Project no. 004758

GORDA

Open Replication Of Databases

Specific Targeted Research Project

Software and Services

Replication modules reference implementation

GORDA Deliverable D3.3

Due date of deliverable: 2006/09/30
Actual submission date: 2006/09/30
Revision date: 2006/12/30

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

Universidade do Minho

Revision 1.1

Project co-funded by the European Commission within the Sixth Framework Programme (2002-2006)		
Dissemination Level		
PU	Public	X
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Contributors

Alfrânio Correia Júnior
José Pereira
Ricardo Vilaça
Rui Oliveira

Abstract

This document describes the ESCADA Toolkit, a modular reference implementation of replication protocols built on the GORDA Architecture and Programming Interfaces. The ESCADA Toolkit can be configured to provide both optimistically as well as conservatively synchronized protocols, with different consistency criteria, and exploiting optimizations for different environments.

Chapter 1

Architecture

The ESCADA Toolkit provides the means to assemble the target replication protocols from a set of common components. By combining this toolkit with a GAPI provider, such as PostgreSQL/G or Derby/G, a complete replicated DBMS solution is achieved.

In detail, the ESCADA Toolkit provides a concurrency control mechanism, a communication infra-structure, fault tolerance, recovery and configuration strategies. In particular, group communication primitives such as reliable and atomic broadcast are used as building blocks even for replication protocols that are not originally based on group communication. By doing this, we ease the reasoning on protocols and their development as issues such as failure detection and retransmission are delegated to the group communication abstraction. To further ease the development, we use an event-driven architecture in which both downward and upward communication between any pair of layers is done exchanging events [17]. This event-driven architecture enables better software design that facilitates reuse, loose coupling, and easy testing of software components. In addition, we use the inversion of control pattern to glue the layers together [8]. Specifically, this pattern is used to instantiate each layer and to register for events. Even internally in each layer, the objects are combined by means of the inversion control pattern.

Figure 1.1 depicts the architecture used by the ESCADA Toolkit. The database engine implements the GAPI [7], which allows to transparently develop different replication protocols suited to different application semantics. The jGCS layer [4] enables the use of any Group Communication Toolkit providing another generic interface in which a replication protocol might rely on to synchronize information among replicas. The Group Communication layer [2] is responsible for propagating information among replicas.

In what follows, we detail this event-driven architecture focusing on the ESCADA Toolkit. Further details on the GAPI or its PostgreSQL or Derby rendering can be found at [1].

1.1 GAPI Layer

The GAPI defines five contexts: (*i*) database management system, (*ii*) database, (*iii*) connection, (*v*) transaction and (*iv*) request. The database management system, or simply dbms, identifies an instance of an application which might have different databases such as a human resource database and a marketing database. Databases allow clients to connect to, retrieve and change information. Clients are identified for a set of connection properties where among them, one can find user identification, password and encoding information. For transactional systems, a set of interactions between clients and servers are guarded in the context of a transaction and inherits the following properties: (a) atomicity, (c) consistency, (i) isolation and (d) durability. Each interaction is identified by a request. For instance, if a client sends a batch to be processed, the set of statements in the batch are in the scope of the same request.

A request passes through different phases organized as a pipeline. First, a request is split into one or more statements which are individually sent to the next phases, to be parsed, optimized and executed. Results from an execution can be directly observed in a logical or physical format. In the former case, updates might be easily transferred among different operating systems and databases. In the latter case, updates from different transactions might be grouped and are represented in a machine dependent format.

Different events are generated for each context and phase, triggering notifications, which are handled by components registered for receiving them. In the ESCADA Toolkit, the capture process receives the GAPI notifications and forwards them as described as follows.

1.2 Processes

A process represents a container that allows to interconnect different pieces of code in order to build a replication protocol. Five canonical processes are used to do this task: (*i*) capture process, (*ii*) kernel process, which is usually known as distribution process, (*iii*) apply process, (*iv*) recovery process and (*v*) configure and monitor process.

In particular, different protocols require different implementations of the capture, kernel and apply processes. Our current prototype handles the DBSM and the Primary Backup protocols while the Conservative and State Machine approaches are being tested.

1.2.1 Capture Process

The capture process receives events from the GAPI, converts them to appropriate events in the ESCADA Toolkit and notifies the other process. In particular for the DBSM (Database State Machine), it receives a transaction begin request and registers the current transaction context. Before sending a commit request, the GAPI should send an ObjectSet which has updates and for some consistency criteria reads too. The capture process gathers such information and upon a commit request constructs an internal transaction event that carries the transaction identification along with updates and reads. Then it notifies the kernel process which is responsible for implementing the concurrency control mechanism.

The class **CaptureKernel** captures events on behalf of all protocols.

1.2.2 Kernel Process

The kernel process is usually known as the distribution process, but in the ESCADA Toolkit, we name it differently as it also provides core functions with respect to a replication protocol. In other words, the kernel process is the core of the ESCADA Toolkit as it directly implements the code that propagates information among replicas and guarantees a consistency criterion. In particular for the DBSM, it certifies a transaction deciding its outcome. In detail, it receives notifications from the capture process, propagates them among replicas by means of the jGCS and, after certifying a transaction, notifies the apply process.

There are three variations of the DBSM protocol. The first variation is implemented by the class **DbmsKernel**, provides snapshot isolation as a consistency criterion and does not exploit optimistic delivery. In contrast, the class **OptmisticDbmsKernel** exploits optimistic delivery. The class **SerializabilityKernel** provides serializability while at the same time exploits optimistic delivery.

There are two different classes implementing the Primary Backup protocol. The class **AsyncPbKernel** provides asynchronous primary backup replication while the **SyncPbKernel** provides synchronous replication.

The Active protocol is implemented by the class **ActiveKernel** and the Conservative protocol is currently under development.

In order to ease development and testing, the Toolkit has a Loopback protocol that does not use network and is implemented by the class **LoopBackKernel**.

1.2.3 Apply Process

The apply process is a key component to the performance of replication protocols built on the GAPI and must address the issues of parallel execution and transaction priority. Succinctly, it has two different strategies according to a transaction's outcome: abort or commit. In particular for the DBSM, when an abort is received, at the initiator replica (i.e., the site that received the client request), the transaction that were blocked is canceled, and at a remote replica it is disregarded. When a commit is received, this process groups operations from different transactions and execute them as a batch. If a batch that is being processed does not conflict with other batches being created, they are executed in parallel. Otherwise, they are executed as soon as the conflicting batch is finished. At the initiator replica, the execution consists on issuing a continue request. At a remote replica, the apply process uses a JDBC interface to execute remote operations. The GORDA

API assumes the existence of a server-side JDBC API that should be used to inject operations in a database.

The apply process is implemented by the class **ApplyKernel** for the DBSM, Primary Backup and Conservative Protocols. In particular, the Active protocol requires a different implementation as it applies statements and is currently under development.

1.2.4 Recovery Process

The recovery process executes when a replica joins the group and exploits the group membership properties exposed by the jGCS as follows. Every time a replica joins a group, a blocking notification is sent by the jGCS indicating this fact. When a replica receives this notification, it should stop sending and receiving messages, putting them into buffers. Meanwhile, a new notification is sent by the jGCS informing that messages that were sent while the new replica was joining the group were delivered and which is the new replica. Right after this information, the buffered messages should be processed as well as any new message. [1]

Unfortunately, the new replica may not be able to process transactions as its database may have a non-updated state. Considering so, the recovery process in the joined replica chooses an other replica as a database source and carries out a state transfer.

To easier the development of different recovery protocols we developed an abstract Recovery Kernel **RecoveryKernel** that have the common infrastructure to any recovery protocol. Then we implement three different recovery protocols:

- Transfer of the entire database using some mechanism provided by the database engine, reflected by the Database context of the GAPI. To use this protocol the class **RecoveryKernelDBImage** should be used as the RecoveryKernel.
- Transfer of the missed updates to joining replicas using middleware logging. To use this protocol the class **RecoveryKernelUpdates** should be used as the RecoveryKernel.
- Transfer of the last version of changed objects. This implements the protocol Lazy Data Transfer presented in [11]. To use this protocol the class **RecoveryKernelLazy** should be used as the RecoveryKernel.

Furthermore, the chosen replica needs not only to transfer the database state but also information regarding the used replication protocol. In the case of DBSM we need to transfer the information stored in buffers. This is necessary to run the certification at the new replica. Otherwise, the joined replica would decide different on a transaction's outcome as its certification buffer is empty.

[1]The recovery process and the kernel process are sequential.

1.2.5 Configuring and Monitoring

This process provides the ability to negotiate services among distributed replicas such as consistency criterion and which replica should be the master in a primary-backup replication and to act as a monitor by inspecting resources and important variables in the system. For instance, it is possible to check information on storage, memory, cpu and network usage and specific information on replication such as number of transactions being applied in parallel and transactions waiting to be applied. It is worth noticing that detail information on these module shall be provided in an other document specifically designated for configuration and monitoring.

Configuring

The negotiation of services currently supports master election, needed for primary-backup protocol and recovery purposes. We are working on more services namely on consistency criterion and certification type.

Monitoring

The monitor is built on JMX (Java Management Extensions) which provides the means for configuring and monitoring distributed applications. There are two types of resources to be monitored. One that requires an interaction with the operating system to be monitored and others that can be directly monitored by inspecting some variables in the Java code. In both cases, the Escada Toolkit is instrumented with JMX components which export properties associated with the resources. These components are Managed Bean objects, or simply MBeans, with *get()* methods to retrieve properties, *set()* methods to set properties when appropriate and operations that are used to perform tasks. The MBeans are registered in a management server that provides a set of services in order to make them available to remote management applications.

The storage, memory, cpu and network resources are mapped to MBean objects through a JNI (Java Native Interface). The Escada Toolkit and the group communication, in particular, by means of the jGCS provide additional information that can be monitored. In our architecture, the MBean component is know as a sensor and exports a set of properties known as attributes which are made available to remote management applications through a custom BSD socket-based protocol:

- **LIST SENSORS** retrieves which sensors are available.
- **LIST SENSORS ATTRIBUTES sensorId** retrieves which attributes might be monitored for a sensor.
- **GET sensorId attrId** gets value for an specific sensor attribute.
- **SET sensorId attrId value** sets value for an specific sensor attribute.

- **OPERATIONS sensorId** retrieves the operations that can be invoked on behalf of a sensor.
- **INVOKE SENSOR operationId** invokes an operation.
- **CREATE REPLICA replicaId** adds a new replica to a cluster from what this management application is running.
- **STOP REPLICA replicaId** removes a replica from a cluster.
- **PAUSE REPLICA replicaId** pauses execution of a replica.
- **CONTINUE REPLICA replicaId** resumes execution of a replica.
- **REGISTER LISTENER sensorId** registers for notification events which are triggered whenever a attribute changes.
- **UNREGISTER LISTENER sensorId** unregister for notification events.
- **NOTIFY sensorId attrId value** is issued whenever a notification event is sent to a listener.

We are working on designing more operations besides the stop, pause and continue. Most likely this process should provide an interface to the configuration features described in the previous section. For instance, it shall be possible to define which replica should be a master in primary backup scenario.

1.3 Event Layer

This layer is a set of components and interfaces that enables processes to exchanged information. If a process *a* is interested in receiving information from a process *b* the following should be done. First process *a* should implement an interface that allows it to receive such information. There are different interfaces for different events matching the GAPI's contexts and phases. Then it should register in a notifier to receive notifications. Discovering a notifier involves getting in touch with process *b* that informs for an event in which notifier process *b* should be registered.

Its current implementation invokes methods in the same Virtual Java Machine (i.e., in the same address space) due to performance reasons, but it might be easily extended to provide communication among process in distinct Virtual Java Machines without changing the other layers. For instance, one might use a JRMI (Java Remote Method Invocation) to invoke operations on other address spaces.

1.4 Communication Layer

This layer is built on the jGCS augmenting it and provides multiplexing and demultiplexing of messages. In other words, this is done to enable sending messages from a process and delivering them to the counterparts processes, i.e., a process that has a channel with the same identification. For instance, when a message is sent from the kernel process, through a channel with identification c, it is delivered only by channels with the same identification. Most likely to the kernel processes distributed among replicas, if such channels are correctly configured.

Membership, block and view messages from the jGCS are delivered to all channels. This is done as these messages represent special notifications on the group communication, most likely affecting the behavior of all process. For instance, the kernel process while receiving a block notification should temporarily stop sending messages and thus queuing them. Messages being delivered should also be put in a queue. Only after receiving a view message, it should restart its normal behavior.

This layer provides point-to-point communication too. This feature is used by the recovery process to exchanging information between recovering and recoverer replicas.

1.5 jGCS Layer

The jGCS layer provides a generic interface for Group Communication. This interface can be used by applications that need primitives from simple IP Multicast group communication to virtual synchrony or atomic broadcast. Its a common interface to several existing toolkits that provide different APIs.

jGCS implements also a new concept of providing a group communication service. Using the notion of inversion of control pattern, this service provides the separation of configuration and use. Provides also modularity, since applications use a common API that can be implemented using different solutions. The solution that will be used by an application is defined on configuration time.

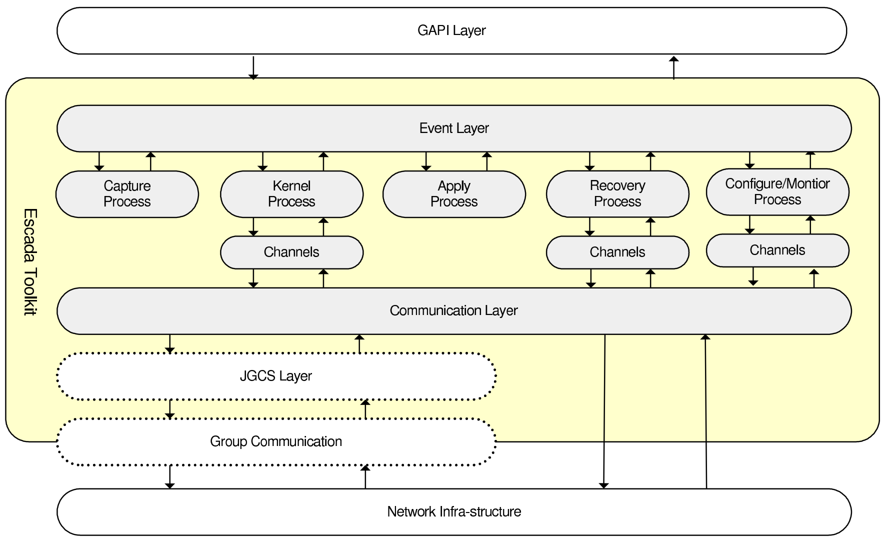

Figure 1.1: ESCADA Toolkit Architecture

Chapter 2

Protocol Implementation

This section describes how the Escada Toolkit provides certification-based and conservatively synchronized replication protocols. As an introduction, we start by showing also how simple state-machine and primary-backup are implemented.

2.1 Primary-Backup

Overview In the primary-backup approach to replication, also called passive replication [13], update transactions are executed at a single master site under the control of local concurrency control mechanisms. Updates are then captured and propagated to other sites. Asynchronous primary-backup is the standard replication in most DBMSs and third-party offers. An example is the Slony-I package for PostgreSQL [19].

Implementations of the primary-backup approach differ whether propagation occurs synchronously within the boundaries of the transaction or, most likely, is deferred and done asynchronously. The later provides optimum performance when synchronous update is not required, as multiple updates can be batched and sent in the background. It also tolerates extended periods of disconnected operation.

The main advantage of this approach is that it can easily cope with non-deterministic servers. A major drawback is that all updates are centralized at the primary and little scalability is gained, even if read-only transactions may execute at the backups. It can only be extended to multi-master by partitioning data or defining reconciliation rules for conflicting updates.

Reflector Components Used Synchronous primary-backup replication requires the component that reflects the Transaction context to capture the instant where the transaction starts executing, commits, or rollbacks at the primary. It also needs the object set provided by the Execution stage to extract the write set of a transaction from the primary and to insert it at the backup replicas.

Events The Escada Toolkit assemblies the Capture, Apply and Kernel process by sending commit and abort notifications that carry information on transactions such as ids and read and write sets. The main classes used to do this are: **CaptureKernel**, **ApplyKernel** and **AsyncPbKernel** or **SyncPbKernel** for asynchronous or synchronous primary backup replication, respectively.

Process Execution The execution of a primary-backup replicator is depicted in Figure 2.1. We start by describing the synchronous variant. It consists of the following steps:

Step 1: Clients send their requests to the primary replica.

Step 2: When a transaction begins, the Capture at the primary is notified, registers information about this event, and allows the primary replica to proceed.

Step 3: Right after processing a SQL command the DBMS server notifies the Capture through the Execution stage component sending an *ObjectSet*. For optimization, one might store the write set in an in-memory structure inside the DBMS server, gathering all updates in the context of a transaction and transmit them to the Capture process when the commit is requested.

Step 4: When a transaction is ready to commit, the Transaction Context component notifies the Capture process which notifies the Kernel and then the gathered updates are atomically broadcasted to all backup replicas (this broadcast should be *uniform* [6]).

Step 5: The write set is delivered at all replicas whereby the Kernel notifies the Apply process. On the primary, the Apply allows the transaction to commit. On the backups, the it injects the changes into the DBMS.

Final Step: After the transaction execution, the primary replies to the client.

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

2.2 State-machine

Overview The state-machine approach, also called active replication [13], is a decentralized replication technique. Consistency is achieved by starting all replicas with the same initial state and, subsequently, receiving and processing the same exact sequence of client requests. Examples of this approach are provided by the Sequoia [5, 18] and PGCluster [16] middleware packages.

The main advantage of this approach is its simplicity and failure transparency, since if a replica fails the requests are still processed by the others. It also trivially handles Data Definition Language (DDL) statements without any special requirements.

On the other hand, the state machine operates correctly only under the assumption that requests are processed in a deterministic way, i.e., when provided with the same sequence of requests, replicas produce the same sequence of output and have the same final state. To start with, this requires that the original SQL command is rewritten to remove non-deterministic expressions and functions such as *now()*.

A second source of non-determinism is the scheduling of concurrently executing conflicting transactions, namely, the order by which locks are acquired is hard to predict. To overcome this problem, it is common to have an external global scheduler that manages which SQL commands can be concurrently processed without undermining the determinism requirement. This introduces additional complexity and may overly restrict concurrency in update-intensive workloads.

The state-machine protocol requires that all replicas receive and process the same sequence of client requests producing a deterministic outcome. To accomplish this, we need to intercept client requests before they are processed enforcing deterministic executions. Specifically, begin, commit and rollback commands, implicitly or explicitly sent, and every SQL command should be intercepted. One possible solution is depicted in Figure 2.2.

Reflector Components Used State-machine replication requires the use of the Transaction context component and Parsing Stage component. On one hand, the transaction component is used to capture the moment where the transaction starts to execute, commits, or rollbacks at one replica. On the other hand, the Parsing Stage component is used to capture and start the execution of transaction statements.

Events The Escada Toolkit assemblies the Capture, Apply and Kernel process by sending begin, execution, commit and abort notifications that carry information on transactions such as ids and statements. The main classes used to do this are: **CaptureKernel**, **ActiveKernel** and **ApplyStatementKernel**, where the class **ApplyStatementKernel** is currently under development.

Process Execution The execution of a state-machine replicator is depicted in Figure 2.2. It consists of the following steps:

Step 1: Clients send their requests to one of the replicas. This replica is called the *delegate* replica.

Step 2: Using the Transaction component the Capture process at the delegate replica is notified of the beginning of the transaction. Then the Capture process notifies the Kernel process which uses a totally ordered broadcast to propagate this notification to all other replicas.

Step 3: All replicators deliver the notification in the same order. The transaction is started in the remote replicas and resumed in the delegate replica by the Apply process.

Step 4: The transaction is executed at the delegate replica. Every time a new command starts the Capture process is notified through the Parsing Stage component of the DBMS server. Then the Capture process notifies the Kernel process which uses a totally ordered broadcast to propagate this notification to all other replicas.

Step 5: The parsed statement is delivered at all Kernel process which notifies the Apply process. The Apply process must implement a deterministic scheduler: each Apply process must ensure that no two concurrent conflicting parsed statements are handled to the underlying DBMS. If such conflict exists, the parsed-statement is kept on hold. Otherwise it is handled to the DBMS at all replicas. It is worth noting two aspects related to this strategy. First, with this approach deadlocks may happen and the replicator should resolve them. Second, should a statement be used instead of a parsed statement, then the replicator would also need to parse it in order to extract information on the tables.

Further steps: Steps 4 and 5 above are repeated.

Step 6: Using the Transaction context component the Capture process at the delegate replica is notified when the transaction is about to commit or rollback. This notification is sent to the Kernel process which broadcasts in total order to all replicas.

Step 7: Upon receiving a commit or rollback notification, remote replicas notify the Apply process which executes the proper command and the delegate replica allows it to proceed.

Final step: Once the processing is completed, the delegate replica replies to the client.

2.3 Certification Based

Overview Certification based approaches operate by letting transactions execute optimistically in a single replica and, at commit time, run a coordinated certification procedure to enforce global consistency. Typically, global coordination is achieved with the help of a total order broadcast communication primitive, that establishes a global total order among concurrent transactions [10, 15, 12, 20].

Multiple variants of the certification based approach have been proposed. Here we briefly describe an approach providing snapshot-isolation [12, 20]. At the time a transaction is initiated, a replica is chosen to execute the transaction (usually, the closest replica to the client which is called the delegate replica). When a transaction intends to commit, its identification, database version read, and the write set are broadcast to all replicas in total order. Right after being delivered, all replicas verify if the received transaction has the same version as the database. If so, it should commit. Otherwise, one needs to check if previously committed transactions do not conflict with it. There is no conflict if previously committed transactions have not updated the same items. If a conflict is detected, the transaction is aborted. Otherwise, it is committed. Since this procedure is deterministic and all replicas, including the delegate replica, receive transactions by the same order, all replicas reach the same decision about the outcome of the transaction. The delegate replica can now inform the client application about the final outcome of the transaction.

This can be extended to serializability by considering also the read-set and then detecting read-write conflicts during certification [10, 15]. Although this might have some impact in performance [9], it is desirable for DBMS in which the consistency criterion is similar. Note also that certification based approaches do not require the entire database operation to be deterministic: Only the certification phase has to be processed in a deterministic manner. Furthermore, they allow different update transactions to be executed concurrently in different replicas. If the number of conflicts is relatively small, certification based approaches can provide both fault-tolerance and scalability.

Certification based approaches operate by letting a transaction to execute optimistically in a single replica and, at commit time, execute a coordinated certification procedure to enforce global consistency.

Reflector Components Used Given its similarity to the Primary-Backup approach, the Certification based replication requires the use of the same components, explicitly the Transaction context and Parsing Stage components.

Events The Escada Toolkit assembles the Capture, Apply and Kernel process by sending commit and abort notifications that carry information on transactions such as ids and read and write sets. The main classes used to do this are: **CaptureKernel**, **ApplyKernel** and **DbmsKernel**. In order to test serializability, the class **SerializabiltyDbmsKernel** was created but we intend to integrate both classes the **DbmsKernel** and **SerializabilityDbmsKernel** in a near feature.

Process Execution The execution of a certification-based replicator is depicted in Figure 2.3. It consists of the following steps:

Step 1-4: Same as in the Primary-Backup solution presented before.

Step 5: Upon receiving the write set, each replica certifies the transaction and decides its outcome: commit or abort. The Kernel process notifies the Apply process its decision. If it is an abort, the delegate replica through the transaction context component cancels the commit and the remote replicas discard it. If it is a commit, the delegate replica allows it to continue and the remote replicas inject updates into the DBMS.

Final Step: The delegate replica returns the response to the client.

2.4 Conservative Replication

Overview In the conservative approach, data is a priori partitioned in conflict classes, not necessarily disjoint. Each transaction has an associated set of conflict classes (the data partitions it accesses) which are assumed to be known in advance. While the conflict classes for a transaction could be determined at runtime, this would require to know the whole transaction before its execution precluding the processing of interactive transactions.

When a transaction is submitted, its id and conflict classes are atomically multicast to all replicas obtaining a total order position. Each replica has a queue associated with each conflict class and, once delivered, a transaction is classified according to its conflict classes and enqueued in all corresponding queues. As soon as a transaction reaches the head of all of its conflict class queues it is executed. Transactions are executed by the replica to which they are submitted.[1]

Conflicting transactions are executed sequentially. Clearly, the conflict classes have a direct impact on the performance. The lesser the number of transactions with overlapping conflict classes, the better the interleave among transactions. Conflict classes are usually defined at the table level but can have a finer grain at the expense of a non-trivial validation process to ensure that a transaction does not access conflict classes that were not previously specified.

When the commit request is received, the outcome of the transaction is reliably multicast to all replicas along with the replica's state changes and a reply is sent to the client. Each replica applies the remote transaction's updates with the parallelism allowed by the initially established total order of the transaction.

It is worth noting that, despite the use of a multi-version database engine, since conflicting transactions are totally ordered and executed sequentially, the protocol ensures 1-copy-serializability as long as transactions are correctly classified by the application. Relaxing the correctness criterion to snapshot-isolation would simple require the reclassification of the transactions by the application.

Reflector Components Used Given its similarity to the Primary-Backup approach and the Certification based replication, it requires the use of the same components, explicitly the Transaction context and Execution Stage components.

Events The Escada Toolkit assemblies the Capture, Apply and Kernel process by sending begin, commit and abort notifications that carry information on transactions such as ids, read and write sets. The classes used to do so are: **CaptureKernel**, **ApplyKernel** and **ConservativeKernel** which is currently under development.

Process Execution The execution of a primary-backup replicator is depicted in Figure 2.4. We start by describing the synchronous variant. It consists of the

[1]When isolated conflict classes exist, dedicating a distinguished replica to the execution of all transactions of such classes, results in a faster processing of those transactions [14].

following steps:

Step 1: Clients send their requests to the delegate replica.

Step 2: When a transaction begins, a begin trigger is fired and a notification is sent to the replicator at the primary, which registers information about this event. Right after, an uniform atomic broadcast is sent and upon being delivered the transaction is classified according to its conflict classes. At the delegate replica, when the transaction gets its turn to execute (i.e. when the transaction is at the head of all its conflict classes) the replicator allows it to continue. At remote replicas, nothing is done.

Step 3: Similar to the Primary-Backup.

Step 4: When a transaction is ready to commit, the Capture process notifies the Kernel process which reliably broadcasts the gathered updates to all backup replicas (this broadcast should be *uniform* [6]).

Step 5: The write set is delivered at all replicas through the Kernel process which notifies the Apply. On the delegate, the Apply process allows the transaction to commit. On the others, the Apply injects the changes into the DBMS.

Final Step: After the transaction execution, the primary replies to the client.

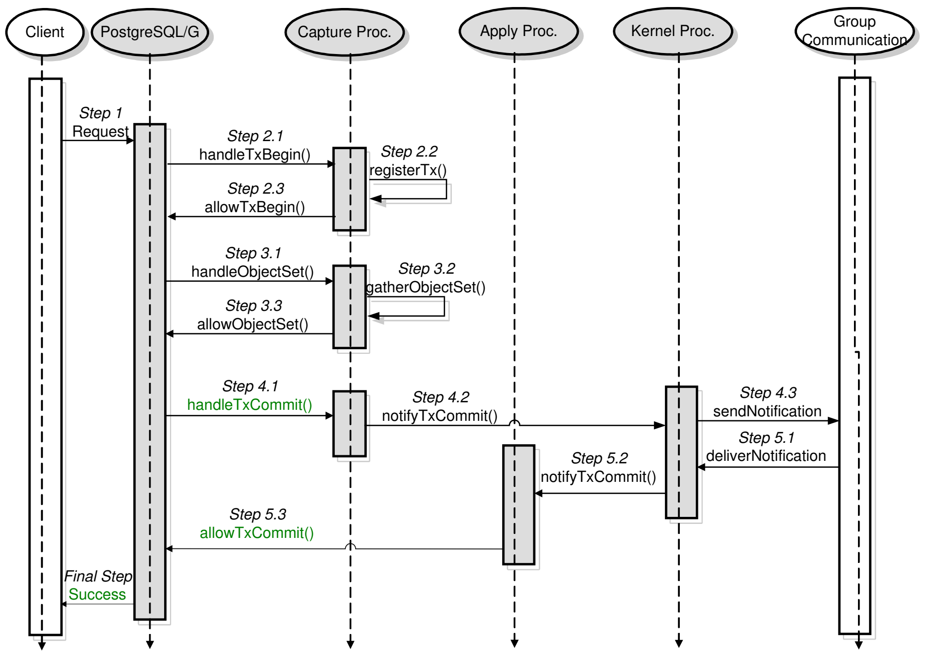

Figure 2.1: Primary-backup replication.

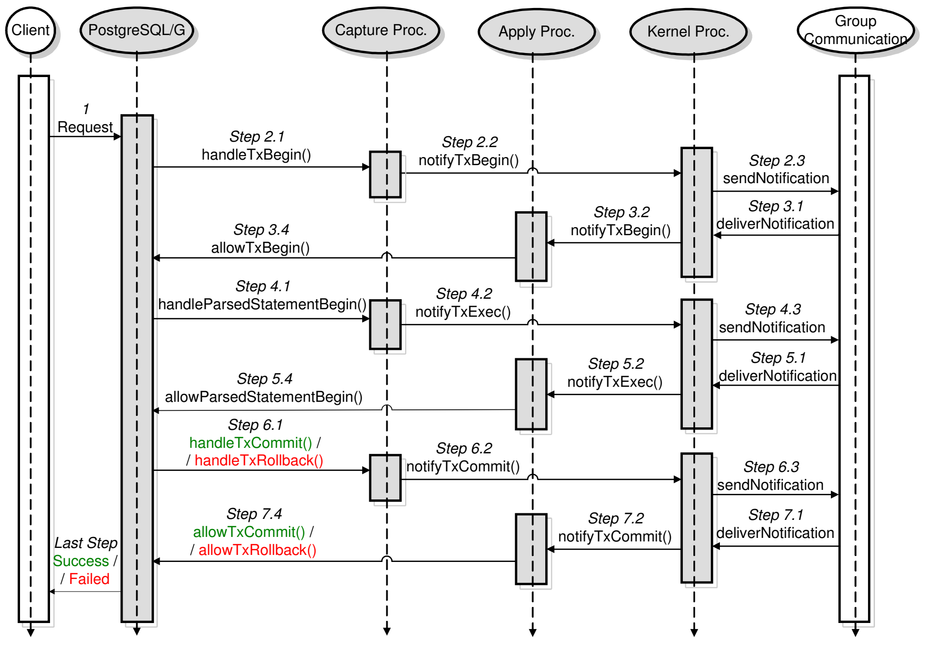
Client
PostgreSQL/G
Capture Proc.
Apply Proc.
Kernel Proc.
Group Communication
1 Request
Step 2.1 handleTxBegin()
Step 2.2 notifyTxBegin()
Step 2.3 sendNotification
Step 3.1 deliverNotification
Step 3.2 notifyTxBegin()
Step 3.4 allowTxBegin()
Step 4.1 handleParsedStatementBegin()
Step 4.2 notifyTxExec()
Step 4.3 sendNotification
Step 5.1 deliverNotification
Step 5.2 notifyTxExec()
Step 5.4 allowParsedStatementBegin()
Step 6.1 handleTxCommit() / / handleTxRollback()
Step 6.2 notifyTxCommit()
Step 6.3 sendNotification
Step 7.1 deliverNotification
Step 7.2 notifyTxCommit()
Step 7.4 allowTxCommit() / / allowTxRollback()
Last Step Success / / Failed

Figure 2.2: State-machine replication.

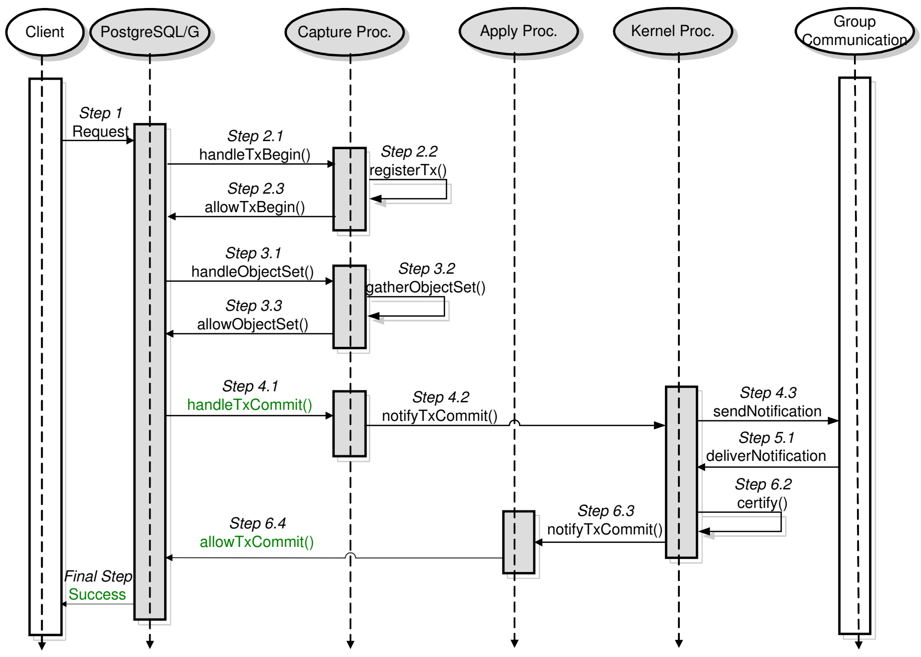

Figure 2.3: Certification-based replication.

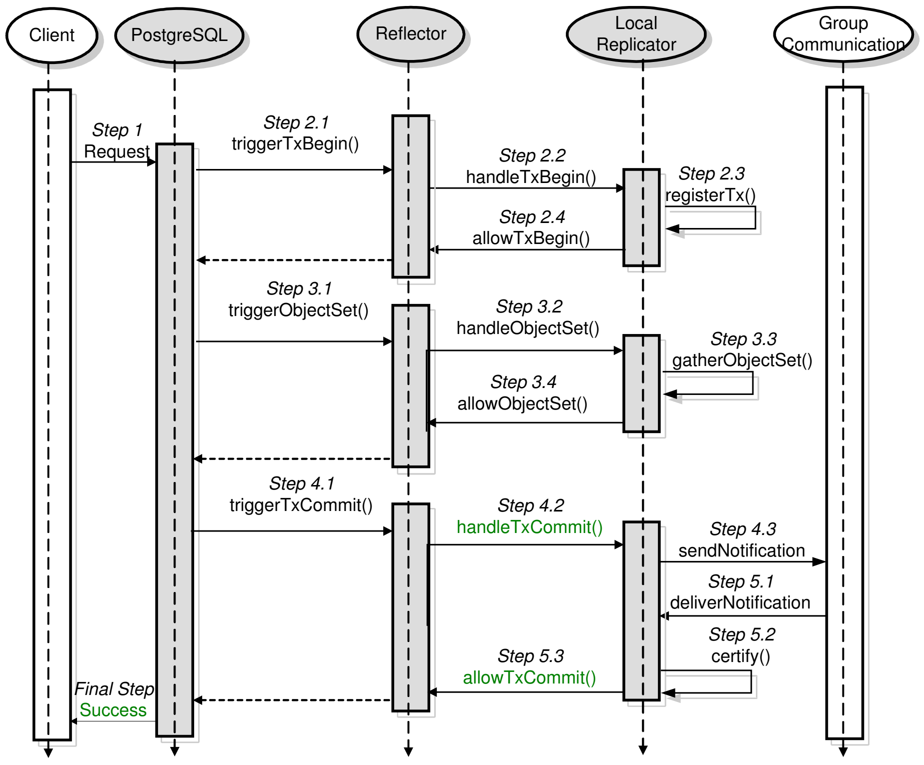
Client
PostgreSQL
Reflector
Local Replicator
Group Communication
Step 1 Request
Step 2.1 triggerTxBegin()
Step 2.2 handleTxBegin()
Step 2.3 registerTx()
Step 2.4 allowTxBegin()
Step 3.1 triggerObjectSet()
Step 3.2 handleObjectSet()
Step 3.3 gatherObjectSet()
Step 3.4 allowObjectSet()
Step 4.1 triggerTxCommit()
Step 4.2 handleTxCommit()
Step 4.3 sendNotification
Step 5.1 deliverNotification
Step 5.2 certify()
Step 5.3 allowTxCommit()
Final Step Success

Figure 2.4: Conservative replication.

Chapter 3

Configuration

This chapter shows how the Escada Toolkit should be configured to implement different replication protocols. To configure the replication engine, we should edit an XML file that specifies information on which databases are replicated, on group communication configuration and on processes. The configuration file have an XML tag to configure each component presented before. In what follows, we shall present details in order to enable the DBSM replication protocol.

3.1 Replica Information

The following excerpt specifies information on a database engine and its databases.

```
<replica>
   <id>0</id>
   <class>gorda.reflector.postgresql.PostgreSQLFactory</class>
   <ccriteria>ww</ccriteria>
   <params>
      <binPath>/usr/local/pgsql/bin/</binPath>
   </params>
   <database>
      <id>0</id>
      <params>
         <pool>10</pool>
         <uri>jdbc:PostgreSQLG://127.0.0.1:5432/tpcc</uri>
         <user>repmanager</user>
         <password></password>
         <binary>true</binary>
      </params>
   </database>
</replica>
```

Each replica should have a unique identification that is assigned through the tag “id”. Note that it is not necessary to have all replicas defined in the configuration file, although this is recommend for documentation purpose. The tag “class” defines a factory that is used to start up the database engine but for some database engines such as PostgreSQL, the factory is only responsible for initiating a rendering of the GAPI as the database engine is started aside. Tags “binPath” and “binary” are used for recovery purposes, defining, respectively, the path for binary tools and the format to be used for data transfer. Tag “ccriteria” defines if read operations should be used to enforce a strong consistency criterion such as Serializability[1].

3.2 Communication Infra-structure

The following excerpt shows the configuration that defines the communication infra-structure that will be used by the DBSM:

```
<jgcs>
   <protocol>
      <class>net.sf.jgcs.appia.AppiaProtocolFactory</class>
      <param></param>
   </protocol>
   <group>
      <class>net.sf.jgcs.appia.AppiaGroup</class>
      <param>/jgcsappia.properties</param>
   </group>
   <services>
      <class>net.sf.jgcs.appia.AppiaService</class>
      <optimistic>seto_total_order</optimistic>
      <regular>regular_total_order</regular>
      <uniform>uniform_total_order</uniform>
   </services>
   <channel>
      <id>0</id>
      <reference>total-distribution</reference>
      <service>vsc+total+services</service>
   </channel>
   <channel>
      <id>1</id>
      <reference>reliable-distribution</reference>
      <service>vsc+total+services</service>
   </channel>
   <channel>
```

[1]It is only available snapshot isolation level among replicas.

```
      <id>2</id>
      <reference>total-recovery</reference>
      <service>vsc+total+services</service>
   </channel>
</jgcs>
```

The tag “protocol” specifies a factory that serves as an interface entry point and triggers the initialization of runtime instances. The “group” encapsulates the address that can be used to open a communication channel that subsequently allows messages to be sent or received, or the membership to be observed. The “service” encapsulates a specification of the possible guarantees to be enforced on a channel. [2] Then, it is possible to define different channels that provide different service qualities. It is worth noticing that a message sent by a channel is delivered by the same channel in the sender process and its remote counterparts. The “id” and the “reference” should be unique. The former is used internally by the communication infra-structure while the latter is used by processes to get a reference to a channel.

3.3 Process configuration

The other elements of the files “replication*.xml” configure processes.

```
<process>
   <id>capture-process</id>
   <class>escada.replicator.process.capture
   .CaptureProcess</class>
   <start>true</start>
   <params>
      <recovery>true</recovery>
   </params>
   <notifier>
      <class>escada.interfaces.process.notifiers
      .ComponentTransactionNotifier</class>
      <notify>
         <url>escada.interfaces.process.notifiers</url>
         <pipeline>Transaction</pipeline>
      </notify>
   </notifier>
   <listener>
      <class>escada.replicator.process
      .capture.CaptureKernel</class>
      <listen>
```

[2] Please, do not try to change this information for now. You are working with the developers of the JGCS and APPIA to improve this configuration and easy its use.

```
         <url>escada.interfaces.process.listeners</url>
         <pipeline>Transaction</pipeline>
      </listen>
      <listen>
         <url>escada.interfaces.process.listeners</url>
         <pipeline>ObjectSet</pipeline>
      </listen>
      <listen>
         <url>escada.interfaces.process.listeners</url>
         <pipeline>Recovery</pipeline>
      </listen>
   </listener>
   <accept>
      <notifications>
         <from>replica</from>
         <id>0</id>
         <pipeline>Transaction</pipeline>
         <url>gorda.reflector.interfaces.context
         .transaction</url>
         <event wait = "false">Begin</event>
         <event wait = "true">Finish</event>
         <processor>gorda.reflector.postgresql
         .PostgreSQLTransactionProcessor</processor>
      </notifications>
      <notifications>
         <from>process</from>
         <id>recovery-process</id>
         <pipeline>Recovery</pipeline>
         <url>escada.interfaces.process.listeners</url>
         <event>StartRecovery</event>
         <event>StopRecovery</event>
      </notifications>
   </accept>
```

For each process, one should define generic information such as an “id”, which “class” implements it, parameters (“params”) and if it should be started in a new thread (“start”). Then, one should specify information on events. Specifically, which components in a process are responsible for notifying events and listening to them. Furthermore, it is required to define which events a process should receive. The events are organized in pipelines and it is possible to define a different “listener” or a different “notifier” for different pipelines. It is important to note that a “url” defines a Java path to a “pipeline” that defines an event. Furthermore, a process should register into other processes or into a database to receive their notifications and this should be done per pipeline. In this phase, it is possible to specify which are the events that a process is interested in. In particular for events from a database, it is possible to define if a database should wait for a confirmation from a process in order to proceed with its execution or not ("wait").

The tag "recovery" is used by any process to determine if it is necessary to the recovery or not. For instance, this should be used for the first database as there is no other database from which it might retrieve its state. Of course, this might be used by an administrator to avoid on-line recovery when it knows for sure that replicas are synchronized. In other words, if one wants to do recovery define it as true, otherwise, as false.

Some process have specific parameters that are described as follows:

- **Capture process** If a primary backup replication is used, one should define this flag in slave replicas as true.

  ```
  <params>
     <recovery>true</recovery>
     <slave>false</slave>
  </params>
  ```

- **Distribution process** The recovery parameter define if a process recovery is defined in this configuration.

  ```
  <params>
    <recovery>true</recovery>
  </params>
  ```

- **Apply process**

  The tag "GroupRemoteCommit" defines the size of the batch of remote transactions. The tag "GroupLocalCommit" defines the size of the batch of local transactions . The tag "GroupRollback" defines the size of the batch of transactions to do rollback . The tag "ConcurrentRemoteCommit" defines the number of concurrent threads applying updates of remote transactions.. The tag "ConcurrentLocalCommit" defines the number of concurrent threads applying updates of local transaction . The tag " ConcurrentRollback" defines the number of concurrent threads doing rollback of transactions.

  ```
  <params>
     <GroupRemoteCommit>6</GroupRemoteCommit>
     <GroupLocalCommit>6</GroupLocalCommit>
     <GroupRollback>6</GroupRollback>
     <ConcurrentRemoteCommit>6</ConcurrentRemoteCommit>
     <ConcurrentLocalCommit>1</ConcurrentLocalCommit>
     <ConcurrentRollback>1</ConcurrentRollback>
  </params>
  ```

- **Recovery process** The tag “port” identifies which port is used to transfer information among replicas during recovery and the tag “peers” defines how may concurrent replicas should be used during a recovering.

  ```
  <params>
     <recovery>true</recovery>
     <port>5000</port>
     <peers>3</peers>
  </params>
  ```

Chapter 4

Issues and Future Work

Our current prototype does not provide the following features:

Replicating meta information Our current prototype does not handle replication of meta information. Thus, if one wants to put a new replica on-line, it should be configured as described in [3].

Grouping local commit Please do not group local commit as this feature might introduce inconsistency into the database.

Serializing transactions from different databases Our current prototype is serializing transactions from different databases. We are currently working with the jGCS developers in order to fix this.

Bibliography

[1] Gorda Project. http://gorda.di.uminho.pt/community.

[2] Appia. http://appia.di.fc.ul.pt/wiki, 2006.

[3] HOWTO PostgreSQL/G. http://gorda.di.uminho.pt/community/pgsqlg/, 2006.

[4] JGCS. http://jgcs.sourceforge.net/, 2006.

[5] E. Cecchet, J. Marguerite, and W. Zwaenepoel. C-JDBC:Flexible Database Clustering Middleware. In *USENIX Annual Technical Conference*, 2004.

[6] G. Chockler, I. Keidar, and R. Vitenberg. Group Communication Specifications: A comprehensive Study. *ACM CSUR*, 2001.

[7] A. Correia, J. Orlando, L. Rodrigues, N. Carvalho, R. Oliveira, and S. Guedes. GORDA: An Open Architecture for Database Replication. Technical report, University of Minho and University of Lisbon, 2006.

[8] Ralph E. Johnson. Frameworks = (components + patterns). *Commun. ACM*, 40:39–42, 1997.

[9] A. Correia Jr., A. Sousa, L. Soares, J. Pereira, F. Moura, and R. Oliveira. Group-based Replication of On-line Transaction Processing Servers. In *2nd Latin-American Symposium on Dependable Computing*, 2005.

[10] B. Kemme and G. Alonso. Don't Be Lazy, Be Consistent: Postgres-R, A New Way to Implement Database Replication. In *VLDB Conference*, 2000.

[11] B. Kemme, A. Bartoli, and O. Babaoglu. Online Reconfiguration in Replicated Databases Based on Group Communication. In *IEEE DSN*, 2001.

[12] Y. Lin, B. Kemme, R. Jiménez Peris, and M. Patiño Martíne. Middleware based Data Replication providing Snapshot Isolation. In *ACM SIGMOD*, 2005.

[13] Sape Mullender. Distributed Systems. In *Distributed Systems*. ACM Press, 1989.

[14] M. Patino-Martinez, R. Jimenez-Peris, B. Kemme, and G. Alonso. Scalable Replication in Database Clusters. In *DISC'00: Proceedings of the 14th International Conference on Distributed Computing*, 2000.

[15] F. Pedone, R. Guerraoui, and A. Schiper. The Database State Machine Approach. In *Journal of Distributed and Parallel Databases and Technology*, 2003.

[16] PGCluster. http://pgcluster.projects.postgresql.org/.

[17] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. Wiley & Sons, 2000.

[18] Sequoia. https://forge.continuent.org/.

[19] Slony. http://slony.info.

[20] S. Wu and B. Kemme. Postgres-R(SI): Combining Replica Control with Concurrency Control based on Snapshot Isolation. In *IEEE ICDE*, 2005.