Project no. 004758

GORDA

Open Replication Of Databases

Specific Targeted Research Project

Software and Services

Modules description and Configuration Guide

GORDA Deliverable D3.4

Due date of deliverable: 2007/03/31
Actual submission date: 2007/09/29

Start date of project: 1 October 2004 Duration: 42 Months

UNISI

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)	

Contributors

Nuno Carvalho, U. Lisbon

Alfrânio Correia, U. Minho

Rui Oliveira, U. Minho

Fernando Pedone, U. Lugano

José Pereira, U. Minho

Luís Rodrigues, U. Lisbon

Luís Soares, U. Minho

Vaide Zuikeviciute, U. Lugano

Abstract

The purpose of this document is to show the necessary steps required to build, configure and deploy the ESCADA toolkit, a modular reference implementation of replication protocols built on the GORDA Architecture and Programming Interfaces. Since ESCADA relies on group communication for propagating information among replicas, a significant part of this document is devoted to jGCS, a generic interface for group communication, and its default implementation, the Appia toolkit.

Contents

Chapter 1

Introduction

The ESCADA toolkit is a modular reference implementation of replication protocols built on the GORDA Architecture and Programming Interfaces. It can be configured to provide both optimistically as well as conservatively synchronized protocols with different consistency criteria and exploiting optimizations for different environments. ESCADA depends on a group communication service (jGCS) [8] to propagate information among replicas. The default implementation of jGCS is provided by the Appia group communication toolkit [4].

Hereafter we describe the aforementioned modules: in Chapter 2 we outline the components of the group communication and ESCADA toolkits; and Chapter 3 provides their configuration details.

1.1 Objectives

The goals of this document are as follows:

- To provide a concise description of ESCADA toolkit and the underlying group communication service;
- To present a complete configuration guide.

1.2 Relationship With Other Deliverables

This deliverable describes and shows how to configure the work specified in D2.3 (APIs Definition Report), and reported in D3.3 (Replication Modules Reference Implementation) and D3.5 (Group Communication Protocols Report).

Chapter 2

Modules description

2.1 Group Communication

Group communication protocols support message passing within groups of processes by offering membership management and reliable multicast services. Membership management keeps track of which processes are operational and mutually reachable taking into account both voluntary requests to join and leave the group, as well as process failures and network partitions. By ensuring that a common membership is observed by all participants, many distributed algorithms are simplified. Message ordering simplifies application programming by ensuring that each message is handled in a predictable context resulting from previous messages. The definition of primary views helps the recovery processes and avoids the divergence of database replicas state.

To ensure all these properties the group communication service is composed of jGCS [8] and Appia [4]. jGCS is a set of interfaces that can be implemented by many group communication toolkits and provides a common service for the replication protocols. The default implementation uses the Appia toolkit. Appia provides the group communication protocols (e.g., reliable multicast, total ordering of messages, primary partitions, membership management) that support various replication solutions.

2.1.1 Group Communication Service

This section briefly describes the Group Communication Service. This service complies to the specification described in the GORDA deliverable D2.3 (APIs Definition Report) and is detailed in [14].

The service is organized in four complementary interfaces, namely: the configuration interface, the common interface, the data interface, and the control interface.

The configuration interface decouples the application code from specific implementations by requiring that a third party, the configurator, matches available services with application requirements. Configuration objects should be easily stored and retrieved in configuration files and directory services.

A protocol session is represented by a *Protocol* instance, obtained from the configuration stored in a *ProtocolFactory*. Using a *Protocol* instance it is possible to obtain *data* and *control* sessions for a specific *GroupConfiguration*. All further operation are invoked through one of these two interfaces. Both data and control sessions identify group members using *java.net.SocketAddress* objects. This directly allows a large number of protocols to be supported without any form of address conversion. Protocols that use different address formats, can easily be wrapped.

The data interface provides the methods for messages to be sent and received. Whenever the application multicasts a message there is always a specific quality of service, i.e. a specific set of guarantees, associated with the request. The guarantees can be implicitly derived from the group or protocol configuration or explictly set using a *Service* parameter.

The control interface provides a flexible modular interface for a wide range of membership management protocols. The simplest interface is suitable only for best-effort multicast protocols: the *ControlSession* provides methods for entering and leaving a group, as well as for registering a listener for control events; the *ControlListener* provides a simple notification of members entering and leaving the group. This interface can be used separately for failure detection or for the management cluster infrastructures, which are not directly related to group communication. Notice also that implementations can choose to distinguish members that have left the group voluntarily in a controlled fashion, from members that have failed and thus have been forcibly excluded. View synchronous group communication can ensure the *sending view delivery* property at the expense of briefly blocking applications before installing a new view.

This service supports the following notions:

- Support for data and membership;
- Support for peer groups and multicast groups;

- Usable from IP Multicast to VSC. It can also be used for autonomous membership services;
- High concurrency/low latency/latency hiding: support for optimistic deliveries and semantic protocols;
- Container-managed concurrency: it is up to the application to decide the thread policy for receiving messages (blocking or concurrent non-blocking).

2.1.2 Appia toolkit

Appia [4] is an open source layered communication toolkit implemented in Java. It is composed of a core and a set of protocols that provide group communication. The toolkit addresses explicitly the issue of enforcing inter channel constraints in multi-channel applications. Most of the new concepts of Appia emerged from several composition problems raised in the context of building fault-tolerant applications [23]. It does so by allowing the same instance of a given protocol to be shared by multiple channels. Appia layers can be composed to define configurations; and Appia sessions can be composed to define channels. Configurations and channels are strictly vertical compositions of layers and sessions, respectively. Lattices are created by declaring sessions to be part of more than one channel. Both configurations and channels may be created in run-time. Configuration and channel specification are defined using XML.

The current Appia configuration used by the replication protocols is depicted in the Figure 2.1. Roughly, the ESCADA replication toolkit uses two communication channels: one channel with optimistic uniform total order delivery of messages and another channel that only provides FIFO order between the group members. The protocols related to group membership management are shared between the two channels. The protocol that provides total order only exists in one channel. Messages are sent to one channel or another depending on the jGCS service [14]. Each protocol provides a specific service.

We now briefly describe the most important layers that compose the communication channels and are used by the ESCADA replication toolkit.

Socket interface with OS. There are several protocols that can be used to interface Appia with the native OS sockets. Currently, Appia supports TCP, UDP and TCP with SSL. The remaining

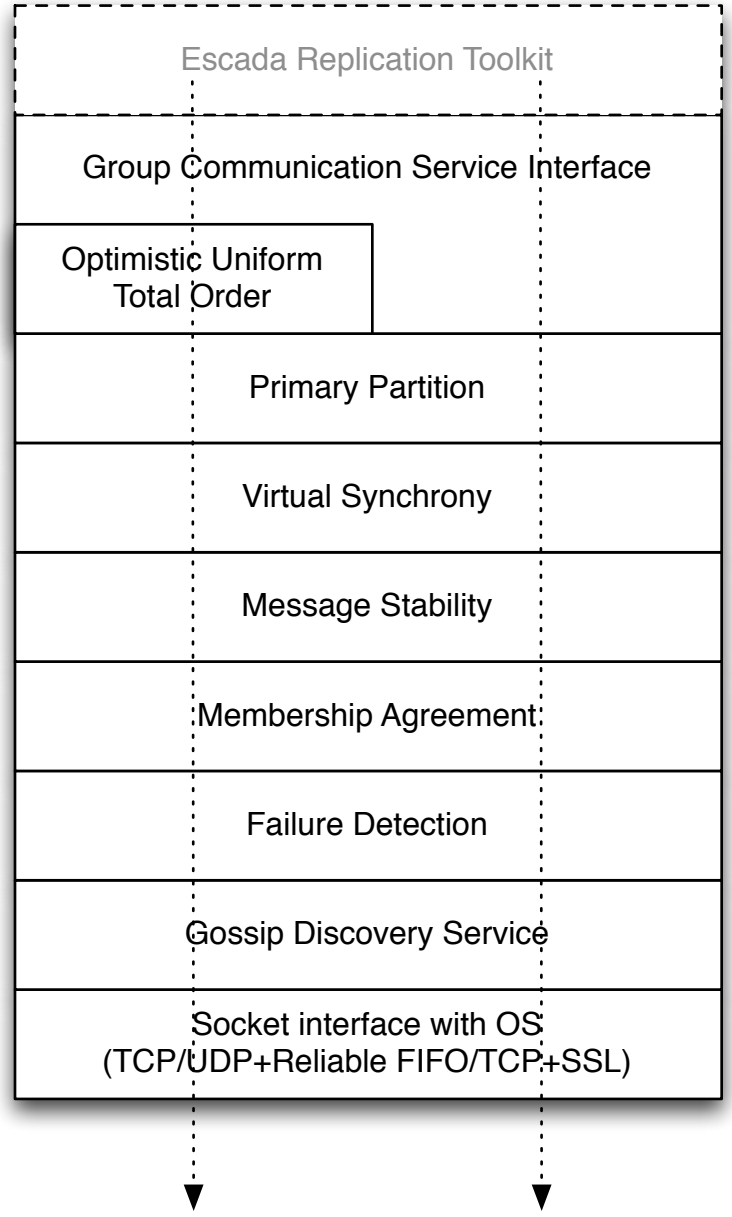

Figure 2.1: Appia configuration used by the ESCADA replication module.

group communication protocols need reliable FIFO channels, so if UDP is used, it must be complemented with a reliable FIFO protocol. This is also provided by Appia. UDP has the advantage of allowing the usage of IP Multicast, if it is available in the network. Having specific protocols that interface with sockets has the advantage of supporting a large number of networks and configurations.

Gossip Discovery Service. This service is responsible for discovering new members in the network. It can use IP Multicast or a dedicated daemon. When a new member starts, it tries to communicate using this service. Upon discovery of new members, this service notifies the Membership Agreement service to start a view change.

Failure Detection. This service constantly sends *alive* messages to other members and monitors the other group members. If a failure of another member is detected this service notifies the Membership Agreement service to start a view change.

Membership Agreement. This service runs an agreement process to define a new membership when a member is considered to have failed or when a concurrent view is detected. The order of the elements in a view is also agreed upon and is the same at all members. Upon defining a new view, this service notifies the above services and applications of the new view. Note that at this level of the protocol stack, the defined views are partitionable.

Message Stability. This service enforces message stability between all group members, allowing all members to see the same set of messages.

Virtual Synchrony. The Virtual Synchrony service provides View Synchrony. With the help of the message stability process, this protocol ensures that all members see the same set of messages in the same view.

Primary Partition. This service provides the definition of primary partitions on the top of a partitionable system. This is done by defining a primary view when the whole the system starts its life cycle and by making decisions on every view change about whether the next view is considered as primary or not. If the view is not considered as primary then it is not delivered to the application. In this case, the application remains blocked while waiting for a new view.

Optimistic Uniform Total Order. This service provides uniform total order of messages with early deliveries and is detailed in [25].

Group Communication Service Interface. This service is just the Appia implementation of jGCS, that allows to hide and wrap all the complexity of the channels defined by jGCS. The ESCADA replication toolkit only interacts with this layer.

2.2 ESCADA replicator

In this section we describe an architecture for generic database replication modeled as a set of components at the database and middleware levels. Features that should be provided by the database are obtained through the GORDA Application Program Interface (GAPI). The ESCADA toolkit enables to easily assemble different components which provide a concurrency control mechanism, a communication layer, and fault-tolerance and recovery strategies. We show that this architecture can efficiently and gracefully accommodate a broad set of replication protocols. In particular, we focus on group-based replication protocols.

Synchronous database replication based on group communication appears as a solution to circumvent the scalability and performance issues related to traditional database replication protocols based on distributed locking. We distinguish two categories of replication protocols based on group communication: conflict-class based and certification-based protocols. In the former, a transaction is annotated with conflict-classes, usually the relations referenced in the transaction's statements, multicast and totally ordered before its execution. Upon delivery the transaction atomically acquires locks at all replicas according to the conflict classes and executes at the replica that originally received its request. Upon commit the result is reliably propagated to the other replicas [15, 22]. In the latter category, a transaction starts executing optimistically at the replica that originally received its request. Upon commit its result is atomically propagated to the other replicas and a certification procedure ensures that only one of the conflicting concurrent transactions commits [21, 19].

Figure 2.2 depicts the architecture used by the ESCADA toolkit. The database engine implements the GAPI [16], which allows to transparently develop different replication protocols suited for different application semantics. The jGCS layer [8] enables the use of any group communication toolkit

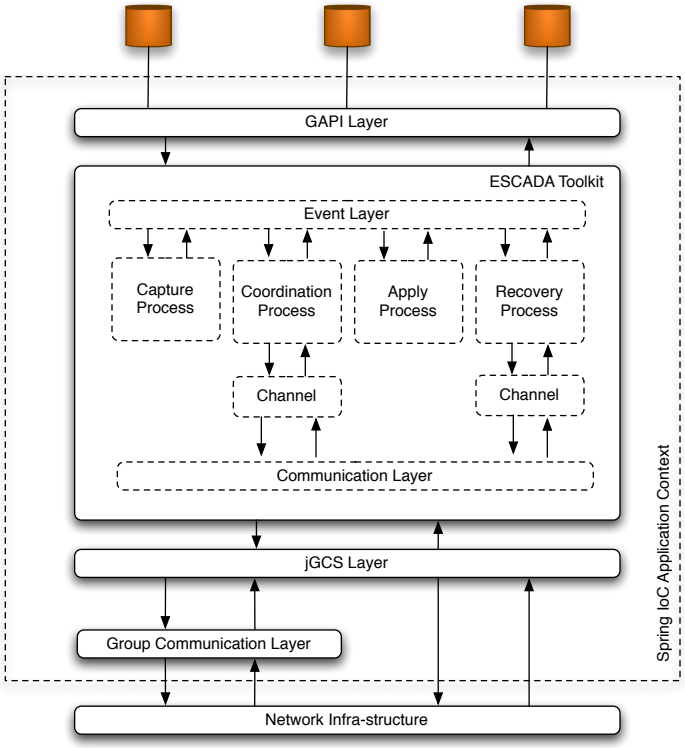

Figure 2.2: ESCADA Toolkit Architecture

providing another generic interface in which a replication protocol might rely on to synchronize information among replicas. The jGCS provides also the ability to use point-to-point communication directly accessing the features provided by the communication infra-structure such as the Transmission Control Protocol (TCP). The Group Communication layer [4] is responsible for propagating information among replicas. Group communication primitives such as reliable and atomic broadcast are used as building blocks even for replication protocols not originally based on group communication such as the 2 Phase Commit [24]. By doing this, we simplify the protocols since 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 by exchanging events [26]. This event-driven architecture allows a better software design that facilitates reuse, loose coupling, and easy testing of software components. In addition, we use the inversion of the control pattern to glue the layers together [18]. 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.

In the following 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 *http://gorda.di.uminho.pt/community*. Before proceeding recall that we shall focus on replication protocols based on group communication, avoiding cluttering the text with different details on different protocols. In particular, we shall focus on the Database State Machine (DBSM). It is worth noticing, however, that our architecture might be used to accommodate different replication protocols.

2.2.1 GAPI Layer

The GAPI defines five contexts: (*i*) database management system, (*ii*) database, (*iii*) connection, (*v*) transaction and (*iv*) request. The database management system (DBMS) identifies an instance of an application which might have different databases such as a human resource database or a marketing database. Databases allow clients to connect to, retrieve and change information. Clients are identified by a set of connection properties such as user identification, password and encoding information. For transactional systems, a set of interactions between clients and servers is managed 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 is 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. The capture process receives the GAPI notifications and forwards them as we describe further on.

2.2.2 Processes Layer

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

Capture Process. The capture process receives events from the GAPI, converts them to appropriate events in the ESCADA toolkit and notifies the other processes. In particular for the DBSM, 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 reads (if required by the consistency criteria). The capture process gathers such information and upon a commit request constructs a internal transaction event that carries the transaction identification along with the updates and reads. Then it notifies the kernel process that is responsible for implementing the concurrency control mechanism.

Kernel Process. The kernel process is usually known as the coordination process, but in the ESCADA toolkit, we name it differently as it also provides core functions regarding a replication protocol. In other words, the kernel process is the core of the ESCADA 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.

Apply Process. The apply process has two different strategies according to the transaction's outcome: abort or commit. When an abort is received at the initiator replica (i.e., the replica that received the client request), the transaction that was blocked is cancelled, and at a remote replica it is disregarded. When a commit is received, this processes 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 GAPI assumes the existence of a server-side JDBC API that should be used to inject operations in a database.

Recovery Process. Every time a replica joins a group, a blocking notification is sent by the jGCS indicating this fact. Hence, when a replica receives this notification, it should stop sending and receiving messages, holding them in 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 identifies the new replica. Right after this information, 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 be in a non-updated state. The recovery process in the joined replica chooses another replica as a database source and carries out a state transfer. In our current implementation, we use the "pg_dump" and "pg_restore" utilities provided by the PostgreSQL to do so. We have developed three different recovery protocols: (*i*) transfer of the entire database using some mechanism provided by the database engine; (*ii*) transfer of missed updates to joining replicas using middleware logging; (*iii*) transfer the last version of changed objects. Furthermore, the chosen replica needs not only to transfer the database state but also information stored in the buffers. This is required for running the certification test at the new replica and for applying transactions that are in the buffers and were not captured in the database state transfer.

Coordination Process. By using our current ESCADA implementation and PostgreSQL/G, different replication protocols might be easily developed. For instance, to develop a lazy replication protocol, one might store the updates and periodically send them to be applied. It is worth noticing that it is necessary to change the capture, coordination and apply components, although it is easy to make such changes and the components can be easily combined to implement the replication protocol. In detail, the capture process should store the updates in order to be periodically processed. The kernel process should read the updates stored by the capture process and propagate them to other replicas by using a reliable multicast primitive. Then the apply process would apply remote transactions in batches and ignore local transactions as they were already committed.

To develop a conflict-class-based protocol, one should annotate each transaction with its conflict classes. This might be easily done by calling a database function that would trigger the replication

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

protocol, acquiring locks according to the conflict classes and enabling transactions to execute. Upon commit the updates would be propagated by using a reliable multicast.

2.2.3 Event Layer

This layer is a set of components and interfaces that enable 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, through an event, to which notifier process *b* should be registered.

2.2.4 Communication Layer

This layer is built on the jGCS, augmenting it, and providing multiplexing and demultiplexing of messages. In other words, it enables 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 to channels with the same identification. Membership, block and view messages from the jGCS are delivered to all channels. These messages represent special notifications on group communication and most likely affect the behavior of all processes. For instance, the kernel process while receiving a block notification should temporarely stop sending messages and thus queue them. Messages being delivered should also be put in a queue. Only after receiving a view message, it should restart its normal behavior.

2.2.5 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. It is 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 separation of configuration and use. It also

provides modularity, since applications use a common API that can be implemented using different solutions. The solution to be used by an application is defined on configuration time.

2.2.6 Management Layer

The management layer provides runtime access to sensors and actuators on almost the entire functional software stack that comprises the ESCADA replicator. Sensors and actuators are exported using Sun's JMX technology and are categorized into three main classes: *system*, *group communication* and *ESCADA processes*.

The *System* class has sensors only. It places probes in the operating system and takes readings of network traffic, processor and memory usage. The *group communication* class has both, sensors and actuators. In this layer, sensors take readings on, for example, number of messages and bytes sent and received. Actuators, are used, for instance, to boot the group. Last, in the ESCADA layer one can also find both, sensors and actuators. Sensors monitor queue sizes, while actuators act flow control primitives (admission control, start, stop, resume and pause of transactions).

2.2.7 Spring: The Application Container

All the software components mentioned in the previous sections need to interact. In order to do this, ESCADA bootstrap is performed by having an Application Container start all the needed classes and hand over the necessary references to the objects instantiated. This pattern is most often referred to as "Dependency Injection" [17], which is a itself a particular from of the Inversion of Control [17] concept. In the Dependency Injection pattern the responsibility of object creation and object linking is transferred from objects into the component that creates them, most often a Factory object.

With this, objects become loosely coupled, whereas interfaces play an important role. Objects are linked together only by interfaces. At instantiation time the object factory, maps these interfaces into implementations and injects them into the right objects. In other words, control is transfered from objects into the factory. This is indeed a form of Inversion of Control, very easily implemented and very appropriate to ESCADA, which is built on top of layers of interfaces (e.g., GORDA API and jGCS).

The Dependency Injection container used was taken from the renowned framework named Spring.

ESCADA makes use of the lightweight Spring IoC module. The container provides an application context, containing a set of Spring beans declared in several XML files. Nonetheless, the Spring configuration files are quite static and should not need to be changed. They can even have parameters declared elsewhere, for instance in a much less obnoxious Java properties file, which get merged into the Spring beans definition when the application context gets created. Moreover, if the user/developer feels that the relations between the objects in the ESCADA framework need some changes, than he/she can alter the Spring beans configuration.

Chapter 3

Configuration Guide

3.1 Group Communication

The following packages were written in Java, thus, they require a java virtual machine installed in your computer. Both jGCS and Appia currently require Java 5. Installing/Upgrading Java is beyond the scope of this document (see `http://java.sun.com` for more information).

3.1.1 Group Communication Service

The package is available as open source and can be downloaded from `http://jgcs.sf.net`. To compile this distribution use the Apache Ant tool (see `http://ant.apache.org`) with the provided *build.xml file*. The targets are:

build compiles the source code and put bytecode files in the bin directory.

jar creates a jar file.

dist creates the dist directory with all jars needed to run the code.

clean Cleans bin and dist directories.

jGCS depends on the Log4j package that needs to be downloaded and placed in the lib directory before compilation. Log4j can be downloaded from `http://logging.apache.org/log4j/`.

To build the required libraries, just use the Ant tool with the *dist* target. The libraries will be binded with the ESCADA toolkit. This is just the service interface without any implementation. The

binding with implementations of jGCS can be done later at configuration time, using mechanisms such as Injection of Control (IoC) and tools such as the Spring framework. The default implementation for the GORDA project is Appia.

3.1.2 Appia toolkit

The Appia toolkit is also available as open source and can be downloaded from `http://appia.di.fc.ul.pt`.

There are two packages: source distribution and binary distribution. The source distribution contains the source code, the libraries that are needed by Appia, an Apache Ant file, the Readme file, the License file and the changes log. It includes also some script files to run applications. One of these applications is the gossip service, needed to form groups when multicast is not available on the network. The binary distribution contains the Appia jar file, the libraries that are needed by Appia, and the Readme, License and changes files. It also includes some script files to run applications.

Appia also runs as a library that needs to be linked with applications that use it. In the case of the GORDA architecture, Appia includes already in the current distribution an implementation of jGCS. It is enough to link both Appia and jGCS to the replication service and configure it properly.

If you choose to download the source distribution you will be able to compile Appia using the Apache Ant tool. The Apache Ant file has the following tags:

dist creates an Appia distribution, building the Appia jar file and placing all the needed jar files in the dist directory. Builds everything except the documentation;

build compiles the source code;

jar creates the Appia jar file;

clean removes all files generated by Ant;

doc compiles the java documentation from the source code.

To create all the needed jar files it is enough to run the Ant tool with the *dist* target. All the jar files will be placed in the *dist* directory.

3.1.3 Binding the Group Communication Service with the Appia toolkit

The Appia toolkit already includes an implementation of jGCS. To use Appia with jGCS one only needs both jar files included in the package and all its dependences. After having all the jar files properly added to the CLASSPATH environment, one needs two configuration files:

appia.xml. This file contains the configuration concerning the channels and protocols used by the ESCADA replication toolkit, as described in the Section 2.1.2;

services.properties. This file configures the existing services that will be provided by the Appia toolkit.

3.2 ESCADA replicator

To proceed, one should be familiar with the following tools and concepts:

- Build and deploy Java software using Maven (version 2) [13];
- Basic understanding of Java Technology [10].

We provide the information necessary to easily compile, install, and configure such packages and programs as long as it is required by the ESCADA toolkit.

The ESCADA Toolkit has the following compile-time dependencies:

- gapi [16];
- jgcs [8];
- log4j [9];
- jdom [7];
- junit [12];
- appia [4];
- spring [2]: beans, context and aop;

3.2.1 PostgreSQL/G, Derby/G or Sequoia/G

PostgreSQL/G, Derby/G or Sequoia/G should be installed before running the ESCADA replication engine. In this document, we assume that at least two different PostgreSQL clusters, Derby databases or Sequoia Controllers are configured (instructions on how to configure and setup these modules can be foun in the companion GORDA deliverable D4.6).

3.2.2 Downloading and Installing Components

The ESCADA toolkit is hosted at `http://escada.sourceforge.net` where both source and binary packages are available. The following instructions assume that the user is performing an installation from the source. When referring to the package version one will be using $VERSION for the rest of the document.

1. Download the source package.
   - wget http://dfn.dl.sourceforge.net/sourceforge/escada/escada-toolkit-$VERSION-src.tar.gz
2. Extract sources from the tarball archive:
   - tar xfvz escada-toolkit-$VERSION-src.tar.gz
3. Change into the resultant directory (escada-toolkit-$VERSION). For the sake of simplicity the escada-toolkit directory will be referred for the rest of the document as $ESCADASRC .
   - cd $ESCADASRC
4. In this directory one will find two sub-directories: starter and toolkit. You will need to compile and install the toolkit first. This is done using maven to build and install the resulting library
   - mvn install
5. Next, change to the starter directory, build the package using maven and create a final package comprising all the dependencies needed to start a ESCADA replicator.
   - cd ../starter

- mvn install assembly:assembly

At this point, under the $ESCADASRC/starter/target one should find several tarballs holding the necessary libraries to launch ESCADA for database backends. The resultant packages are named:

- escada-starter-$VERSION-$DBBACKEND-bin.tar.gz.

$DBBACKEND stands for the database backend and is one of: *i*) PostgreSQL; *ii*) Derby; or *iii*) Sequoia.

If the user chooses to alter source files, he must install the package in which he changed the source and either replace the jar in the resultant assembled package (-bin.tar.gz) or redo the assembly in the starter directory by entering in the command line: mvn assembly:assembly .

In what follows, we assume that the ESCADA toolkit is uncompressed in a directory represented by the shell variable $ESCADA.

3.2.3 Components

The ESCADA toolkit is is built upon the Gorda Programming Interface (GAPI) and thus requires either the PostgreSQL/G [6], Derby/G [5] or Sequoia/G [11]. It has two main components: a set of processes that enables building different replication protocols and a communication infra-structure.

3.2.3.1 Processes

Processes are divided into four major categories: *i*) capture; *ii*) coordination; *iii*) apply; *iv*) and recovery. Together, four instances of these (one of each), builds up the replication protocol.

The capture process receives notifications from a database, converts such notifications in a format known by other processes and notifies them. Communication among a capture process and a database occurs through the GAPI thus enabling the reuse of the capture process. Generally speaking, the GAPI is a common interface that reflects abstract transaction concepts as objects in Java. For instance, when a transaction begins, items are written or when a commit is requested different notifications are sent.

The coordination process aims at propagating data collected by the capture process to all replicas by using a communication infra-structure. Besides being the entry point for remote replicas, it also implements features required by different replication protocols. For instance, in the Database

State Machine [20], it decides a transaction's outcome: abort or commit. Whenever the coordination processes assembles the required information for one given transaction, it notifies the apply process.

The apply process receives messages from the coordination process. It acts on them according to a transaction's outcome. If an abort is received, at the local replica (i.e., the site that received the client request), the transaction that was blocked is canceled. At a remote replica, the message is simply disregarded. If a commit is received, the apply groups operations from different transactions and executes them in a batch.

Finally, the recovery process executes when a replica joins the group and exploits the communication infra-structure, doing the necessary data transfer to update new, or rejoined replicas.

3.2.3.2 Communication Infra-structure

The communication infra-structure is based on group communication greatly easing the development of the ESCADA toolkit by enforcing precise message delivery semantics, specifically in face of faults. The ESCADA toolkit is agnostic when it comes to choosing the group communication toolkit because it relies on a generic group communication service for Java, named jGCS [8]. As such, one is free to use any of the following group communication toolkits without having to rewrite any part of ESCADA: Spread, JGroups or Appia.

3.2.3.3 Management Extensions

The ESCADA toolkit also provides the necessary means to monitor and assess replication protocols. Each component encompassing processes, communication infra-structure, and even the server on which the ESCADA toolkit runs, exports a set of operations and properties of special managed beans (sensors and actuators) by means of the Java Management Extensions framework [1] (JMX). These sensors and actuators may be accessed using the raw *jconsole* tool shipped with JDK, or even through a web-based console developed by the GORDA consortium.

3.2.4 Application Container

The bean configuration XML files, holding component wiring and dependency injection, are shipped inside a jar file and is loaded in runtime as a Java resource from the classpath. Therefore, the large

Configuration File	Description
etc/spring/database.derby.xml	Derby Database descriptor.
etc/spring/database.postgres.xml	PostgreSQL Database descriptor.
etc/spring/database.sequoia.xml	Sequoia Database descriptor.
etc/spring/escada.properties	Spring property configurator.

Table 3.1: ESCADA Configuration files.

XML configuration files are hidden from the end user who has access to plain old Java properties files which are injected in runtime into the bean configuration by means of a Spring property configurator service. This is only possible because component wiring is pretty much static for every replication scenario. What changes are the Java classes implementations used and some of the initial values of their attributes, which are defined in the afore mentioned Java properties files.

3.2.5 Configuration

The ESCADA replicator is configured using the files presented in Table 3.1. The path to the configuration files is relative to ESCADA installation base directory: $ESCADA.

3.2.5.1 Database Configuration

Database descriptors hold configuration parameters concerning databases connections, schemas and users. In these files the user may configure any databases that he wishes to. The only unique field present in these files is the *id* field. This field should uniquely identify a database. The set of possible values are depicted in Table 3.2.

XML Element	Description
databases	Defines the set of databases.
database	Defines data for a given database.
id	Database unique identifier.
url	Connection url.
user	Database user.
password	Password to be used in the connection.
relations	Set of relation names within the given database.
value	Used to specify a value.

Table 3.2: Database descriptor description.

Figure 3.1 shows a sample configuration file for Apache Derby database backend.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE databases SYSTEM "database.dtd">
<databases>
  <database>
    <id>test</id>
    <url>jdbc:derby:test</url>
    <user>refmanager</user>
    <password />
    <relations>
      <value>APP.TEST</value>
    </relations>
  </database>
</databases>
```

Figure 3.1: Sample database.derby.xml

3.2.5.2 Replica Configuration

Replica configuration is performed using a Java properties file named: *escada.properties*. The set of possible values are depicted in Table 3.3. An example of a replica configuration file is given in Figure 3.2

3.2.6 Starting/Stopping the ESCADA Replicator

Before starting the replicator, one needs to change directory into the $ESCADA. At this location there are different scripts to be used with different Operating Systems. On UNIX/Linux derivatives one should use the *startNetworkServer.ksh* and on Microsoft Windows one should use the *startNetwork-Server.bat*.

The start script takes one parameter out of two: *pb* or *optimistic*. The parameter *pb* stands for primary-backup replication protocol and *optimistic* stands for DBSM replication protocol.

To start the replicator, enter the following in the command line (assuming Linux OS):

- startNetworkServer.ksh <pb|optimistic>

To stop the replicator, enter the following in the command line (assuming Linux OS):

- stopNetworkServer.ksh

Property	Description
replica.group.master	Sets the replica as the master (PB Scenario).
replica.group.recovering	true, if recovering, false otherwise.
replica.group.management	true if JMX is to be activated, false otherwise.
replica.group.name	Logical group logical name.
replica.group.configuration	Path to the group comm. configuration file.
replica.dbms.id	DBMS unique id.
replica.dbms.user	DBMS user.
replica.dbms.password	DBMS password.
replica.dbms.connectionaddress	DBMS bind address.
replica.dbms.connectionport	DBMS listening port.
replica.dbms.connectionpool	Connection pool size.
replica.dbms.backuputility	Path to the backup tool binary.
replica.dbms.backupbinary	Sets binary or ASCII backup mode.
replica.dbms.systemHome	Database management system home.
replica.dbms.factory	GORDA Reflector class to instantiate.
replica.dbms.process	GORDA DBMS Reflector class to instantiate.
management.cache.storage.path	Path to the database descriptor file.
management.port	Management port.
recovery.port	Recovery port.
recovery.recoverers	Max. simultaneous recovery donors.
apply.remote.commit.threads apply.local.commit.threads apply.rollback.threads	Thread pool size.
apply.remote.commit.group apply.local.commit.group apply.rollback.group	Maximum batch size.

Table 3.3: Replica configuration properties.

```
replica.group.master=true
replica.group.recovering=false
replica.group.management=true
replica.group.name=hipo-hipo-gsd
replica.group.configuration=etc/appia/appia-master.xml
replica.dbms.id=1
replica.dbms.user=refmanager
replica.dbms.password=
replica.dbms.connectionaddress=0.0.0.0
replica.dbms.connectionport=1527
replica.dbms.connectionpool=10
replica.dbms.backuputility=
replica.dbms.backupbinary=true
replica.dbms.systemHome=.
replica.dbms.factory=gorda.reflector.derby.impl.GenericDerbyReflector
replica.dbms.process=gorda.reflector.derby.impl.DerbyDbmsImpl
management.cache.storage.path=etc/spring/database.derby.xml
management.port=2000
recovery.port=3000
recovery.recoverers=3
apply.remote.commit.threads=6
apply.local.commit.threads=1
apply.rollback.threads=1
apply.remote.commit.group=6
apply.local.commit.group=6
apply.rollback.group=6
```

Figure 3.2: Sample database.derby.xml

3.2.7 Logging

The log framework used is *log4j* [9]. Log settings may be changed by tweaking the log4j configuration file, which is in $ESCADA/etc/log4j.xml, according to the log4j specification.

3.2.8 Notes

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 [6]. Furthermore, the transactions from different databases are serialized. We are currently working with the jGCS developers in order to fix this.

Bibliography

[1] Java Management Extensions (JMX). http://java.sun.com/javase/technologies/core/mntr-mgmt/javamanagement/download.jsp.

[2] Spring framework. http://www.springframework.org/.

[3] Spring IoC. http://www.springframework.org/docs/reference/beans.html.

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

[5] HOW-TO Derby/G. http://gorda.di.uminho.pt/community/derbyg/, 2006.

[6] HOW-TO PostgreSQL/G. http://gorda.di.uminho.pt/community/pgsqlg/, 2006.

[7] JDOM. http://www.jdom.org/, 2006.

[8] jGCS. http://jgcs.sourceforge.net/, 2006.

[9] Log4J. http://logging.apache.org/log4j/docs/, 2006.

[10] The Source for Java Technology. http://www.java.sun.com, 2006.

[11] HOW-TO Sequoia/G. http://gorda.di.uminho.pt/community/sequoiag/, 2007.

[12] JUnit. http://www.junit.org/, 2007.

[13] Maven. http://maven.apache.org/, 2007.

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

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

[16] 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.

[17] M. Fowler. Inversion of control containers and the dependency injection pattern. http://martinfowler.com/articles/injection.html.

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

[19] B. Kemme and G. Alonso. A Suite of Database Replication Protocols Based on Group Communication Primitives. In *IEEE ICDCS*, 1998.

[20] F. Pedone. *The Database State Machine and Group Communication Issues*. PhD thesis, Département d'Informatique, l'École Polytechnique Fédérale de Lausanne, 1999.

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

[22] R. Jiménez Peris, M. Patiño Martínez, B. Kemme, and G. Alonso. Improving the Scalability of Fault-Tolerant Database Clusters. In *IEEE ICDCS*, 2002.

[23] A. Pinto, H. Miranda, and L. Rodrigues. Light-weight groups: an implementation in ensemble. In *Fourth European Research Seminar on Advances in Distributed Systems (ERSADS'01)*, 2001.

[24] M. Raynal and M. Singhal. Mastering Agreement Problems in Distributed Systems. *IEEE Software*, 18:40–47, 2001.

[25] L. Rodrigues, J. Mocito, and N. Carvalho. From spontaneous total order to uniform total order: different degrees of optimistic delivery. In *Proceedings of the 21st ACM symposium on Applied computing (SAC'06)* (to appear). ACM Press, April 2006.

[26] Douglas C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. Wiley & Sons, 2000.