Project no. 004758

GORDA

Open Replication Of Databases

Specific Targeted Research Project

Software and Services

GORDA Interfaces Definition

GORDA Deliverable 2.3

Due date of deliverable: 2006/03/31
Actual submission date: 2006/04/24

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

Fundação da Faculdade de Ciências da Universidade de Lisboa

Revision 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

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

The GPI 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.
- Section **Error! Reference source not found.** describes GORDA Programming Interfaces exhibited by each component.
- Section 4 illustrates the usefulness of the GPI 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.

A rendering of these interfaces in the Java programming language and detailed documentation is provided in the companion document “GORDA PI in Java – D2.3 Annex” for reference.

2 Assumptions and Approach

2.1 Database architecture

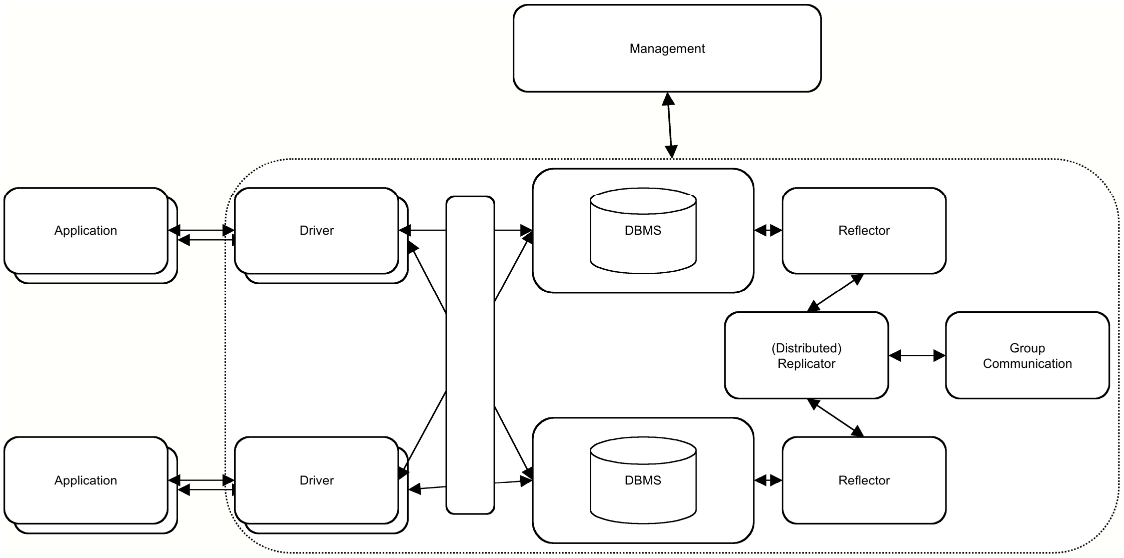


Figure 1 - Generic database replication architecture

The GORDA replication architecture is shown in the Figure 1 and builds on the following main components:

- 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.
- The reflector provides the means for intercepting, observing and modifying transaction processing by exposing the path through which a transaction passes.
- The replicator captures events associated to a transaction processing (e.g, begin, commit, updates, statements being processed) in a database by using the reflector, propagates them among other databases and also by means of the reflector applies the events remotely captured. In order to receive such events, the replicator registers callbacks in the reflector.

- The group communication is the transport mechanism used to send events among databases.

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 Programming Interface (GPI) is based on existing data management standards. Namely:

- ISO/IEC 9075-3:1995 - Call Level Interface (SQL/CLI) and X/Open XA Distributed Transaction Processing (DTP) specify client interfaces.

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 Programming Interfaces stands on the following general principles:

- Independence of operation and configuration (Inversion of Control pattern). 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 GPI 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 is achieved by adding an optional layer on basic interfaces.

3 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.

3.1 Client Interfaces

Client interfaces allow applications to query and modify replication metadata through the standard call-level and SQL interfaces. However, it is out of the scope of this document to define metadata regarding replication.

3.2 Database Interfaces

Database interfaces provided by GORDA allows replication components to inspect and modify transaction processing. Figure 2 maps such interfaces to the execution flow of a transaction and Figure 3 to Figure 11 present in detail each interface.

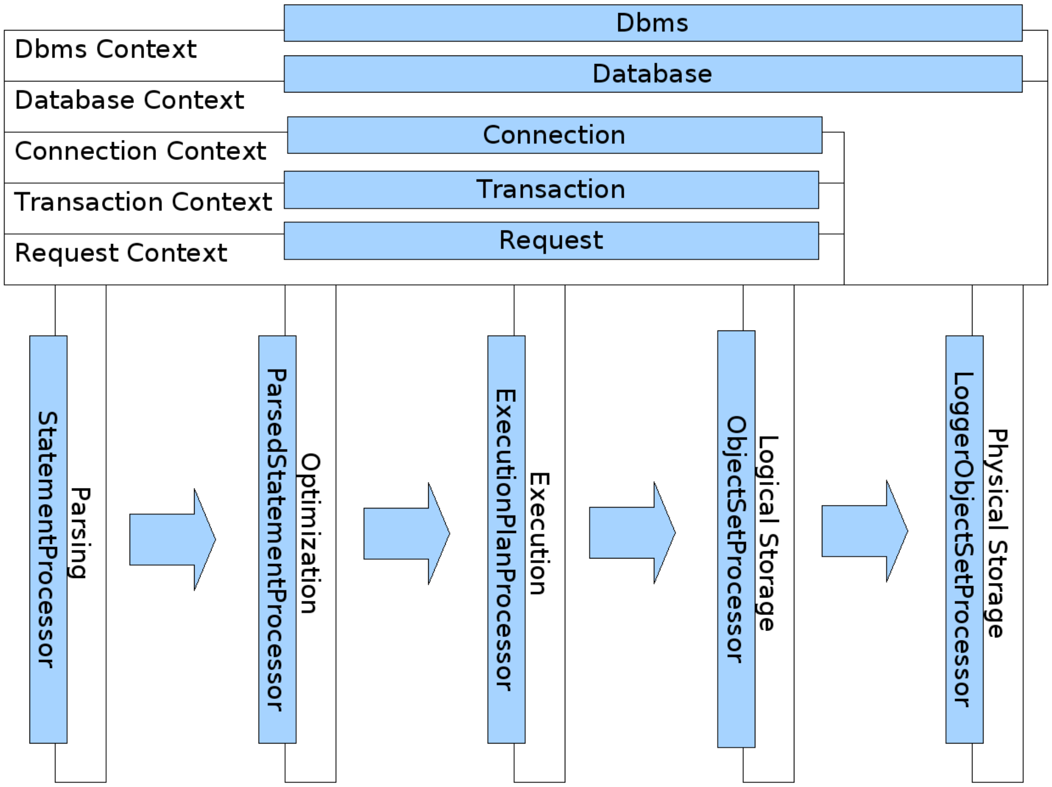


Figure 2 – Transaction processing model

The context layers provide the means to uniquely identify the events exposed by the interfaces. There are five distinct contexts that together form a hierarchy:

- Dbms Context: identifies the database management system.
- Database Context: identifies a database with its users, tables, triggers, etc. The Dbms has at least one Database.
- Connection Context: identifies a user accessing a database and multiple users may access a Database simultaneously.
- Transaction: Groups a set of requests sent by a user into a logical unit that can be committed or aborted. It should be created together with the first Request sent by a user.
- Request: identifies a request sent by a user in the context of a Transaction.

Requests sent by a user are processed by the abstract pipeline presented in Figure 2. The *StatementProcessor* allows replication components to intercept a request before it is parsed. The other stages work as follows. The *ParsedStatementProcessor* intercepts a parsed statement before it is optimized. The *ExecutionPlanProcessor* intercepts a optimized execution plan before it is executed. The *ObjectSetProcessor* intercepts read and written information. The *LoggerObjectSetProcessor* intercepts only written information right before it is ready to be flushed to disk.

Interfaces in each context and pipeline have three basic elements: processors, events and listeners. Processors are used to register callbacks and provides methods to continue or cancel executions associated with events. Listeners defines the callbacks methods. While registering listeners, it is possible to define if a database should proceed in parallel without waiting that a listener handles an event (i.e., wait equals true) or not (i.e., wait equals false). This configuration information is available through this interface, instead of being delegated to a directory service, as it has direct impact on the ability to handle events.

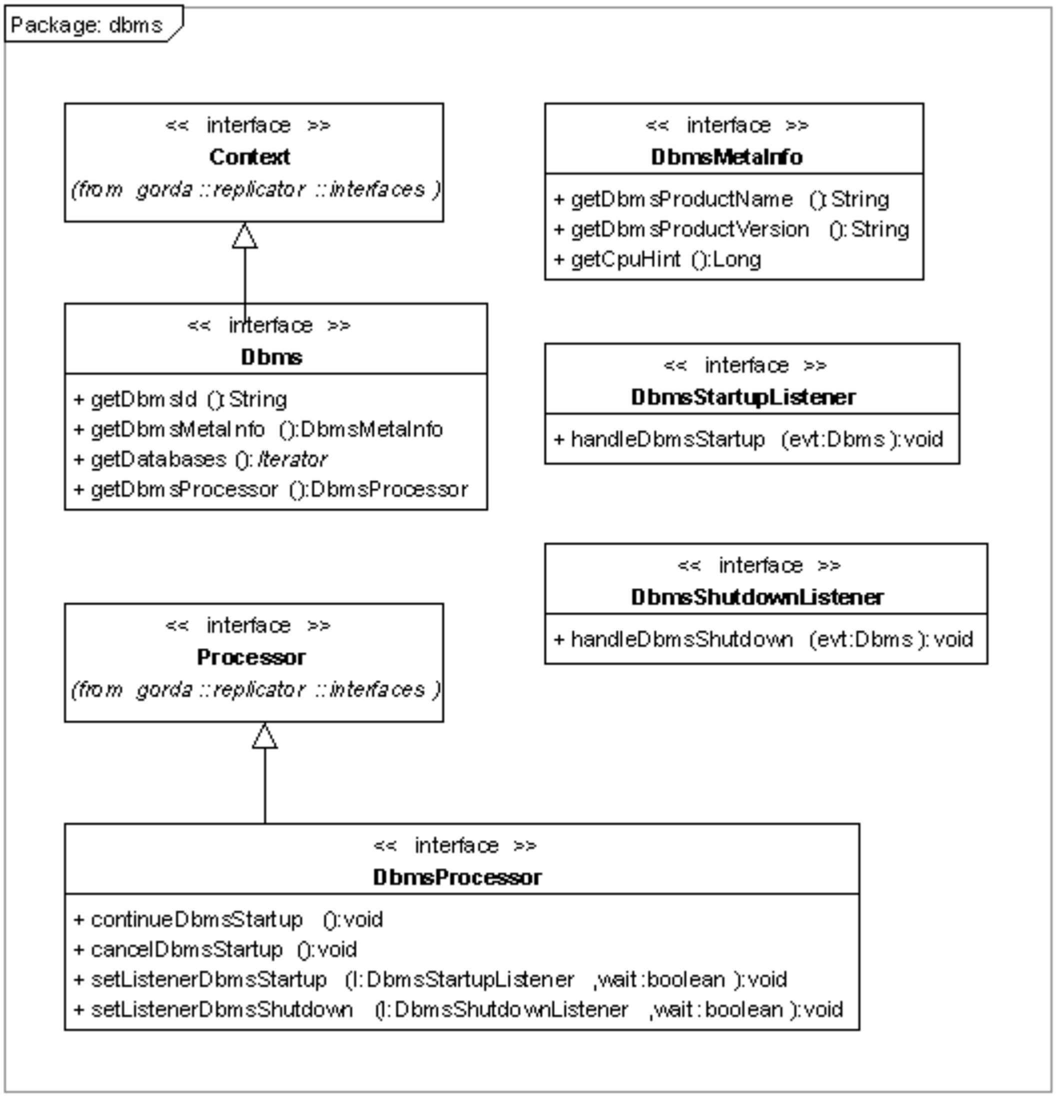

Package: dbms
<< interface >>
Context
(from gorda :: replicator :: interfaces)
<< interface >>
DbmsMetaInfo
+ getDbmsProductName ():String
+ getDbmsProductVersion ():String
+ getCpuHint ():Long
<< interface >>
Dbms
+ getDbmsId ():String
+ getDbmsMetaInfo ():DbmsMetaInfo
+ getDatabases ():Iterator
+ getDbmsProcessor ():DbmsProcessor
<< interface >>
DbmsStartupListener
+ handleDbmsStartup (evt:Dbms):void
<< interface >>
DbmsShutdownListener
+ handleDbmsShutdown (evt:Dbms):void
<< interface >>
Processor
(from gorda :: replicator :: interfaces)
<< interface >>
DbmsProcessor
+ continueDbmsStartup ():void
+ cancelDbmsStartup ():void
+ setListenerDbmsStartup (l:DbmsStartupListener ,wait:boolean):void
+ setListenerDbmsShutdown (l:DbmsShutdownListener ,wait:boolean):void

Figure 3 – Dbms Context

The Dbms context is built on the set of interfaces depicted in Figure 3. The *DbmsProcessor* enables the replication component to register callbacks for the startup and shutdown events. While handling the startup event, the replication component is enabled to continue or cancel it. Dbms interface provides access to an id, meta information, processor interface and to all databases available.

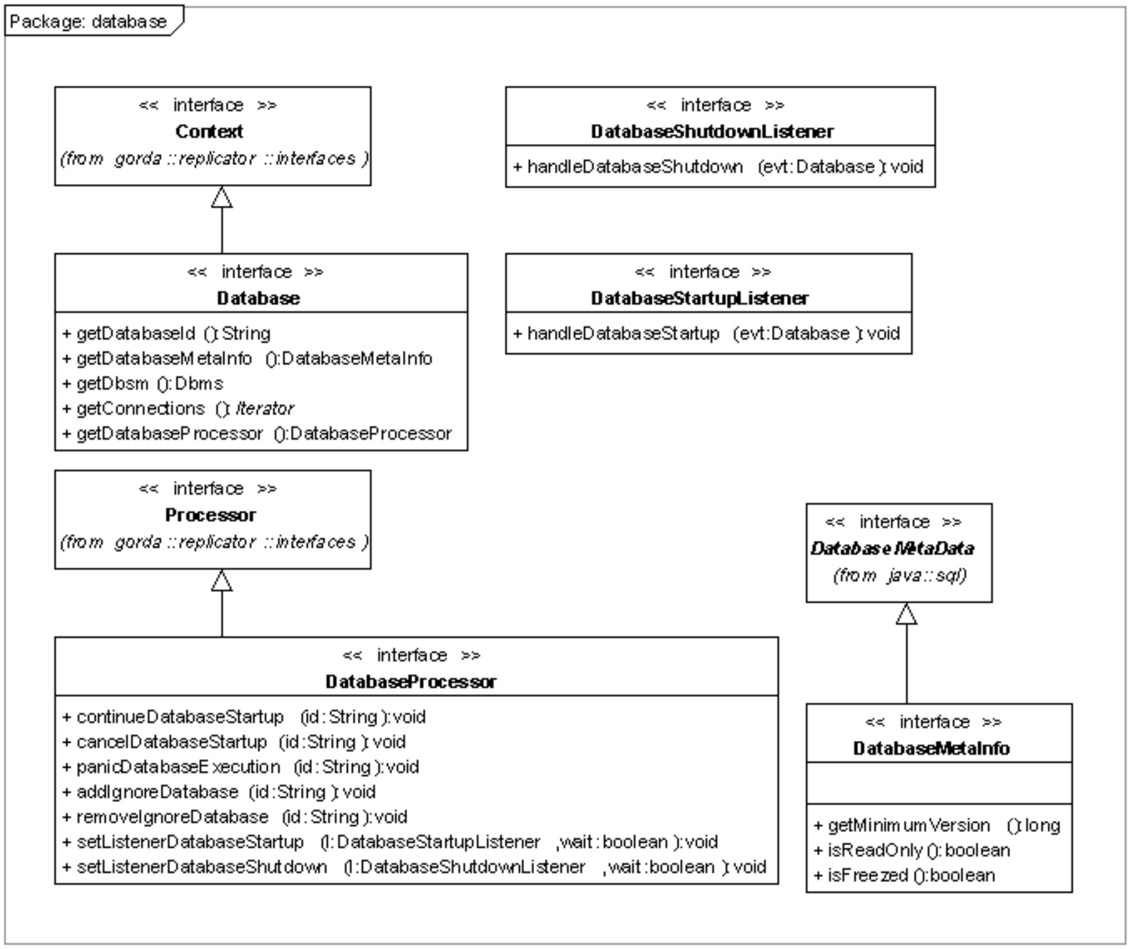

Package: database
<< interface >>
Context
(from gorda ::replicator ::interfaces)
<< interface >>
DatabaseShutdownListener
+ handleDatabaseShutdown (evt: Database): void
<< interface >>
Database
+ getDatabaseId (): String
+ getDatabaseMetaInfo ():DatabaseMetaInfo
+ getDbsm (): Dbms
+ getConnections (): Iterator
+ getDatabaseProcessor ():DatabaseProcessor
<< interface >>
DatabaseStartupListener
+ handleDatabaseStartup (evt:Database): void
<< interface >>
Processor
(from gorda ::replicator ::interfaces)
<< interface >>
DatabaseMetaData
(from java::sql)
<< interface >>
DatabaseProcessor
+ continueDatabaseStartup (id: String):void
+ cancelDatabaseStartup (id:String): void
+ panicDatabaseExecution (id: String):void
+ addIgnoreDatabase (id:String): void
+ removeIgnoreDatabase (id: String):void
+ setListenerDatabaseStartup (l: DatabaseStartupListener ,wait : boolean):void
+ setListenerDatabaseShutdown (l:DatabaseShutdownListener ,wait :boolean): void
<< interface >>
DatabaseMetaInfo
+ getMinimumVersion (): long
+ isReadOnly (): boolean
+ isFreezed ():boolean

Figure 4 – Database Context

The Database context is depicted in Figure 4. It is quite similar to the Dbms, but there are important differences that should be described. The Database context defines meta information based on the *DatabaseMetadata* from the java.sql package. This meta information should be used to configure the replication component and should provide whatever information it is necessary. However, the information available is implementation dependent. This context also enables access to all connections available. It has the ability through the *DatabaseProcessor* to put a database in a panic state. The panic method should be used if a database for some reason becomes inconsistent. For instance, if a remote update fails the panic method should be called in order to avoid that unprivileged users have access to the database defining that it should only be accessed by an administrator in order to fix problems. It allows the replication component by means of the *addIgnoreDatabase* to define if its events should be notified or not.

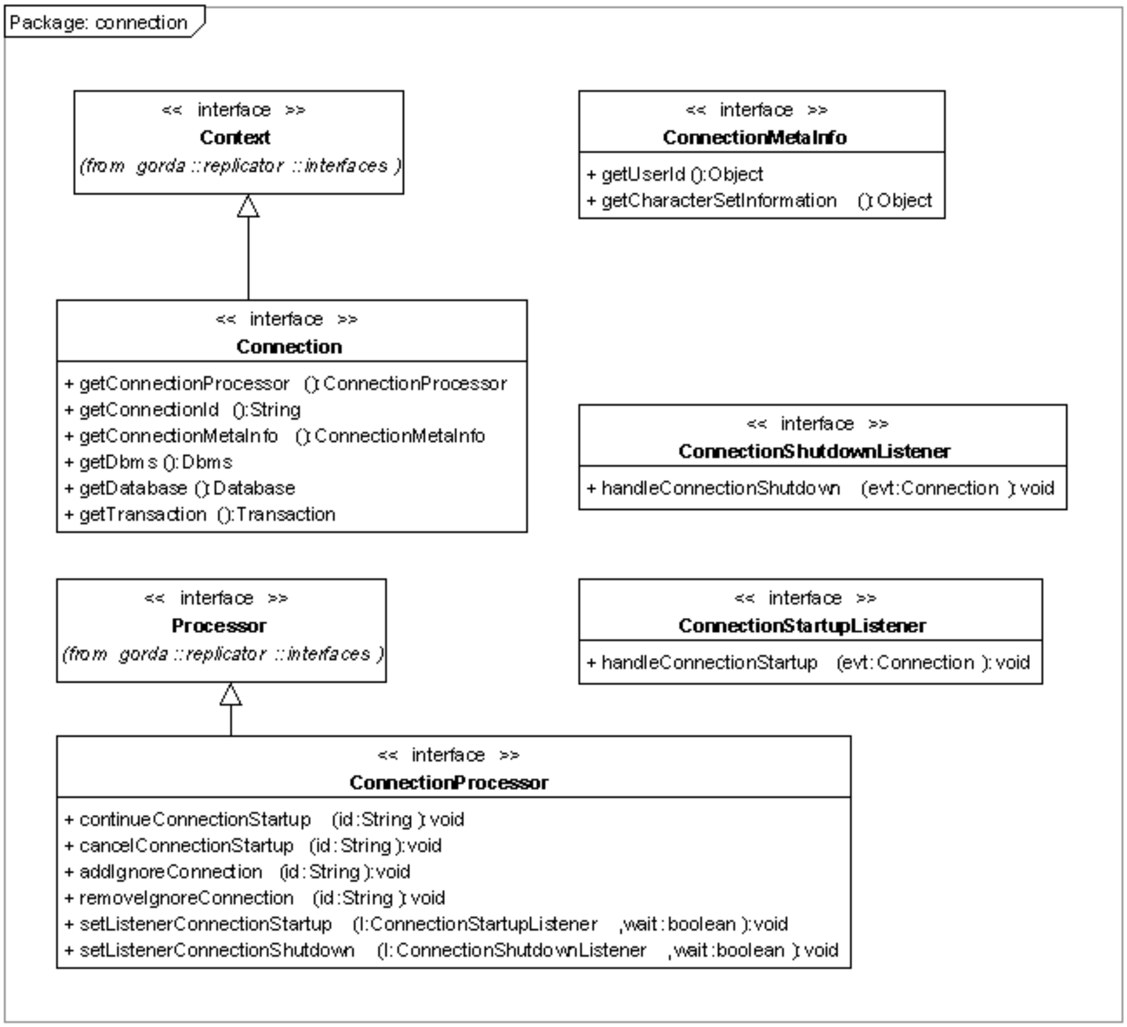

Package: connection
<< interface >>
Context
(from gorda ::replicator ::interfaces)
<< interface >>
ConnectionMetaInfo
+ getUserId ():Object
+ getCharacterSetInformation ():Object
<< interface >>
Connection
+ getConnectionProcessor ():ConnectionProcessor
+ getConnectionId ():String
+ getConnectionMetaInfo ():ConnectionMetaInfo
+ getDbms ():Dbms
+ getDatabase ():Database
+ getTransaction ():Transaction
<< interface >>
ConnectionShutdownListener
+ handleConnectionShutdown (evt:Connection):void
<< interface >>
Processor
(from gorda ::replicator ::interfaces)
<< interface >>
ConnectionStartupListener
+ handleConnectionStartup (evt:Connection):void
<< interface >>
ConnectionProcessor
+ continueConnectionStartup (id:String):void
+ cancelConnectionStartup (id:String):void
+ addIgnoreConnection (id:String):void
+ removeIgnoreConnection (id:String):void
+ setListenerConnectionStartup (l:ConnectionStartupListener ,wait:boolean):void
+ setListenerConnectionShutdown (l:ConnectionShutdownListener ,wait:boolean):void

Figure 5 – Connection Context

The *Connection* context is depicted in Figure 5. It is used to catch startup and shutdown events associated to a connection. Regarding this context, it is worth mentioning its ability to cancel connections through the *ConnectionProcessor* interface. For instance, it could be possible to inspect and cancel connection events from specific users (e.g., non-administrator users) while other users are allowed to proceed (e.g., administrator users). It also provides the ability to identify which connections should not generate events by using the *addIgnoreConnection* method. This is particular interesting when one needs to execute queries and updates from the replication component and does not want that such events are capture by the interface.

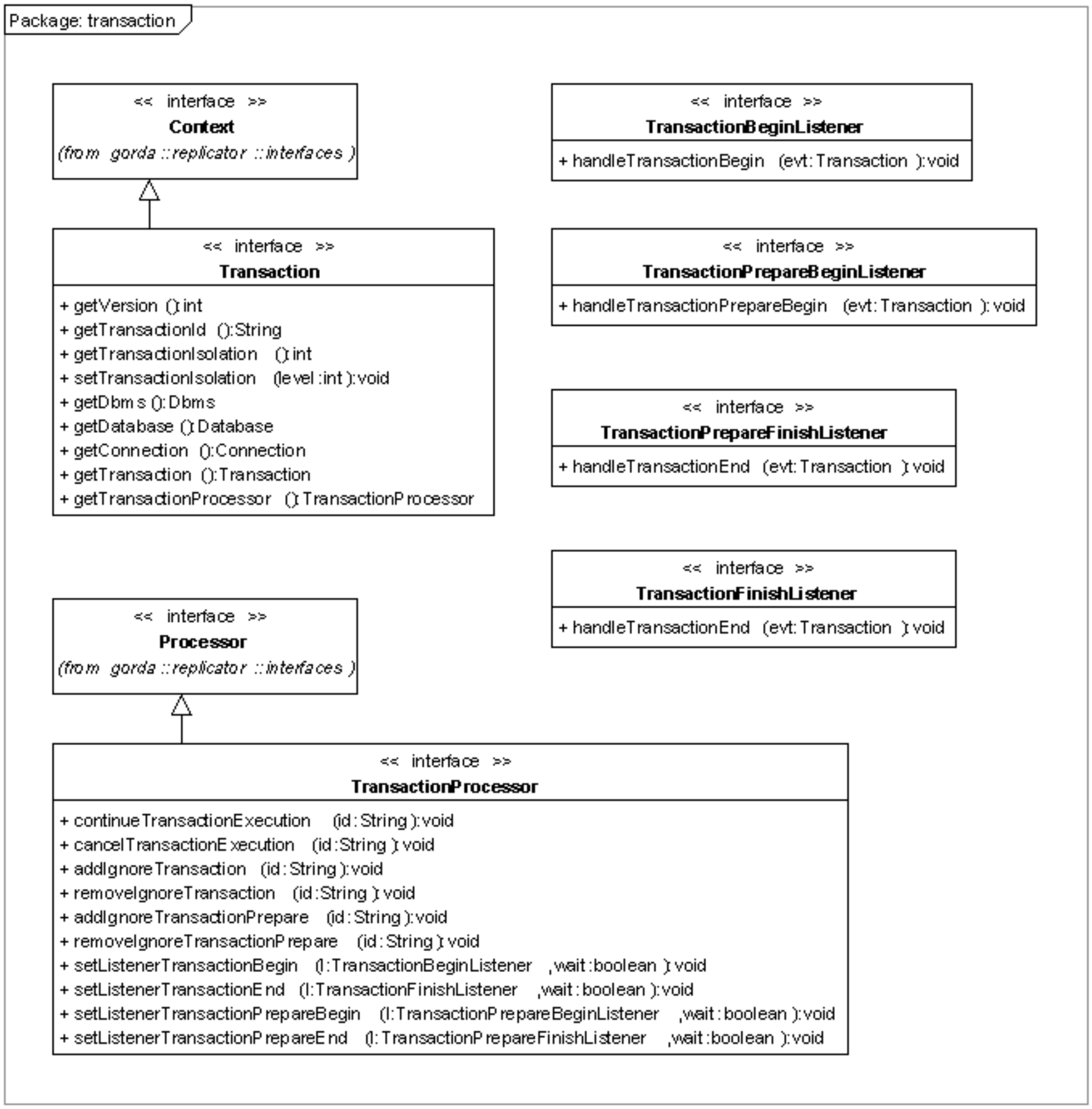

Package: transaction
<< interface >>
Context
(from gorda :: replicator :: interfaces)
<< interface >>
TransactionBeginListener
+ handleTransactionBegin (evt: Transaction):void
<< interface >>
Transaction
+ getVersion ():int
+ getTransactionId ():String
+ getTransactionIsolation ():int
+ setTransactionIsolation (level :int):void
+ getDbms ():Dbms
+ getDatabase ():Database
+ getConnection ():Connection
+ getTransaction ():Transaction
+ getTransactionProcessor ():TransactionProcessor
<< interface >>
TransactionPrepareBeginListener
+ handleTransactionPrepareBegin (evt: Transaction): void
<< interface >>
TransactionPrepareFinishListener
+ handleTransactionEnd (evt: Transaction):void
<< interface >>
TransactionFinishListener
+ handleTransactionEnd (evt: Transaction):void
<< interface >>
Processor
(from gorda :: replicator :: interfaces)
<< interface >>
TransactionProcessor
+ continueTransactionExecution (id : String):void
+ cancelTransactionExecution (id : String):void
+ addIgnoreTransaction (id : String):void
+ removeIgnoreTransaction (id :String):void
+ addIgnoreTransactionPrepare (id : String):void
+ removeIgnoreTransactionPrepare (id : String):void
+ setListenerTransactionBegin (l:TransactionBeginListener ,wait :boolean):void
+ setListenerTransactionEnd (l:TransactionFinishListener ,wait : boolean):void
+ setListenerTransactionPrepareBegin (l:TransactionPrepareBeginListener ,wait : boolean):void
+ setListenerTransactionPrepareEnd (l:TransactionPrepareFinishListener ,wait :boolean):void

Figure 6 – Transaction Context

The *Transaction* context groups requests from a single-threaded user application. It is depicted in Figure 6. Similar to previous contexts, it provides the ability to disregard events, in this case, events associated to a transaction. It allows to define callbacks on a begin transaction, commit and rollback. If a coordinated transaction is executed (e.g., 2PC or 3PC), it is also possible to defines call backs on a prepare transaction, commit prepared or rollback prepared. To handle save points, it provides the *getTransaction* method available which returns the outer-transaction if there is one.

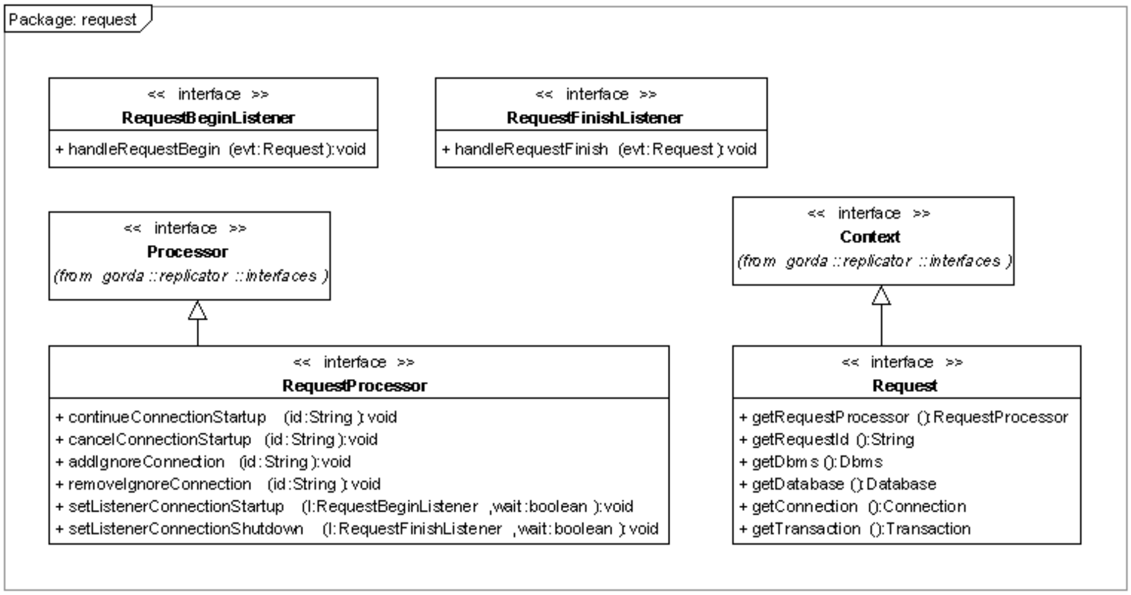

Package: request
<< interface >>
RequestBeginListener
+ handleRequestBegin (evt:Request):void
<< interface >>
RequestFinishListener
+ handleRequestFinish (evt:Request):void
<< interface >>
Processor
(from gorda::replicator::interfaces)
<< interface >>
Context
(from gorda::replicator::interfaces)
<< interface >>
RequestProcessor
+ continueConnectionStartup (id:String):void
+ cancelConnectionStartup (id:String):void
+ addIgnoreConnection (id:String):void
+ removeIgnoreConnection (id:String):void
+ setListenerConnectionStartup (l:RequestBeginListener ,wait:boolean):void
+ setListenerConnectionShutdown (l:RequestFinishListener ,wait:boolean):void
<< interface >>
Request
+ getRequestProcessor ():RequestProcessor
+ getRequestId ():String
+ getDbms ():Dbms
+ getDatabase ():Database
+ getConnection ():Connection
+ getTransaction ():Transaction

Figure 7 – Request Context

The *Request* context is depicted in Figure 7 and it may have several statements. This context provides a simple way to identify each request sent by a user in the context of a transaction.

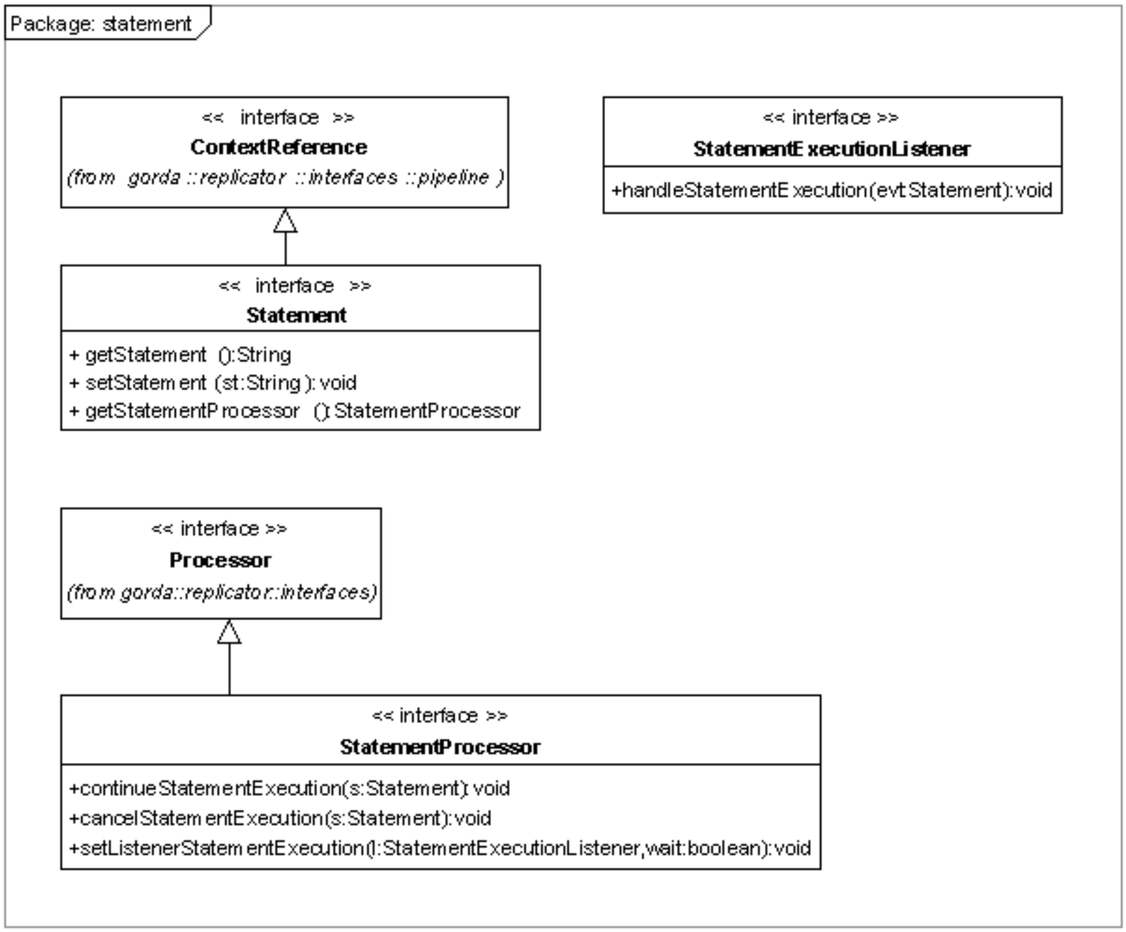

Package: statement
<< interface >>
ContextReference
(from gorda::replicator::interfaces::pipeline)
<< interface >>
StatementExecutionListener
+handleStatementExecution(evt:Statement):void
<< interface >>
Statement
+ getStatement ():String
+ setStatement (st:String): void
+ getStatementProcessor ():StatementProcessor
<< interface >>
Processor
(from gorda::replicator::interfaces)
<< interface >>
StatementProcessor
+continueStatementExecution(s:Statement): void
+cancelStatementExecution(s:Statement):void
+setListenerStatementExecution(l:StatementExecutionListener,wait:boolean):void

Figure 8 – Statement Phase

Figure 8 depicts the first stage of the pipeline. It intercepts single statements and allows one to inspect and modify them. If a request has several statements, the *StatementProcessor* sends a notification for each statement.

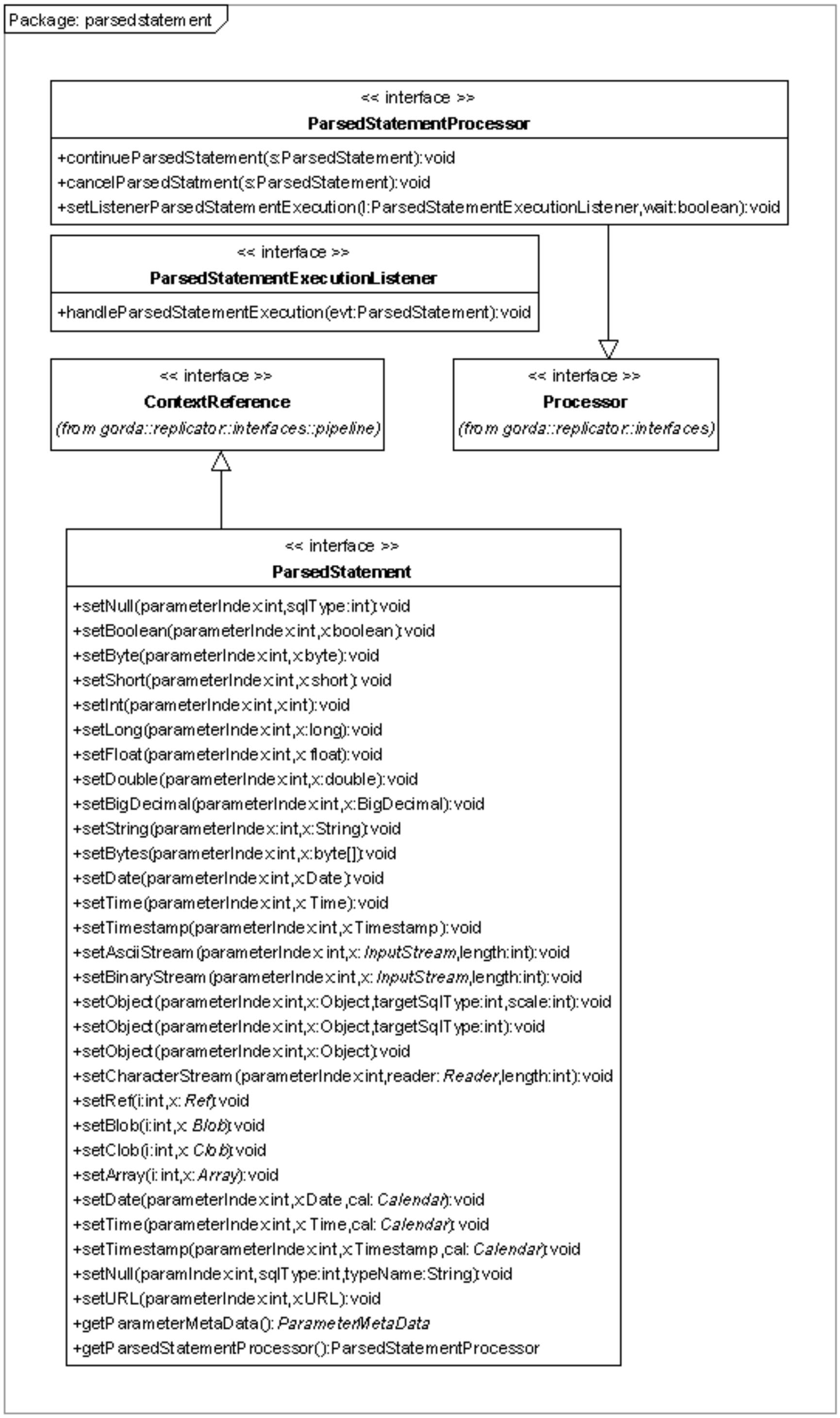

Package: parsedstatement
<< interface >>
ParsedStatementProcessor
+continueParsedStatement(s:ParsedStatement):void
+cancelParsedStatment(s:ParsedStatement):void
+setListenerParsedStatementExecution(l:ParsedStatementExecutionListener,wait:boolean):void
<< interface >>
ParsedStatementExecutionListener
+handleParsedStatementExecution(evt:ParsedStatement):void
<< interface >>
ContextReference
(from gorda::replicator::interfaces::pipeline)
<< interface >>
Processor
(from gorda::replicator::interfaces)
<< interface >>
ParsedStatement
+setNull(parameterIndex:int,sqlType:int):void
+setBoolean(parameterIndex:int,x:boolean):void
+setByte(parameterIndex:int,x:byte):void
+setShort(parameterIndex:int,x:short):void
+setInt(parameterIndex:int,x:int):void
+setLong(parameterIndex:int,x:long):void
+setFloat(parameterIndex:int,x:float):void
+setDouble(parameterIndex:int,x:double):void
+setBigDecimal(parameterIndex:int,x:BigDecimal):void
+setString(parameterIndex:int,x:String):void
+setBytes(parameterIndex:int,x:byte[]):void
+setDate(parameterIndex:int,x:Date):void
+setTime(parameterIndex:int,x:Time):void
+setTimestamp(parameterIndex:int,x:Timestamp):void
+setAsciiStream(parameterIndex:int,x:InputStream,length:int):void
+setBinaryStream(parameterIndex:int,x:InputStream,length:int):void
+setObject(parameterIndex:int,x:Object,targetSqlType:int,scale:int):void
+setObject(parameterIndex:int,x:Object,targetSqlType:int):void
+setObject(parameterIndex:int,x:Object):void
+setCharacterStream(parameterIndex:int,reader:Reader,length:int):void
+setRef(i:int,x:Ref):void
+setBlob(i:int,x:Blob):void
+setClob(i:int,x:Clob):void
+setArray(i:int,x:Array):void
+setDate(parameterIndex:int,x:Date,cal:Calendar):void
+setTime(parameterIndex:int,x:Time,cal:Calendar):void
+setTimestamp(parameterIndex:int,x:Timestamp,cal:Calendar):void
+setNull(paramIndex:int,sqlType:int,typeName:String):void
+setURL(parameterIndex:int,x:URL):void
+getParameterMetaData():ParameterMetaData
+getParsedStatementProcessor():ParsedStatementProcessor

Figure 9 – Parsed Statement Phase

When a statement is parsed, this information is exposed by the second stage of the pipeline which is depicted in Figure 9. By using this interface it is possible to identify and modify parameters in a statement that could be changed during an execution. For instance, if a cached plan or an ordinary parse-statement is about to be processed, it would be possible to redefine input parameters in order to guarantee determinism. This interface is similar to the *PrepareStatement* provided by the JDBC. However, it is worth noticing that the latter does not provide access to output parameter and does not provide methods to execute statements.

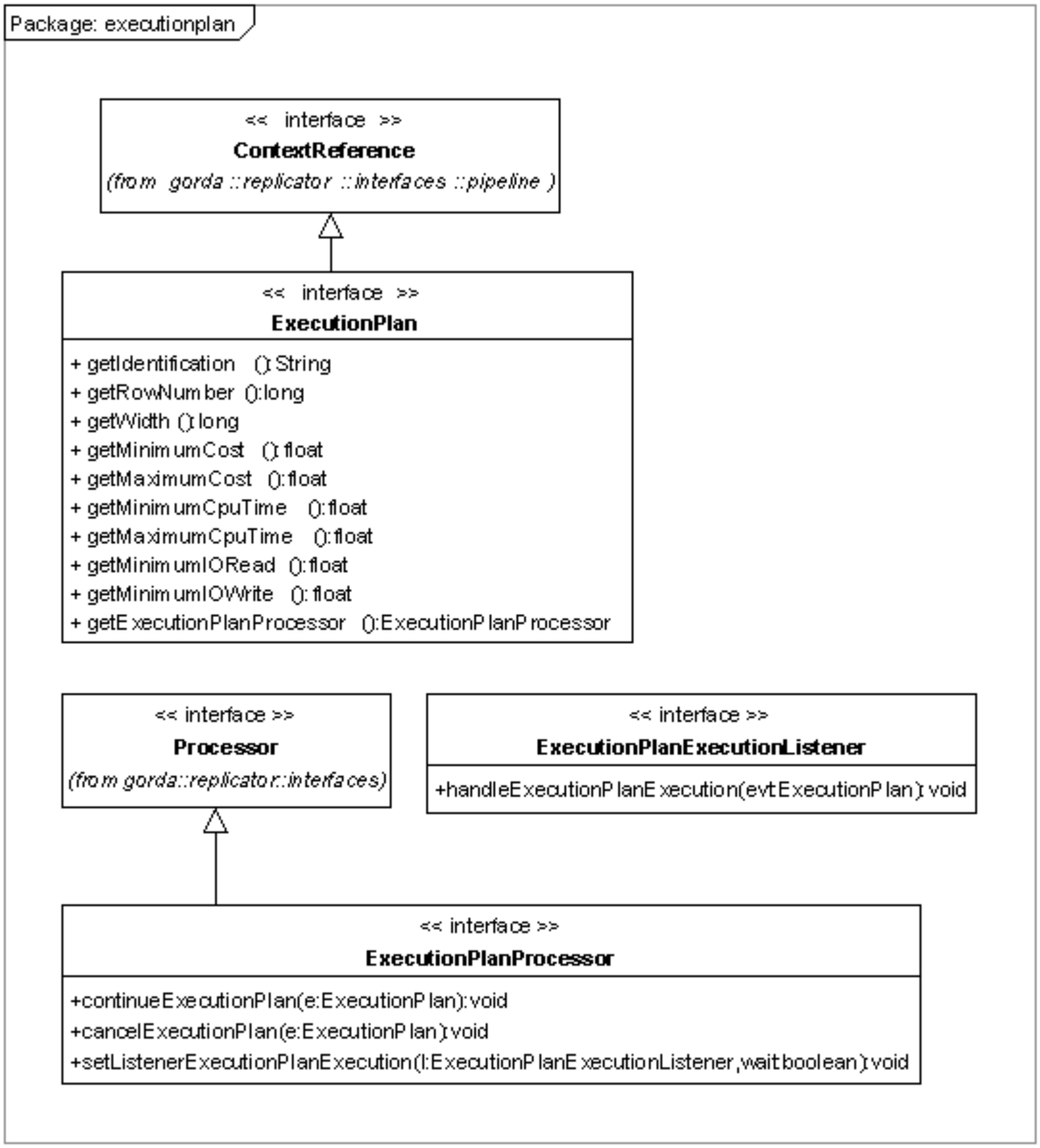

Package: executionplan
<< interface >>
ContextReference
(from gorda ::replicator ::interfaces ::pipeline)
<< interface >>
ExecutionPlan
+ getIdentification ():String
+ getRowNumber ():long
+ getWidth ():long
+ getMinimumCost ():float
+ getMaximumCost ():float
+ getMinimumCpuTime ():float
+ getMaximumCpuTime ():float
+ getMinimumIORead ():float
+ getMinimumIOWrite ():float
+ getExecutionPlanProcessor ():ExecutionPlanProcessor
<< interface >>
Processor
(from gorda::replicator::interfaces)
<< interface >>
ExecutionPlanExecutionListener
+handleExecutionPlanExecution(evt:ExecutionPlan):void
<< interface >>
ExecutionPlanProcessor
+continueExecutionPlan(e:ExecutionPlan):void
+cancelExecutionPlan(e:ExecutionPlan):void
+setListenerExecutionPlanExecution(l:ExecutionPlanExecutionListener,wait:boolean):void

Figure 10 – Execution Plan Phase

Figure 10 depicts the pipeline stage that provides access to execution plan information. It defines CPU, I/O cost, number of tuples and tuple's width associated to a statement execution. It is important to notice that a rendering of this interface should configure such information with values that are valid among distinct replicas as a variety of machines and dbms(s) may be used.

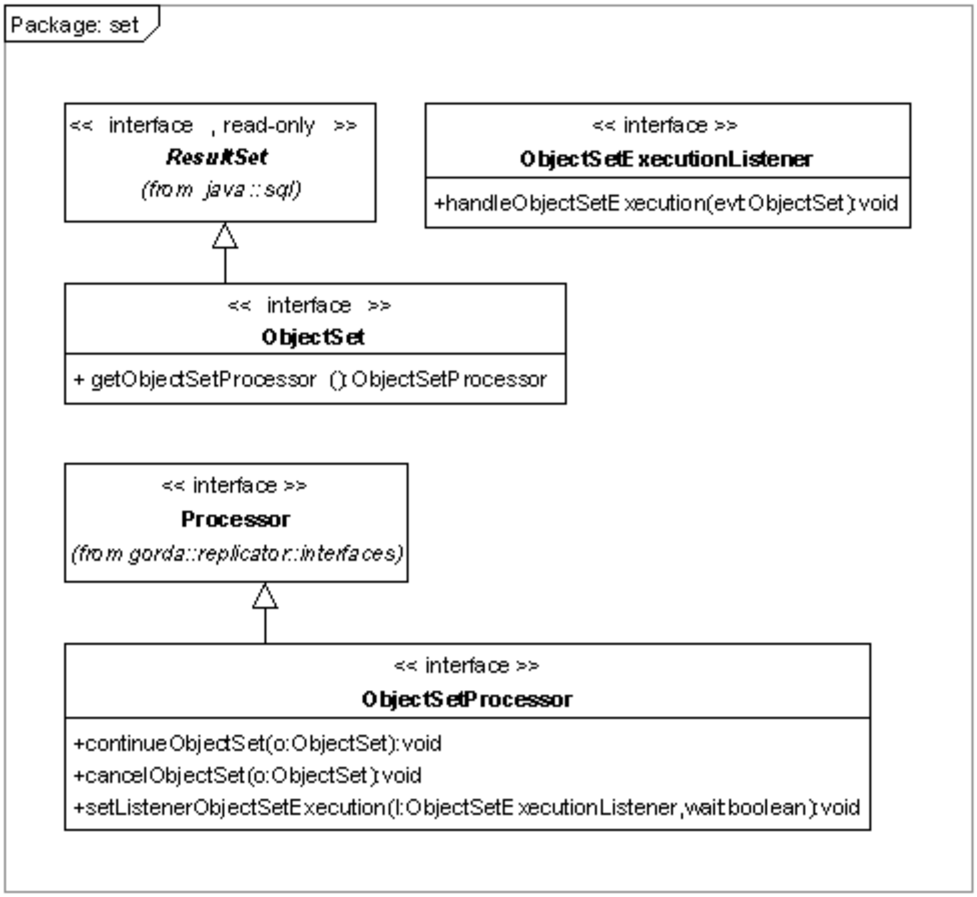

Package: set
<< interface , read-only >>
ResultSet
(from java :: sql)
<< interface >>
ObjectSetExecutionListener
+handleObjectSetExecution(evt:ObjectSet):void
<< interface >>
ObjectSet
+ getObjectSetProcessor ():ObjectSetProcessor
<< interface >>
Processor
(from gorda::replicator::interfaces)
<< interface >>
ObjectSetProcessor
+continueObjectSet(o:ObjectSet):void
+cancelObjectSet(o:ObjectSet):void
+setListenerObjectSetExecution(l:ObjectSetExecutionListener,wait:boolean):void

Figure 11 – Result Set Phase

Figure 11 presents an interface that provides access to result sets by means of an interface that extends the *ResultSet* interface provided by JDBC. The result of queries, updates and other set of statements can be retrieved by using this interface. The result however depends on the implementation. For instance, it could be rendered in a binary format with one tuple and one column in the result set, with multiple tuples and columns or as a lock set.

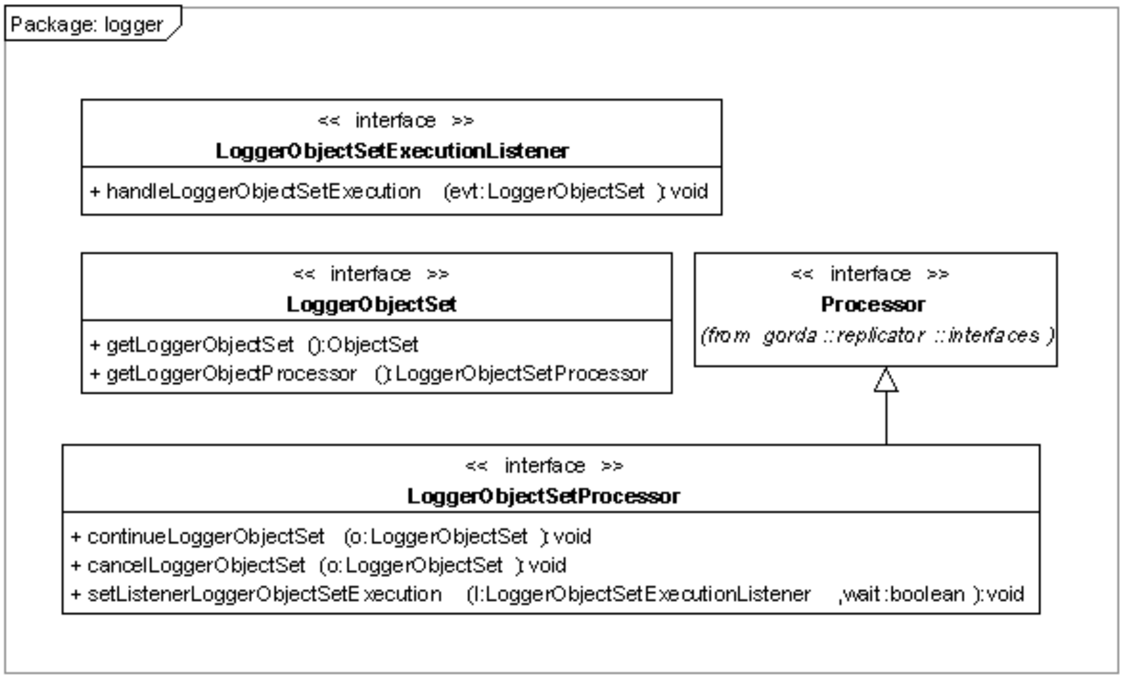

Package: logger
<< interface >>
LoggerObjectSetExecutionListener
+ handleLoggerObjectSetExecution (evt: LoggerObjectSet): void
<< interface >>
LoggerObjectSet
+ getLoggerObjectSet ():ObjectSet
+ getLoggerObjectProcessor (): LoggerObjectSetProcessor
<< interface >>
Processor
(from gorda :: replicator :: interfaces)
<< interface >>
LoggerObjectSetProcessor
+ continueLoggerObjectSet (o: LoggerObjectSet): void
+ cancelLoggerObjectSet (o: LoggerObjectSet): void
+ setListenerLoggerObjectSetExecution (l:LoggerObjectSetExecutionListener ,wait :boolean):void

Figure 12 – Logger Phase

Figure 12 depicts the *LoggerObjectSetProcessor* stage, which groups updates from different transactions in order to put themn in a stable storage. Most likely the information provided this stage depends on the architecture of the machine at which the database is running. However, one could transform this information to an independent form and this decision is an implementation detail.

It is worth mentioning that, when a replication component wants to submit queries or updates to the database, it should use a server-side interface to do that. In this document, we suggest that this interface should be based on the JDBC and we provide a *DriverManager* class depicted in Figure 13 that allows us to get access to it. The Driver interface is an abstraction that follows the pattern proposed by the JDBC but in a server-side scenario, it is not necessary as it is possible to send statements or other objects to be directly processed by a database. For instance, we could use this interface to insert log information in a metadata table.

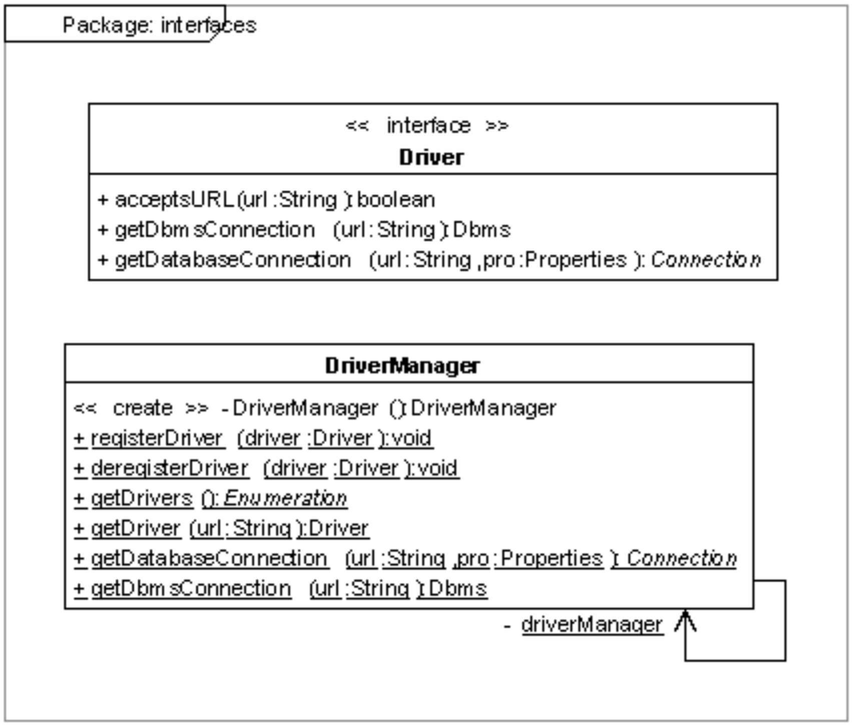

Package: interfaces
<< interface >>
Driver
+ acceptsURL(url :String):boolean
+ getDbmsConnection (url : String):Dbms
+ getDatabaseConnection (url : String ,pro :Properties): Connection
DriverManager
<< create >> - DriverManager ():DriverManager
+ registerDriver (driver :Driver):void
+ deregisterDriver (driver :Driver):void
+ getDrivers (): Enumeration
+ getDriver (url : String):Driver
+ getDatabaseConnection (url :String ,pro :Properties): Connection
+ getDbmsConnection (url :String):Dbms
- driverManager

Figure 13 – Server-Side JDBC access

3.3 Recovery interfaces

Database recovery interfaces (Figure 14) provide operations to gather and apply the data required for consistent replica recovery.

The *RecoveryManager* interface exposes methods to orchestrate the recovery scenario chosen:

- *setRecoveryStrategy* – chooses recovery strategy.
- *selectPeerHost* – selects one or several active replicas to participate in the recovery.

The *ImageCapturer* interface provides operations for the recovery scenario that uses database image transfer:

- *getDatabaseImage* – retrieves database image from a selected peer replica.
- *installDatabaseImage* – installs the given image at the recovering database server.

The *LoggerObjectSetRecovery* contains operations required for the recovery strategy that uses database updates log:

- *getTransactions* – retrieves updates from the log that satisfy specified conditions.
- *getLastCommittedTransaction* - retrieves the last committed transaction from the recovering database log
- *installLostUpdates* – applies missed updates to the recovering database.

- *getLogSize* – retrieves the size of the log.
- *decideTransactionStatus* – when a database is recovering and finds entries in a log referencing "*prepared transaction commands*", a notification has to be sent to a listener, that will decide automatically or by human intervention on the transaction fate: commit or abort.

The *TransactionRecovery* extends *Transaction* Interface to extract additional transactions information required for recovery protocols.

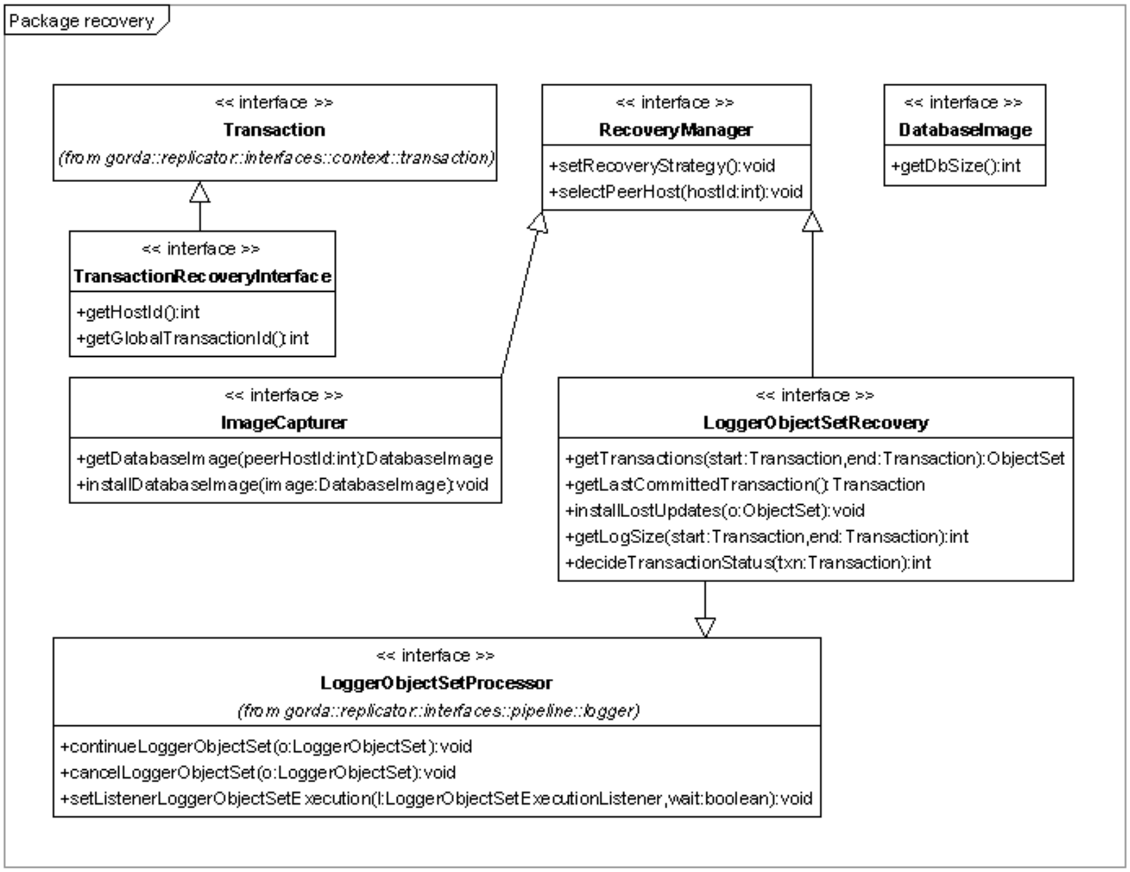

Package recovery
<< interface >>
Transaction
(from gorda::replicator::interfaces::context::transaction)
<< interface >>
RecoveryManager
+setRecoveryStrategy():void
+selectPeerHost(hostId:int):void
<< interface >>
DatabaseImage
+getDbSize():int
<< interface >>
TransactionRecoveryInterface
+getHostId():int
+getGlobalTransactionId():int
<< interface >>
ImageCapturer
+getDatabaseImage(peerHostId:int):DatabaseImage
+installDatabaseImage(image:DatabaseImage):void
<< interface >>
LoggerObjectSetRecovery
+getTransactions(start:Transaction,end:Transaction):ObjectSet
+getLastCommittedTransaction():Transaction
+installLostUpdates(o:ObjectSet):void
+getLogSize(start:Transaction,end:Transaction):int
+decideTransactionStatus(txn:Transaction):int
<< interface >>
LoggerObjectSetProcessor
(from gorda::replicator::interfaces::pipeline::logger)
+continueLoggerObjectSet(o:LoggerObjectSet):void
+cancelLoggerObjectSet(o:LoggerObjectSet):void
+setListenerLoggerObjectSetExecution(l:LoggerObjectSetExecutionListener,wait:boolean):void

Figure 14 - Recovery Interfaces

3.4 Management interfaces

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

- The GORDA Management Interface (GMI)
- The Database Management Interface (DBMI)
- The sensor interfaces (for QoS and failure sensors)
- The actuator interfaces

- The Autonomic Manager interface

Figure 15 outlines the relationship between the GMS and the DB cluster, via DBMI. It conveys the aggregation-type association between the GMS instance and DBMS instances that implement the DBMI interface. This implies that a GMS will communicate with a set of DBMS for management purposes. The figure also illustrates that the GMS contains a list of references to available and running replicas, identified by generic objects that are not manageable DBMS entities. These replicas can be used to host a deployed DBMS, as shown in the diagram.

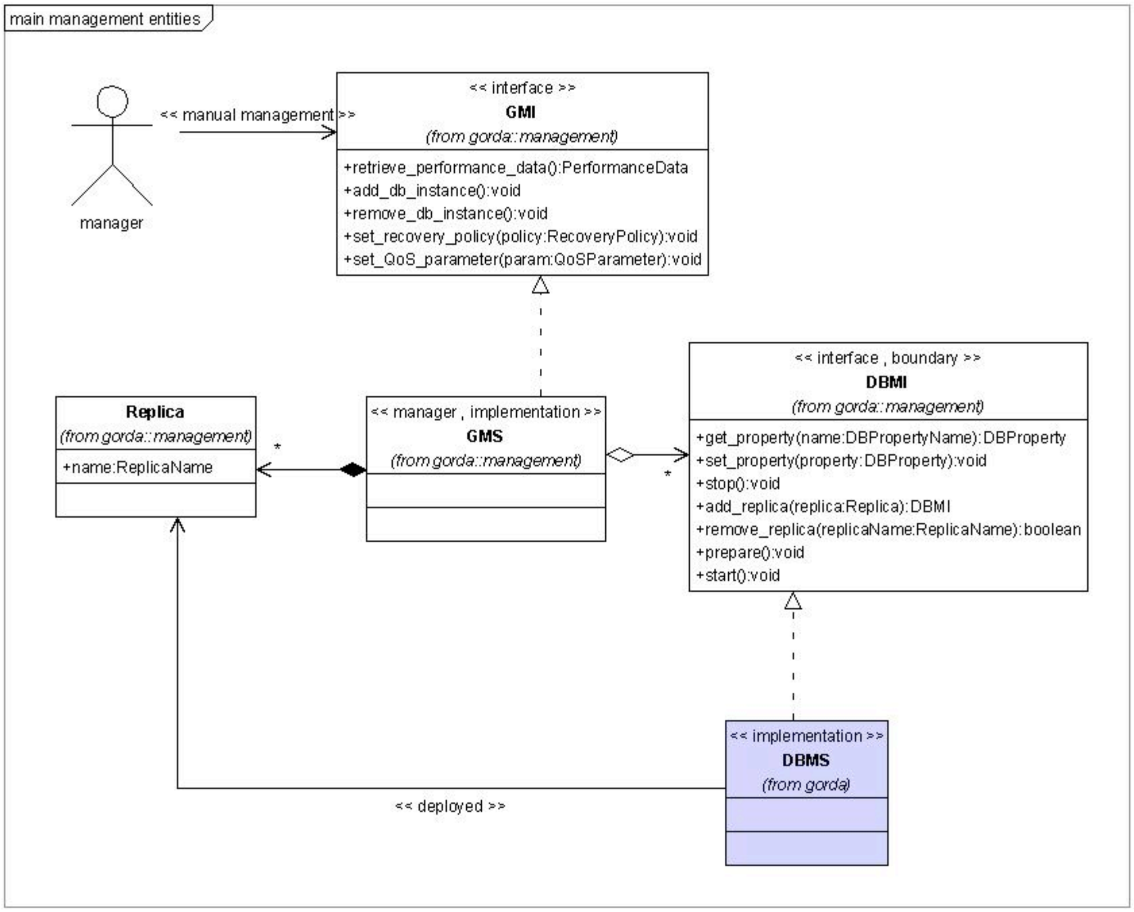

main management entities
<< manual management >>
manager
<< interface >>
GMI
(from gorda::management)
+retrieve_performance_data():PerformanceData
+add_db_instance():void
+remove_db_instance():void
+set_recovery_policy(policy:RecoveryPolicy):void
+set_QoS_parameter(param:QoSParameter):void
Replica
(from gorda::management)
+name:ReplicaName
*
<< manager , implementation >>
GMS
(from gorda::management)
*
<< interface , boundary >>
DBMI
(from gorda::management)
+get_property(name:DBPropertyName):DBProperty
+set_property(property:DBProperty):void
+stop():void
+add_replica(replica:Replica):DBMI
+remove_replica(replicaName:ReplicaName):boolean
+prepare():void
+start():void
<< implementation >>
DBMS
(from gorda)
<< deployed >>

Figure 15. Relationship between GMS and DBMI

The GORDA Management System (GMS) exposes the GMI interface to external operators (humans or third-party systems). This allows the specification of different management policies and in some cases, the overriding of particular autonomic management operations performed by the GMS.

- *retrieve_performance_data*: retrieves a snapshot of the system's performance which can be useful for creating or altering management policies.
- *add_db_instance*: instructs the manager to add a new DB replica to the cluster.

- *remove_db_instance*: instructs the manager to remove a DB replica from the cluster.
- *set_recovery_policy*: configures the recovery policies used by the GMS in case of failures.
- *set_QoS_parameter*: specifies the value of a particular QoS parameter (such as performance thresholds).

The Database Management Interface (DBMI) contains operations that the GORDA Management System uses for managing 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*: instruct this DB instance to start serving requests.
- *stop*: instructs this DB instance to stop its execution.
- *prepare*: instructs this DB instance to get ready for a future start (this might imply synchronizing its state with the other replicas).
- *add_replica*: adds a new replica to the cluster of DB replicas.
- *remove_replica*: instructs the cluster to remove a given replica from the set of DB instances.

The DBMS is a sample class that implements the DBMI interface and it is used to showcase management functionality.

The Replica corresponds to the physical replica entity and provides replica control operations. This does NOT represent a manageable DB instance, rather a representation of a machine that can be used in different purposes by autonomic managers. It has an attribute which holds its name in the GMS namespace.

In Figure 16, the basic management loop is illustrated with the entities available in the API. The GMS is shown interacting with autonomic managers via the *AutonomicManager* interface. In turn, all managers implementing this interface will subscribe to a set of sensors and will control the DB instances via actuators.

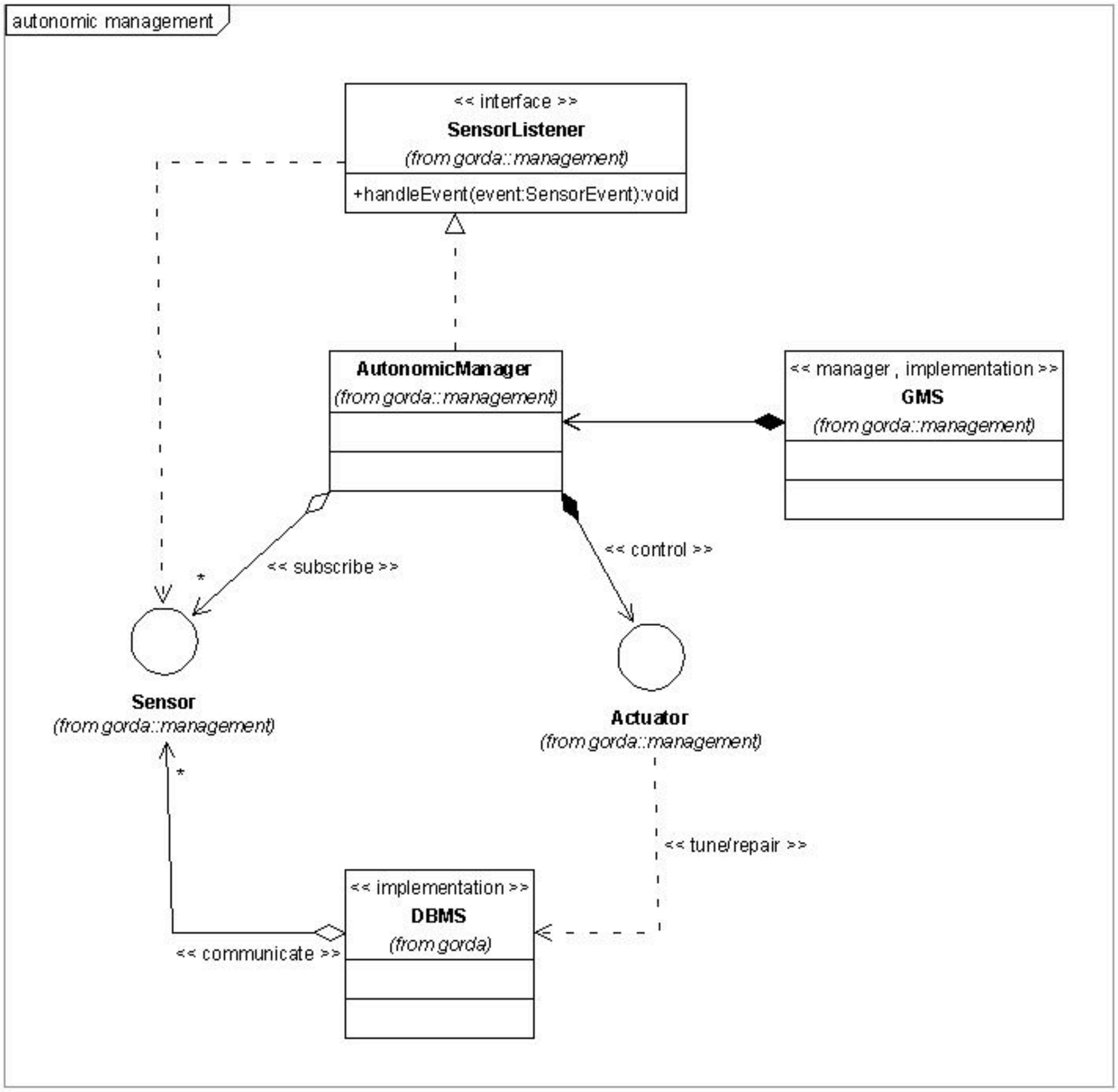


Figure 16. Autonomic Management Loop

The *AutonomicManager* is the super class of all autonomic managers. It provides common operations such as registering sensors.

The Sensor interface will be implemented by sensor classes used by autonomic managers. It has operations related to sensor and listener setup.

- *addListener*: adds a notification listener for events originating from this sensor.
- *removeListener*: removes a previously added listener, thus preventing it from receiving further notifications.
- *setSamplingInterval*: instructs the sensor to consider the new interval when checking the state of the monitored resource (DB instance).
- *getSamplingInterval*: returns the sampling interval currently in consideration by the sensor.

The *SensorListener* interface must be implemented by specific sensor listeners. Its *handleEvent* method acts as a callback method from the sensor, to notify that a measurement has occurred.

The *Actuator* interface must be implemented by all the actuators corresponding to the autonomic managers. Its implementation is very "manager-specific" and the actuators may have direct hooks for the DB instances. The core actuators, however, can use the DBMI interface to perform management operations.

The core autonomic managers to be provided in the reference implementation, together with their actuators and sensors, are illustrated in Figure 17. This puts their actual implementations into the context of the management APIs.

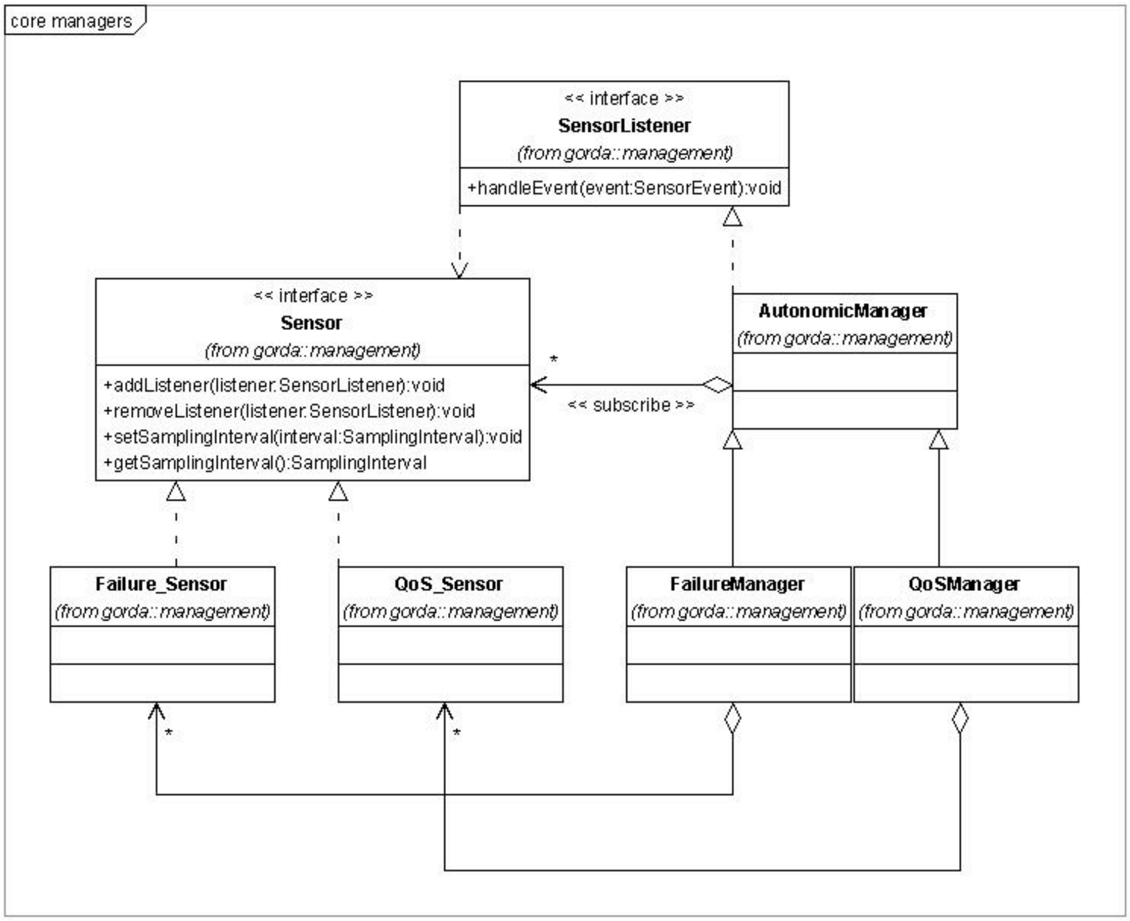


Figure 17. Core Autonomic Managers, Actuators and Sensors

3.5 Group communication interfaces

Communication interfaces, shown from the Figure 18 to the Figure 21, 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. This interface was built to be generic and can be implemented using several existing solutions.

Although configuration of Quality of Service (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.

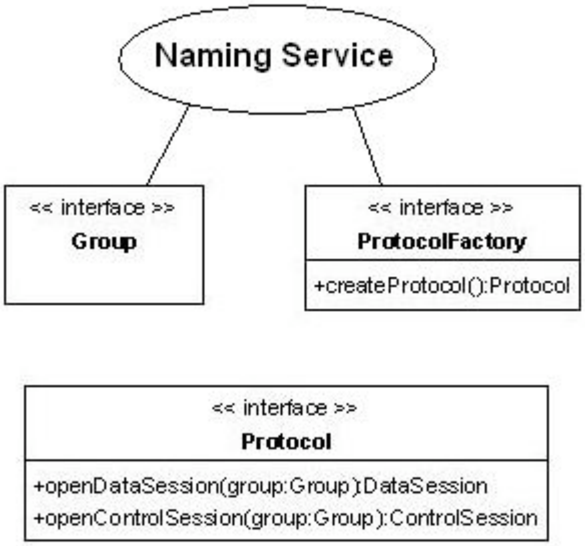
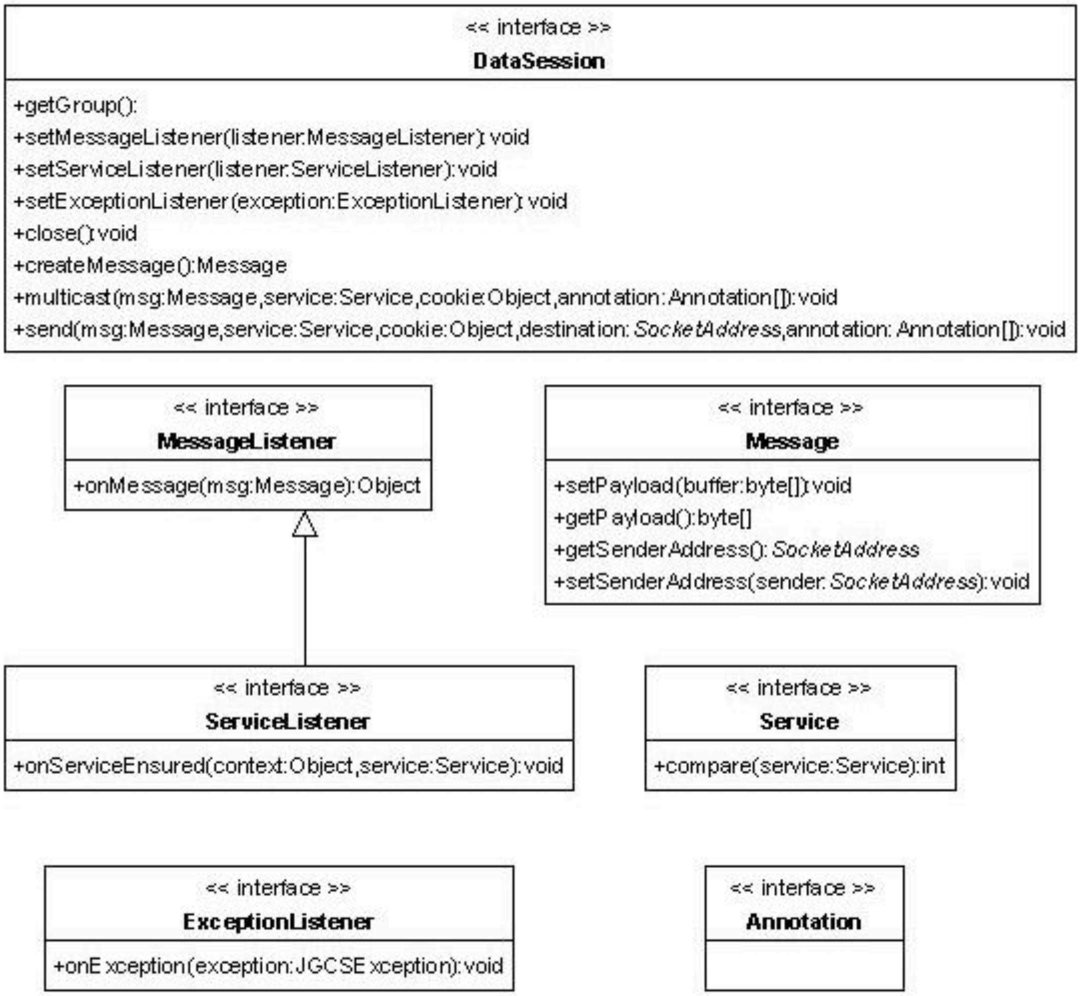

Naming Service
<< interface >>
Group
<< interface >>
ProtocolFactory
+createProtocol():Protocol
<< interface >>
Protocol
+openDataSession(group:Group):DataSession
+openControlSession(group:Group):ControlSession

Figure 18 - Protocol creation interfaces.

Communication sessions are previously configured and available on execution time with several types of QoS. A session can be created through a Directory and Naming interface (e.g. JNDI) and exposes methods to send messages and register message listeners. To use the communication interfaces, the user must first obtain instances of two classes: a *ProtocolFactory* and a *Group*. The *ProtocolFactory* creates instances of protocols (class *Protocol*) with a given configuration that is defined by the *Group* object. The *Protocol* is an instance of the solution used to provide group communication. From the *Protocol* the user can create instances of *DataSession* and *ControlSession*. The Figure 18 shows the described interfaces. The *DataSession* is used to send and receive messages and the *ControlSession* is used to join and leave the group and to receive notifications about other members (failure or explicit leaving and joining).



<< interface >>
DataSession
+getGroup():
+setMessageListener(listener:MessageListener):void
+setServiceListener(listener:ServiceListener):void
+setExceptionListener(exception:ExceptionListener):void
+close():void
+createMessage():Message
+multicast(msg:Message,service:Service,cookie:Object,annotation:Annotation[]):void
+send(msg:Message,service:Service,cookie:Object,destination:SocketAddress,annotation:Annotation[]):void
<< interface >>
MessageListener
+onMessage(msg:Message):Object
<< interface >>
Message
+setPayload(buffer:byte[]):void
+getPayload():byte[]
+getSenderAddress():SocketAddress
+setSenderAddress(sender:SocketAddress):void
<< interface >>
ServiceListener
+onServiceEnsured(context:Object,service:Service):void
<< interface >>
Service
+compare(service:Service):int
<< interface >>
ExceptionListener
+onException(exception:JGCSException):void
<< interface >>
Annotation

Figure 19 - Data handling interfaces.

Using the *DataSession*, messages can be sent to the group (multicast) or just to one member of the group (send). To receive messages, the programmer must register a listener (*MessageListener*) on this session. The *DataSession* exposes the following interface:

- *createMessage* – used to create empty messages to fill in the payload and send it;
- *multicast* – used to send a message to the group;
- *send* – used to send a message to a specific member of the group;
- *close* – closes this session. After this call, no messages can be send or received;
- *setMessageListener* – sets a listener to asynchronously receive messages from the group;
- *setExceptionListener* – set a listener to receive exceptions that can occur upon message reception;

- *setServiceListener* – sets a listener to receive future notifications of services guaranteed by the protocol. This functionality is detailed on Session 3.7.

The *Message* class exposes the following interface:

- *setPayload* – sets the payload of the message;
- *getPayload* – upon reception, this method gets the payload of the message;
- *getSenderAddress* – gets the address of the sender of the message;
- *setSenderAddress* – sets the address of the sender, if needed.

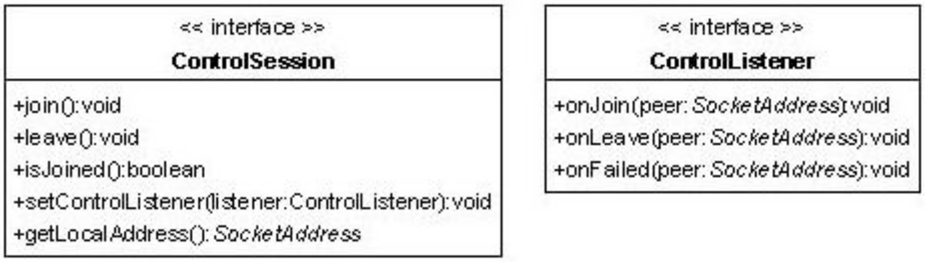

<< interface >>
ControlSession
+join():void
+leave():void
+isJoined():boolean
+setControlListener(listener:ControlListener):void
+getLocalAddress(): SocketAddress
<< interface >>
ControlListener
+onJoin(peer: SocketAddress):void
+onLeave(peer: SocketAddress):void
+onFailed(peer: SocketAddress):void

Figure 20 - Simple group management interfaces.

The *ControlSession* is used to join and leave the group and receive notification about other members. To receive these notifications, the programmer must register a *ControlListener* in this session. This Session exposes the following interface:

- *getLocalAddress* – gets the address binded by the protocol that identifies the member;
- *isJoined* – verifies if the member is joined or not;
- *join* – joins the group;
- *leave* – leaves the group;
- *setControlListener* – sets a listener to receive notifications about other members (joining and leaving).

3.6 Support for View Synchrony and Extended View Synchrony

The Figure 21 depicts the extensions of the basic control session to provide virtual synchrony and extended virtual synchrony. The *MembershipSession* extends the *ControlSession* with methods that give information about the membership and a method to register a listener to the membership. The *BlockSession* extends the *MembershipSession* with the block functionality: before view change, the application is notified to flush pending messages, using the previously registered *BlockListener*. The application flushes all messages and notifies the membership service, invoking the *blockOk* method. After this process, a view change is performed.

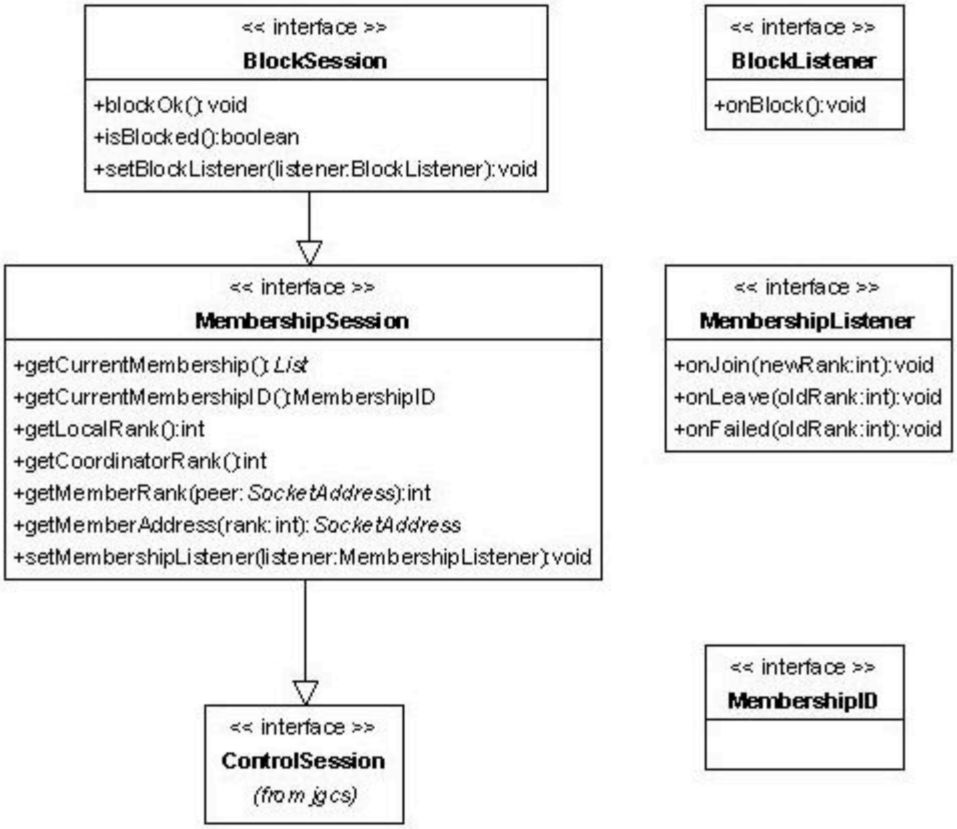

<< interface >>
BlockSession
+blockOk():void
+isBlocked():boolean
+setBlockListener(listener:BlockListener):void
<< interface >>
BlockListener
+onBlock():void
<< interface >>
MembershipSession
+getCurrentMembership():List
+getCurrentMembershipID():MembershipID
+getLocalRank():int
+getCoordinatorRank():int
+getMemberRank(peer:SocketAddress):int
+getMemberAddress(rank:int):SocketAddress
+setMembershipListener(listener:MembershipListener):void
<< interface >>
MembershipListener
+onJoin(newRank:int):void
+onLeave(oldRank:int):void
+onFailed(oldRank:int):void
<< interface >>
MembershipID
<< interface >>
ControlSession
(from jgcs)

Figure 21 - Interface extensions for view synchrony.

3.7 Support for Optimistic and Semantic Protocols

As shown in the Figure 19, the group communication interfaces specification contains two interfaces, namely the *Service* and the *Annotation*. (e.g. a message is non-uniformly delivered and notified when the message is already uniform). This is provided by the *ServiceListener* interface and the *Service* interface and a *Context object*. The *ServiceListener* 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 cookie object is provided by the application upon the delivery of the message. Implementations of the *Service* interface must implement the compare method to define a partial order between service guarantees.

A Message can be sent tagged with an *Annotation* which is semantic information that can be used by content based optimization protocols.

3.8 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.

4 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 *LoggerObjectSetProcessor* 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.

4.1 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 *LoggerObjectSetRecovery* 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. 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 *LoggerObjecSet* 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 *LoggerObjectSetRecovery* 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 determinism problem similar to the state machine approach, as it would be if the other interfaces rather than the *LoggerObjectSetProcessor* would be used to develop an asynchronous replication.

Once the changes are propagated to the replica, a component that renders a server-side JDBC API 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.

4.2 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 allows that. If for some reason the listener wants to cancel the execution, it calls the *cancelTransactionExecution*. This feature is enabled when a listener is register and sets the parameter wait to true for the event., 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.

4.3 Replicas recovery

An essential aspect of replicated databases is the need to allow failed nodes to recover and rejoin the system without interrupting the ongoing transaction processing on the available nodes. In particular, before a joining site can execute transactions, an up-to-date replica has to provide the current state of the data to the joining site. Traditionally, this is done by copying the entire database state to a new or recovering site. This consumes bandwidth and time, as normal update operations of the database system are interrupted. If the amount of data to be copied is large, the availability requirements of the system might be violated. Obviously it is impractical to perform data transfer as a single atomic step. Furthermore, the database consistency must be ensured after transfer: if some replicas are available for transactions processing during state transfer, the new or recovering site might never be able to catch up with such nodes. Since this depends on the update transactions ratio in the workload and the database size, one way to solve the problem would be to consider the workload while designing a state transfer mechanism.
The suggested recovery scenarios can be divided depending on

- what is transferred between replicas
    - missed updates.

If a recovering replica was offline for relatively short period of time or the transaction workload is not writes intensive, only the changes made during replica's downtime need to be applied. In this case the *LoggerRecoveryInterface* retrieves from the log the identifier of the last committed transaction on the recovering site. The *RecoveryManager* chooses one or more peer sites to participate in the recovery. The *LoggerRecoveryInterface* extracts missed updates from the recoverer replica's log and applies them to the recovering database.

  - an image of the whole database.

    If a replica was offline for a long time and the transaction load is updates intensive, it might take too long to apply the updates at the recovering site. For such case and when a new replica is bootstrapped, an image of the whole database is sent. In this case the typical recovery scenario is as follows: GCS or GMS initiates recovery protocol. The *RecoveryManager* selects an active replica to act as a peer site for the recovering replica. *ImageCapturer* contacts the recoverer to get the database image and installs it on the recovering site.

- how many up-to-date replicas participate in recovery procedure
  - only one active replica
  - several active replicas in the system. To avoid overloading a single active node several up-to-date replicas participate in recovery: each node sends only a part of updates or database image.

The system automatically chooses the recovery strategy that is expected to take less time depending on the database size and the changes the joining site needs to install.

The recovery protocols can be triggered by system view changes provided by group communication interfaces or by operations exposed by management interfaces (DBMI).

4.4 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 it self as a listener for events originating at instances of sensors such as *QoS_Sensor* and *Failure_Sensor* implementations 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 control the sensors to obtain this information with the required sampling rates. For this, it can call the *setSamplingInterval* method of the sensors.

If the desired QoS is not maintained, the manager can decide to add a replica to the cluster. This operation is illustrated in Figure 22 with a UML sequence diagram.

The manager first needs to obtain and setup an available replica in its set of managed machines. It can then call the *add_replica* operation on **any** managed DB instance to instruct the cluster to take the new replica into consideration. When the manager decides that the need for the new replica to become active is incumbent, it can either "prepare" the replica or directly “start” it by using the prepare and start operations respectively. 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. When preparing or starting a replica, the Replicator will bring the newly activated replica up to date. As an example, when using C-JDBC, 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. The start operation also performs an “atomic” prepare operation so there is no need to precede start operations by prepare operations. The advantage of having an explicit prepare operations is that one can pre-emptively get a replica into an “almost-ready” state so that when it has to become active it does not need to perform lengthy synchronization operations.

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 stop operation on the DBMI.

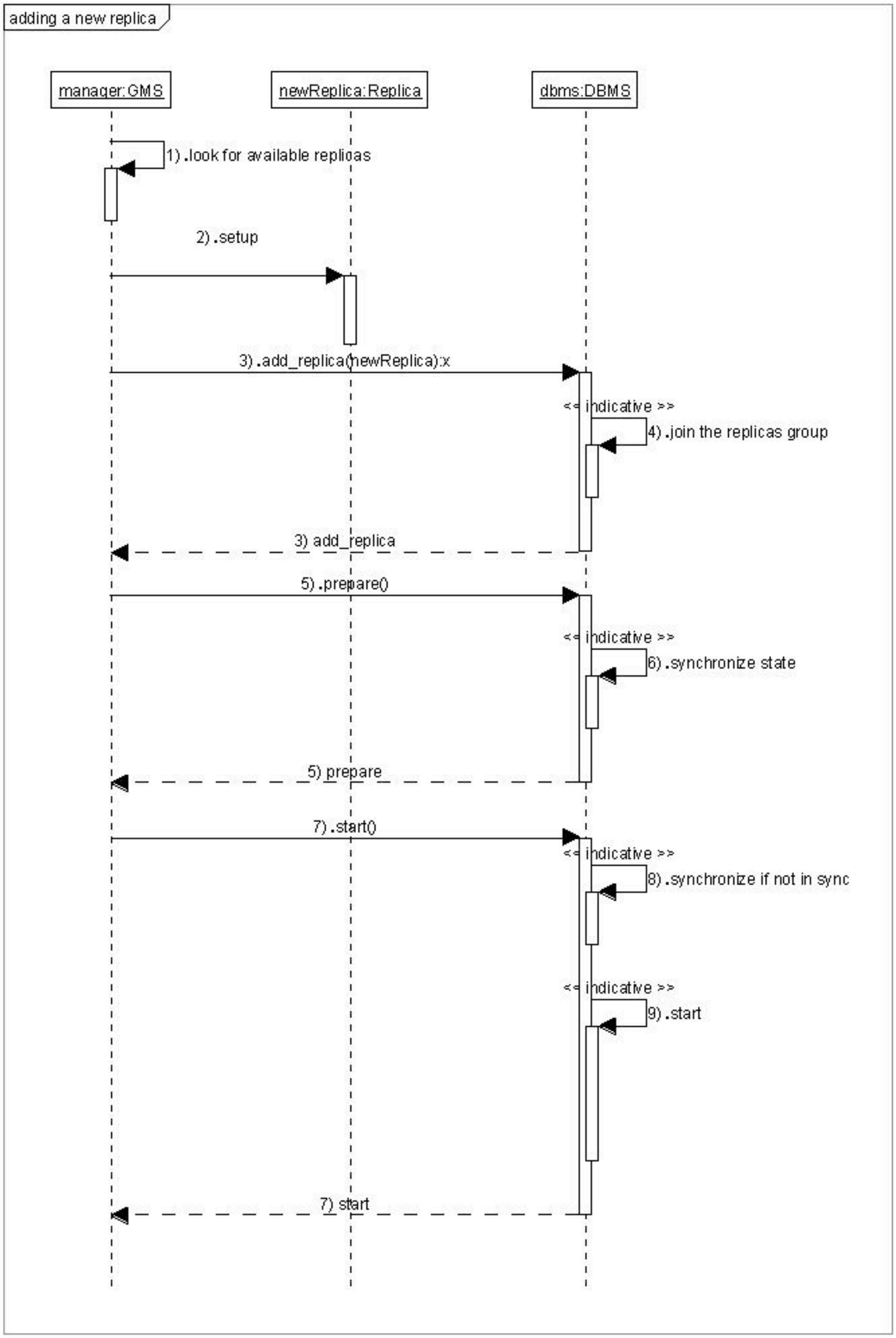


Figure 22. Adding a New Replica: UML Sequence Diagram.

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.

4.5 Group communication

A replication component uses the communication interfaces to join the group of distributed replicas. Using these interfaces, the component must create a *Protocol* through the *ProtocolFactory* and then create a *DataSession* and a *ControlSession* with a given configuration using the *Group* interface. To receive messages of other members of the distributed replication component, the replica must register a *MessageListener* in the *DataSession*. To be notified of new joining, failure and leaving of replicas, it must register a *MembershipListener* in the *ControlSession*.

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 *MessageListener*. When a replica fails, all other replicas will be notified, receiving a notification on the *onFailed* method.

If one replica wants to use optimistic deliveries on totally ordered messages, it must also register in the channel using the *ServiceListener* interface. In this case, upon reception of a message, the replica must return a cookie object (that will identify the message) to the communication service and will be notified later when the order of the messages is stabilised.