



Project no. 004758

GORDA

Open Replication Of Databases

Specific Targeted Research Project

Software and Services

Preliminary GORDA Architecture and Interfaces GORDA Deliverable D2.1

Due date of deliverable: 2005/10/31
Actual submission date: 2005/09/12

Start date of project: 1 October 2004

Duration: 36 Months

Fundação da Faculdade de Ciências da Universidade de Lisboa

Revision draft 1.0

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)	

Contents

1	Scope.....	3
1.1	Limitations of the preliminary specification	3
2	Assumptions and Approach	5
2.1	Database architecture	5
2.2	Relevant standards	6
2.3	Design principles	6
3	Architecture.....	7
3.1	Generic replication architecture.....	7
3.2	Generic management architecture	11
4	Programming Interfaces.....	14
4.1	Client interfaces.....	14
4.2	Database interfaces	14
4.3	Management interfaces	18
4.4	Communication interfaces.....	20
4.5	Threading.....	22
5	Use cases	24
5.1	Asynchronous replication.....	24
5.2	Synchronous replication.....	25
5.3	Managing Replicas.....	25
5.4	Group communication.....	26

1 Scope

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

The GAPI provides the means for efficiently intercepting, observing, and modifying transaction processing in a DBMS independent fashion. Generic interfaces for management and communication are also provided, respectively, to support autonomic management and group communication.

In detail, this document is structured as follows:

- Section 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.
- Section 3 describes the generic GORDA Architecture, its components and relations. It also describes several concrete refinements for different implementation scenarios.
- Section 4 describes GORDA Programming Interfaces exhibited by each component.
- Section 5 illustrates the usefulness of the GAPI by showing how it can be used to implement various replication protocols.

This document does not include information on database management systems, communication protocols, or tools themselves and how these meet the proposed interfaces. This information can be found in companion mapping documents “GORDA In-core Mapping – D4.1” and “GORDA Middleware Mapping – D4.2”.

A rendering of these interfaces in the Java programming language and detailed documentation is provided in the companion document “GORDA PI in Java – D2.1 Annex” for reference.

1.1 Limitations of the preliminary specification

The current preliminary form of the GORDA Architecture and Programming Interfaces specification has the following limitations that will be overcome in the final specification:

- Only a call-level interface expressed in Java is provided. No mapping to other languages or remote access protocols is provided.
- No client interface to selectively break transparency to applications is provided.

No support for distributed transactions (XA); savepoints and partial rollback for savepoints; or.

Blobs and Clobs.

- No interface for external utility modules that should be useful when implementing replication systems or the GAPI itself, such as persistent logging or generic scheduling.

Specific interfaces may vary in the final version according to the experience gathered during later stages of implementation and evaluation.

2 Assumptions and Approach

2.1 Database architecture

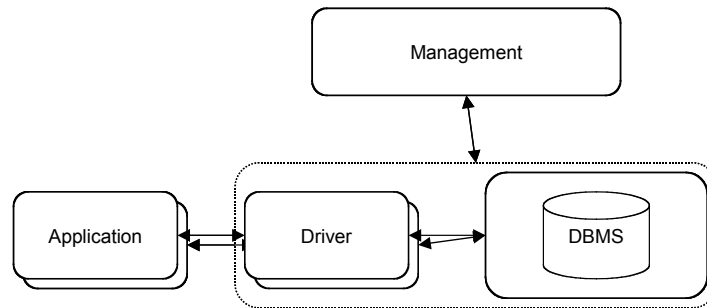


Figure 1 - Generic database management architecture

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 Figure 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 and Programming Interfaces 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 pattern.
- 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.
- Façade 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.

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.

3.1 Generic replication architecture

The GORDA Architecture considers a multi-stage model of transaction processing which can be replicated by observing and modifying it through a reflector interface. Basically, it allows that the database observes its own state so that it can change its own behavior dynamically. The main intent of the reflector is to allow the database to observe its own state. 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.

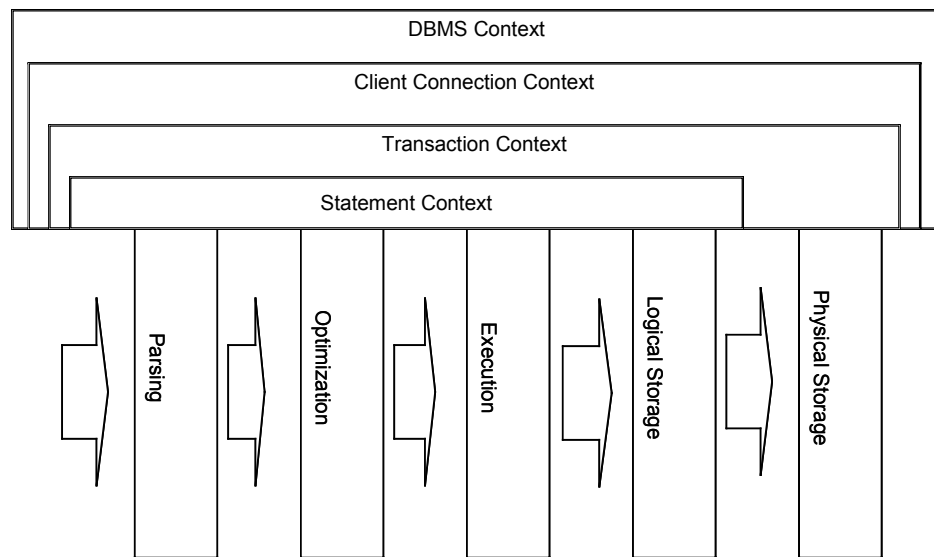


Figure 2 - Transaction processing model

The transaction processing pipeline assumed is shown in Figure 2 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 layer deals mostly with synchronization of commit requests, guaranteeing that a reply is issued to the client only after durability is ensured.

Notice 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.
- Client Connection Context identifies a specific client within a DBMS. Multiple connections may be active within a DBMS.
- Transaction Context identifies a specific transaction within a Client Connection Context. It is assumed that at most a single top-level transaction exists at any given time within the same connection context.
- Statement Context identifies a specific interaction step with the client. It is assumed that at most a single active statement exists at any given time within the same transaction context; otherwise the transaction is in an idle state.

According to the variable geometry design principle, a given implementation might choose to expose different subsets of the pipeline, Client Connection, Transaction and Statement contexts. Most implementations will however expose at least the Client to Parser steps or/and the Execution to Storage steps.

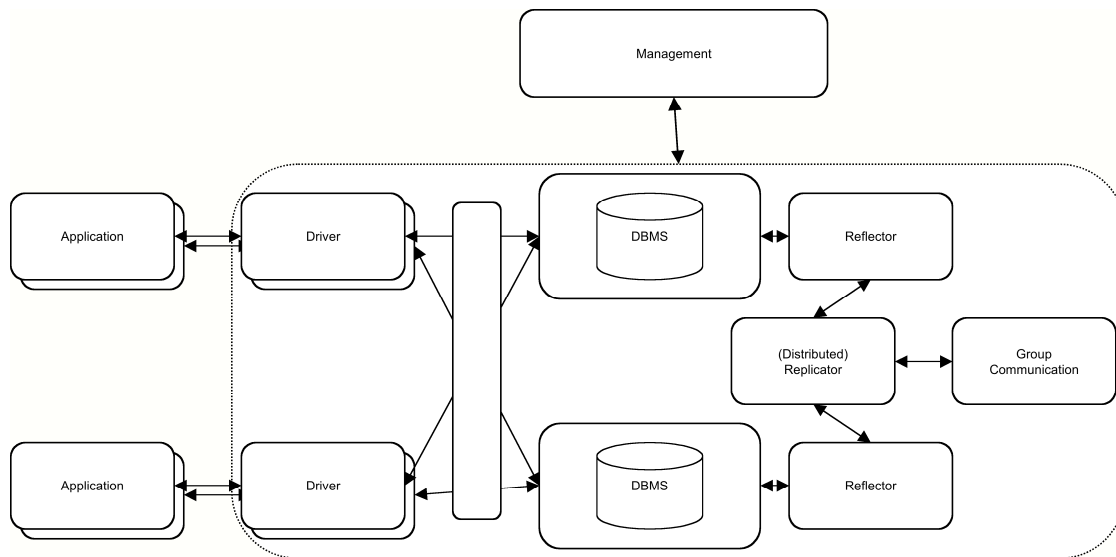


Figure 3 - Generic database replication architecture.

The generic replication architecture shown in Figure 3 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.

Specializations of generic architectures 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, and management components are portable and can therefore be combined with different DBMS.

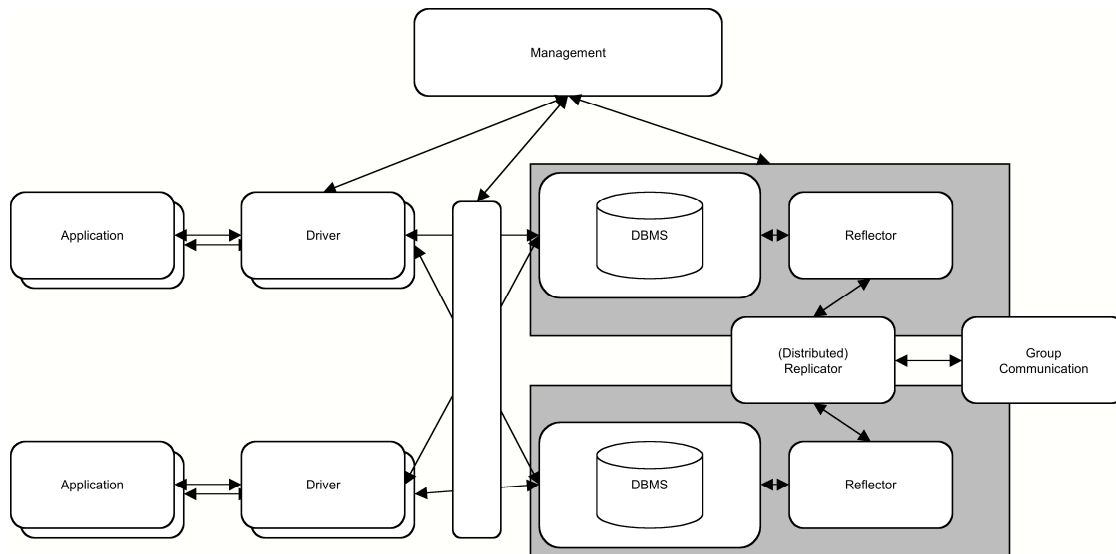


Figure 4 - Specialization for an in-core implementation.

As an example, consider in Figure 4 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.

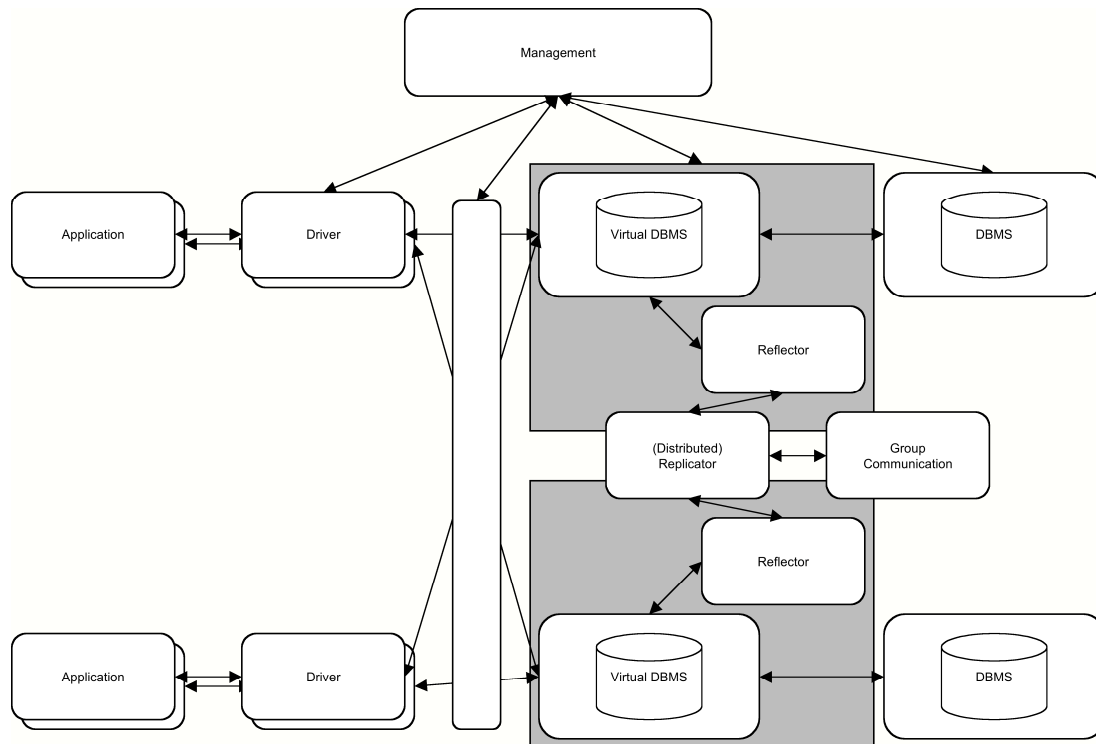


Figure 5 - Specialization for a middleware-based architecture

A DBMS independent implementation can be achieved strictly by intercepting the remote database access interface as suggested in Figure 5. 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 6.

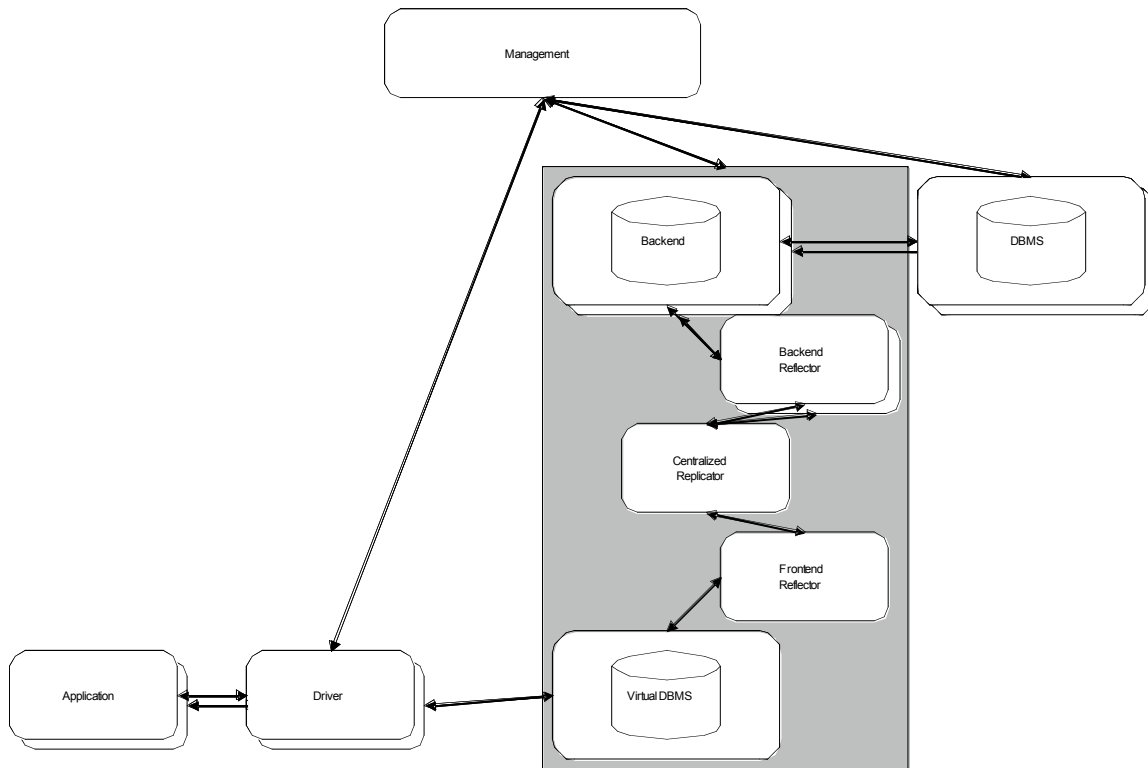


Figure 6 - Specialization for the single controller C-JDBC architecture.

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.

3.2 Generic management architecture

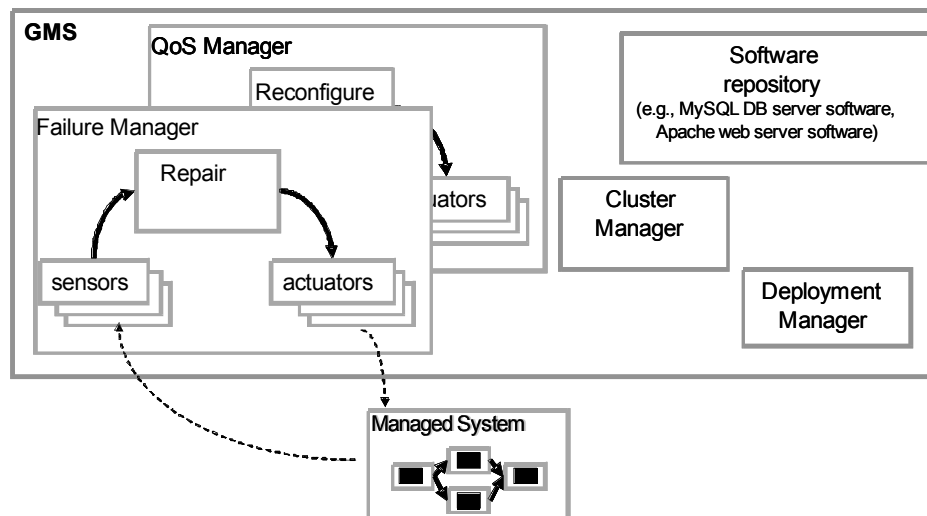


Figure 7 - Generic replicated database management architecture.

Figure 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.
- 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.

4 Programming Interfaces

GORDA Programming Interfaces are provided for each of the components of GORDA Architecture in order to allow reusable components. Although these interfaces are presented as a call-level interface, they can be implemented as a remote invocation and in a variety of languages by selecting an appropriate interoperability standard such as CORBA.

4.1 Client interfaces

Client interfaces allow applications to query and modify replication metadata through the standard call-level and SQL interfaces. These are not provided in the preliminary version of the API.

4.2 Database interfaces

Database interfaces allow replication components to inspect and modify transaction processing. The interfaces in Figure 9 to Figure 16 expose contexts and transaction processing stages as a number of events that can be captured and actions that can be applied to DBMS state. Figure 8 maps the interfaces shown in this section with the execution flow control of the transaction.

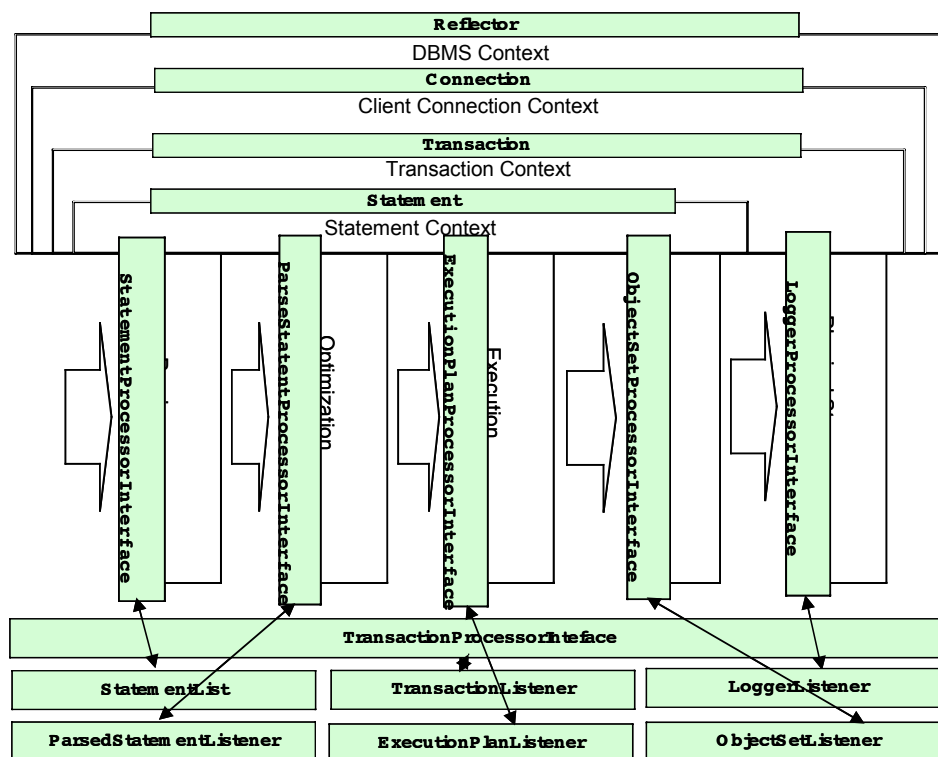


Figure 8 - Mapping of interfaces on the architecture.

In particular, the interfaces in Figure 13 can expose detailed information on write-sets, read-sets, result-sets and lock-sets that can be iterated in an implementation independent manner using an interface modeled on the standard client call-level interface for result sets. Which information is exactly available depends on the DBMS and on the configuration of the reflector components. This is determined at runtime and does not fit in the scope of the GORDA specification.

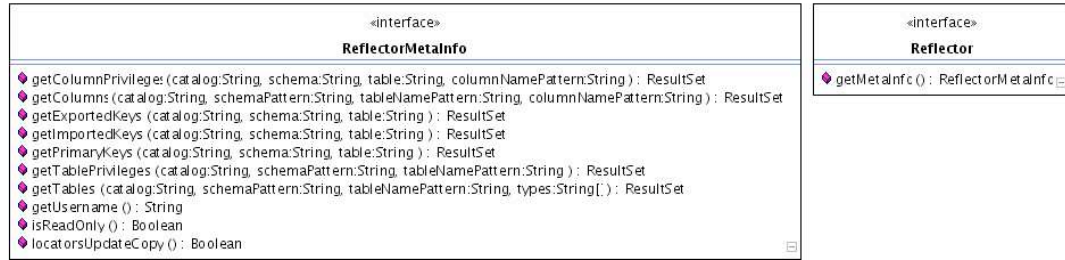


Figure 9 - Top-level reflector interface.

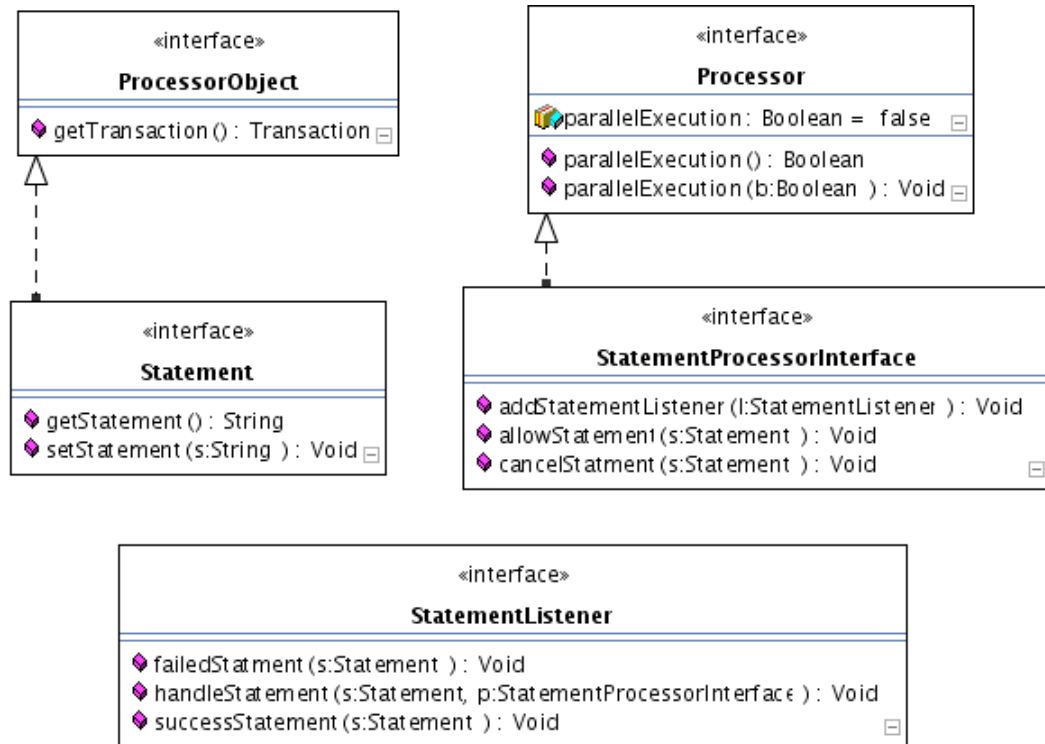


Figure 10 - Interface of statement stage.

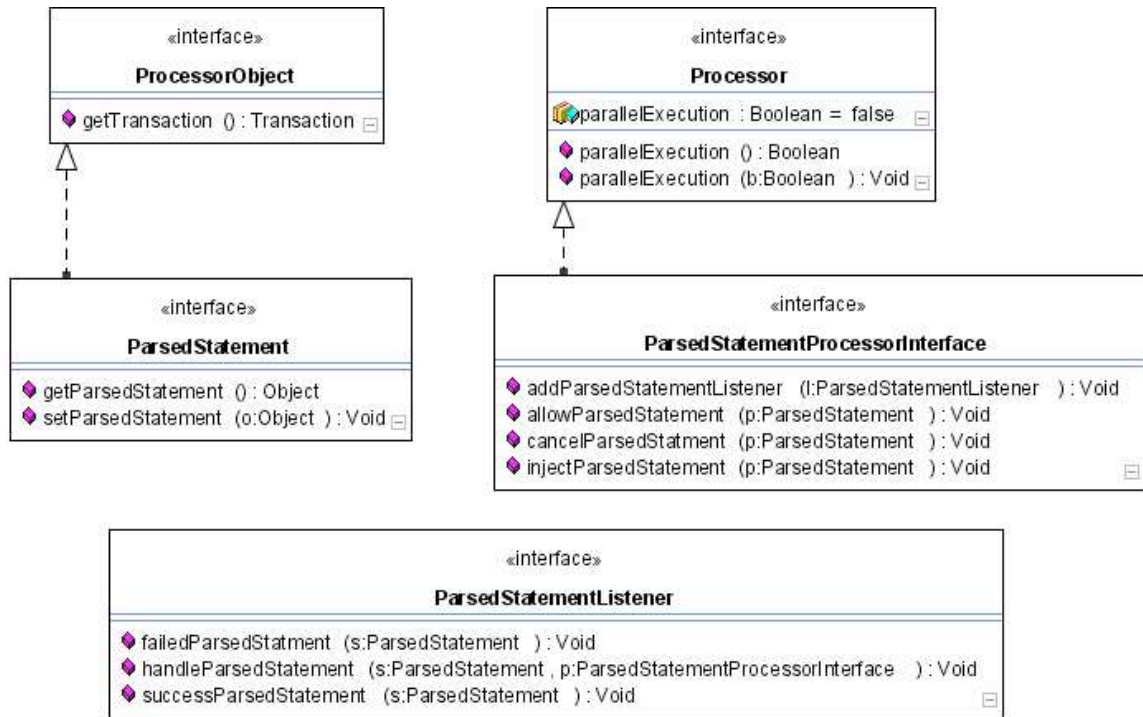


Figure 11 - Interface of parsing stage.

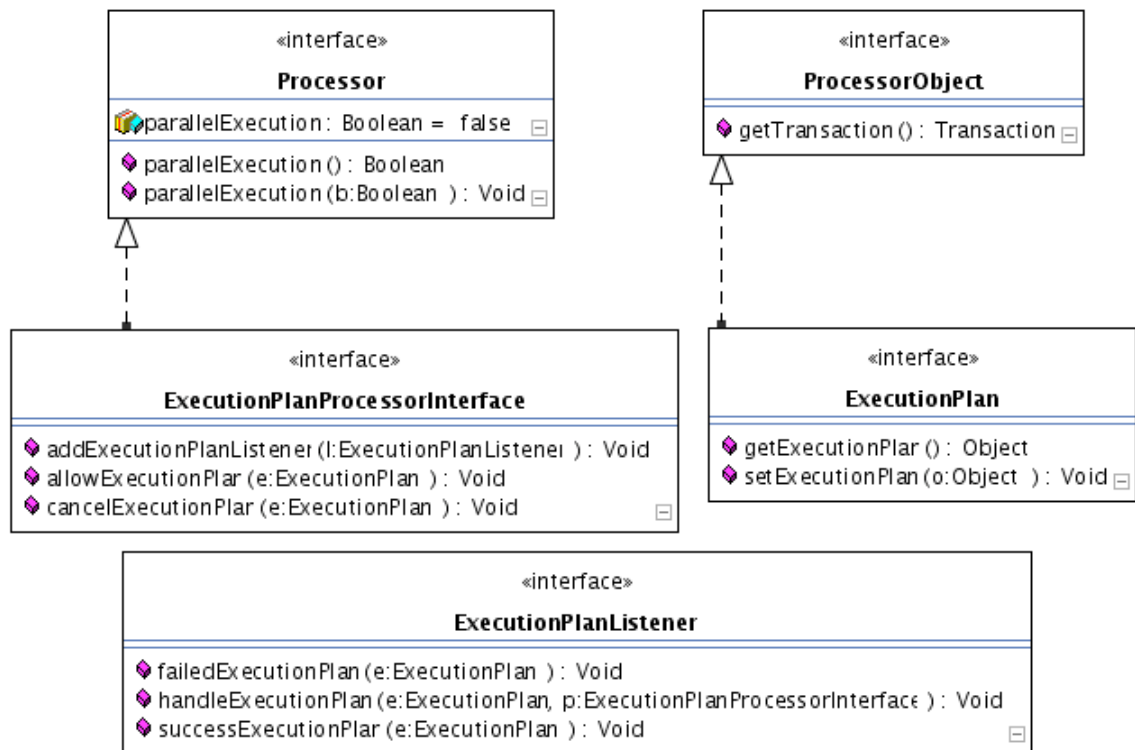


Figure 12 - Interface of execution stage.

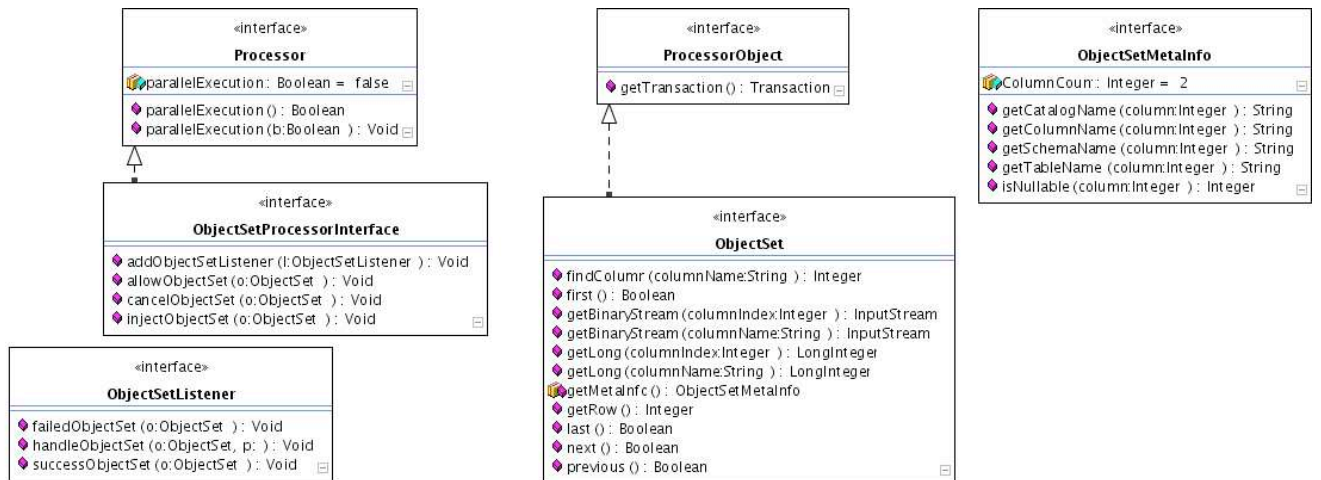


Figure 13 - Interface of committing stage.

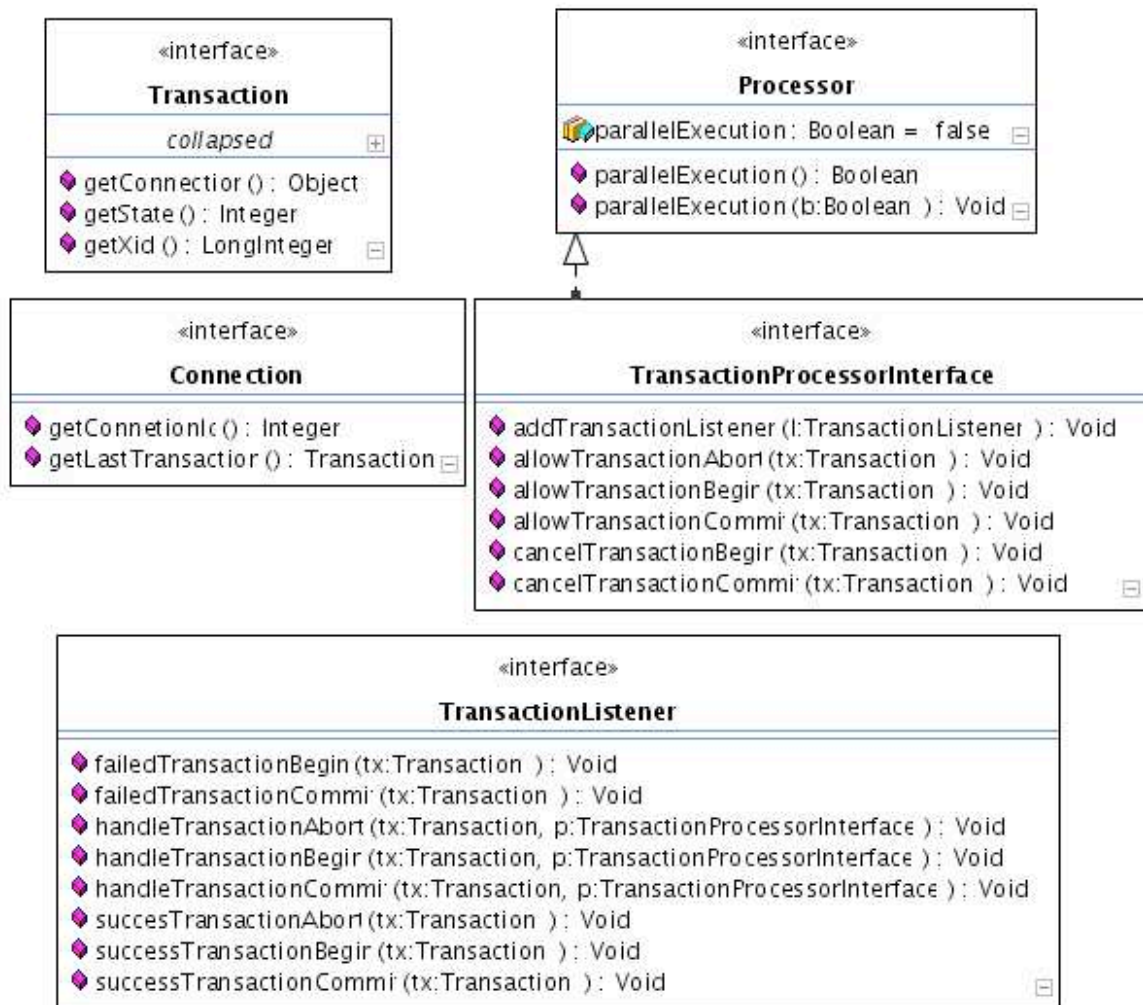


Figure 14 - Interface of transaction context.

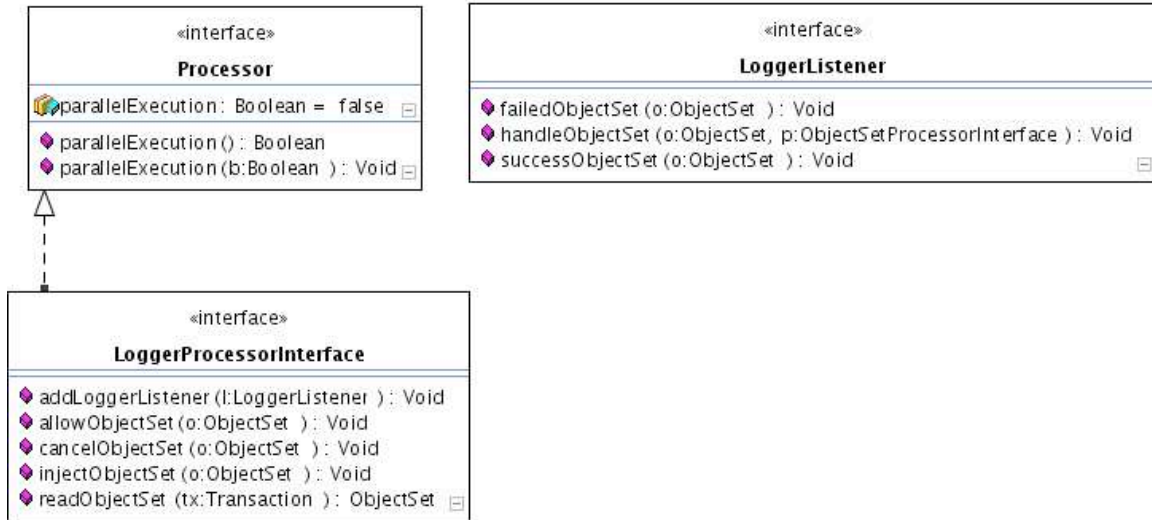


Figure 15 – Interfaces of the Logger.

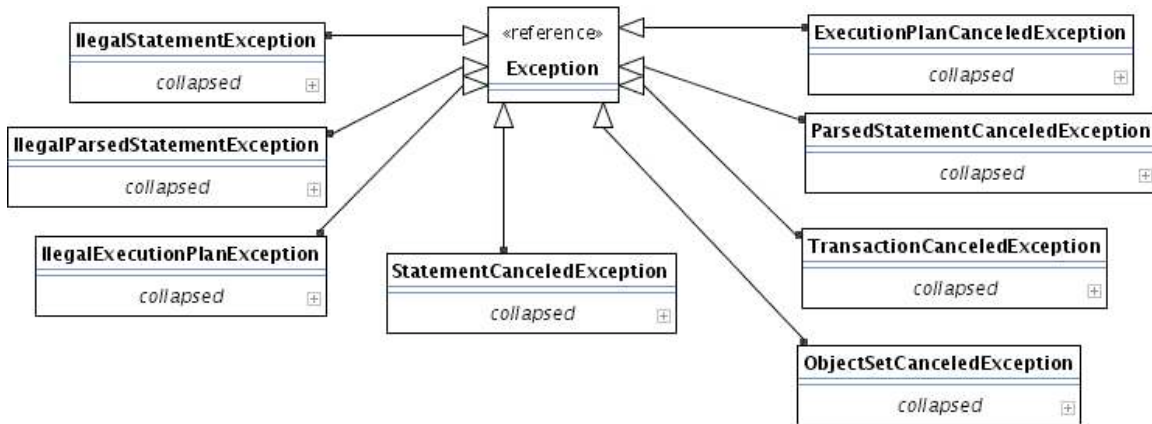


Figure 16 - Exception hierarchy.

4.3 Management interfaces

Management interfaces allow interoperability of management tools and other system components. These include:

- The Cluster Controller Management Interface(CCMI)
- The Database Management Interface (DBMI)
- The sensor interfaces (for QoS and failure sensors)

Figure 17 details the CCMI, DBMI and sensor interfaces and the multiplicity of the association relations with the GMS. For one cluster there is a “one to one” relationship with the Controller Interface whereas the relationship with the individual DB Interface is of the type “one to many”. In addition, the DBMI has “one to many” relationships with

the sensor interfaces, illustrating that there can be multiple sensors managed by one DBMI.

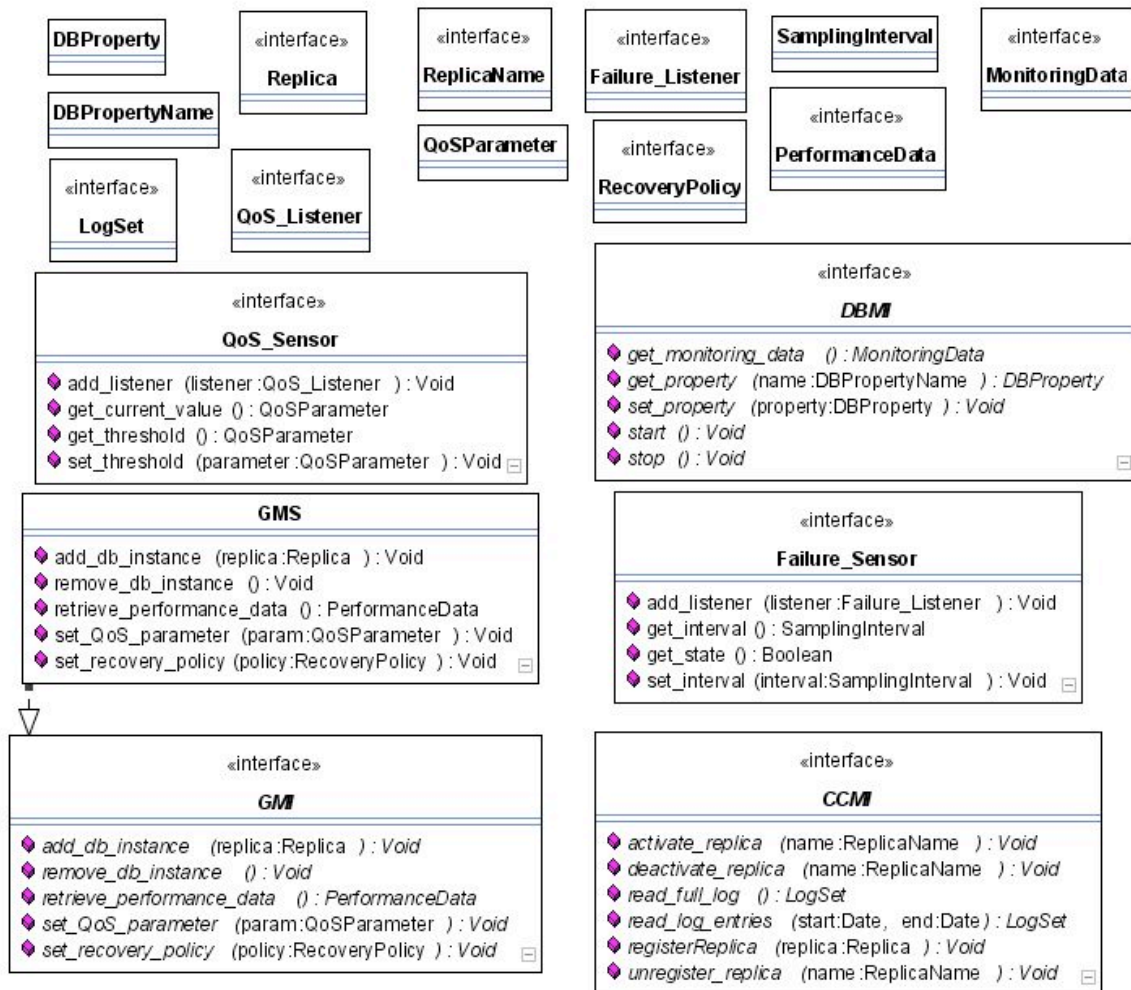


Figure 17 - Component interfaces.

The CCMI contains the operations that GMImpl will require in order to manage the cluster controller:

- `register_replica`: instructs the cluster manager to add the new replica as part of the cluster, effectively increasing the cluster size by 1 replica.
- `unregister_replica`: instructs the cluster manager to remove the new replica from the cluster, effectively decreasing the cluster size by 1 replica.
- `activate_replica`: instructs the cluster manager to activate an existing replica (i.e. to begin using this replica for serving requests).
- `deactivate_replica`: instructs the cluster manager to deactivate an existing replica (i.e. to stop using this replica for serving requests).

- `read_full_log`: reads the entire log of requests and events since the cluster start-up (this could be useful in situations where the cause of an error needs to be discovered and historical data can help).
- `read_log_entries`: reads a subset of the cluster log corresponding to requests and events between given dates (this is required upon reconciliation a newly added replica with the rest of the cluster replicas).

The DBMI contains operations that GMImpl will require in order to manage individual DB servers:

- `get_property`: returns the requested DB server configuration property
- `set_property`: sets a given property of the DB servers. For instance, it resets the file-system location of the DB server configuration (used when deploying the required packages onto a newly added replica) or adds a new database (including the structure and the content of the tables) to the DB server configuration.
- `start`: launches the DB server with the current configuration (used for instance after adding a new replica).
- `stop`: terminates the DB server (used for instance when removing a replica).

The QoS_Sensor interface exposes the operations that QoS sensors will provide, for monitoring the quality of service attributes of database instances:

- `set_threshold`: instructs the sensor to consider the new threshold as a trigger for QoS violations.
- `get_threshold`: returns the current QoS threshold.
- `get_current_value`: returns the current QoS value that the managed entity (database instance) operates under.
- `add_listener`: adds a notification listener for QoS events originating from this sensor.

The Failure_Sensor exposes the operations that failure sensors will provide for monitoring the state of database instances:

- `set_interval`: instructs the sensor to consider the new interval when checking the health of the monitored resource (DB instance).
- `get_interval`: returns the sampling interval currently in consideration by the sensor.
- `get_state`: returns true if the DB instance is alive or false otherwise.
- `add_listener`: adds a notification listener for failure events originating from this sensor.

4.4 Communication interfaces

Communication interfaces in Figure 18 **Error! Reference source not found.** allow replication and management components to interact in a distributed system. Interfaces are provided both for point-to-point and multicast communication. These can be provided by

stand-alone communication toolkits or embedded within a distributed DBMS or clustering toolkit.

Although configuration of QoS implementations is out of the scope of the specification, generic interfaces are provided such that pre-configured QoS can be selected when sending and receiving messages. Protocol specific information on messages (e.g. for content based optimization) can be provided.

Communication channels are previously configured and available on execution time with several types of QoS. A Channel is created through a JDNI interface and exposes methods to send messages and register message listeners.

The Message interface exposes the following methods:

- `setPayload` – sets the payload of the message;
- `getPayload` – upon reception, this method gets the payload of the message;
- `getSenderRank` – gets the rank that identifies the sender of the message inside a group membership;
- `getSenderAddress` – gets the address of the sender of the message.

Depending on the QoS needed, a system component can create different types of Channels. The Basic channel is used for point-to-point communication and the Group channel is used for group communication.

The Callback interface is used by system components to register methods to receive messages, view changes and notifications about insurance of a certain service in a message (e.g. a message is non-uniformly delivered and notified when the message is already uniform). This is provided by the ServiceCallback interface and the Service interface and a Context object. The ServiceCallback interface exposes the following method:

- `serviceEnsured` – notifies the system component that a previously delivered message, identified by a context, has one certain service ensured.

The Context object is provided by the application upon the delivery of the message. Implementations of the Service interface must also implement the Comparable interface.

A Message can contain an Annotation which is semantic information that can be used by content based optimization protocols.

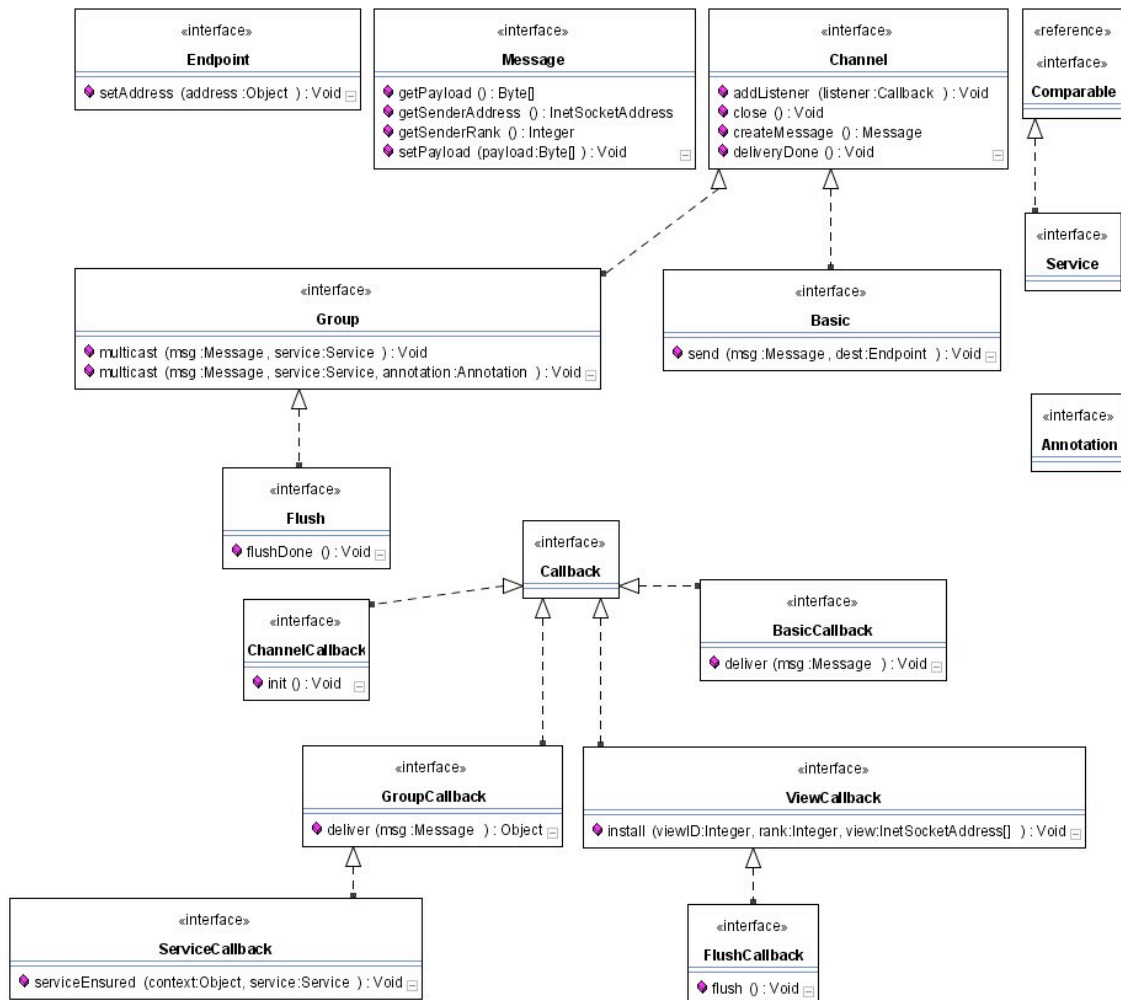


Figure 18 - Communication interfaces.

4.5 Threading

Accommodating efficient transaction processing requires allowing concurrent operations. This is done in first place by ensuring that all implementations of GORDA Programming Interfaces are thread safe and can thus be invoked concurrently.

For each event, the detailed specification of the interfaces mentions also if multiple notifications can be issued concurrently. This applies to all interfaces: DBMS, management, and communication.

As an example, notifications of transaction commit are serialized to allow replication code to determine serialization order. Notification of transactions starting can be issued concurrently and is up to the handler to serialize execution of possible critical sections.

A second example is that a group communication protocol can concurrently issue notifications for unordered message delivery but must serialize notification of totally ordered messages.

Concrete implementations may provide additional serialization guarantees, always or as configuration options. For instance, it may be relevant to serialize transaction start notification with transaction commit to determine causality.

Additionally the DBMS reflector interfaces allow each stage to be configured to generate events asynchronously without implicitly blocking the DBMS thread by means of the Processor interface.

5 Use cases

In all the scenarios, the Reflector Interface enables a generic interface that allows the configuration management and the replication tools to access metadata regarding the replicated databases. A rendering of the ObjectSet Interface is used to represent the information retrieved which is stored by using an independent representation enabling to propagate the updates among different database vendors and architectures.

In general, the ObjectLogProcessor is directly related to asynchronous replication. The StatementProcessor, ParsedStatementProcessor and ExecutionPlanProcessor allow state machine replication as the operations that update the database are intercepted. In contrast, the ObjectSetProcessor propagates the changes itself.

The TransactionProcessor is used to notify events related to a transaction such as its startup, commit or rollback. The concern with such events is related to the synchronous replication protocols presented in this document. In the asynchronous replication protocols, the transaction context is implicitly defined by using the attribute Transaction.

In what follows, we present use cases for asynchronous and synchronous protocols.

5.1 Asynchronous replication

The asynchronous replication eliminates the additional overhead that would be imposed by the propagation of changes within a transaction's execution. It decouples the update of the replicas from the transaction that originated the changes. Basically, different transactions handle the refreshment of the replicas.

Based on this assumption, different replication scenarios may be proposed.

We suggest using the ObjectLogProcessor Interface which may be rendering actively or passively. In the first case, an external component access the DBSM through the interface provided to retrieve the last committed transactions in the log since the last access. The interval between successive retrievals depends on the requirements imposed by the application. In the second case, the data is retrieved from the log as soon as an entry is produced by the DBSM. The component that implements ObjectSetListener Interface is notified during the creation of the entries in the log in order to keep up the replicas synchronized. Unfortunately, uncommitted information may be read and thus, transactions will abort in the replicas whenever they abort in the origin. It is important to notice that this approach does not imply that the propagation must be handled in the same thread that creates the entries into the log or that the updates must be copied to an intermediate storage before being processed. Basically, the updates' rate and the number of threads used to propagate the changes will determine if it is necessary to use an intermediate storage and if so, its size.

Probably, when combined with a synchronous replication protocol, the asynchronous replication may use the ObjectSetProcessor. The idea is similar to the ObjectLog Processor used passively and further information will be provided in the next section.

Other pipeline's stages, such as the `ExecutionPlanProcessor`, `StatementProcessor` or `ParsedStatementProcessor`, could be used to provide information to the asynchronous replication protocols. However, most likely, the `ObjectLogProcessor` Interface is the best solution in terms of performance, as it avoids any additional processing inside a transaction's execution and reduces duplicated information. In contrast to the `ExecutionPlanProcessor`, `StatementProcessor` or `ParsedStatementProcessor`, it does not have the problems related to the state machine approach.

Once the changes are propagated to the replica, a component that renders the `ObjectLogProcessor` Interface applies the changes. This component may apply different transactions together as a single transaction and may apply different transactions in parallel in order to improve performance.

5.2 Synchronous replication

Synchronous replication attempts to ensure that when there are transaction commits the replicas are already updated. The changes are propagated during the transaction's execution which means that an additional overhead is imposed to transactions.

During a transaction's execution whenever a transaction begins, commits or aborts a notification is sent by the `TransactionProcessor` Interface. The processor is allowed to proceed when the listener calls the `successTransactionBegin`, `successTransactionCommit` or `successTransactionAbort` according to the event. If for some reason the listener wishes to cancel the execution, it calls the `failedTransactionBegin`, `failedTransactionCommit` or `failedTransactionAbort`. This feature is enabled when the attribute `parallelExecution` is false, otherwise it is ignored.

Furthermore, it is possible to notify the listener that for some reason a processor failed or otherwise, it succeeded. This information is quite useful to the garbage collection and to ensure that unprocessed events such as those related to the statements, parsed statements and execution plans were correctly executed thus avoiding inconsistencies among the replicas.

Every processor but the active `ObjectLogProcessor` can be used to retrieve the information to be propagated. Let us consider the case that the `ObjectSetProcessor` is rendered. In this case, a component that implements the `ObjectSetListener` Interface is notified whenever a tuple is read or written.

In order to apply the remote updates, a replica provides a `TransactionProcessor` Interface to execute the events related to the transaction and a `ObjectSetProcessor` Interface to effectively update the changes.

5.3 Managing Replicas

An autonomic manager can be built that uses the GORDA Management Interfaces to dynamically orchestrate the allocation of replicas in the system at runtime, based on varying performance and availability conditions.

The autonomic manager can register itself as a listener for events originating at instances of `QoS_Sensor` and `Failure_Sensor` interfaces by calling their respective `add_listener` methods. In addition to receiving events when a quality of service agreement has been

violated or when a replica has failed, the autonomic manager can also poll the sensors to obtain this information at arbitrary moments. For this, it can call the `get_current_value` and `get_state` methods of the QoS and failure sensors respectively.

If the desired QoS is not maintained, the manager can decide to add a replica to the cluster and calls the `register_replica` method on the CCMI interface. In addition, it calls the `activate_replica` operation on CCMI, in order to enable requests to be served by this replica. Note that when adding a DB replica, it is essential to synchronize the data of the new replica with the data in the already running replicas. Therefore, the manager uses operations provided by the Replicator to bring the newly activated replica up to date. As an example, when using CJDBC, the differential log file of all the SQL write statements can be replayed in order to rebuild the state of the new replica to match the other replicas. This assumes that the new replica contains an initial DB with most read-only data already in place.

Similarly, when the QoS values are exceeded and the amount of resources currently in use can be reduced, the manager can deactivate replicas and free them for use in other applications. For this it calls the `deactivate_replica` operation on the CCMI.

In both cases, the DBMI is used to implement the required operations on individual DB servers (by calling its start and stop methods among others).

Similarly to the above scenario, the manager may decide to add replicas to the system when an existing replica fails, after receiving the appropriate event from the failure sensor or after polling the replica's status (`get_state` method on the `Failure_Sensor` interface).

5.4 Group communication

A replication component uses the communication interfaces to join the group of distributed replicas. Using these interfaces, the component must create a channel of type `Group` and register it self in that channel using a callback of type `GroupCallback` to receive messages of other members of the distributed replication component. To be notified of new views (replicas that joined or leaved the group) it must register in the channel also with the `ViewCallBack` interface.

When a replica finished these first steps, it can start broadcasting messages to all other replicas and messages from other members are delivered in the `GroupCallback`. When a replica fails, all other replicas will be notified, receiving a new view.

If one replica wants to use optimistic assumptions on totally ordered message delivery, it must also register in the channel using the `ServiceCallback` interface. In this case, upon reception of a message, the replica must return a context object to the communication service and will be notified later when the order of the messages is stabilised.