Project no. 004758

GORDA

Open Replication Of Databases

Specific Targeted Research Project

Software and Services

GORDA Architecture Definition

GORDA Deliverable 2.2

Due date of deliverable: 2006/03/31
Actual submission date: 2006/04/24
Revision 1.1 date: 2007/09/29

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

FFCUL

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

Sara Bouchenak, INRIA
Alfranio Correia Jr., U. Minho
Nuno Carvalho, U. Lisboa
Nuno A. Carvalho, U. Minho
Emmanuel Cecchet, Continuent
Susana Guedes, U. Lisboa
Adrian Mos, INRIA
Rui Oliveira, U. Minho
José Pereira, U. Minho
Luís Rodrigues, U. Lisboa
Luís Soares, U. Minho
Ricardo Vilaça, U. Minho

Abstract

This document describes the GORDA Architecture, which enables independent development of database management systems (DBMS) and database replication systems. The architecture builds on the classic database management system's architecture with remote database access, a standard call-level interface and supporting SQL requests. The proposed architecture models the system through two articulated plans: one that reflects the standard transaction pipeline and the other a set of contexts reflecting the scope of the various operations through the transaction processing pipeline. The replicated database management system architecture is complemented by a generic management architecture.

Contents

Chapter 1

Introduction

This document describes the GORDA Architecture, which enables independent development of database management systems (DBMS) and database replication systems.

The document is structured as follows. Chapter 2 summarizes concepts and assumptions, referring to available standards where appropriate. This includes describing the architecture of a non-replicated DBMS that serves as a baseline for comparison. Chapter 3 describes the generic GORDA Architecture, its components and relations. It then describes several concrete refinements for different implementation scenarios and shows how to use the presented architecture in a specific use case scenario.

We do not include information on database management systems, communication protocols, or tools themselves.

1.1 Relationship With Other Deliverables

This document, along with its companion deliverable D2.3 - APIs Definition Report, refines deliverable D2.1 - Preliminary Architecture and API and serves as the basis for most of the development of workpackages 3 (Replication Protocols) and 4 (Database Support).

With respect to the preliminary report D2.1, it obsoletes its sections 2 and 3 (the companion document D2.3 does the same for the remaining sections 4 and 5).

Since the general architecture and guidelines proposed in D2.1 have been considered to be correct, the current document makes only minor updates that reflect experience earned while implementing both the replication protocols and database support components. Namely:

- A new processing context, Database Context, has been added to support multiple active databases within the same DBMS.
- The Statement Context has been renamed Request Context and its meaning changed slightly to better handle different database implementations.

- The meaning of the Physical Storage Stage has been changed. While initially it was supposed to allow synchronization of commit requests, it is now used only to inspect logs after transaction commit. Synchronization with commit requests is now done through the Transaction Context.
- The scope of processing contexts has been changed to be consistent with the revised definition of the Physical Storage Stage.

Chapter 2

Assumptions and approach

2.1 Generic database architecture

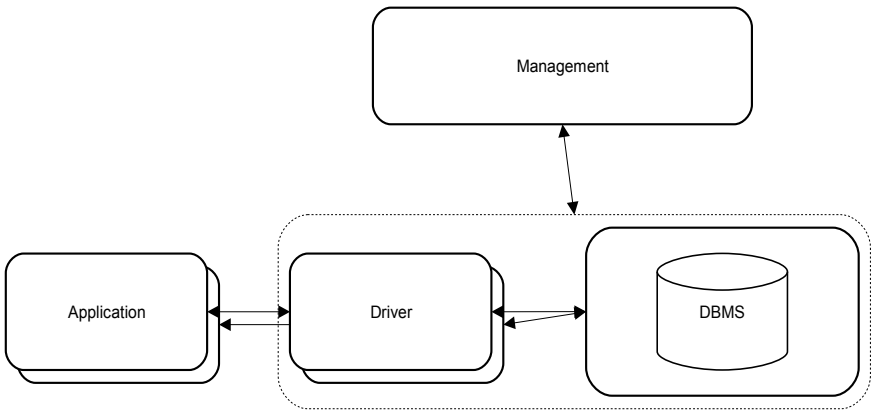

Figure 2.1: Generic database architecture.

The GORDA Architecture builds on the assumption of the generic database architecture with remote database access, a standard call-level interface, and SQL as shown in the Figure 2.1.

The main components of this model are:

- The Application, which might be the end-user application or a tier in a multi-

tiered application.

- The Driver provides a standard call-level interface (CLI) for the application and remotely accesses the database itself using a communication mechanism. The communication protocol is hidden from the application and can be proprietary.
- The DBMS holds the database content and handles remote requests expressed in standard SQL to query and modify data.
- Management tools are able to control the Driver and DBMS components independently from the Application using a mixture of standard and proprietary interfaces.

Further assumptions on these systems are that:

- The call-level interface and SQL should not be changed, and cannot be changed at all in a backward incompatible manner.
- Some DBMS implementations can be modified in a backward compatible manner, but some others cannot be modified at all.
- The remote database access protocol should not be changed to maintain compatibility with third party tools.
- The driver can easily be changed with minor impact.

This simple architecture can easily be mapped to a Java system, using JDBC as the call-level interface and driver specification, any remote database access protocol encapsulated by the driver and a DBMS, and an external configuration tool for the JDBC driver.

2.2 Relevant standards

The GORDA Architecture and Programming Interface (GAPI) are based on existing data management standards. Namely:

- ISO/IEC 10032:1995 - Reference Model of Data Management (RMDM) specifies typical data management architectures and nomenclature.
- ISO/IEC 9075-3:1995 - Call Level Interface (SQL/CLI) and X/Open XA Distributed Transaction Processing (DTP) specify client interfaces.
- ISO/IEC JTC1/SC32 working drafts on Distributed Database Access and Schemas for client information.

The rendering of interfaces in the Java language is therefore based on existing implementations of such standards in the Java platform, in particular, in Java Enterprise Edition (JEE).

2.3 Design principles

The design of the GORDA Architecture stands on the following general principles:

- Independence of operation and configuration. All configuration interfaces are assumed to be out of the scope of the present specification. Configuration of components and relevant parameters is available by means of an embedded directory service and the factory design patterns.
- Variable geometry interfaces. Each implementer is free to provide only a subset of the whole GAPI that is adequate for each situation. Each component should therefore explicitly state requirements and check for the availability of all requirements. This is however part of configuration and thus out of the scope of this specification.
- Facade interfaces that allow manipulation of the internal state of the DBMS without forward and backward format conversions. Conversion to a DBMS independent representation if necessary is achieved by an optional layer on basic interfaces.

Chapter 3

Architecture

The GORDA Architecture specifies the building blocks for a replicated DBMS. At an abstract level, these building blocks can be mapped to existing monolithic implementations regardless of the implementation. The GORDA Architecture is however the basis for the definition of interfaces between such components that allows them to be reused in different contexts.

This Section discusses the GORDA architecture. Implementation issues are discussed in the Deliverables D4.3, D4.4 and D4.5 of the project.

3.1 Reflective architecture for replication

In this section we outline the reflection usage in the GORDA architecture, as well as the underlying rationale. The GORDA architecture considers a multi-stage model of transaction processing that can be replicated by observing and modifying it through a reflector interface. The main intent of the reflection is to allow the replication component to observe the state of the DBMS. Specifically, the reflector exposes transaction-processing concepts such as parse trees, write sets, or transactions as first class entities in the target programming language. Replication protocols can register for significant events to be notified of relevant state transitions and call methods to alter the state.

3.1.1 Target Reflection Domain

Existing reflective facilities in database management systems are targeted at application programmers using a relational model. Its domain is therefore the relational model itself. With it, one can intercept operations that modify relations by inserting, updating, or deleting tuples, observe the tuples being changed and then enforce referential integrity by vetoing the operation (all at the meta-level) or by issuing additional relational operations (base-level).

In contrast, a replication protocol is concerned with details that are not visible in the relational model, such as modifying query text to remove non-deterministic or the precise scheduling of updates to achieve a given isolation level. For instance, one may be interested in intercepting a statement as it is submitted, whose text can be inspected, modified (meta-level) and then re-executed, locally or remotely, within some transactional context (base-level). Therefore, a more expressive target domain is required. The fact that a series of activities (e.g. parsing) is taking place on behalf of a transaction is reflected as a transaction object, which can be used to inspect the transaction (e.g. wait for it to commit) or to act on it (e.g. force a rollback). Meta-level code can register to be notified when specific events occur. For instance, when a transaction commits a notification is issued, containing a reference to the corresponding transaction object (meta-level). Actually, handling notifications is the way that meta-level code dynamically acquires references to meta-objects describing the on-going computation.

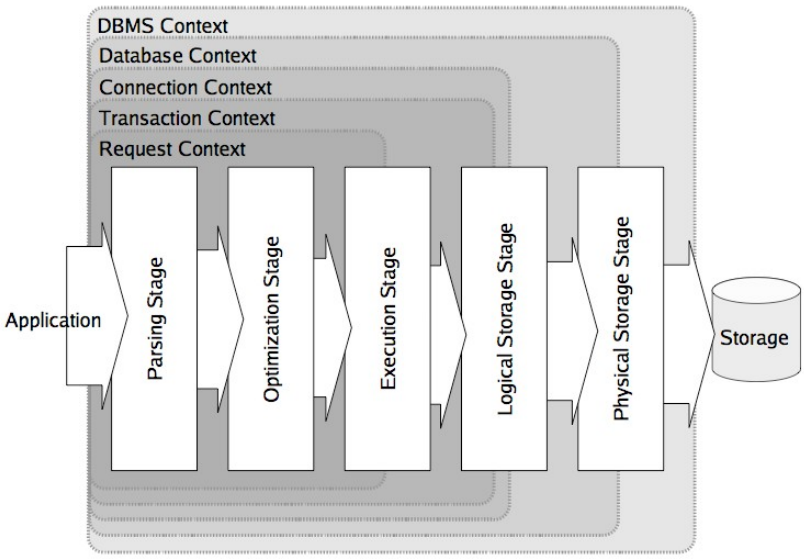

Figure 3.1: Transaction processing model.

The transaction-processing pipeline assumed is shown in Figure 3.1 with the following stages after acceptance of requests from a client, which includes dealing with the appropriate call-level interface or most likely, a remote database access protocol:

- Parsing of the SQL statement received, resulting in a parse tree.
- Optimization, in which the parse tree is transformed according to optimization criteria and statistics. The result is an execution plan.
- The execution stage executes the plan and produces write sets and result sets. It might also produce lock-grabbed and lock-blocked sets in order to inform which locks were acquired and which locks are blocking the execution respectively. The format of the sets is highly dependent on the implementation.
- The logical storage layer deals with access to logical objects.
- The final physical storage stage allows asynchronous capture of updates to the database and access to database logs for recovery.

Note that such stages are not mutually exclusive. It is likely that different parts of the same transaction or even of the same statement are in different stages of the pipeline.

All events and operations on transaction processing are provided in four nested contexts:

- DBMS Context identifies the reflector and thus the DBMS originating events.
- Database Context (introduced in this Deliverable) identifies a specific database within a DBMS. Multiple databases may be active within a DBMS.
- Connection Context identifies a specific client within a Database. Multiple connections may be active within a Database.
- Transaction Context identifies a specific transaction within a Connection Context. It is assumed that at most a single top-level transaction exists at any given time within the same connection context.
- Request (also introduced in this Deliverable) identifies a specific request within a Transaction Context. It is assumed that at most a single request exists at any given time within the same transaction context.

3.1.2 Processing Stages

The usefulness of the meta-level interface depends on what is exposed as meta-objects. If a very fine granularity is chosen, the interface cannot be easily mapped to different DBMSs and the resulting performance overhead is likely to be high. On the other hand, if a very large granularity is chosen, the interface may expose too little to be useful.

Therefore, we abstract transaction processing as a pipeline as it is commonly accepted [3] (Figure 3.1). In detail, the Parsing stage parses raw statements received thus producing a parse tree. The parse tree is transformed by the Optimization stage according to various optimization criteria, heuristics and statistics to an execution plan. The Execution stage executes the plan and produces object-sets. The Logical Storage stage deals with mapping from logical objects to physical storage. Finally, the Physical Storage stage deals with block input/output and synchronization.

In general, one wants to issue notifications at the meta-level whenever computation proceeds from one stage to the next. For instance, when write-sets are issued at the execution stage, a notification is issued such that they can be observed. The interface thus exposes meta-objects for each stage and for data that moves between them.

In contrast, previous approaches assume that reflection is achieved by wrapping the server and intercepting requests as they are issued by clients [5]. By choosing beforehand such implementation approach, one can only reflect computation at the first stage, i.e. with a very large granularity. Exposing further details requires rewriting large portions of DBMS functionality at the wrapper level. As an example, Sequoia [1] does additional parsing and scheduling stages at the middleware level.

3.1.3 Processing Contexts

The meta-interface exposed by the processing pipeline is complemented by nested context meta-objects, also shown in the Figure 2. These show on behalf of whom some operation is being performed. In detail, the DBMS and Database context interfaces expose metadata and allow notification of lifecycle events. Connection contexts reflect existing client connections to databases. They can be used to retrieve connection specific information, such as user authentication or the character set encoding used. The Transaction context is used to notify events related to a transaction such as its startup, commit or rollback. Synchronous event handlers available here are the key to the synchronous replication protocols presented in this document. Finally, to ease the manipulation of the requests within a connection to a database and the corresponding transactions one may use the Request context interface.

Events fired by processing stages refer to the directly enclosing context. Each context has then a reference to the next enclosing context and can enumerate all enclosed contexts. This allows, for instance, determining all connections to a database or which is the current active transaction in a specific connection. Notice that some contexts are not valid at the lowest abstraction levels. Namely, it is not possible to determine on behalf of which transaction a specific disk block is being flushed by the physical stage.

3.1.4 Base-level and Meta-level Calls

An advantage of reflection is that base- and meta-level code can be freely mixed, as there is no inherent difference between base- and meta-objects. In detail, the application programmer can force a direct call to meta-level code by registering it as a native procedure and then using the CALL SQL statement. This causes a call to the meta-level code to be issued from the base-level code within the Execute stage. The target procedure can then retrieve a pointer to the enclosing Request context and thus to all relevant meta-interfaces. The reason for allowing this only from the Execute stage is simplicity, as this is inherently supported by any DBMS, and does not seem to impact generality. A second reason is that this is where the pipeline can be reentered, should the meta-level procedure need to callback into the base-level.

Meta-level code can callback into base level in two different situations. The first is within a direct call from base-level to issue statements in an existing enclosing request context. This can be achieved using the JDBC client interface by looking up the "jdbc:default:connection" driver, as is usually done in Java procedures. The second option is to use the enclosing Database context to open a new base-level connection to the database. The reason for allowing base-level to use the JDBC interface is again simplicity, as this avoids the need to have interfaces that build contexts and inject external data into internal structures. This may however have an impact on performance, and is thus the subject of future work.

A second issue when considering base-level calls is whether these also get reflected. The proposed option is to disable reflection on a case-by-case basis by invoking an operation on context meta-objects. Therefore, meta-level code can disable reflection for a given request, a transaction, a specific connection or even an entire database. Actually this can be used on any context meta-object and thus for performance optimization. For one, consider a replication protocol, which is notified that a connection will only issue read-only operations, and thus ceases monitoring them.

A third issue is how base-level calls issued by meta-level code interact with regular transaction processing regarding concurrency control. Namely, how are conflicts that require rollback resolved, namely, in multi-version concurrency control where the first commiter wins or, more generally, when resolving deadlocks. The proposed interface solves this by ensuring that transactions issued by the meta-level do not abort in face of conflicts with regular base-level transactions. Given that replication code running at the meta-level has a precise control on which base-level transactions are scheduled, and thus can prevent conflicts among those, has been sufficient to solve all considered use cases. The simplicity of the solution means that implementation within the DBMS resulted in a small set of localized changes.

3.1.5 Design Patterns

The design of meta-level interfaces leverages patterns that have proven useful in object oriented middleware. The first is the façade, which allows inspection of diverse data structures through a common interface. A very well known example is the ResultSet, which allows results to be stored in a DBMS native format. The proposed architecture suggests using this for most of the data that is conveyed between processing stages (e.g. object sets).

The second is the inversion-of-control pattern, which eases deployment of software components. In detail, meta-objects such as transactions are exposed to an object container, which is configured with replication components. The container is then responsible for injecting the required meta-objects into each replication component during initialization.

The third pattern is the container-managed concurrency. The container implementation schedules event notifications according to performance and correctness criteria. For instance, by ensuring that no two transactions commit notifications are issued concurrently, implicitly exposes a commit order. Notification of available write-sets of two different transactions can be issued concurrently.

3.2 Generic architecture and composition scenarios

This section describes how the components can be used in several specialization scenarios of the architecture. It overviews the components and shows their interaction and roles in the architecture.

The generic replication architecture shown in Figure 3.2 extends the data management architecture with the following components:

- A Reflector is attached to each DBMS allowing inspection and modification of the process. This is achieved by reflecting transaction processing concepts to objects in the target language.
- A Replicator component attaches to multiple DBMS by means of reflector interfaces and ensures their consistency. Most likely, this is a distributed component which makes use of a communication service.
- Clients may not be directly attached to a single DBMS. Instead, they may be dispatched dynamically and transparently by means of a load-balancer component that intercepts client requests.

Specialization of the generic architecture provide direct mappings with possible or actual replicated DBMS. Namely, multiple logical components can be provided by a single physical component. What is required is that replication, communication,

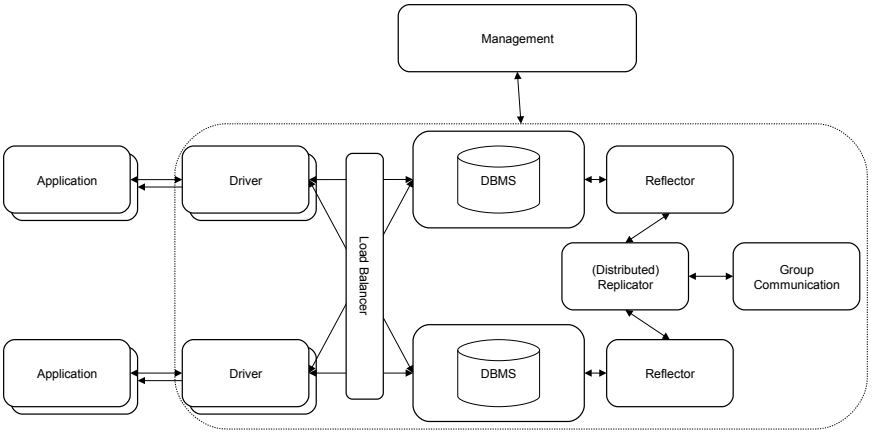

Figure 3.2: Generic database replication.

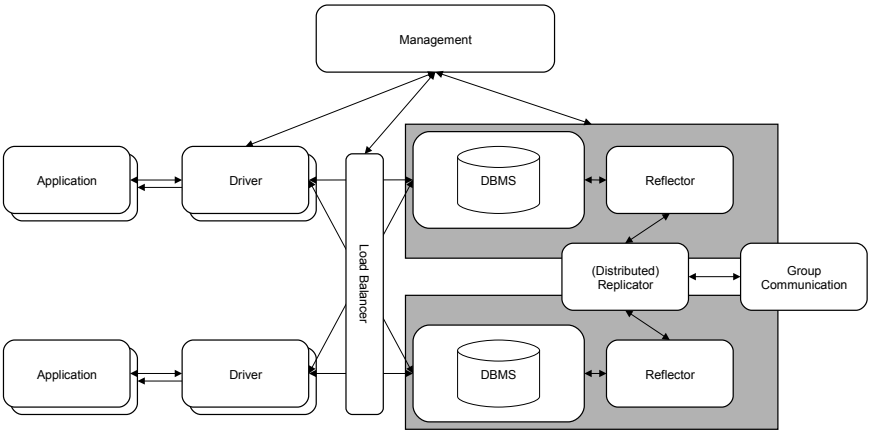

Figure 3.3: Specialization for an in-core implementation.

and management components are portable and can therefore be combined with different DBMS.

As an example, consider in Figure 3.3 the situation in which the reflector is provided within the same physical component as the DBMS, where replication and communication components can be installed to control replication.

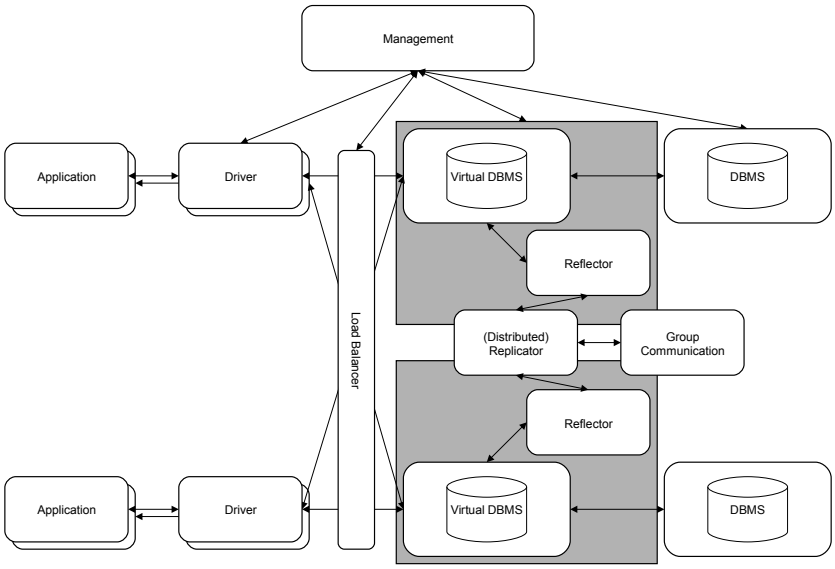

Figure 3.4: Specialization for a middleware-based architecture.

A DBMS independent implementation can be achieved strictly by intercepting the remote database access interface as suggested in Figure 3.4. In this situation, clients connect to a virtual DBMS which implements the reflector interface. The virtual DBMS is itself implemented by relying on client interfaces provided by the real DBMS.

There is no need that each virtual DBMS directly maps to a backend DBMS or that all virtual DBMS accept clients, or that at most a single reflector exists in each physical package. This is the case of C-JDBC with a single controller implementing RAIDb protocols, which is depicted in Figure 3.5.

Note that the previous examples show how the GORDA Architecture maps possible implementations but they do not specify exactly what the interfaces of the reflector are or how these interfaces are implemented in concrete DBMS.

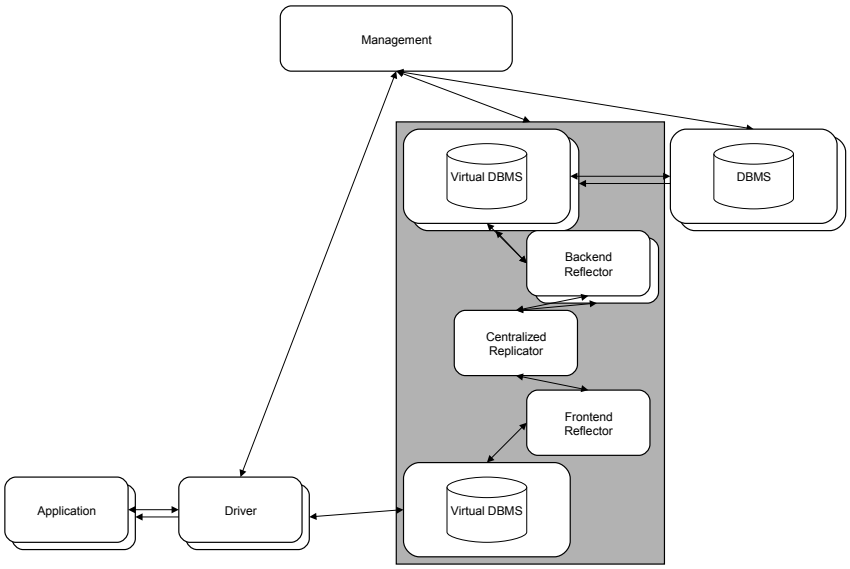

Figure 3.5: Specialization for the single controller C-JDBC architecture.

3.3 Architecture components and usage

This Section describes the architecture components. We will give more focus on the reflection, replication and Management components. The clients could be connected directly to a single DBMS or they may be dispatched dynamically and transparently by means of a load-balancer component that intercepts client requests. In any case, its standard execution remains the same. The group communication is used by the replication component and its interaction is described in that scope.

3.3.1 Reflection and Replication

These two components work together. The reflection component reflects the execution of the DBMS as already described in this document. The reflected execution is used by the replication component to replicate the requests on all databases. The group communication component is an important building block for the replication component, since it provides properties such as atomic broadcast in the messages that are exchanged by the several replication instances.

The execution of the replication component always use these other components but with different behaviors, depending on the replication protocol used. To illustrate how the replication, reflection and group communication components can be used in the architecture, we will use the primary-backup replication protocol. In the primary-backup approach to replication, also called passive replication [4], update transactions are executed at a single master site under the control of local concurrency control mechanisms. Updates are then captured and propagated to other sites. Asynchronous primary-backup is the standard replication in most DBMSs and third-party offers. An example is the Slony-I package for PostgreSQL [2]. Implementations of the primary-backup approach differ whether propagation occurs synchronously within the boundaries of the transaction or, most likely, is deferred and done asynchronously. The latter provides optimum performance when synchronous update is not required, as multiple updates can be batched and sent in the background. The Primary-Backup protocol has a primary replica where all transactions that update the database are executed.

Synchronous primary-backup replication requires the reflection component for the Transaction context to capture the moment where the transaction starts executing, commits, or rollbacks at the primary. It will also need the object set provided by the Execution stage to extract the write set of a transaction from the primary and insert it at the backup replicas.

The execution of the primary-backup replication in the GORDA architecture is depicted in the Figure 3.6. We start by describing the synchronous variant. It consists of the following steps:

1. Clients send their requests to the primary replica;

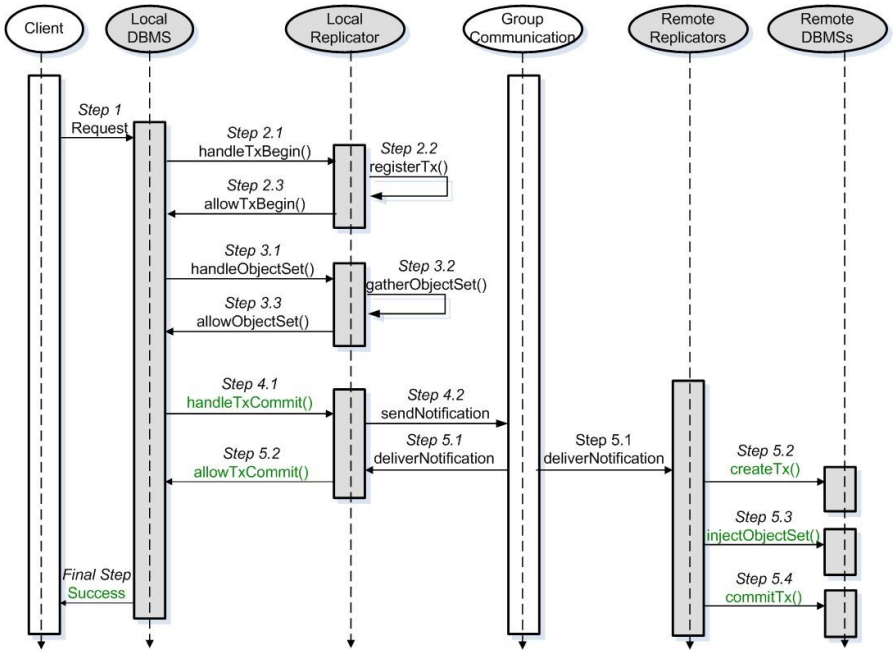

Figure 3.6: Primary backup using the GORDA architecture.

2. When a transaction begins, the replication component at the primary replica is notified by the reflection component; registers information about this event, and allows the primary replica to proceed;

3. Right after processing a SQL command the reflection component notifies the replication component through the Execution stage with an ObjectSet. Roughly, the ObjectSet provides an interface to iterate on a result set (e.g., write set). Specifically, in this case, it is used to retrieve statement's updates which are immediately stored in a in-memory structure with all other updates from the same transaction context;

4. When a transaction is ready to commit, the reflection component notifies the replication component of the primary replica. The replication component uses the group communication component to atomically broadcast the gathered updates to all backup replicas (this broadcast should be uniform); the write set is received at all replicas. On the primary replica, the replication component allows the transaction to commit. On the backups, the replication component injects the changes in the DBMS;

5. (Final Step) After the transaction execution, the primary replica replies to the client.

An asynchronous variant of the algorithm can be achieved by postponing Step 4 (and, consequently, Step 5) for a tunable amount of time.

3.3.2 Management

This Section overviews the architecture of the management component, focusing in the management of a cluster instead of a single DBMS. The management component should manage a cluster of DBMSs that were enriched with replication components. The consistency of data, availability and failure detection/reaction are key issues that must be taken into account in the management component.

Figure 3.7 describes the generic GORDA Management Architecture (GMS) and its main features and reconfiguration mechanisms, namely the *QoS Manager* and the *Failure Manager*. Roughly speaking, both are based on a control loop with the following components:

- First, *sensors* responsible for the detection of the occurrence of particular events, such as database failures, or QoS requirement violations.

- Second, *analysis/decision* components that represent the actual reconfiguration algorithm, e.g. replacing a failed database by a new one, or increasing the number of resources in a cluster of replicated databases upon high load.

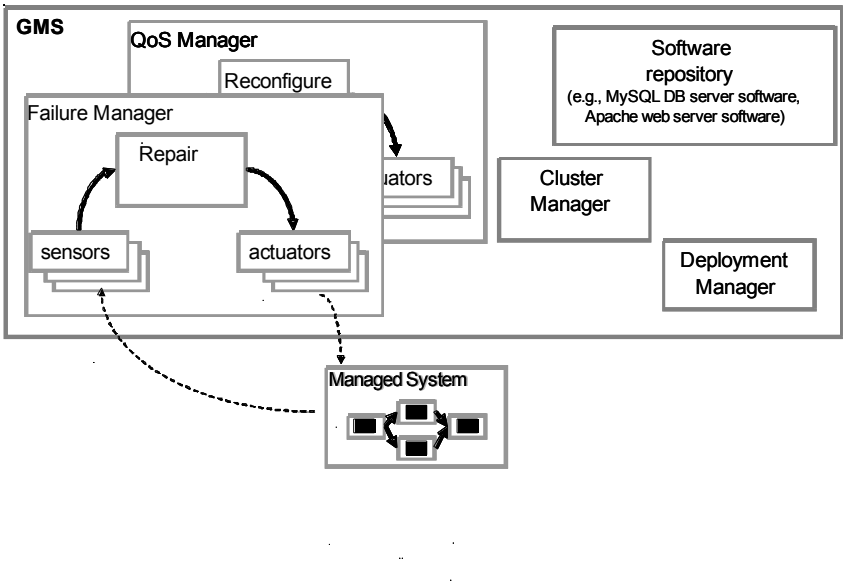

Figure 3.7: Generic replicated database management architecture.

- Finally, *actuators* that represent the individual mechanisms necessary to implement reconfiguration, e.g. allocation of a new node in a cluster.

The *Failure Manager* is used for self-repair. In a replication-based system, when a replicated resource fails, the service remains available due to replication. However, we aim at autonomously repairing the managed system by replacing the failed replica by a new one. Our current goal is to deal with fail-stop faults. The proposed repair policy rebuilds the failed managed system as it was prior to the occurrence of the failure. Sensors are used to detect failures in the replicas (such as a node crash) and decision policies are analyzed in order to determine the appropriate action to be taken. Once an action has been decided, actuators are used to enforce it (i.e. allocating a free node and deploying the appropriate software packages on it).

The *QoS Manager* is used for self-optimization. Self-optimization is an autonomic behavior which aims at maximizing resource utilization to meet the end user needs with no human intervention required. A classical pattern is when a given resource R is replicated statically at deployment time and a front-end proxy P acts as a load balancer and distributes incoming requests among the replicas. GMS aims at autonomously increasing/decreasing the number of resources used by the application when the load increases/decreases. This has the effect of efficiently adapting resource utilization (i.e. preventing resource overbooking). This can be done at different levels:

- In a shared server, allocated resources such as connection pools and memory buffers can be adjusted.
- In a cluster with shared storage, replicas can be migrated to servers with appropriate capacity.
- In a cluster with virtualized storage, replicas can be provisioned and discarded efficiently.
- In a wide area network, replicas and fragments can be repositioned to adjust to traffic patterns.

The *Deployment Manager* automates and facilitates the initial deployment of the managed system. For this purpose, the *Deployment Manager* makes use of two other mechanisms in GMS: the *Cluster Manager* and the *Software Repository*.

The *Cluster Manager* is responsible for the management of the resources (i.e. nodes) of the cluster on which the managed system is deployed. A node of the cluster is initially free, and may then be used by an application component, or may have failed. The *Cluster Manager* provides an API to allocate free nodes to the managed system/release nodes after use. Once nodes are allocated to an application, GMS deploys on those nodes the necessary software components that are used by the managed system.

The *Software Resource Repository* allows the automatic retrieval of the software resources involved in the managed application. For example, in case of an

e-business multi-tier JEE web application, the used software resources may be a MySQL database server software, a JBoss enterprise server software, and an Apache web server software.

Once nodes have been allocated by the *Cluster Manager* and software resources necessary to an application retrieved from the *Software Resource Repository*, those resources are automatically deployed on the allocated nodes. This is made possible due to the API provided by nodes managed by GMS, namely an API for remotely deploying software resources on nodes.

3.3.3 Recovery and security aspects

Before a joining replica can start executing transactions, the state of its database must be brought up-to-date with respect to the rest of the system. Site that crashes and recovers will perform some clean-up on its copy of the database and then will join the group again.

Recovery in the GORDA architecture is made at the replication component. The replication component uses the interface provided by the reflection component for retrieving the full database image implemented depending on the dump/restore tools of the DBMS. It is also possible to use the GORDA pipelines to do logging of missed transactions at the replication component.

One of the features provided by the group communication component is a failure detector and a mechanism to merge concurrent views in the system. This feature is used to detect joining/failure of replicas and for synchronizing the state transfer with normal processing.

One final aspect that should be taken into account is how to maintain security on the connections between the several components of the system. As this document should focus on the replication architecture, we will not focus on the security issues in the database management systems. Instead, we will focus on the components that were added to support replication.

The connections between the several replicas of the system are maintained by the group communication component. This component is responsible for creating and maintaining communication channels using sockets. The sockets are used by the group communication protocols that ensure reliability, ordering guarantees, failure detection and view synchrony. The toolkit that creates these sockets should use the Secure Socket Layer (SSL), among with trusted certificates, when it's necessary to provide secure channels between the replicas in a GORDA compliant system.

Bibliography

[1] Continuent. Sequoia version 2.9.

[2] PostgreSQL Global Development Group. Slony-i version 1.1.5.

[3] J. Ullman H. Garcia-Mollina and J. Widom. *Database Systems: The Complete Book*. Prentice Hall, 2002.

[4] S. Mullender, editor. *Distributed Systems*. ACM Press, 1989.

[5] J. Salas, R. Jimenez-Peris, M. Patino-Martinez, and B. Kemme. Lightweight reflection for middleware-based database replication. In *SRDS '06: Proceedings of the 25th IEEE Symposium on Reliable Distributed Systems (SRDS'06)*, pages 377–390, Washington, DC, USA, 2006. IEEE Computer Society.