Project no. 004758

GORDA

Open Replication Of Databases

Specific Targeted Research Project

Software and Services

Middleware proof-of-concept

GORDA Deliverable D4.4

Due date of deliverable: 2006/09/30
Actual submission date: 2007/04/11
Revision date: 2008/03/30

Start date of project: 1 October 2004 Duration: 36 Months

Continuent

Revision 1.1

Project co-funded by the European Commission within the Sixth Framework Programme (2002-2006)		
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 Junior, U. Minho
Edward Archibald, Continuent
Emmanuel Cecchet, Continuent
Robert Hodges, Continuent
Seppo Jaakola, Continuent
José Pereira, U. Minho
Nuno Carvalho, U. Lisboa
Rui Oliveira, U. Minho
Ricardo Vilaça, U. Minho

Abstract

This document describes the middleware proof-of-concept implementation of the GORDA Architecture and Programming Interfaces. The prototype is built as part of Sequoia, an open source database clustering middleware. Emphasis is put on the description of the fundamental components of Sequoia that serve to wrap relational database engines and allow to offer a GORDA compliant DBMS which are the middleware level concurrency control and the capture of the transaction changes to the underlying databases, and on how the GORDA Reflector interfaces have been implemented in Sequoia.

Contents

Chapter 1

Introduction

The GORDA middleware proof-of-concept is built as part of Sequoia[1], an open source database clustering middleware written in Java that allows applications to transparently access a cluster of databases through JDBC ™.

In the next chapter we start by describing the Sequoia middleware. In Chapter 3 we detail the fundamental components of Sequoia that serve to wrap relational database engines and allow to offer a GORDA compliant DBMS. Finally, in Chapter 4 we show how Sequoia enables a middleware-based implementation of GORDA and detail the GORDA's architecture main component, the reflector, implementation in Sequoia.

This report is part of GORDA Deliverable D4.4. The companion software package can be downloaded from the project's website at http://gorda.di.uminho.pt.

1.1 Objectives

The middleware proof-of-concept implementation of the GORDA Architecture and Programming Interfaces (GAPI) has the following goals:

- to provide a compatibility layer to deploy GORDA protocols on a wide range of standard off-the-shelf database management systems;
- to provide full compatibility with the JDBC client interfaces;
- to demonstrate the feasibility and evaluate the consequences of implementing a large subset of the GAPI by relying only on a standard JDBC client interface, and
- to leverage as much as possible existing open-source components, namely, to maintain backward compatibility with Sequoia RAIDb protocols and tools.

1.2 Relationship With Other Deliverables

This document depends on deliverables D2.2 - GORDA Architecture Definition and D2.3 - GORDA Interfaces Definition, which specify the architecture and interfaces implemented here. The current deliverable supersedes deliverable D4.2 - Middleware Mapping Report, which has outlined the general approach of this proof-of-concept implementation within Sequoia. In detail, it improves on the previously proposed mapping as follows:

[1] http://sequoia.continuent.org

- Improvements to the scheduler for optimistically synchronized protocols providing priority scheduling (Section 3.2), which wasn't initially envisioned. This allows the middleware proof-of-concept to run all proposed reference replication protocols.
- Improvements to the scheduler for supporting stored procedures, thus widening the applicability of the prototype to legacy DBMS and applications.
- Support for ODBC clients, thus widening the applicability of the prototype and strengthening its value when integrating legacy DBMS and applications.

Chapter 2

Sequoia

Sequoia implements the concept of Redundant Array of Inexpensive Databases (RAIDb) [2]. A database is distributed and replicated among several nodes and Sequoia load balances the queries between these nodes.

Sequoia provides a generic driver to be used by the clients. This driver forwards the SQL requests to Sequoia that balances them on a cluster of databases following a read-one write-all approach (reads are load balanced and writes are executed on all copies). In the following, queries or statements (used interchangeably) refer to SQL statements in the context of a transaction, explicitly or implicitly defined.

2.1 Architecture

Figure 2.1 shows an overview of the Sequoia architecture with drivers and controllers. The Sequoia architecture is open, allowing to plug in custom requests schedulers, load balancers, connection managers, caching policies, etc. Sequoia can be used with any RDBMS (Relational DataBase Management System) providing a JDBC driver, that is to say almost all existing open source and commercial databases. Sequoia allows to build any cluster configuration including mixing database engines from different vendors and is also flexible in the sense that it can be configured with a single controller and a single backend, providing a bare-bones JDBC interceptor.

2.1.1 Sequoia Driver

The Sequoia JDBC Driver is a type 4 JDBC driver, which forwards all database queries to the Sequoia Controller. When using Sequoia with a Java client, the client applications connect to the cluster through the Sequoia driver, which replaces the database-specific driver originally used by the client application.[1]

The Sequoia JDBC Driver provides an almost complete coverage of the JDBC 3 specification. The driver also provides transparent fail-over capabilities to reconnect and restore a transactional context in case of a controller failure.

2.1.2 Sequoia Controller

The Sequoia Controller is a Java program that acts as a proxy between the Sequoia Driver and the Database Backends. The controller enables the managed databases to be presented to the client application as a single Virtual Database. The Sequoia Controller uses the native database JDBC driver to access the underlying database server(s).

[1] The basic Sequoia installation only includes a JDBC driver for Java clients. However, the Carob project (http://carob.continuent.org) provides additional connection for other client applications and, through the DBD::JDBC Perl module, Perl clients can also be served.

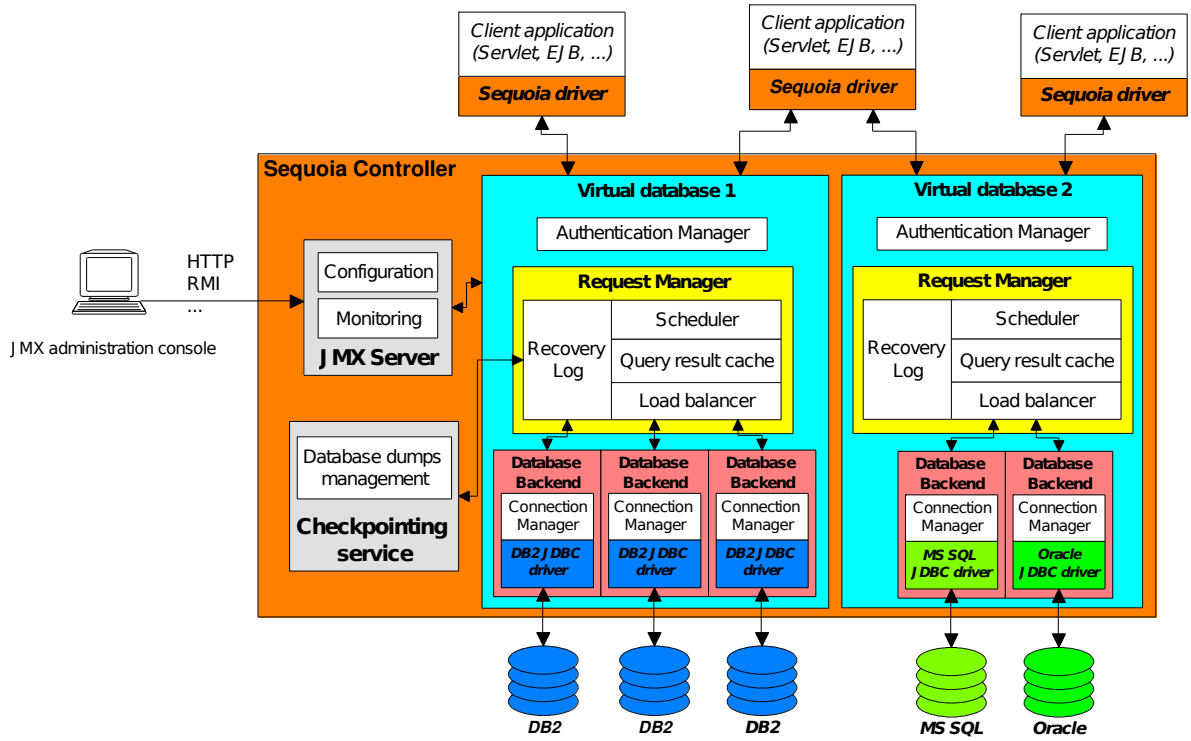

Figure 2.1: Sequoia overview

2.1.3 Virtual Database

A Virtual Database virtualizes a single database but the Sequoia controller virtualizes a Relational Database Management System (RDBMS). In other words, just as a RDBMS can host multiple databases, the controller can host multiple virtual databases.

A Virtual Database consists of the following components:

Authentication Manager authenticates the virtual database username and password during connection establishment and checks that it is correctly mapped to the real database server username and password.

Request Manager handles the incoming client requests forwarded by the Sequoia connector (Section 2.2.1)

Database Backend the database backend(s) are used to administer the underlying database servers(s) (Section 2.2.2).

Backup Manager performs database backup and restore operations and transfers backup files from one controller to another (Section 2.2.3, not shown in Figure 2.1)

A Virtual Database has a virtual name that matches the database name used in the client application. The client therefore perceives there to be a single (virtual) database camouflaging the multiple backends.

A Virtual Database and its components are configured in a controller-specific virtual database configuration file. To configure one Virtual Database that is hosted by two controllers, two unique controller-specific configuration files for that Virtual Database are thus needed.

2.2 Main Components

2.2.1 Request Manager

The Request Manager contains the core functionality of the controller. When a client request arrives from the Sequoia Driver, it is first routed to the Request Manager associated with the Virtual Database. The Request Manager consists of the following components, which are described in more detail in the following sections:

- Request Scheduler
- three optional Query Result Caches:
  - a metadata cache that caches result set metadata (such as column names, types, etc.) that is used when building result sets;
  - a parsing cache that caches the parsing results, usually table names extracted for locking and caching purposes. The parsing cache is useful especially with prepared statements;
  - a query result cache that caches the result sets of read queries: if a given query is executed several times, it only needs to be sent to the database once.
- Load Balancer
- Recovery Log

Request Scheduler

The Request Scheduler manages the requests and ensures query consistency. Sequoia uses a pass-through scheduling method, where queries are assigned a unique identifier and forwarded as-is to the load balancer. This identifier is used later to ensure that the writes are sent in the same order to all backends. Each database server ultimately performs the scheduling and the locking. Thus, the locking granularity depends on the database engine.

Load Balancer

Client requests arrive at the Load Balancer through the Request Scheduler. Sequoia's load balancing mechanism increases the overall performance of the database cluster. It distributes requests between backends according to a predefined load balancing method: users can choose a method most suitable for their system.

The Database Backends (Section 2.2.2) are attached to the Load Balancer. Each Database Backend contains a total order queue in which it receives the requests.

Recovery Log

The Recovery Log is a transactional log that records all requests and transactions that update the Virtual Database for database recovery and synchronization purposes. The log also maintains information about checkpoints that can be local to a controller or global to the cluster.

The Recovery Log information is used to resynchronize failed nodes or add new nodes to the cluster by replaying queries from a consistent checkpoint that is usually associated to a database dump.

2.2.2 Database Backend

A Database Backend is a Sequoia object that is used to administer an underlying database server. The term backend refers to Sequoia's view of a database server instance.

When a backend is disabled, the underlying database server instance may remain operational. A backend is disabled for example for the time of performing a database backup in order to prevent the execution of queries during the backup procedure and to ensure database consistency.

Each Sequoia Controller hosts a dedicated set of backends. No backends are shared between controllers for the sake of database consistency.

Database backends can be dynamically added to or removed from a Virtual Database, transparently to the user application.

2.2.3 Backup Manager

When a new backend is added to the cluster, its database must be brought to a state that is consistent with the databases of the other active cluster nodes.

In Sequoia, when a node is brought into the cluster, entries logged into a recovery log are used to bring the node back into exactly the same state as the other nodes in the cluster. This process does not interfere with the other operations of the cluster and makes it possible to recover from failures without downtime.

The Backup Manager, together with the Recovery Log, allows the dynamic addition of new backends to the Virtual Database without the need to stop and restart the system. Similarly, one can use the Backup Manager and recovery log to easily re-enable a backend when recovering from a backend failure. The Sequoia installation includes both a generic and several RDBMS-specific backupers. Backupers should ensure that a full database snapshot can be taken and restored on another backend to provide the exact same state.

2.2.4 Group Communication

When using RAIDb across multiple controllers, the controllers use a group communication protocol to exchange information and maintain consistent state information between each other. This controller replication prevents controllers from representing a possible single point of failure.

Only database write and commit/rollback commands are sent using the group communication protocol. All other commands are executed locally by the controller.

Sequoia allows multiple group communication implementations to be used. By default, Sequoia controller communication is implemented using the Appia group communication library, although any jGCS compliant group communication protocol [1], such as Spread can also be used.

2.3 Request Processing

Figure 2.2 illustrates the internals of Sequoia which we will use to exemplify how requests are processed. The configuration consists of two client JVMs, each having three connections running different transactions on two controllers replicating a virtual database using RAIDb-1 on 4 database backends. The circled arrows represent Java threads.

In diagrams, T1, T3, T4 and T5 are write transactions whereas T2 and T6 are read-only transactions. Transactions that are sent in parallel are ordered by the group communication primitives. In this example, the transactions are delivered in the following order: T1, T4, T3 and T5. When we say that transactions are delivered in a specific order, in fact we mean requests belonging to these transactions. A VirtualDatabaseWorkerThread is assigned to each client connection. It handles the protocol with the driver and

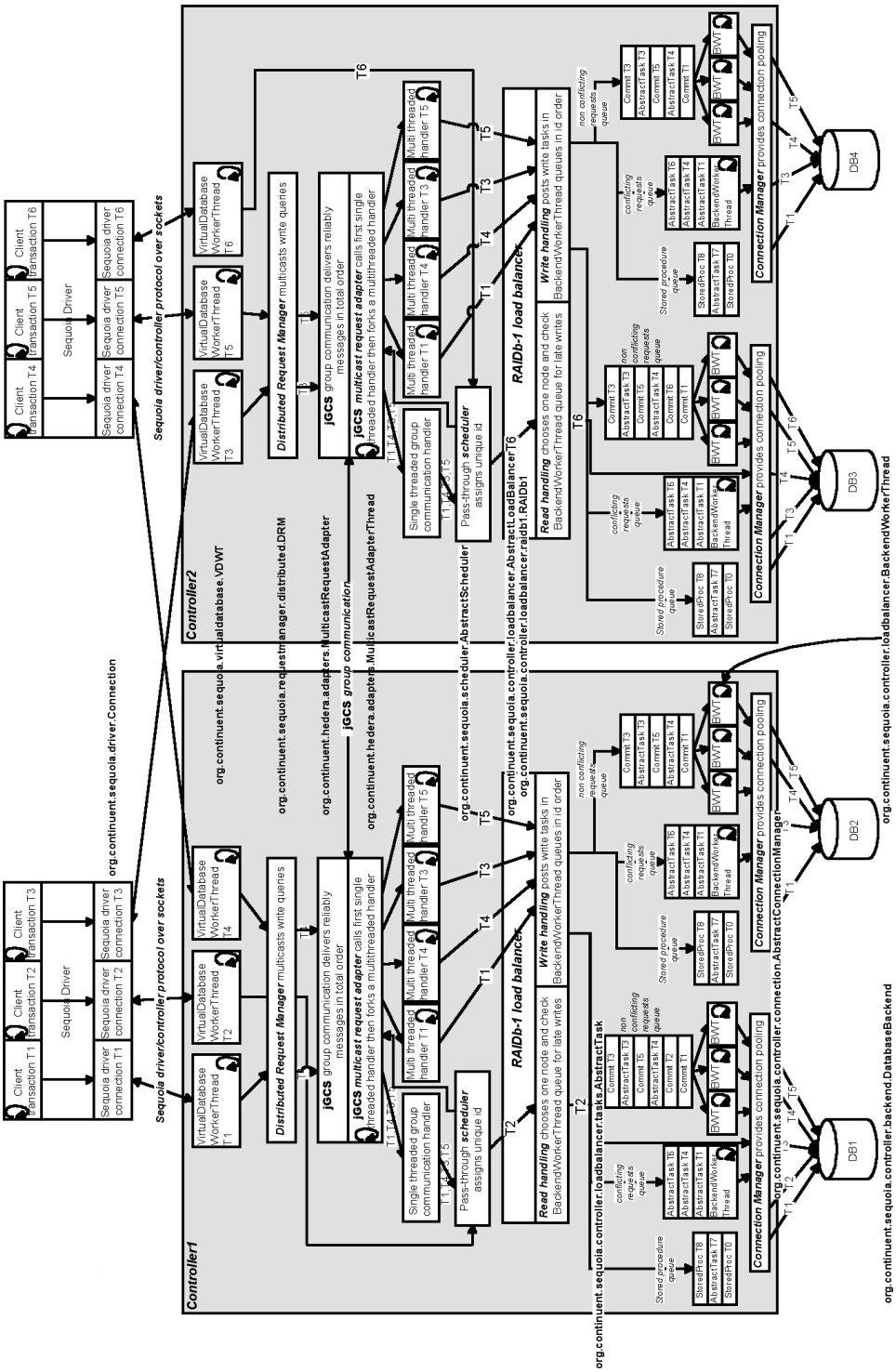

Figure 2.2: Sequoia internals overview

it is responsible for executing the queries on the controllers. You can notice that read transactions are only executed at one backend on the local controller whereas write transactions are executed at all nodes and have a dedicated connection on all nodes.

Each request is represented by one of the objects in the org.continuent.sequoia.controller.requests.-AbstractRequest hierarchy (only used on the controller side). The request object also carries the connection context to be used in all components of the controller. These objects are serializable to be sent to remote controllers. Each command (StatementExecute, StatementExecuteUpdate, etc.) maps a JDBC call and is encapsulated in a group message represented by a org.continuent.sequoia.controller.-virtualdatabase.protocol.DistributedRequest object that handles the query execution through the different components of the distributed request manager. Each command is posted in the BackendWorkerThread queue as an org.continuent.sequoia.controller.loadbalancer.tasks.AbstractTask that contains the code for the execution on the backend.

2.3.1 Read Transaction Handling

Read requests do not require any group communication since they execute locally to a controller. A read request will completely execute inside the VirtualDatabaseWorkerThread by executing first the code of the scheduler and then the load balancer code to choose a node according to the load balancing policy. Note that if a query result cache is available (not shown in the Figure 2.2), the result would be returned on a cache hit before going to the load balancer. Read-only transaction commits only occur on the nodes that have started the transactions. Usually there is only one node but it might happen that in multi-statement transactions reads are distributed on several nodes.

2.3.2 Write Transaction Handling

Write requests are sent through the group communication to be delivered in the same total order at all nodes. When the group communication delivers the messages in total order, it executes, in mutual exclusion, the single threaded group communication handler that inserts the query in a total order queue (not shown in Figure 2.2). It is only when the single threaded handler has completed that the next message (usually a request) can be delivered. The ultimate execution of each request is then handled in a separate thread that executes the code of the multi-threaded handler, on behalf of the client application, for each VirtualDatabase managed by the controller (code found in DistributedVirtualDatabase). The handler executes the code of the load balancer that posts the query to a request queue and coordinates the completion of the requests between the associated client application, via the VirtualDatabaseWorkerThread, and the BackendWorkerThread. The load balancer ensures that all queries are posted in the same order to all queues. The request completion locally to a controller is handled by the org.continuent.sequoia.-controller.loadbalancer.tasks.AbstractTask methods. The completion of a distributed request is handled in org.continuent.sequoia.controller.requestmanager.distributed.DistributedRequestManager. The same methods can also detect inconsistencies for update queries returning an update count. The first result is considered as the final result and all backends that have a different result are disabled.

2.4 Client APIs

Sequoia includes two types of client APIs designed to provide simple and transparent access to the cluster.

2.4.1 JDBC APIs

Sequoia implements its own wire protocol for clients. This protocol is supported by the Sequoia JDBC driver, which clients must use to connect directly to controllers. This approach was a natural design for Sequoia and its predecessor C-JDBC, which are written in Java and intended to support Java applications.

JDBC drivers are highly standardized and therefore quite transparent. Java client applications can switch to the Sequoia driver easily and then connect directly to the cluster.

The main advantage of the Sequoia JDBC driver is that it has built-in failover and connection load balancing capabilities. This raises client application availability and balances load across controllers without necessity of application load-balancing logic or extra hardware.

2.4.2 Native Client Access

The majority of clients that connect to databases served by GORDA are not written in Java (specifically PostgreSQL and MySQL). Instead, these applications are written in languages like PHP, Perl, C, Ruby, and a host of other languages that communicate with the database using native APIs. There is no standard for database access across such languages, and ports of the native libraries themselves can be very problematic for a number of reasons that range from difficulty in relinking applications to performance problems.

The Myosotis connector is an open source proxy developed by Continuent to solve the native client access problem. Myosotis serves as a gateway between MySQL and PostgreSQL clients and JDBC. For example Myosotis can accept a SELECT request from a MySQL client, execute the query through JDBC and return results to the client again. This approach greatly increases the simplicity of using the cluster, as clients do not need to switch libraries and all languages and OS platforms are supported. In addition, Myosotis is quite performant. Query throughput is better than comparable proxies like pgpool-II (a PostgreSQL proxy) as well as Sequoia itself.

Continuent has made considerable improvement to Myosotis over the course of 2007 and early 2008, including addition of support for prepared statements. Transparency is now very high, with current work focused on addressing arcane details of data types and character sets.

Chapter 3

Database Middleware Wrapper

In this chapter we describe in detail three fundamental mechanisms built into Sequoia allowing it to take the role of a replicated DBMS in a GORDA setting. Sequoia acts as a GORDA compliant DBMS wrapping up a database engine where the GAPI interfaces cannot be implemented.

3.1 Logical Locking

We start by discussing how scheduling of potentially conflicting transactions is performed. As the JDBC interface provides no mean to detect if a query will block at the database level or not, we have to reproduce the database locking at the middleware level in order to guess potential conflicts. This is achieved in Sequoia by using logical locks, granted to transactions, based on the database table namespace. This requires parsing each query to know which database tables are accessed. If the parsing fails or if it is not possible to determine which tables are accessed (e.g. a stored procedure call) the query's transaction will require locks for all tables at once.

3.1.1 Write Scheduling

In detail, the scheduler maintains a list of locks acquired by the different transactions on each database table (one lock per table). The first transaction to acquire a lock on a table will execute in non-blocking mode (possibly in a dedicated thread) whereas if the lock is already held by another transaction, the request will be marked as potentially blocking. This way, non-conflicting transactions can be executed in parallel and potentially blocking transactions will execute sequentially from a conflicting queue.

Being based on table-level locking the middleware scheduler is fated to be too conservative especially when the underlying databases use a more fine grained row-level locking scheme. To avoid false conflicts and mitigate this phenomenon, the Sequoia scheduler allows to execute in parallel conflicting queries

Conflicting requests queue	Non-conflicting requests queue	Lock queue		T1	T2
		A	B	Begin	Begin
	T1W(A1)	T1		W(A1)	
T2W(A2)		T2 T1			W(A2)
	T1W(A2)	T2 T1		W(A2)	

Table 3.1: Example of non-conflicting and conflicting accesses to the same table

Conflicting requests queue	Non-conflicting requests queue	Lock queue		T1	T2	T3	T4
		A	B	Begin	Begin	Begin	Begin
	T1W(A)	T1		W(A)			
T2W(A)		T2 T1			W(A) blocked		
T2W(A)	T3W(B)	T2 T1	T3			W(B)	
T4W(B) T2W(A)		T2 T1	T4 T3				W(B) blocked
T4W(B) T2W(A)	T3Commit	T2 T1	T4			Commit	
T2W(A)	T4W(B) (priority inversion)						
T2W(A)	T4Commit						Commit
	T1Commit	T2	T4	Commit			
	T2Commit		T4		Commit		

Table 3.2: Example of order inversion

by relying on the concurrency control of the backend (see example of Table 3.1). This optimization however can be troublesome when queries have conflicts at the row level, that is, real conflicts. Since Sequoia cannot guarantee that such statements execute by the same order on all backends, this may lead to a deadlock or an inconsistency problem. If Sequoia waits for an answer from all backends to continue the execution of a transaction, this may generate a deadlock as some backends may be waiting to acquire a lock to proceed. On the other hand, if the answer of the first backend is used, one may end up with backends executing transactions by different orders. To circumvent this, one may check if the updates returned the same results on all backends. If not, the backends that report a different result are automatically disabled.

3.1.2 Order Inversion on Transaction Completion

When a transaction completes (commit/rollback/abort), it releases all locks it had acquired during its execution. These locks might have caused other queries to be flagged as possibly blocking since the transaction was holding the locks. It is then necessary to re-check if some of the blocked queries are not unnecessarily held in the conflicting requests queue. Order Inversion extracts a query from the conflicting requests queue to place it in the non-conflicting requests queue for faster execution eventually inverting the natural order of execution.

Table 3.2 shows an example of order inversion with four transactions (T1 and T2 writing to A and T3, T4 writing to B). If the queries are executed in the order used in the example, T4 will be stacked in the conflicting requests queue after T2 which is waiting for T1 to release its lock. As T4 is only waiting for a lock held by T3, when T3 completes, it notifies the next waiting transaction on the lock for B (that is T4). T4 request does not block anymore and is promoted to the non-conflicting requests queue for immediate execution and thus overtaking T2.

3.1.3 Table Based Deadlock Detection

Table 3.3 presents a scenario with two transactions trying to write to two tables in different orders. This is a typical deadlock problem, but, in our case, not all requests are sent to the database which would not

Conflicting requests queue	Non-conflicting requests queue	Lock queue		T1	T2
		A	B	Begin	Begin
	T1W(A)	T1		W(A)	
	T2W(B)	T1	T2		W(B)
T2W(A)		T2 T1	T2		W(A) blocked
T1W(B) T2W(A)		T2 T1	T1 T2	W(B) blocked	

Table 3.3: Example of a deadlock scenario

Conflicting requests queue	Non conflicting requests queue	Lock queue		T1	T2
		A	B	Begin	Begin
	T1W(A)	T1		W(A)	
T2W(B)R(A)		T2 T1	T2		W(B)R(A) not blocked if snapshot isolation
	T1Commit	T2	T2	Commit	
	T2Commit				Commit

Table 3.4: Example of a multi-table locking scenario

be able to detect the deadlock. We need to check the list of acquired locks to detect a conflicting order between transactions. In this example, T1 took A before T2 but T2 took B before T1. The deadlock detection algorithm builds a graph of dependencies for all lock queues and then checks for cycles in the graph. Transactions are aborted and removed from the graph until the graph does not contain any cycles. The graph computation and deadlock detection is only triggered when the conflicting requests queue has not progressed for a given amount of time (equivalent to a deadlock detection timeout). This way, the request execution does not incur in any overhead until a real deadlock happens.

3.1.4 Queries Accessing Multiple Tables

Some queries access multiple tables either to read or to write. This is for example the case of queries using a sub-select statement in their where clause (e.g. a statement like `UPDATE table1 WHERE id in (SELECT id FROM table2)` accesses table1 to write it and table2 to read it). Some databases also accept updating simultaneously multiple tables such as `DELETE FROM table1,table2`.

All locks should always be acquired in the same order to prevent any deadlock. For example, `DELETE FROM table1,table2` and `DELETE FROM table2,table1` always take the locks in the same order. Sequoia uses a SortedSet of table names, so that locks are always acquired in the alphabetical order of the table names.

If all locks are free, the query can go in the non-conflicting queue, else it has to go to the conflicting queue. Extra care has to be taken when all locks cannot be taken. Note that in Table 3.4 example, the result of the update on table B will depend on whether T1 commits before or after the read on table A. In this case, it is necessary to ensure that all backends will commit T1 in the same order with regard to T2's update operation. An option consists of systematically waiting for all locks to be available.

Conflicting requests queue	Non conflicting requests queue	Lock queue			T1	T2	T3
		A	B	C	Begin	Begin	Begin
	T2ST1	T2	T2	T2		ST1	
T1W(A)		T1 T2	T2	T2	W(A) blocked		
	T2W(A)	T1 T2	T2	T2		W(A) blocked	
T3W(A) T1W(A)		T3 T1 T2	T2	T2			W(A) blocked
T1ST1 T3W(A) T1W(A)		T3 T1 T2	T1 T2	T1 T2	ST1		

Table 3.5: Deadlock scenario with a simple locking for stored procedures

3.1.5 Stored Procedures Handling

Stored procedures are very specific in the sense that simple parsing of the stored procedure invocation request will not tell us which tables will be accessed. One approach is to assume that the stored procedure will access all tables and then lock all tables. If we handle stored procedures this way, it increases the likelihood that a deadlock can occur, at the middleware level, because we are locking all tables. This is demonstrated in Table 3.5. In this example, T2 executes a stored procedure (ST1) and then locks all tables. T1 writes to A and is flagged as non-blocking however, let us assume, the write executes successfully in the conflicting requests queue since it did not conflict with tables accessed in ST1. When T2 tries to write to A, it is allowed to proceed in the non-conflicting requests queue since it already has the lock on A. When T1 executes a stored procedure, it is sent to the conflicting requests queue, however it can still execute. If T3 tries to write to A, it will be blocked in the conflicting requests queue. If T1 tries to execute a stored procedure it will be blocked after T3's write and will never be able to progress. The system is deadlocked and the deadlock detection mechanism will not detect any cycle in the lock graph.

The proposed solution, for the near term, consists of using a separate queue for stored procedures. This queue does not have any BackendWorkerThread to process it. Queries in the stored procedure queue are just in standby waiting for all current transactions to release their lock to be able to lock the whole database. When a stored procedure executes, no other transaction is allowed to take any lock (and thus execute writes) until the transaction completes. Note that the deadlock detection mechanism has to be enhanced to take into account transactions that are on hold because they are waiting for a stored procedure to execute. Table 3.6 gives an example of a deadlock where T1 waits for T2's completion before executing its stored procedure. If T2 tries to acquire a resource already held by T1, there is a deadlock that needs to be detected.

The deadlock detection algorithm has to augment the lock queue graph with locks of the stored procedure queue so that stored procedure locks are represented in the graph. Note that when a cycle is detected, rollback priority should be given to transactions trying to execute a stored procedure. Since such transactions are trying to acquire all locks, they are more likely to cause deadlocks than other transactions. A longer term solution to the execution of stored procedures is anticipated to involve determining, to the degree possible, which tables *may* be referenced by a given stored procedure and to then acquire locks only on those tables. This approach has some limitations as well and is only applicable to stored procedures for which the source code is available and which exhibit other properties such as

Stored procedure queue	Conflicting requests queue	Non conflicting requests queue	Lock queue		T1	T2
			A	B	Begin	Begin
		T1W(A)	T1		W(A)	
		T2W(B)	T1	T2		W(B)
T1ST1			T1	T2	ST1	
	T2W(A) blocked		T2 T1	T2		W(A) blocked

Table 3.6: Deadlock scenario with a simple locking for stored procedures

deterministic execution. Given that this type of approach is feasible, the handling of stored procedures can then be re-factored so that they are handled the same way as any other database write request.

3.2 Priority Scheduling

A priority scheduling mechanism eases the task of ensuring that a commit order can be enforced regardless of competing unrestricted transactions being submitted by other connections. In detail, this requires that in the event that a high priority transaction is about to block due to a lower priority transaction, the latter is aborted.

To enable such features in Sequoia, logical locking 3.1, deadlock detection and transaction properties take into account *master* transactions. Such an attribute is set simply by issuing a `SET TRANSACTION MASTER` command.

Sequoia defines two levels of priority, namely, "master" and "normal" (i.e., default). Once a transaction is defined as master, its priority is only brought to normal, after being rolled back or committed.

The information about the priority of a transaction is stored in org.continuent.sequoia.controller.-requestmanager.TransactionMetaData. Using logical locks, when a master transaction tries to acquire a lock on a table and it has already been acquired by a normal transaction then this normal transaction is added to the queue of transactions to abort on org.continuent.sequoia.controller.locks.AbortThread. This thread is responsible for request the abort of transactions in its queue to RequestManager.

Consider two different transactions: T1 and T2, where T1 is a master (or high priority transaction) that is about to block due to locks held by T2 on table t. When T1 tries to acquire a lock on table t, T1 finds T2 holds a lock on t and notifies AbortThread to abort T2 and therefore releasing its locks.

3.3 Write Set Extraction

To capture the changes of a transaction to the database, Sequoia needs to be able to extract the write-set during a transaction execution. There are three types of tuple values that need to be saved: the new values in the case of an INSERT, the old values in the case of a DELETE, and both the old and the new values in the case of an UPDATE. To achieve this at the middleware level, one needs to set triggers in the backends capable of capturing the required values and saving them in the backend itself and, just before the transaction commits, to access and export them.

These triggered procedures are highly dependent on the real database engine used. Therefore, there is an interface to perform these operations which is the gorda.reflector.sequoia.utils.objectset.-SequoiaObjectSet:

- **enableGlobalWriteSetExtraction** registers a trigger to write to temporary tables the inserted, updated and deleted values and to perform a cleanup routine on commit;

- **enableLocalWriteSetExtraction** creates, for each connection, a temporary table that will collect the data;
- **getWriteSet** retrieves the inserted, updated and deleted values for each table.

Currently these trigger procedures are defined for PostgreSQL and for MySQL. They use temporary tables to save the required data. In the former for each connection, a temporary table is created for each original table in the database, with its original schema plus a field with the type of operation performed, a sequential timestamp that records the order of the operations, and the user id. In latter, due to restrictions in the use of temporary tables and the inability to change the primary keys with AUTO_INCREMENT configured, three temporary tables are defined for each original table in the database, one for each type: INSERT, DELETE and UPDATE. Each table has its original schema plus a field a sequential timestamp that records the order of the operations. Trigger procedures for other database management systems can easily be defined.

Chapter 4

GAPI Compliance

Figure 4.1 shows the high level architecture of a middleware-based implementation of GORDA. Compared to an in-core implementation (GORDA Deliverable D4.3 - In-Core Proof-of-concept) clients are not directly attached to a DBMS. Instead, clients attach to a Sequoia Controller (shaded box) through a Sequoia Driver that provides the application with a standard JDBC driver that replaces the original database JDBC driver.

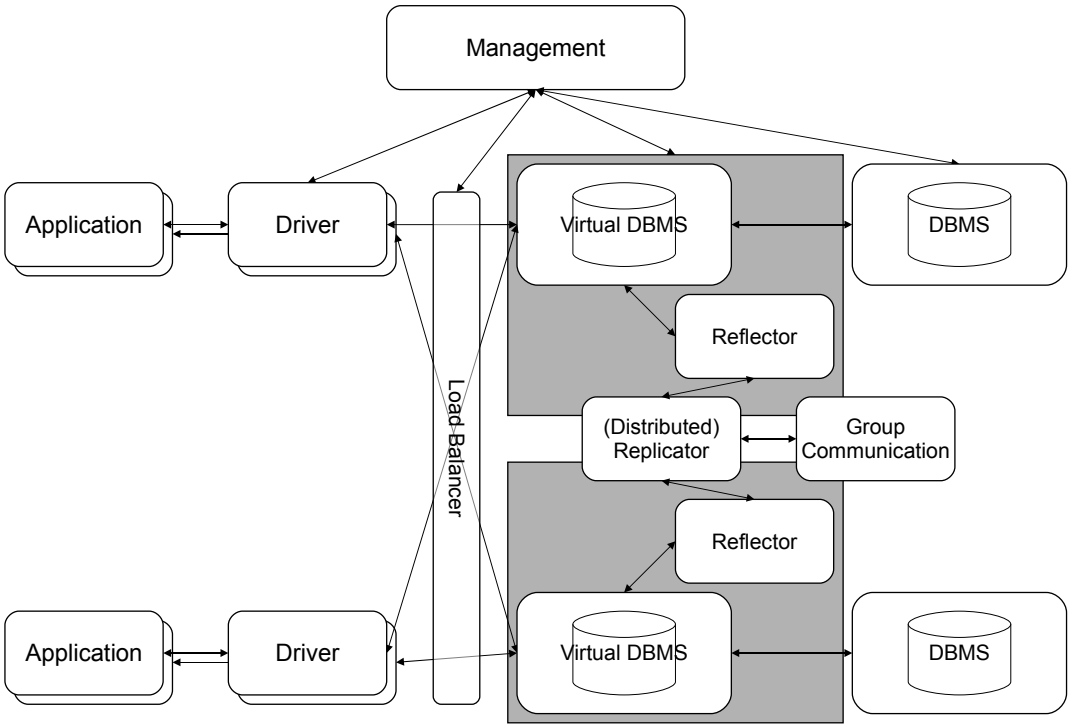

Figure 4.1: Generic GORDA architecture for a middleware implementation

The Sequoia Controller exposes the Reflector interfaces (as defined in GORDA Deliverable D2.3 - APIs Definition Report) to the Replicator component. The Replicator is a component that is implemented in each instance of the GORDA middleware and relies on the Group Communication component (as defined in GORDA Deliverable D2.2 - Architecture Definition Report). In its current state, the Sequoia implementation allows it to be used with all reference replication protocols described in GORDA

Deliverables D3.1 (Wide-Area Oriented Protocols Report) and D3.2 (Cluster Oriented Protocols Report).

The source code of the reflector interface has been placed in a different package (gorda.db.sequoia). The package gorda.db.sequoia contains the implementation of the GORDA reflector processing contexts and GORDA reflector processing stages. The package gorda.db.sequoia.demo contains the class that enables to run the GORDA reflector demos within sequoia. The package gorda.db.sequoia.escada contains the class that enables to run the replicator within sequoia. The package gorda.db.sequoia.utils contains classes used for writeset extraction and parse module.

The reflection service is disabled by default. The service can be enabled by specifying the location of class registering GORDA reflector events.

In the following we describe how the the reflector component (GORDA Deliverable D2.2 - Architecture Definition Report) is provided by Sequoias's Virtual Database implementation. Figure 4.2 describes how the different GORDA interfaces are mapped onto Sequoia building blocks.

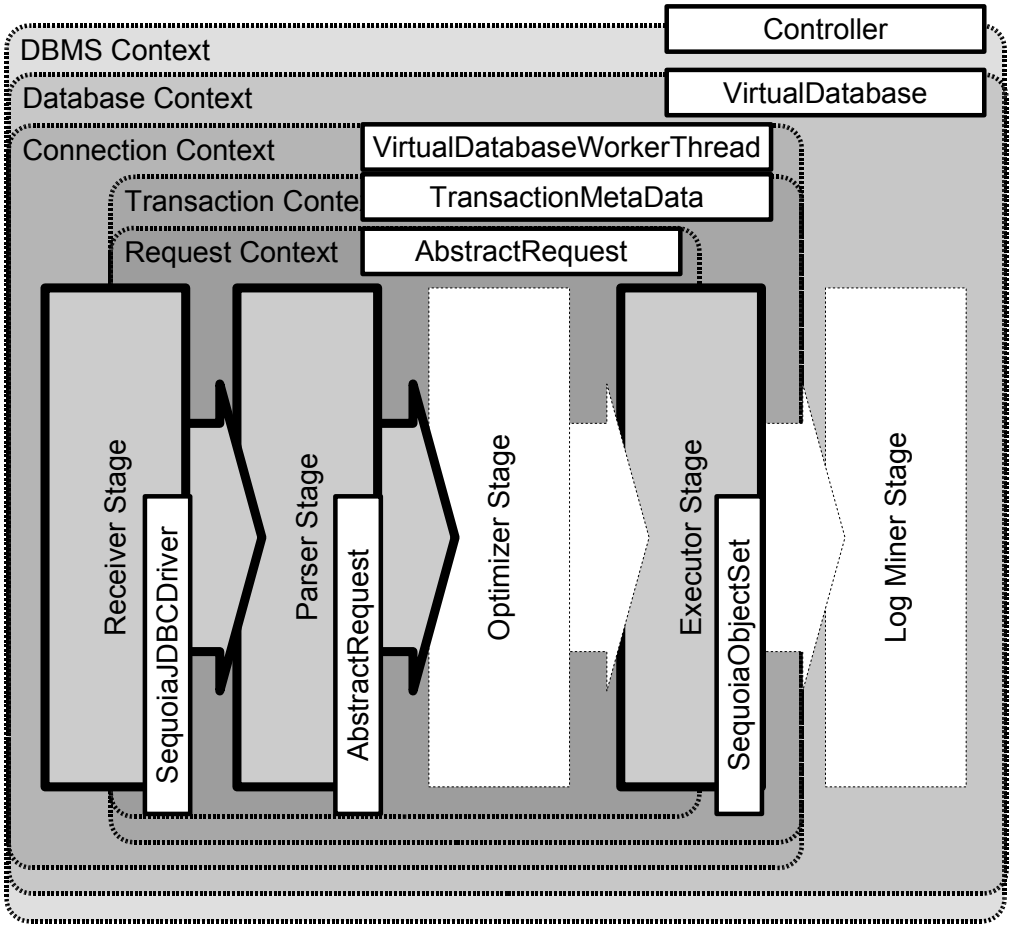

Figure 4.2: Transaction processing model

4.1 Processing Contexts

With respect to the processing contexts, the DBMS Context is a subset of the JDBC DatabaseMetadata that is fully supported by the Sequoia Controller. The client Connection Context builds upon the JDBC

Connection information (fully JDBC 3 compliant) with additional runtime information. The Transaction Context is represented internally in Sequoia through its TransactionMetadata container that includes cluster-wide unique transaction ids. The Request Context is essentially mapped to the Sequoia JDBC Statement implementation but specific implementations exist for PreparedStatement and CallableStatement (stored procedures).

The VirtualDatabaseWorkerThread handles a connection with a Sequoia driver thus plays an important role in the interception of the following events:*connectionStartUp*, *connectionShutDown*, *beginTransaction*, *beforeCommit*, *beforeRollback*, *afterRollback*, *afterCommit*, *beginRequest* and *finishRequest*.

4.1.1 Support and Implementation of the DBMS Module

The DBMS Context is started when the Sequoia Controller starts. At this point, administrative tasks are carried using Sequoia console. It is worth noticing that the DBMS identification is assigned upon start up of the Sequoia Controller and should be unique. Its assignment and management is not done automatically but it is done by the administrator. There is no problem in reusing ids as long as distinct systems have distinct ids. Canceling a DBMS startup involves invoking $System.exit()$ and aborting the Sequoia controller.

4.1.2 Support and Implementation of the Database module

Each Virtual Database represents a Database Context. The database start up event is generated by Sequoia whenever Virtual Database have at least on enabled Backend for this Virtual Database. The database shutdown is generated by the Sequoia whenever Virtual Database has no enabled Backends.

As detailed in Section 2.1.3 similar to a RDBMS a controller can host multiple virtual databases.

By calling $getDatabaseSource()$ one has access to a JDBC connection. If the URL used to acquire a connection is $jdbc{:}default$, statements are processed in the context of a transaction if there is one, otherwise an exception is raised.

This module is also responsible for providing access to the system meta-information. This information is accessible using org.continuent.sequoia.controller.virtualdatabase.VirtualDatabaseDynamicMetaData that gathers the dynamic metadata for a Virtual Database, that is, all the metadata subject to changes during the lifetime of the application.

The methods related to the transfer of the entire database image, used for recovery purposes, are implemented using the Backup Manager that performs database backup and restore operations. Currently the Backup Manager has several backupers implemented. In the context of GORDA, the Apache Derby, the PostgreSQL and the MySQL backupers are used.

4.1.3 Support and Implementation of the Connection Module

Sequoia creates a new thread org.continuent.sequoia.controller.virtualdatabase.VirtualDatabaseWorkerThread for each user. One might allow a connection to proceed or cancel it, by calling *continueExecution()* or *cancelExecution()*, respectively. In the latter case, the new connection is safely ended. The ability to cancel a connection is quite interesting when a database is doing recovery refusing user transactions.

4.1.4 Support and Implementation of the Transaction Module

Sequoia's Request Manager maintains a mapping between active transaction ids and transactions' meta data (TransactionMetaData class). This class carries transaction metadata including the transaction state.

While handling a notification, one might abort a transaction by calling *cancelExecution()*. In particular, before a commit request is issued, besides aborting or allowing a transaction to continue, one might

still update, delete and insert information on the transaction's context. This feature is exploited in our prototype to log information used during recovery.

4.1.5 Support and Implementation of the Request Module

A new Request is generated for each transaction Statement. Similarly to other modules, it is possible to cancel a request by calling *cancelExecution()*. When this method is invoked the request's transaction is aborted.

4.2 Processing Stages

Sequoia's JDBC driver provides receiving stage and parsing is fully contained in the Sequoia's controller. There is no optimization module for Sequoia since these operations are delegated to and performed by the underlying database engine. At the Execution stage the write set extraction mechanism (Section 3.3) provides the required data.

4.2.1 Support and Implementation of the Receiver Module

This module is fully implemented by Sequoia's JDBC driver and statements can be changed. A particularity of Sequoia, due to the request determinism required by the active replication on RAIDb, is that time/random macros values are set prior to sending the requests to the group communication protocol.

4.2.2 Support and Implementation of the Parse Module

Sequoia parses SQL requests and extracts the selected columns and tables given the DatabaseSchema of the database targeted by this request and previously acquired from the database backends. The depth of parsing depends on the parsing granularities defined in configuration.

Sequoia have the following parsing granularities:

- **NO_PARSING** the request is not parsed;
- **TABLE** table granularity: only table dependencies are computed; minor granularity required for logical locking and priority scheduling;
- **COLUMN** column granularity: column dependencies are computed (both select and where clauses), and
- **COLUMN_UNIQUE** column granularity with UNIQUE queries: same as COLUMN except that UNIQUE queries that select a single row based on a key are flagged UNIQUE (and should not be invalidated on INSERTs).

Sequoia defines the following hierarchy of requests after parsing:

- **AbstractRequest** defines the skeleton of an SQL request. Requests have to be serializable (at least) for inter-controller communications.
  - **AbstractWriteRequest** is the super-class of all requests which do NOT return any ResultSet. They do may have side-effects.
    - **AlterRequest** is a SQL request of the following syntax: `ALTER { AGGREGATION | CONVERSION | DATABASE | FUNCTION | GROUP | LANGUAGE | OPERATOR | SCHEMA | TABLE INDEX | TRIGGER | ROLE | USER | TABLESPACE } RENAME TO target`

* **CreateRequest** is a SQL request of the following syntax: `CREATE [TEMPORARY] TABLE table-name [(column-name column-type [,column-namecolumnm-type] * [,table-constraint-definition]*)]`
  Sequoia also supports SELECT INTO statements.
* **DeleteRequest** is an SQL request with the following syntax: `DELETE [table1] FROM table1,table2,table3,... WHERE search-condition` or `DELETE t WHERE search-condition`
  Note that DELETE from multiple tables is not supported as this is not part of the SQL standard.
* **DropRequest** is an SQL request with the following syntax: DROP TABLE table-name
* **InsertRequest** is an SQL request of the following syntax: `INSERT INTO table-name [(column-name[,column-name]*)] {VALUES (constant|null[,constant|null]*)}|{SELECT query}`
* **UnknownWriteRequest** is an SQL request that does not match any SQL query known by this sequoia.
* **UpdateRequest** is an SQL request with the following syntax: `UPDATE table-name SET (column-name=expression[,column-name=expression]*) WHERE search-condition`

– **SelectRequest** is an SQL request returning a ResultSet. It may also have database side-effects. It has the following syntax:

```
SELECT [ALL|DISTINCT] select-item[,select-item]*
FROM table-specification[,table-specification]*
[WHERE search-condition]
[GROUP BY grouping-column[,grouping-column]]
[HAVING search-condition]
[ORDER BY sort-specification[,sort-specification]]
[LIMIT ignored]
```

* **UnknownReadRequest** This class defines an UnknownReadRequest used for all SQL statements that are not SELECT but should be executed as read requests.
  An UnknownReadRequest is a request that returns a ResultSet and that we are not able to parse (we cannot know which tables are accessed, if any).

– **StoredProcedure** it encodes a stored procedure call. It can have the following syntax: `call <procedure-name>[<arg1>;,<arg2>;, ...]` or `?=call <procedure-name>[<arg1>,<arg2>, ...]`

4.2.3 Support and Implementation of the Executor Module

For performance reasons the write set is collected while executing a transaction as explained in Section 3.3 and before commit is triggered.

When a new connection is established the write set extraction for that connection is enabled as explained in Section 3.3.

In Sequoia each backend has a thread responsible for sequentially process a set of tasks. The Commit task defines the commit of a transaction in some database backend. When this task is executed the write set is gathered and the GAPI notified before triggering the *onCommit* event.

The method handling the notification of object set, clean the temporary table used for that temporary table.

Chapter 5

Maintenance and Further Development of GAPI in Sequoia

Continued maintenance of the GAPI work from GORDA will require investment from both Continuent, the main sponsor of the Sequoia project, as well as the open source community that uses and helps develop Sequoia. This chapter contains a summary of on-going work to ensure continued maintenance and use of the GAPI interfaces.

5.1 Long-Term Value of GAPI Model

GAPI provides a very general interceptor model that has value for a variety of applications beyond direct support for change set extract for replication purposes. Continuent has been able to confirm the commercial value of such interceptors to help support integration of Sequoia with MySQL replication to create clusters that operate between multiple sites. Continuent has introduced very simple interceptors into the current stable branch of Sequoia but needs the generality of the GAPI model. We believe this would also be of interest to the open source community, which can use the model to support a variety of interesting uses of Sequoia that are currently impossible due the monolithic processing model.

5.2 Preparations for Further Development on Sequoia

Continuent has over the course of 2007 and 2008 engaged in two important activities to prepare for the next round of development on Sequoia.

First and foremost, Continuent has invested considerable effort in stabilizing the Sequoia core to allow it to be used in production deployments. This included addition of numerous features as well as fixing well over 100 bugs in areas like recovery log handling, failover, SQL parsing, and error handling. These are a necessary prerequisite for engaging in further major developments on Sequoia.

Second, the Sequoia project, its codelines, and associated wikis and documentation are now hosted in a central location, which provides excellent network connectivity as well as high availability. The infrastructure at LogicWorks includes a significant test bed for builds and test based on Vmware.

These activities are pre-requisites for long-term maintenance and extension of GAPI interfaces.

5.3 Full Incorporation of GAPI into Sequoia

The current GAPI implementation for GORDA-M is part of Sequoia 3.0, an experimental version of Sequoia. This is a good environment for the initial development, which was designed to prove the

concept of GAPI interfaces. However, to incorporate GAPI fully, it needs to be based on the more stable production code maintained by Continuent.

Continuent and members of the open source community plan to handle incorporation of GAPI interfaces over two releases.

- Sequoia 4.0 - This will be a new release of Sequoia based on the current Continuent version 2.10 commercial code pruned to eliminate a number of features that are no longer necessary, harmful, or just not useful. Design work for this release is currently in progress. One of the most important "features" of the release will be an open source test suite that can be run by the community and will be the basis of builds to ensure that the new Sequoia 4.0 release is not subject to major regressions.
- Sequoia 5.0 - The GAPI will be formally released in this version. Our analysis shows that some reimplementation of the current GAPI will be required to match the cleaned-up code provided by Sequoia 4.0. The 4.0 clean-up will in fact make GAPI much cleaner as there is considerable duplicated code in Sequoia that makes interceptor implementations quite fragile.

5.4 Looking Forward

GAPI interfaces look like a great edition to Sequoia. We look forward to the additional flexibility that will be provided by GAPI when it is baked into Sequoia, backed by solid tests and documentation. We hope to enable both new open source uses (and innovation) as well as new commercial applications.

Bibliography

[1] N. Carvalho, J. Pereira, and L. Rodrigues. Towards a generic group communication service. In *Proceedings of the 8th International Symposium on Distributed Objects and Applications (DOA)*, Montpellier, France, October 2006.

[2] Emmanuel Cecchet. Raidb: Redundant array of inexpensive databases. In Jiannong Cao, Laurence Tianruo Yang, Minyi Guo, and Francis Chi-Moon Lau, editors, *ISPA*, volume 3358 of *Lecture Notes in Computer Science*, pages 115–125. Springer, 2004.