Project no. 004758

GORDA

Open Replication of Databases

Specific Targeted Research Project

Software and Services

In-Core Proof-of-concept

GORDA Deliverable D4.3

Due date of deliverable: 2008/03/31
Actual submission date: 2008/04/21
Resubmission date: 2008/05/28

Start date of project: 1 October 2004 Duration: 42 Months

FCUL

Revision 1.1

Project co-funded by the European Commission within the Sixth Framework		
Dissemination Level		
PU	Public	X
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Contributors

Alfrânio Correia Jr., U. Minho
Susana Guedes, U. Lisboa
José Pereira, U. Minho
Luís Rodrigues, U. Lisboa
Luís Soares, U. Lisboa
Nuno Carvalho, U. Lisboa
Paulo Jesus, U. Minho
Rui Oliveira, U. Minho

Abstract

This document describes the mapping of the GORDA Architecture and Programming Interfaces to the open-source Apache Derby and MySQL database management system. It aims at providing an example of an in-core implementation, thus reflecting current experience and roadmap for ongoing work on the prototype.

Contents

Chapter 1

Scope

This Chapter describes the scope of the Derby (Derby/G) and MySQL (MySQL/G) prototypes, regarding its objectives and the components of the GORDA Architecture and Programming Interfaces actually being implemented.

1.1 Apache Derby

1.1.1 Objectives

The in-core mapping of the GORDA Architecture and Programming Interfaces (GAPI) to the open source Derby database management has the following goals:

- To allow the implementation of a wide range of group communication based database replication protocols thus requiring an extensive mapping of the GAPI, in particular, in the storage phase.
- To allow full compatibility with all Derby clients interfaces and tools.
- To demonstrate the feasibility of efficiently extracting read-sets.
- Modifications to the core Derby source code, although unavoidable, should be as little intrusive as possible in order to facilitate dissemination by proposing patches for acceptance by the Derby open source community.

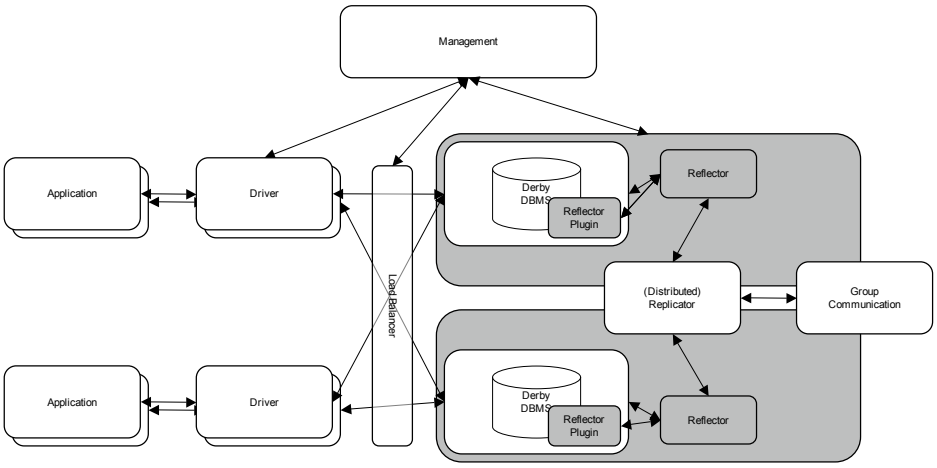

Figure 1.1: Mapping architecture.

1.1.2 Components

The mapping to Derby is outlined in Figure 1.1. The Derby DBMS is patched with the basic code to allow the communication between the Reflector and the Derby engine. The Reflector is placed inside a container. This container holds replication and communication components as well as the client side of the reflection interface. This container is started in the same address space of the Derby database, but the implementation was designed to support Remote Method Invocations (RMI) for the case that the user want to have the replication protocol running as a separate process.

Communication between applications and Derby are performed using a legacy driver provided by the database vendor/supplier thus ensuring compatibility with all existing clients and tools.

As shown in Figure 1.2 the current mapping of the GORDA Programming Interfaces includes all contexts and the main stages.

1.1.3 System

Derby [1] is a java based DBMS. This system is based on standard technologies, such as JDBC and ANSI SQL. Derby is able to handle the SQL language, transaction management, concurrency control, triggers and backups. It has also functionality's such as user authentication, access control, validation of certificates, among other security issues.

The Figure 1.3 shows the Derby architecture. The Derby architecture is composed by the following modules:

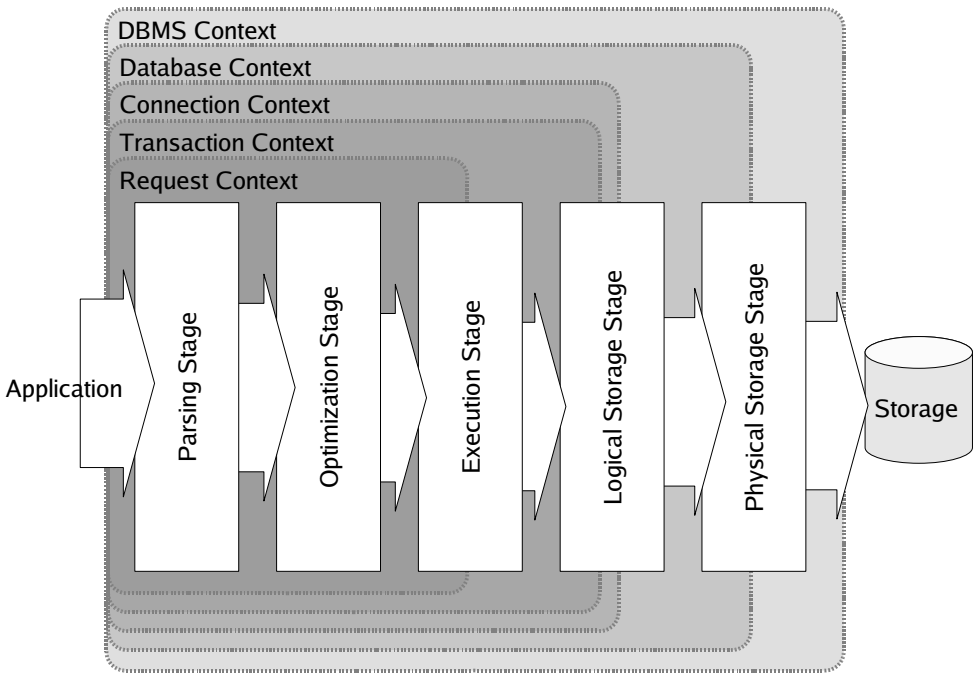

Figure 1.2: Mapped interfaces.

Monitor The Derby DBMS is based on modules. Each module is created and retrieved by a factory and the monitor manages and configures all the Derby modules.

JDBC Layer – client interface This layer is used by the clients to access the databases. and it implements the Java packages *java.sql* and *javax.sql*, corresponding to the JDBC 2 and JDBC 3 versions, respectively.

SQL Layer – relational engine The relational engine supports tables, indexes, views, procedures, functions, triggers, temporary tables, restrictions, and keys. It allows to cache of data and SQL. It allows also the definition of several isolation levels for concurrency control and supports locks for tables and rows, detecting deadlocks when they occur.

Access Layer It provides the access to tables. It queries rows based on the data, indexes, data ordering, access control, transactions and isolation levels.

Storage Layer It manages the physical storage of data into files and logs the transactions. It allows file encryption using JCE [2].

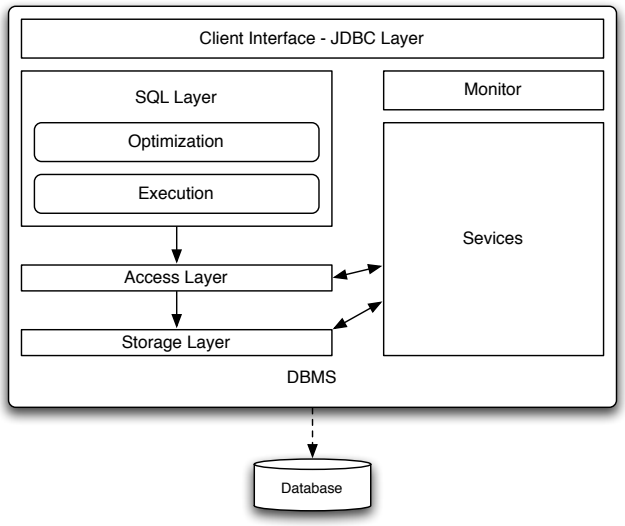

Figure 1.3: The Derby architecture.

Services A service is a collection of co-operative modules with some functionality. One of the modules of the service is considered the primary and defines the service interface. These services are started by the Monitor and can be persistent (defined in configuration time) or non persistent (created on execution time).

The Derby DBMS can be used as an embedded system or as a server. The Derby DBMS automatically starts when it is loaded by the JDBC Driver Manager. The Derby supports configuration on three abstraction levels: the system, the database and the physic storage. The properties can be static or dynamic. One example of a static property is the location of a database. One example of a dynamic property is the write permission of a database. The system properties are defined using Java properties.

The properties that are database specific are stored inside its database. This allows that each database can be configured with different properties. It allows also that the properties are still valid even when the database is dumped to another location. These properties can be configured and read using SQL commands. An example is shown here:

```
-- Procedure to read a property value
SYSCS_UTIL.SYSCS_GET_DATABASE_PROPERTY(<property_name>)

-- Configuration procedure
SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(<property_name> , <property_value>)
```

1.2 MySQL

1.2.1 Objectives

The in-core mapping of the GORDA Architecture and Programming Interfaces (GAPI) to the open source MySQL database management has the following goals:

- Allow the implementation of a primary-backup replication prototype based on group communication;
- Allow full compatibility with all MySQL clients interfaces and tools;
- Demonstrate the feasibility and simplicity for efficiently insert extension points on MySQL core that allow exporting a binding of the GORDA API.
- Modifications to the MySQL core are very little intrusive and agree with MySQL development guidelines towards creating a generic interface for pluggable replication modules.

1.2.2 Components

MySQL mapping is implemented in two distinct components: *i*) in-core extensions points are placed in the codebase and a replication handler is responsible to map these extension points and into a generic MySQL replication plugin interface; *ii*) An implementation of a replication plugin maps MySQL replication interface into GORDA API implemented in Java (through JNI - Java Native Interfaces). MySQL replication plugin interface was developed by the GORDA team. Furthermore, it allowed reusing the plugin architecture in MySQL 5.1.X and the work previously done with Apache Derby, Sequoia and PostgreSQL mappings.

Implementing a plugin and having it glued with previous available GAPI implementations using JNI, allowed us to reuse available Java-based implementations and mix them with C++ within the same address space. Furthermore, JNI introduced little performance penalty.

1.2.3 System

MySQL [3] is a C++ based DBMS. This system is based on standard technologies, such as JDBC and ANSI SQL. MySQL is able to handle the SQL language, transaction management, concurrency control, triggers and backups. It has also functionality's such as user authentication, access control, validation

of certificates, among other security issues. It has a multi-thread architecture. In its most recent stable version, it has a pluggable architecture supporting several plugin types: storage engines, information schema, daemon, etc.

Chapter 2

Implementation

2.1 Apache Derby

This section describes the implementation of the GAPI in the Apache Derby database management system. This is a proof-of-concept implementation that shows also that implementing the GAPI is not very intrusive for kernel of the DBMS.

This interface is subsetable and it can be implemented only some parts of it. To show the functionality of the GAPI, it was implemented the necessary interfaces to be used by a replication protocol in order to validate this implementation in a real system.

The Derby DBMS is composed by modules and the GAPI was built as another Derby module. The other important modules where extended to provide the functionalities needed to implement the GAPI.

2.1.1 Challenges

There are several issues that are needed to solve when implementing the GAPI interface, regardless of the DBMS where the interface is going to be implemented.

The Figure 2.1 shows the generic model for the implementation of the Reflector interface. Because the implementation should be minimally intrusive as possible, the only thing that should be implemented in the DBMS is the capture of the necessary information, like the moment that a transaction starts, the read set, the write set, and so on. The Reflector itself should be implemented as a separate module.

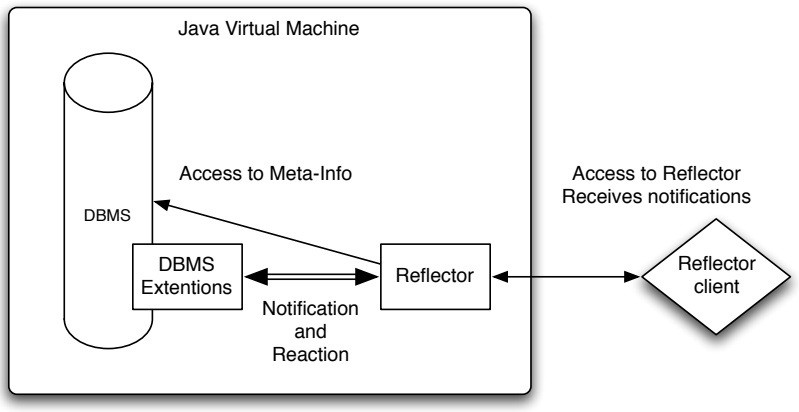

Figure 2.1: Generic model for the Reflector implementation.

Design Options

The main considerations that one must have when designing the Reflector interface are:

Integration of the Reflector in the DBMS In a modular system, the reflector implementation can be just another module. It should be possible to query the reference for the Reflector in any module of the DBMS.

Reflector configuration The Reflector configuration should always be made using configuration files, avoiding code editing to change the reflector parameters.

References between the Reflector stages It is necessary to define how the different stages of the Reflector interface will access the other stages.

Notification Management It is necessary to define a mechanism that supports parallelism. It must allow that several notifications can be done simultaneously. For each notification, it's necessary to find the client to notify, taking into account that the notification, regarding a certain context, could have been cancelled directly, or as a cascading effect at the level of a superior context.

Unique identifiers for contexts It's necessary to define unique identifiers for each execution context.

DBMS extensions

The first stage of the Reflector interface implementation was to extend the DBMS with the capture of all the necessary events and the support for the functionality's that allowed to react to those events. For each

module of the Reflector, the needed events were captured inside the DBMS engine. The components of the Derby DBMS that were extended are the following:

DBMS components corresponding to contexts, connection and client request These components must be extended to inform the Reflector of an expected state change, allowing that the processing can be cancelled in the state INITIALISING, ESTABLISHING and STARTING.

DBMS component corresponding to the database context This component must be extended to inform the Reflector of expected changes, allowing that can be cancelled the processing of STARTING a database.

DBMS transaction component This component must be extended in a way that allows that the change of state to INITIALISING, PREPARING, COMMITTING and ABORTING can be cancelled. It must be also possible to configure the isolation mode of a transaction.

DBMS relational processing module It is necessary to capture the points of the analysis, optimization and execution in the relational processing. It is necessary to define how to cancel the processing in the beginning of each one of these phases. It should be possible to change the SQL request in the first two phases.

Component responsible by the read and write operations It's necessary to identify in the Derby engine where the data is read, written and updated, allowing to obtain the read sets and write sets.

Component responsible by the storage operations It's necessary to identify where the data is logged, in order to reflect it in the physical storage module.

Reflector support

After capturing all the information inside the DBMS engine, it is necessary to build the Reflector itself. To build the Reflector, the following guidelines must be followed:

DBMS module This module is responsible to materialise the method that allow the notification of changes in the DBMS. It's necessary to get the context meta-information, such as the URL, the name, the version of the DBMS and the supported client versions.

Database context module This module is responsible to materialise the state changes of a database and to allow to retrieve meta-information, in particular the URL, the size and the database mode. It should be also possible to retrieve the related contexts and a JDBC DataSource.

Client connection context module This module is responsible to materialise the state changes of a client connection and to allow to retrieve meta-information, in particular the user ID and the used language. It should be also possible to retrieve the related contexts.

Transaction context module This module is responsible to materialise the state changes of a transaction and to allow to retrieve meta-information, in particular the ID, the version and the isolation mode. It should be also possible to retrieve the related contexts.

Client request context module This module is responsible to materialise the state changes of a request and to allow to retrieve meta-information, in particular the ID and the related contexts.

Processing stage modules These modules should materialise the methods that allow to inform the beginning and end of the processing stages.

2.1.2 Mapping on Derby

This Section presents the implementation of the Reflector interface in the Derby DBMS. This implementation was designed to have the best performance possible and at the same time to maintain the compatibility with its design patterns, tools and client interfaces. The current implementation is based on the implementation challenges and options previously described. Only the minimum functionality's were implemented inside the Derby engine. The interface itself was implemented in a separated source folder. To maintain the Derby structure and design patterns, the Reflector implementation was built as a Derby service – the Reflection Service. Each module of this service corresponds to a execution context or to a processing phase of the reflector.

Each Derby service exports a well defined interface. The interface that needs to be exported by each module of the reflection service corresponds to an extension of the reflector interface itself, with methods that allow to retrieve information about the database engine. This option allows the Derby engine to use the reflection service. This service can be used by the Derby engine for self configuration and monitoring. The primary module of the reflection service is called *ReflectionModule* and is responsible

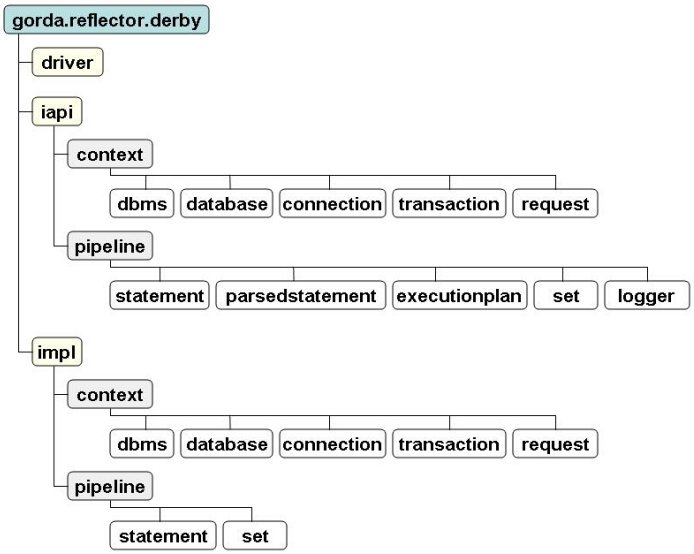

Figure 2.2: Reflector source code organisation.

for the definition of the service interface, its name (*GordaReflection*) and the property that is used to configure the service (*derby.service.ReflectionModule*).

The source code of the reflector interface was placed in a different package (*gorda.reflector.derby*), apart from the Derby engine. Its organisation is illustrated in the Figure 2.2. The package *gorda.reflector.derby.iapi* contains the specification of each module of the reflection service, using interfaces and the package denominated *gorda.reflector.derby.impl* contains the implementation of these interfaces.

The reflection service is disabled by default. The service can be enabled by specifying the location of its primary module:

```
-- Reflection service configuration
derby.service.GordaReflection=gorda.reflector.derby.iapi.ReflectionModule
```

It is also possible to configure each one of the other modules, indicating if they should be enabled or not:

```
-- Reflector modules configuration
derby.gordaReflection.Dbms            = < true / false (default) >
derby.gordaReflection.Database        = < true / false (default) >
derby.gordaReflection.Connection      = < true / false (default) >
```

```
derby.gordaReflection.Transaction     = < true / false (default) >
derby.gordaReflection.Request         = < true / false (default) >
derby.gordaReflection.Statement       = < true / false (default) >
derby.gordaReflection.ParsedStatement = < true / false (default) >
derby.gordaReflection.ExecutionPlan   = < true / false (default) >
derby.gordaReflection.ObjectSet       = < true / false (default) >
derby.gordaReflection.Logger          = < true / false (default) >
```

The configuration of the reflection service and its modules is done at the system level, as described previously.

The Derby DBMS architecture is based on modules and there are already mechanisms to allow, to each module, to obtain the reference to other modules. This can be done using the Monitor and its context manager. When the Reflector service is created, it receives an execution context with a reference to the other modules. On the other side, all the Derby components that were extended to build the reflector contain, on their execution context, a reference to the reflector module.

The notifications to the Reflector clients are made from its execution thread, allowing that a blocking notification stop only the execution of the captured event and not other events. This allows also that the client of the Reflector interface uses its own thread policy.

All execution context must have a unique identifier. These identifiers are created by a Derby tool that creates identifiers in the same format of the Microsoft UUIDGEN:

```
-- Identifier generated by the module UUIDFactory of Derby
E4900B90-DA0E-11d0-BAFE-0060973F0942
```

Support and implementation of the DBMS module

To support the functionality's of the DBMS reflector context, its necessary to extend the corresponding Derby component, capturing the events of STARTUP and SHUTDOWN. The Derby DBMS is started by the DriverManager, invoking:

```
Monitor.startMonitor(bootProperties, logging);
```

This method is responsible for starting up one instance of the Derby Monitor. This is done using an abstract class called *BaseMonitor* that contains the method *runWithState* for starting up the system and the *shutdown* method for shutting down the system.

This module was extended with the functionality's for the supporting the communication between the Derby engine and the Reflector. Each state change is captured in the Derby engine, received by the reflector and delivered to the client modules. This module is also responsible for giving access to the system meta-information. This information is also accessible using the interface *DatabaseMetaData* defined in JDBC.

Support and implementation of the database module

To support the functionality's of the database reflector context, its necessary to extend the corresponding Derby component, capturing the events of start up and shutdown of a database and defining the panic (freeze) mode of a database.

The Derby system defines an interface called *Database*, located in the package *org.apache.derby.database*, that allows the control of a database, such as the data itself and the files where the data is stored. This interface is internally extended by the interface *org.apache.derby.iapi.db.Database* with operations that are not accessible for the users. It only exists one implementation of this interface, the *BasicDatabase*, that implements the *boot* and *stop* methods. These class was extended to notify the reflector about state changes of a database. The extension of this component also extends the panic mode of a database. This mode indicates that the database state is not coherent and cannot be accessed by the database users. For this to be supported, the reflector implementation uses procedures already provided by Derby:

```
CALL SYSCS_UTIL.SYSCS_FREEZE_DATABASE()
CALL SYSCS_UTIL.SYSCS_UNFREEZE_DATABASE()
```

Support and implementation of the transaction module

To support the functionality's of the transaction reflector context, its necessary to extend the corresponding Derby component, capturing the events of start and end of a transaction and defining its isolation mode.

The Derby defines an interface called *org.apache.derby.iapi.store.raw.Transaction* that represents a transaction and provides the methods to manage it. It is possible, for example, to abort a transaction. This interface is partially implemented by the abstract class *RawTransaction*. This class already implements functionality's that allow the notification of external elements when a transaction is aborted, committed, or rolled back until the last save point. This class is extended by the class *Xact*, that implements all the

other life cycle states of a transaction.

Support and implementation of the analysis module

To support the functionality's of the analysis module, its necessary to extend the Derby relational processing component, capturing the events of start and end of an analysis phase and allowing that the processing is cancelled at the beginning of the logical storage stage.

The *GenericStatement* class corresponds to a SQL request in text format and it's responsible for the analysis of the request. This class was extended to notify the reflector about the beginning and end of this stage.

Support and implementation of the storage module

To support the functionality's of the storage module, its necessary to extend the Derby in the points where data is read, inserted, updated and removed. This is needed to obtain the write and read sets. At this moment, only the write sets are exported to the reflector interface.

On Derby, the byte code that processes the execution plan is dynamically generated. The generated class extends an internal class called *BaseActivation* that implements the *Activation* interface. During the execution, it's generated a tree with the sets of updated, inserted, removed and added data. The events are captured in these classes and exported to the Reflector.

The *Activation* interface was extended to allow the configuration of capturing the write sets; if it should be captured and, in the case that is captured, which columns to capture.

2.2 MySQL

This section describes the implementation of the GAPI in the MySQL database management system. This is a proof-of-concept implementation that shows also that implementing the GAPI is not very intrusive for kernel of the DBMS, specially if benefiting from the plugin architecture.

This interface is subsetable and it can be implemented only some parts of it. To show the functionality of the GAPI, it was implemented the necessary interfaces to be used by a primary-backup replication protocol in order to validate this implementation in a real system.

The MySQL DBMS is composed by an execution engine, and several plugin types that provide different storage engines. Since no plugin type existed for replication, the GORDA team has created one

and used it to keep GAPI mapping as little intrusive as possible.

2.2.1 Challenges

There are several issues that are needed to be solved when implementing the GAPI interface, regardless of the DBMS in which the interface is going to be implemented. Additionally, the implementation must be as little intrusive as possible, so it should take advantage of existing pre-extension points and procedures that ease the final goal.

The major challenge was to minimize intrusion by having implementation to agree with the current plugin architecture provided by MySQL 5.1.X. Using this approach, one just needed to pinpoint the appropriate extension points inside the core and place hooks. These hooks are managed by a set of extensions that provide a direct mapping into the a replication plugin interface. An implementation of the plugin interface provides a GAPI mapping for MySQL.

Another challenge was reusing the codebase for replication (eg, ESCADA, APPIA) that we had already used in other prototype implementations. The procedure should provide no catastrophic performance hit, and preserve 100% compatibility among existing components and MySQL.

2.2.2 Mapping on MySQL

This Section presents the implementation of the replication plugin interface in MySQL as well as the GAPI mapping as an external plugin implementation. This implementation was designed to have the best performance possible and at the same time to maintain the compatibility with its design patterns, tools and client interfaces. The current implementation is based on the implementation challenges and options previously described. Only the minimum functionality's were implemented inside the MySQL engine and already existing interfaces for capturing events were used (binlog).

Inside MySQL mainline there are two major changes: *i*) definition of a new plugin type for replication. *ii*) extension hooks placed correctly that enable extraction of information to be feed to replication plugin implementations. The replication plugin interface was implemented resorting to the plugin architecture defined by MySQL. A new plugin definition was added to the mainline source code. This definition included an interface, a entry in the existing plugin table and functions to register and unregister plugin implementations. The extension points, or hooks, were placed according to the lifecycle and execution workflow of MySQL. Calls to these hooks are handled by handlers that are responsible to find

registered replication plugins and feed them with the information extracted. Should no replication plugin implementation be registered/installed in the MySQL instance, then the information is discarded, and performance negative performance impact is quite negligible.

The actual GORDA API, GAPI, was implemented as a MySQL replication plugin. It builds on the replication plugin interface and maps that interface to GAPI. GAPI itself was a good candidate for MySQL replication plugin interface, but it was far to complete, for the demonstration process, risking cluttering the end result. Additionally, one was able to reuse much of the implementation already done for PostgreSQL and put it alongside MySQL with minimal effort. Furthermore,

Support and implementation of the DBMS context

The DBMS context is ultimately implemented by extracting information from MySQL initialization function, *main*. Hooks are placed so that start and stop of the DBMS end up mapped into MySQL plugin and eventually into GAPI interface. The shutdown hook is placed inside the signal handler responsible to clean up MySQL global state. Startup hooks are mapped into the replication plugin that is responsible to start the Java Virtual Machine (JVM) loading GAPI mapping for MySQL and ESCADA. Shutdown hook is responsible to clear the JVM in an orderly fashion.

Support and implementation of the Database context

The database context is not completely implemented and is still marked as work in progress. Nevertheless, this is a minor issue, as in MySQL connections are target DBMS and not databases, and for prototype demonstration there were no use cases requiring database notifications.

Support and implementation of the Connection context

The connection context is bounded by connection startup and shutdown in mysqld main loop function. The context is uniquely identified by the thread identifier that handles the connection. Connection startup and shutdown are notified to the replication extension handler that drives this notifications to the replication plugin and ultimately to MySQL GAPI binding.

Support and implementation of the Transaction context

There is no startup hook for transactions. Instead, every time a transaction issues an object set notification, and no transaction context is associated with the given transaction id, a new context is started implicitly. The transaction context is uniquely identified by MySQL transaction identifier.

Support and implementation of the Request context

Request context is implemented by grouping object set notifications. The group of object sets defines the boundaries of each context and is provided as a bundle to MySQL GAPI implementation.

Bibliography

[1] Apache Derby web page.

`http://db.apache.org/derby/`.

[2] JCE Manual.

`http://java.sun.com/products/jce/`.

[3] MySQL AB / SUN Microsystems. Mysql.

`http://www.mysql.com/`.