be queued at any one time so a single variable is used to keep track of these tasks.

Each Consumer Resource has a Tuple Queue. The Tuple Queue is entirely private to the Consumer Resource - the Streaming Receiver Thread (see Sec.8 calls the resource's `insert()` method (defined by its Streaming Sink Interface) to add tuples, and the Consumer Service calls the resource's `pop()` method to remove them. Although the Consumer Service will authorize each call to the pop operation itself, access to the data in the Tuple Queue requires no further authorization. There is no timeout or cleanup of tuples in the Tuple Queue - the termination interval of the Consumer Resource itself is sufficient. The Tuple Queue holds a number of tuples in memory. If the maximum tuples in memory is exceeded, a group of tuples is taken off the queue and stored in a DB table with a unique incrementing ID to preserve the order. A "read" and "write" pointer to the RB table is kept in memory so that a group of tuples can be popped with a single query. Tuples are taken first from the data base table and then from memory to preserve the queue ordering.

If the tuple queue fills up, the consumer's query is aborted. It does this by calling the abort(RGMAException) on the consumer so that the exception can be stored and returned at the next pop() call.

5 REGISTRY SERVICE

Package: `org.glite.rgma.server.services.registry`

The registry service has a registry instance for each VDB hosted locally and a RemoteRegistry singleton to deal with forwarding requests to remote registry services. Each instance is responsible for dealing with its own replication and makes use of a RegistryDataBase interface. An implementation of this interface is needed for each supported RDBMS.

5.1 REGISTRY INSTANCE

The principal components of the Registry Instance are shown in the picture at the start of this chapter. When the instance is created, it creates a new instance of a Registry Database and creates and starts a new Registry Cleanup and Replication thread. The instance has a status (`OFFLINE`/`ONLINE`) set to `OFFLINE` when it is created. The registry instance is set to `ONLINE` when the initialisation is complete.

All operations other than those associated with replication have the same signature as those in the Registry Service except that the vdbName is omitted. The first part of all operations is to check the caller is authorized to call this operation.

5.1.1 REGISTRY REPLICATION

The Registry Instance is updated either by direct calls or upon receipt of replication messages. A registry "owns" those records that were last changed by direct calls and is responsible for pushing updates of these records to other registries within the same VDB. Records have an `isMaster` flag that `isMaster` flag is set by any direct add request (`getMatchingProducersForTables` or `registerProducerTable`) and is cleared by a replication message.

Records have a `lastContactTimeSec` which controls the purging of old records however it is also used to detect a delayed replication message. This is set by the registry receiving a direct call and is subsequently replicated as part of the record.

Direct change requests are always respected, whereas replicated change requests are ignored if they are out of date (i.e. before the `lastContactTimeSec` associated with the record).

If a registry should be unavailable then direct update requests will get routed to a different registry instance and records in the new registry will get the `isMaster flag` set. The original registry will initially be unaware that another registry has mastership of that record. When it receives replication messages from the new master the `isMaster flag` will be cleared.

The algorithm may result in more than one registry thinking that it is master - however this is not a problem. It also possible to have no master as the master may go down. However in all cases the system will clean up when a registry assumes mastership for the record and replicates its records.

If the clock on a registry is running too far behind time its replication messages will be ignored if the receiving registry already has an entry. If the clock is running fast then the replication message will be received and the `lastContactTimeSec` will be set in the future which will delay the purging of this record if it is not explicitly deleted. This may result in non-existent resources being returned by the registry.

Each `VDB` registry instance should log if the `lastContactTimeSec` for a record is ahead of the current time.

Note that where registry entries don't already exist, updates are treated as inserts by the Registry Database and deletions of entries that no longer exist are handled silently.

Full updates The algorithm described above assumes that all replication messages are transmitted and processed correctly. Time stamps are associated with the replication messages so that if a message has been lost the recipient will know when the message is received. In this case a full update is transmitted. This is generated by asking the Registry Database for all records with the `isMaster` flag set.

The RegistryReplicationTable A hash table is used to hold the addRegistration and deleteRegistration requests keyed on the primary key of the resource entry. This means that if several addRegistration requests are followed by a deleteRegistration request only the delete will be held.

When a request for a resource registration or deletion is received by the Registry Instance, then in addition to passing the request to the Registry Database it is inserted in the hash table.

When the Registry Instance replicate() operation is requested by the Registry Cleanup and Replication a new hash table is created to take new entries and the old one can be processed. This should be achieved by a synchronisation lock between the call which adds to the hash table and the code which switches the references.

A message is created from the hash table to be sent asynchronously to all replicas using the `addReplica` call. If at the beginning of the next cycle any messages have not been sent they are purged and a full update will be sent instead to those replicas.

If the registry is restarted it will lose its hash table. When it starts it must send a full update to all replicas. If the registry has been down for some time other registries will have taken mastership and most records will be ignored as they will be too old.

The addReplica message The addReplica message contains UTC timestamps of the last replication cycle for this instance (`lastReplicationTime`) and the current one (`currentReplicationTime`). This allows the recipient to recognise if it has missed a message and to request a full update.

For a full update the `lastReplicationTime` is set to 0 so that the recipient will accept the message as the latest.

This is followed by an add section containing entries to add and a delete section with keys to be deleted.

5.1.2 Replication Operations

This code is intended to create an `addReplicaMessage` or `fullUpdateAddReplicaMessage` only when needed. If a remote registry is down for a while repeated attempts will be made to send it an `addReplicaMessage`. Once it comes back and receives a message it will respond with `RGMAReplicaUpdateException` and next time will get a `fullUpdateAddReplicaMessage`.

Note that the replicate call has a boolean flag `full` which is set on the first call to request a full update.

Note also that when an `addReplica()` call is sent it is sent asynchronously to another replica of this VDB. It does not use the normal remote call mechanism via the `RemoteRegistry` object but sends directly to the specified replica.

5.2 Registry Database

The RegistryDatabase is an interface to implementations for different vendor specific databases with all of the SQL needed to read and update it. The implementations uses vendor specific connection wrappers in org.glite.rgma.server.services.registry. The database table definitions and the methods in this component are described here.

5.3 REMOTEREGISTRY

This is the API used to forward calls to another Registry Service. It has exactly the same interface as Registry Service but makes no use of the authz context.. It uses the Remote Call Subsystem described in section 11.

Each VDB which is not supported locally has a current registry service defined to which calls will be forwarded. If the current registry is not defined all services are pinged in parallel and the first to reply is selected as the current registry. If it turns out that the current registry is set but not reachable the current registry is cleared and a new one is sought.

6 SCHEMA SERVICE

Package: `org.glite.rgma.server.services.schema`

The SchemaService inherits from Service and owns a set of Schema Instances. Each instance holds the schema for a single VDB.

If an R-GMA server supports a VDB then it must offer a schema instance for that VDB. For each VDB one instance is the master and this is controlled by a configuration file.

All schema updates are made first to the master and then an update is requested from the master. Other slaves are responsible for getting updates from the master.

Each Schema Instance creates an instance of Schema Database to contact the underlying RDBMS. Different Schema Database implementations will be written - each peculiar to a single RDBMS to make effective use of that system. Each schema instance has its own replication thread.

Most operations are passed to the schema instance corresponding to the VDB name except for those which may modify the state of the instance. These calls are all passed via the RemoteSchema to the master schema - where they are then processed locally. Before passing the call on to the RemoteSchema the canForward flag must be checked. If the flag is not true an error must be returned. All calls to the Remote Service must have canForward set to false.

The schema instance does very little - most of the work is done by the Schema Database which it creates when it starts.

6.1 SCHEMA REPLICATION

Slaves poll the master to receive updates. The master does not know the slaves and does not hold any queues. The slaves ask for changes since a certain time and if they don't get a reply keep asking the same question on each replication loop. Eventually a reply will be received and the time stamp from the master is stored. Subsequent requests will ask for updates since the new time. To allow the master to respond correctly, major items in the schema (Table, View and Index) have a time stamp associated with them. This time stamp is set when the tuple is added or updated. Child tuples, such as the columns of a table have implicitly the same stamp. This means that if a column is removed from a table, the table and all its remaining columns will be transmitted upon request.

Note that when a Table, View or Index is dropped from the schema, a time-stamped skeleton is left behind which will then have no related columns. This makes transmitting updates very easy at the cost of a few entries in the database. Evidently a table, view or index with no columns should normally be treated as not present - except in replication operations.

6.2 SCHEMA DATABASE

The SchemaDatabase is an interface to implementations for different vendor specific databases with all of the SQL needed to read and update it. The implementations uses vendor specific connection wrappers in org.glite.rgma.server.services.schema.

7 MEDIATOR SUBSYSTEM

The mediator acts as an intermediary between the Consumer/Secondary Producer and the Registry, as illustrated in the picture below.

The Mediator is not exposed as a service.

7.1 CONCEPTS

PLAN

A Plan describes the producers a consumer needs to contact to answer its query. It consists of a list of PlanEntry structures, together with an optional warning string. An empty string is assumed to mean "no warning".

PLANENTRY

A plan entry is a query to send to a Producer. It consists of the `ProducerDetails` object, the SQL SELECT statement, a start time and an end time for the query.

PLANINSTRUCTION

A plan instruction describes a modification to a plan. There are two types:

- An `ADD` instruction adds a PlanEntry to a Plan.
- A `REMOVE` instruction removes a PlanEntry from a Plan.

Both types of instruction may have a warning which will be attached to the modified query plan if it does not already have one.

7.2 OPERATIONS

7.2.1 GETPLANSFORQUERY

This is the main mediation operation. It is called by a consumer when its query is started and is responsible for registering the consumer (for a `CONTINUOUS` query) and returning the query plan (the producers that must be contacted to answer the query).

Continuous queries

A single plan consisting of all relevant primary producers that can answer continuous queries. If the query has an associated time interval and not all the relevant producers have a sufficient HRP to respect it, a warning will be attached to the plan. Continuous queries must be *simple*, as a result only one table need be considered (although VDB union is possible, these have been removed at an earlier stage).

`getMatchingProducersForTables` is called on the registry instance for the VDB in the query. This has the side-effect of registering the consumer in the specified VDB. The matching producers are looped through and if the producer supports continuous queries and is a primary producer, added to the list. If the producer does not have a sufficiently long HRP, a flag is set to generate a warning. Each producer is sent the query.

Latest/History queries

Loop through all VDBs referenced in the query and get all matching producers for the query from the registry instance for this VDB then update the list of producers, adding producers which are not in the list and adding vdb-table details to those that are already in the list.

From the list of producers, extract a list of complete producers, i.e. those secondary producers which publish all the tables in the query, which support the query type and have no predicate (or a predicate which is complete with respect to the query). If at least one complete producer was found, form a separate plan for each one. Attach a warning to any plan if the query is a history query and the secondary producer has an insufficient HRP. Return this list of plans.

If no complete producers were found, but primary producers do exist, create a fallback plan consisting of all primary producers which publish all the tables and support the query type. Attach a warning if there are other primary producers which are not included in this list, or if there is more than one primary producer and the query is complex (since complex queries cannot be answered completely by merging results from separate producers).

If no primary producers exist, return an empty plan with no warning - there is no information in the system to answer the query so an empty plan is correct.

Static queries

Loop through all VDBs referenced in the query and get all matching producers for the query from the registry instance for this VDB then update the list of producers, adding producers which are not in the list and adding vdb-table details to those that are already in the list.

Create a plan consisting of all on-demand producers which publish all the tables. Attach a warning if there are primary producers which are not included in this list, or if there is more than one on-demand producer and the query is complex

7.2.2 REFRESHPLAN

This operation is called by continuous consumers to update their plan and reaffirm their registration. It checks for any new relevant producers at the same time. In principle the consumer should get notification directly from new producers, but this method provides a fallback solution if this message is lost.

Get all matching producers for the query from the registry instance for the VDB. This has the side-effect of re-registering the consumer.

If the plan consists only of secondary producers, continue to the next VDB, ignoring the list of returned producers.

If the plan contains primary producers, check that each returned continuous primary producer is included in the plan (either directly or via a secondary producer). For any that are not included, create an 'add' instruction for the producer. Attach a warning to the instruction if the producer's HRP is insufficient to answer the query.

7.2.3 ADDPRODUCERTOPLAN

This is called when a continuous consumer receives notification from a new, potentially relevant producer.

If the producer is relevant and a primary producer and if the producer is not already covered by the plan (by a secondary producer) then return an 'add' instruction for the producer. Attach a warning to the instruction if the producer's HRP is insufficient to answer the query.

7.2.4 REMOVEPRODUCERFROMPLAN

This is called when a consumer receives notification that a potentially relevant producer no longer exists, or when any kind of consumer determines that a producer in its current plan is not functioning. Both these forms of notification are assumed to be more reliable than whether or not the producer exists in the registry.

If the producer is currently part of the plan and is a primary producer return a 'remove' instruction for the producer.

If the producer is currently part of the plan and is a secondary producer we are likely to need a new plan so:

Call `getPlansForQuery` to obtain a list of new plans. Remove any plans that include the removed producer (it may still be in the registry but we trust an explicit removal more).

For each plan, form a set of add/remove instructions that will change the current plan into the new one. Choose the smallest set of instructions that results in a new plan with no warning. If all the new plans have warnings, choose the smallest set of instructions.

7.2.5 CLOSEPLAN

If the query is continuous call `unregisterContinuousConsumer` on the registry for the VDB referenced in the query.

8 QUERY ANSWERING SUBSYSTEM

8.1 PRINCIPAL COMPONENTS

This subsystem handles the interaction between producer and consumer services required to answer queries. It covers the entire streaming process that starts (for a query) just after a producer receives a start call from a consumer and ends either when all tuples have been sent, or the query is aborted. It also covers timing out of queries and streaming connections.

8.2 STREAMING PROTOCOL

Every new streaming connection that is opened must begin with a 4 byte header containing an integer identifying the streaming protocol being used. This is the negative of a version number of the streaming protocol.

The header must be sent at the start of every new streaming connection but not in between result sets. If a connection is broken and has to be reopened, the reopened connection must send the header bytes again.

A StreamingProtocol is a system for encoding result sets into a byte stream and decoding them from a byte stream. It has the methods: getTupleEncoder and getTupleDecoder.

A tuple encoder takes result sets and encodes them into a byte stream suitable for streaming. It also provides a `getHeader` method to retrieve the 4 byte header which must begin the streaming message. It encodes complete result sets only via the `encode` method.

A tuple decoder takes a byte stream and decodes it into result sets. The byte stream is taken directly from the streaming connection but should **not** include the 4 byte header. Since the streamed bytes may arrive in chunks which do not correspond to complete result sets, the TupleDecoder has two methods: `pushBytes` to read streaming data, and `popResults` to return complete result sets. `popResults` should normally be called after each call to `pushBytes` and may return zero, one or multiple result sets. Once a chunk of bytes has been pushed to a TupleDecoder it must either decode them into a complete result set or store incomplete data internally. The original data passed to `pushBytes` may be discarded or overwritten.

8.3 PRODUCER SIDE

STREAMINGSENDER

The streaming sender is a single thread which iterates over a set of StreamingSource objects, each of which is associated with a single streaming connection. For each StreamingSource, it calls `popBytes`, passing a buffer to be filled with data to send. The size of this buffer defines the maximum amount of data that may be sent with each iteration and is analagous to the chunk size in the StreamingReceiver (but again is unrelated to the result set chunk size). The bytes returned are written to the socket connection.

If a socket connection fails, the `reset` method is called on the associated StreamingSource. This triggers the StreamingSource to re-send the header bytes, and possibly re-send bytes corresponding to an incomplete result set.

Producers call `addQuery` to register a new RunningQuery. If a StreamingSource object already exists to the target consumer using the same streaming protocol (new protocol only), the query is added to this StreamingSource, otherwise a new StreamingSource is created for this query.

STREAMINGSOURCE

The StreamingSource is associated with a single streaming connection. It has one or more RunningQuery objects which it pops results from to be sent on the streaming connection. It is instantiated with a particular StreamingProtocol.

With each call to popBytes, the StreamingSource attempts to fill the supplied buffer with bytes. If the header bytes have not yet been sent, these are sent first. Then, the StreamingSource iterates through its RunningQuery object, popping a result set, encoding it to bytes using its TupleEncoder, and writing the bytes to the send buffer. When the send buffer is full, it is returned and the encoded bytes from the current result set are saved. The next time `popBytes` is called, remaining bytes from this result set are sent before moving on the the next RunningQuery.

If a RunningQuery returns a result set with the `endOfResults` flag set, it is deleted from the list. If the StreamingSource has no RunningQueries, it returns `false` to `popBytes`, indicating to the StreamingSender that the connection may be closed. To avoid connections being closed at the same time as new queries being added, the `addQuery` and `popBytes` methods must be synchronized and the StreamingSource must set an internal flag rejecting any further calls to `addQuery` before returning `false` to `popBytes`.

If the StreamingSender calls `reset`, the StreamingSource takes the following actions:

- It sets a flag so that the header bytes are sent again on the next call to `popBytes`.
- It resets the send position on the byte buffer containing the current result set so that this result set is sent again from the beginning.

This method does not need to be synchronized with `popBytes`, since both methods are only ever called by the main thread in the StreamingSender.

RUNNING QUERY

A Producer Resource (of any type) has a set of Running Query instances, one for each consumer to which it is currently streaming. This holds a Tuple Cursor and details of the Consumer to which it is streaming. It enforces query timeouts irrespective of whether notification is received from the Consumer.

A RunningQuery is associated with a single StreamingSource object which represents the connection (although it may not be the only RunningQuery using this StreamingSource). It pops tuples from its cursor in response to pop requests from the StreamingSource.

8.4 CONSUMER SIDE

STREAMINGRECEIVER

The streaming receiver is a singleton which listens on the streaming port. It receives connections from producers and forwards the associated byte stream to a StreamingSink object, one of which is created for each connection that is accepted. The byte stream is forwarded in chunks of a configurable size. This block size does not affect the parsing of the stream or the chunk size of the results (number of tuples per result set), it is purely an optimization parameter and is likely to be fixed at a reasonable value initially.

The StreamingReceiver has a list of RunningReply objects which receive tuples for consumers. It passes this list to each new StreamingSink so that it can identify the correct reply for each decoded result set. Consumers register their queries by calling `addReply` for each producer they send a query to. When a

consumer is destroyed or wishes to abort a query, it calls `removeReply` to remove relevant RunningReply objects from the streaming receiver.

The StreamingReceiver also maintains a list of IP addresses of producers which it expects to be contacted by. If a new connection is received from a host whose address is not in this list, the connection is dropped immediately with no response. This is intended to be a basic security precaution to block attacks from machines which are not R-GMA servers, however it does not protect against malicious connections from machines registered in the Registry as genuine R-GMA servers.

STREAMINGSINK

A streaming sink is created for each connection accepted by the StreamingReceiver. It handles the stream of bytes as they arrive, initially reading the header and instantiating the appropriate TupleDecoder for the requested streaming protocol. Once this has been done, it passes the byte stream to the decoder and pops the decoded result sets. For each result set, it locates the appropriate RunningReply object and pushes the results on to it.

Location of the appropriate RunningReply is done using the `matches` method which tests if a reply is relevant to a query intended for a specific consumer, from a specific producer in response to a specific query. The first matching RunningReply found receives the tuples. For connections from older R-GMA servers, the metadata included in the result set is limited, in general only the recipient resource ID is specified. To accommodate this, Consumers should submit a special RunningReply object which only matches results with this limited information and will receive all tuples from old producers using the Classic streaming protocol.

Location of the appropriate RunningReply object must be synchronized with the `addReply` method from the StreamingReceiver. This is done by synchronizing on the list object. This should not be a performance problem since only one StreamingSink will try to synchronize at a time (because the StreamingReceiver is single-threaded), and neither will hold the lock for a long time.

RUNNING REPLY

A Consumer Resource has a set of Running Reply instances, one for each producer from which it is currently streaming. Its purpose is to encapsulate the consumer's connection to the producer which may be tested (using `testProducer`) or aborted. It receives result sets from one or more StreamingSink objects and generally adds them to the consumer's tuple stack, although it may ignore them if the query has been aborted or timed out.

9 SQL PARSING AND VALIDATION SUBSYSTEM

Packages: `org.glite.rgma.server.services.sql` and `org.glite.rgma.server.services.sql.parser`

9.1 SQL PARSING

SQL strings musct conform to the R-GMA System Specification document.

A recursive descent parser is generated from a grammar which is processed by JavaCC.

The parsing subsystem contains a class for each different supported SQL statement. Each class has a static parse() method using to create a statement object from an SQL String and setter/getter methods to access the components of the statement. In each case the parse method instantiates a JavaCC based parser with the statement to parse and returns the result which the JavaCC parser has built.

In addition there are a number of utility classes for SQL constructs which are not statements in their own right but provide appropriate get and set methods.

The supported SQL statements and their uses are:

SelectStatement	Represents all SELECT queries that the user can make on R-GMA
InsertStatement	Represents all INSERT statements that the user can pass to R-GMA
UpdateStatement	Represents all UPDATE statements that the system generates to update Latest tables in the Tuple Store Database
CreateTableStatement	Represents all CREATE TABLE statements that the user can pass to R-GMA
CreateViewStatement	Represents all CREATE VIEW statements that the user can pass to R-GMA
CreateIndexStatement	Represents all CREATE INDEX statements that the user can pass to R-GMA
ProducerPredicate	Represents all producer predicates that the user can pass to R-GMA when declaring a table. These are far simpler than the predicate/WHERE clause allowed within a SELECT statement

9.2 SCHEMA VALIDATION

A Validator class is provided to check that a SelectStatent is consistent with the schema. This is quite complex as a query can span more than one VDB.

This class has a constructor taking the SelectStatement and table definitions from the schema. It has one operations: validate() to perform the validation. It checks that:

- SELECT * is not used in conjunction with a join.
- all column names are only specified once (this is necessary for the creation of temporary tables in one-time queries and is not actually invalid SQL).
- all tables and columns referenced in SELECT, FROM and WHERE exist in the schema tables specified.
- the types of expressions in the predicate match.

10 TASK MANAGEMENT SUBSYSTEM

Package: `org.glite.rgma.server.services.tasks`

10.1 OVERVIEW

A task is used for a sequence of operations that can be queued to run at some time in the future. A task will be tried and if it fails may be retried as explained below. The tasks are processed by threads in a pool. Each task has a key value that is associated with the resource that the task uses. This key is used to ensure that the finite thread pool does not have many threads trying to make use of a malfunctioning resource.

When tasks are submitted to the `TaskManager` they are added to a queue to await processing. The `TaskManager` has a pool of `TaskInvocators` which periodically pop a task off the queue to process it. The processing algorithm establishes a set of "good" keys. Only if a key is in the good key set will more than one task be run at a time with that key. If the tasks key is not in the goodKey set and any other `TaskInvocator` is using that key the Task is re-queued. Only when an invocation is successful on the *first* attempt is the key added to the goodKey set. If a task fails its key is removed from the goodKey set. If a task returns a `HARD_ERROR` then it and all other tasks on the queue with the same key will all be removed from the queue.

To ensure that threads are always available to process tasks using resources that are performing well, some threads will be configured to only take tasks with a key in the goodKey set.

The use of timers is made to ensure that a task does not run indefinitely. There is a timer in the `TaskInvocators` that will interrupt a task if it over runs. There is another timer in the `TaskManager` that will abandon a `TaskInvocator` and create a new one if necessary.

An algorithm has been devised to ensure the `TaskInvocators` do not continually loop furiously through the queue when the only tasks in the queue are tasks that cannot currently be run.

The **user** of the `TaskManager` must provide a class that extends the `Task` class and provide an implementation of the `invoke()` method. The user should set a sensible value for the maxRunTimeMillis in the Tasks constructor; this is the time period that the task will be allowed to run before the `TaskManager` considers the task to have hung and interrupts it. A value for the maximum attempt count also has to be included, it is recommended that this is set to 2. An instance of the extension of the `Task` needs to be added to the singleton `TaskManager`. *The user should keep a reference to the task if he wishes to interrogate it*. On completion of the task, the tasks extension should ensure that it returns `SUCCESS`, `SOFT_ERROR` or `HARD_ERROR`. If `SOFT_ERROR` is returned the `TaskManager` will re-queue the task for retry.

The task is created with a zero `attemptNumber`. After each time it is invoked the `attemptNumber` is incremented by the `TaskInvocator`. It is the responsibility of the Task's extension to make use of the `attemptNumber` and use an appropriate strategy to achieve success or to return a `HARD_ERROR` after a finite number of attempts.

The `TaskManager` also offers a look up mechanism against an `ownerId` provided by the user when creating the task. This ownerId associated with the task is not unique and is intended for diagnostics.

10.2 TASK

Task is an abstract class with an `invoke()` method that must be implemented to carry out the task. The derived class must encapsulate all data and functionality required to support the `invoke()` call. The constructor of the derived class must call the superclass constructor with the owner id, key and maximum run time milliseconds as arguments.

If the task is still running after the `maxRunTimeMillis` then it will be sent an interrupt signal from the `TaskInvocator`. Upon receiving an interrupt the task should exit gracefully. Once an interrupt is sent the task will be regarded as having returned a hard error irrespective of the value it returns.

Task(ownerId, key, maxRunTimeMillis) Public constructor that takes the specified ownerId and key as strings, maxRunTimeMillis as a long and maxAttemptCount as an int.

```
ownerId - owner associated with the task, intended for diagnostics

key - an identifier of the resource to be used, i.e.  a URL

maxRunTimeMillis - time period in milliseconds after which the task will be
interrupted

maxAttemptCount - the number of attempts that the TaskInvocator will attempt
a task before it sets the tasks ResultCode to textttHARD_ERROR


attemptNumber = 0
```

abstract invoke() Invokes this task returns a status code (`SUCCESS`, `SOFT_ERROR` or `HARD_ERROR`).

abort() This is a public method that marks the task as aborted by setting the status code to `HARD_ERROR`, only if the current status is `NULL` or `SOFT_ERROR`.

getCurrentAttemptNumber() This is a protected method that returns the number of times that the task has been invoked.

getResultCode() This is a public method that returns the result status. This may be called at any time. When a task is created the status is NULL. Return values may be: `SUCCESS`, `SOFT_ERROR`, `HARD_ERROR` or `NULL`.

setResultCode(Result) This is a *synchronized* package method used to set the result status. If the result status is already `SUCCESS` or `HARD_ERROR` then the status is not updated.

statusInfo() This is a public method that returns a map of status information, used for monitoring purposes.

toString() This is a public method that returns the key and owner id of the task.

10.3 TASKMANAGER

This class is a singleton. It uses data structures and threads as indicated in the constructor.

Constructor is private, throws RGMAConfigurationException.

> create the task queue - a linked list for tasks needing to be run.
>
> create the goodKey set - a *synchronized* hash set for keys that are working well.
>
> create the currentTasks map - a *synchronized* hash map keyed on the task.key giving the count of the TaskInvocators currently processing tasks with that key.
>
> create a pool of TaskInvocators. Some of these are able to deal with any task and some only with tasks with key in the goodKey set. The values are taken from a configuration file containing the total number of threads and the number for good only threads. There must be at least one TaskInvocator that can invoke any task.
>
> create a taskInvocatorDataList - a *synchronized* hash map keyed on the TaskInvocator.Thread.currentThread. These data are used by the InteruptHangingTaskInvocators timer.
>
> create a firstRejectedTaskAnyInvocator queue - a linked list that will contain null or only one entry, the first task encountered that an 'Any' TaskInvocator cannot process due to the tasks key.
>
> create a firstRejectedTaskGoodOnlyInvocator queue - a linked list that will contain null or only one entry, the first task encountered that a 'Good Only' TaskInvocator cannot process due to the tasks key.
>
> create InteruptHangingTaskInvocators timer - a timer thread that will check for hung TaskInvocators.
>
> initialize lastTaskTime, successfulTasks and failedTasks.

getInstance() - *synchronized* - public static method that returns the singleton instance - creating it if necessary.

add(task) This is a public method that adds a task to the queue. Requires *synchronization* on the task queue.

getGoodKeys() This is a public method that returns a list of keys designated as good keys by this TaskManager.

getNotGoodKeys() This is a public method that returns a list of keys not designated as good keys for queued and running tasks. Requires *synchronization* on the task queue.

getQueuedTasks() This is a public method that returns a list of tasks that are currently queued awiating execution. Requires *synchronization* on the task queue.

getTasks(ownerId) This is a public method that returns a list of queued tasks, and their status, that have been added to the `TaskManager` by the specified owner id. Requires *synchronization* on the task queue.

getTasksPerKey() This is a public method that returns a map of number of queued and running tasks, associated with each key, keyed on the task.key. Requires *synchronization* on the task queue.

statusInfo() This is a public method that returns a map of the status information where each property serves as a key pointing to its value.

10.3.1 INTERUPTHANGINGTASKINVOCATORS

This is a timer thread created by the TaskManager that will check for hung TaskInvocators. It will try and clean up a hung `TaskInvocator` and before recreate new one.

Constructor(threadInterruptDelayMillis) A time period in milliseconds, used to give the `TaskInvocator` a chance to clean up a task, after the task has timed out, before clobbering the `TaskInvocator`.

10.4 TASKINVOCATOR

Pops a task from the queue and processes it.

Constructor The constructor takes seven arguments: currentTasks map, firstRejectedTaskAnyInvocator, firstRejectedTaskGoodOnlyInvocator, goodKey set, goodOnlyInvocator which is a boolean which if true causes the thread to only process tasks with a key in the goodKey set, taskInvocatorDataList and task queue

Access to all queues and sets needs to be synchronized. Note that it is important to give up locks as quickly as possible - and not to hold locks while a task is being invoked.

run

```
while not interrupted:
    synchronize on queue:
        if queue is empty:  queue.wait()
        pop task from queue
    if task not aborted:  # i.e.  not HARD_ERROR
        synchronize on currentTask map
            if (task.key in goodKey set) or
            (not goodOnly TaskInvocator and task.key not in currentTask map):
              add task.key to currentTask map
              set process flag
        if not process flag:
            requeue the task
        else:
            increment task.attemptNumber
            start timer # send interrupt to task after maxRunTimeMillis
            result = task.invoke
            update lastTaskTime
            if task interrupted:  result = HARD_ERROR
            if task.getResultCode aborted:  # i.e.  HARD_ERROR
              remove task.key from currentTask map
            else:
              task.resultCode = result
              if result = SUCCESS:
                if task.attemptNumber = 1:  add task.key to goodKey set
                else:  remove task.key from goodKey set
                remove task.key from currentTask map
                increment successfulTasks
              else if result = SOFT_ERROR:
                remove task.key from goodKey set
                remove task.key from currentTask map
                if not task aborted:  requeue the task
              else:  # HARD_ERROR
                remove task.key from goodKey set
                removeSimilarTasks
                remove task.key from currentTask map
                increment failedTasks
```

10.5 DEALING WITH HANGING TASKS

Tasks are added to the task queue via the `TaskManager` and run via one of many `TaskInvocators`. If the task fails to complete in the allocated time (`task.maxRunTimeMillis`) a timer task, created by the `TaskInvocator`, will try and interupt the task. If successful in interupting the task the `TaskInvocator`

will set the the ResultCode to HARD_ERROR. All other tasks on the task queue with the same key will also have their ResultCodes set to HARD_ERROR.

The InteruptHangingTaskInvocators timer task of the TaskManager periodically checks the taskInvocatorDataList for TaskInvocators running tasks that have been running for task.maxRunTimeMillis + InteruptHangingTaskInvocators.threadInterruptDelayMillis, such TaskInvocators are deemed to have hung. The InteruptHangingTaskInvocators will set the the tasks ResultCode to HARD_ERROR and all other tasks on the task queue with the same key will also have their ResultCodes set to HARD_ERROR. *N.B.* A new TaskInvocator will then be created to replace the hung one. If the old TaskInvocator regains control from the task it will notice that it no longer has an entry in the taskInvocatorDataList and die gracfully.

10.6 AVOIDING TASKINVOCATORS FURIOUSLY LOOPING AROUND THE TASK QUEUE

Under certain conditions it would be possible for a TaskInvocator to loop furiously around the task queue. This could happen if none of the tasks in the task queue had keys in the goodKey set and there was one occurrence of each queued task.key already being executed by other TaskInvocators. It would also occur if the TaskInvocator was a goodOnlyInvocator and none of the tasks in the queue had keys in the goodKey set.

In order to guard against this a record is kept of the first task encountered which could not be run and was put back onto the queue. When any TaskInvocator gets back round to this 'first' rejected task it will go to sleep. At any point when a task is added to the queue, removed from the queue for processing, or removed from the currentTasks map, then the record of the first rejected task will be set to null and a notification sent to the sleeping TaskInvocators. As the TaskInvocators also have to sleep when the task queue is empty the same wake up trigger is used.

Further complication is introduced as there are two types of TaskInvocators 'Any' and 'Good Only'. Therefore, two variables are needed to record the 'first' rejected task for the TaskInvocators, firstRejectedTaskAnyInvocator and firstRejectedTaskGoodOnlyInvocator.

11 REMOTE CALL SUBSYSTEM

For each service to be called a class is provided which has only static methods and which is responsible for generating the http(s) request to the remote service and unpacking any results. These classes throw exceptions as needed but do not themselves try to repeat calls if they fail (see section **??**) Three subexceptions of `RemoteException`: `ConnectTimeoutException`, `WriteTimeoutException` and `ReadTimeoutException` provide more information in the event of the remote connection failing for network reasons.

Each class should have methods for the system interface and for those user calls which may be made by other services.

The class should also detect any calls that are local to the machine and make a direct call. This both saves time and facilitates testing.

Remote calls are inherently unreliable, so as far as possible, remote calls should be inside tasks managed by the task management system described in section 10.

12 LOGGING

The R-GMA server uses the log4j package to manage logging. Messages are written to a named logger at a specified severity level and output to one or more logging files.

12.1 SEVERITY LEVELS

Logging messages on the server are categorized using the log4j severity levels `FATAL`, `ERROR`, `WARN`, `INFO` and `DEBUG`. Messages should be assigned to a level according to the following guidelines:

`FATAL`	The server is shutting down.
`ERROR`	Either an unanticipated failure (i.e. a bug in R-GMA), or an anticipated failure that may result in loss of information from the system (e.g. database failure, tuple not streamed).
`WARN`	Anticipated failure (e.g. firewalled connection, network error) or a failure caused by bad user input (e.g. invalid tuple, attempt to contact expired resource).
`INFO`	Reporting of an operation executed by the server. The level of detail should be approximately the same as the specification and where appropriate use the same terminology. Any given operation should only be reported once at this level (after completion).
`DEBUG`	Reporting of an implementation-level operation. This may include details beyond what is described in the specification and an operation may generate more than one message at this level.

12.2 GENERAL GUIDELINES

- Stack traces should not be logged unless the exception was unanticipated.
- Don't log a warning and throw an exception for the same error. Exceptions should be logged when they are caught if they cannot be handled without error.
- Log messages should be brief but include as much relevant information as practical.
- Completed operations should be logged rather than entry/exit into particular methods.
- When concatentating more than two strings for an `INFO` or `DEBUG` message, the log4j methods `isDebugEnabled()` and `isInfoEnabled()` should be used to avoid the cost of constructing the message.