Project no. 004758

GORDA

Open Replication of Databases

Specific Targeted Research Project

Software and Services

Prototype of the Integrated System

GORDA Deliverable D5.2
(Companion Report)

Due date of deliverable: 2008/03/31
Actual submission date: 2008/04/21

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

Universidade do Minho

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

Alfrânio Correia Júnior, U. Minho
Ana Nunes, U. Minho
António Sousa, U. Minho
Emmanuel Cecchet, Continuent
José Pereira, U. Minho
Luís Soares, U. Minho
Luís Rodrigues, U. Lisboa
Miguel Matos, U. Minho
Nuno Carvalho, U. Lisboa
Nuno Carvalho, U. Minho
Rui Oliveira, U. Minho
Ricardo Vilaça, U. Minho
Sara Bouchenak, INRIA

Abstract

This document accompanies the project's Prototype of the Integrated System. It presents a succinct description of the various modules current implementations and the detailed module dependencies in the demonstration scenarios.

Contents

Chapter 1

Introduction

The GORDA Prototype of the Integrated System (Deliverable 5.2) includes the three main GORDA services, the current GORDA compliant DBMSs and a workload generator:

- **GORDA Replication Service (GRS)** Based on the ESCADA replication system [2] as described in GORDA Deliverable D3.3.
- **GORDA Communication Service (GCS)** Composed of the jGCS [4] generic interface for group communication and binding for the APPIA group communication system [6] as described in GORDA Deliverable 6.4b.
- **GORDA Management Service (GMS)** Based on the JADE autonomic management system [3] with OSGi, JMX, Fractal/Julia as underlying technologies as described in GORDA Deliverable D5.4.
- **Gorda compliant DBSMs** The prototype of the integrated system include all the GORDA API implementations in Apache Derby, Sequoia and PostgreSQL described in detail in GORDA Deliverables D4.3, D4.4 and D4.5, respectively.
- **Workload generator** Consists of a generator to simulate the TPC-C benchmark.

  In the next chapters we provide relevant details and limitations of the current implementations of the above components.

1.1 Relationship With Other Deliverables

This document presents a succinct description of the implementation of the various modules that make the GORDA Prototype of the Integrated System (Deliverable 5.2). As such it directly relates to deliverables D3.3 - Replication Modules Reference Implementation, D3.4 - Modules Description and Configuration Guide, D4.3 - In-Core Proof-of-concept, D4.4 - Middleware Proof-of-concept, D4.5 - Hybrid Proof-of-concept, and D4.6 - DB Support Description and Configuration Guide.

Chapter 2

GORDA Replication Service (GRS)

The GORDA Replication Service is a modular reference implementation of replication protocols built on the GORDA Architecture and Programming Interfaces. The GRS can be configured to provide any of the protocols described in deliverable D3.3 - Replication Modules Reference Implementation.

The GRS is mainly composed of:

- **escada-interfaces** Utility classes for marshaling/unmarshaling and interface with the GCS and the GAPI compliant databases.
- **escada-protocols** Implementation of the replication protocols.
- **escada-replicator** Implementation of the processes that implement the GAPI processing contexts and rgister and handle GAPI events (as per deliverables D2.2 - Architecture Definition Report and D2.3 - APIs Definition Report).

The current GRS prototype consists of 24171 lines of Java Code. This includes the escada-interfaces (13913), escada-protocols (3822) and escada-replicator (6436).

2.1 Known Bugs and limitations

- Replication of meta information: our current implementation does not handle replication of meta information. Thus, if one wants to put a new replica online, it should be configured as described in [1] prior to begin any state transfer;
- Grouping local commit: the existing optimization for data flushing by grouping local commits is not safe as of yet and might introduce inconsistencies;
- Forcing commitment ordering among different databases: the current prototype forces commitment ordering among transactions from different databases as the same communication channel is used to atomically propagate transactions. We are currently working with the JGCS developers to circumvent this by providing different atomic channels, one per database, with no ordering guarantees among them.
- **Recovery Protocols** The prototype presented at the demo only supports full database transfer.

Chapter 3

GORDA Management Service (GMS)

The GORDA Management Service (GMS) is based on Jade, a framework that provides autonomic management capabilities for distributed systems [3]. GMS handles (i) the management of legacy database systems, and (ii) the management of clusters of replicated database systems.

The GMS is composed of a dynamic configuration and deployment module, a monitoring module, an automatic recovery module, an optimization module and a supervision console.

The implementation of the GMS has 16254 lines of Java Code, splitted in the dynamic configuration and deployment module (1241), the monitoring module (6327), the automatic recovery module (3468), and the dynamic optimization module (5219). Additionally, the dashboards developed for the supervision console consist of 320 lines of XML.

3.1 Known Bugs and limitations

- The current implementation of the deployment module keeps all logs in memory. This leads to their loss in cas of a crash and may lead to memory exhaustion in the event of a long JManage crash;
- The supervision console freezes the browser if the dashboard of replica *n* is opened when replica *n* fails.

Chapter 4

GORDA Communication Service (GCS)

GORDA Deliverable 6.4b (GORDA Group Communication Service Specification) specifies the programming interfaces for generic group communication systems.

In the scope of the project, the presented interfaces were implemented with several group communication toolkits in mind. The prototype of the integrated system uses the Appia binding.

4.1 jGCS

GCS specifies a programming interface for Group Communication as well as minimum semantics that allow application portability. This interface accommodates existing group communication services, enabling implementation independence. The interface is called jGCS.

The jGGS interface consists of 2160 lines of Java Code. Additionally, the implementations of the interface for the different bindings (Appia, JGroups, NeEM, Spread) have 6134 lines of Java code and the demonstration applications 2204 lines of Java code.

4.1.1 Known Bugs and limitations

- Annotations in messages are not implemented in any binding;
- The Appia implementation of the *Service* interface and usage is limited to the set required by the project.

4.2 Appia

Changes made to the Appia communication toolkit had been reported in the GORDA deliverable D3.5 (Group Communication Protocols).

We make an informal evaluation of the effort required to implement the GORDA requirements and GCS interface (jGCS) in current version of Appia, 4.0.1, by counting the lines of code modified relatively to the size of the code before GORDA project, Appia 3.0. Additionally Appia had several bugs fixed and improvements during GORDA project.

The size of Appia 3.0 was 36442 lines of Java code, where 422 files were changed by inserting 23785 lines and deleting 12189 lines, resulting in 48038 lines of code.

4.2.1 Known Bugs and limitations

- The amount of memory spent by a communication channel is defined statically;

- Communication channels are defined on configuration time, not being possible to create new channels on the fly;
- When the system bootstraps, the first (primary) view must be defined by a system administrator.

Chapter 5

GAPI implementations

5.1 In-core proof of concept

We make an informal evaluation of the effort required to implement the GAPI in Derby by counting the lines of code modified relatively to the size of the original code base.

The effort required to implement such subset of the interface can roughly be estimated by the amount of lines changed in the original source tree as well as the amount of new code added: the size of Apache Derby is 514941 lines, where 29 files were changed by inserting 1250 lines and deleting 25 lines; 9464 lines of code were added in new files.

5.1.1 Known Bugs and limitations

- Our current implementation on Derby is not properly handling write set when explicit transactions are executed;
- The log miner phase is not implemented, and thus the recovery process is not implemented as it requires a hot backup integrated with a log miner;
- It is not possible to add and remove Derby replicas as recovery is not implemented.

5.2 Middleware proof of concept

The size of the generic portion of Sequoia is 137238 lines, which includes the Sequoia Controller and the JDBC driver. Additionally, Sequoia contains 29373 lines implementing plugable replication and partitioning strategies which are not used when used as a GORDA DBMS. The GAPI implementation in Sequoia, consisted in changing the Sequoia Controller. 7 files were changed by inserting 180 lines and deleting 23 lines and 8625 lines of code were added in new files.

The Sequoia GORDA compliant version, Sequoia 3.0 Beta 2, evolved, in the context of the project, from version 2.9. It was subject to numerous bugs fixes and improvements. A comparison of the two versions reveals changes in as many as 667 files. These changes were mainly done in lock, prepared statements and stored procedures semantics.

5.2.1 Known Bugs and limitations

- The current implementation of PostgreSQL write set extraction through triggers calls the PostgreSQL CLI (*psql*) explicitely due to limitations on JDBC driver.

5.3 Hybrid proof of concept

We make an informal evaluation of the effort required to implement the GAPI in PostgreSQL by counting the lines of code modified relatively to the size of the original code base.

Regarding PostgreSQL, its code base in version XXXX is 667586 lines of C code plus the 7574 lines of C code and 16331 of Java code added by the PL/J package. 21 files were changed by inserting 569 lines and deleting 152 lines. Additionally, 1346 lines of C code and 11512 lines of Java code were added in new files.

5.3.1 Known Bugs and limitations

- PL-J does not currently support all valid PostgreSQL data types (e.g., date, datetime, timestamp, float). Some of these were implemented, in both C and Java languages to suit both ends. The remaining were solved by adding an interim conversion to strings, as PostgreSQL converts them back as needed;
- Running clients concurrently to a *VACUUM FULL* crashes PostgreSQL;
- PL-J does not successfully supports *SELECTs* in the context of ongoing transaction.

Chapter 6

Workload Generator

The workload generator is based on the TPC-C benchmark, that proposes a wholesale supplier with a number of geographically distributed sales districts and associated warehouses as an application. This environment simulates an OLTP workload with a mixture of read-only and update intensive transactions.

TPC-C determines that the database size is configured for each simulation run according to the number of clients as each warehouse supports 10 emulated clients [7]. This ensures a realistic amount of conflicting updates and I/O requirements.

The workload generator can also be used in a mode that does not enforce standard TPC-C scaling guidelines.

The workload generator component spawns multiple threads using a pool of connections spread evenly across active database replicas. Upon failure of a replica, connections are reallocated to remaining active replicas. Upon start of a replica, load is re-balanced. The workload generator is monitored and controlled using a JMX interface.

For more details see Section 2.1 on GORDA Deliverable 7.1 (Deployment Plan).

The effort required to implement the workload was 10260 lines of Java Code which includes the workload generator and the initial populate of the TPC-C database.

6.1 Known Bugs and limitations

- Workloads does not stricly follows some TPC-C specification parameters like boundaries on execution time and standard deviation.

Chapter 7

Software Dependencies

All software developed in the context of the GORDA project were integrated with Maven [5]. Maven is a software project management tool. It manages information regarding library dependency tracking, software building, reporting, project meta-data and documentation. Data is kept in an XML file (pom.xml) which stands for Project Object Model. It has a pluggable architecture, and several plugins exist that rely on pom.xml to actually perform actions like build and test source code, generate reports and documentation. In the context of GORDA, maven has been used primarily as a dependency tracking, build and testing framework. It was also used to generate documentation and, with the appropriate plugin, to visually keep track of all the software modules and libraries used or developed in the GORDA prototypes.

We present a dependency graph generated with Maven plugins of the integrated prototype needed to run each scenario of the demo as described in the GORDA Deliverable D7.1 (Deployment Plan).

7.1 Scenario 0:Basic System

This scenario aims at demonstrating a GORDA system by setting it up from publicly distributed software packages, as would be done by an end-user or system integrator while evaluating GORDA technology. In detail, this scenario shows a simple primary-backup replication setup using Apache Derby, ESCADA, and Appia. The outline of the software dependencies graph is presented in Figure 7.1.

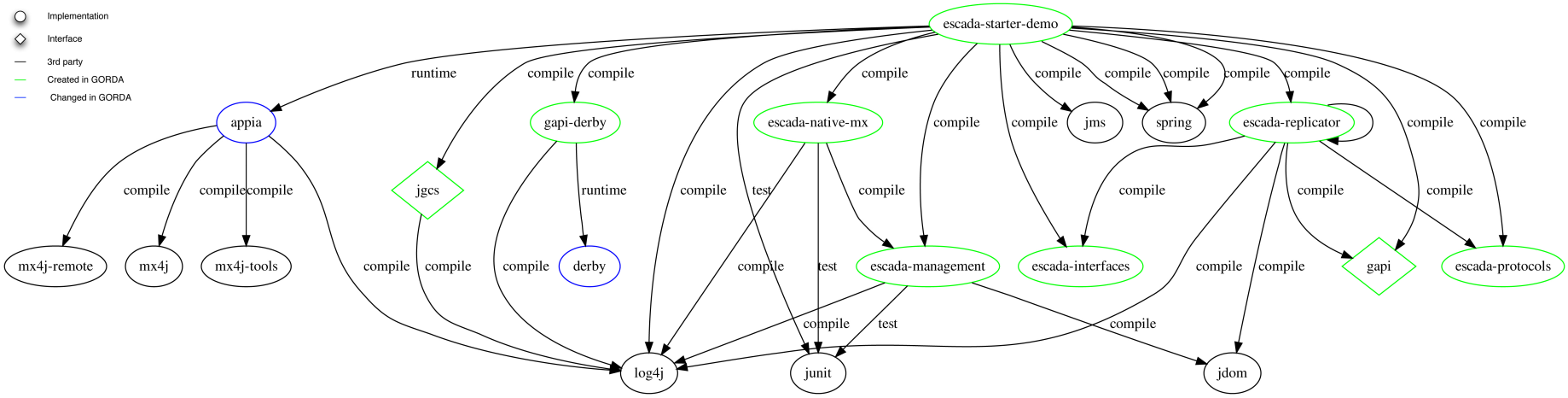

Figure 7.1: Scenario 0 software dependencies graph

7.2 Scenario 1:Complete System

This scenario demonstrates the complete GORDA Integrated Prototype in a cluster, including a multi-master/update everywhere replication protocol, automated deployment, recovery, selftuning, and a complex workload. In detail, it shows PostgreSQL, ESCADA, Appia, and Jade. The outline of the software dependencies graph is presented in Figure 7.2.

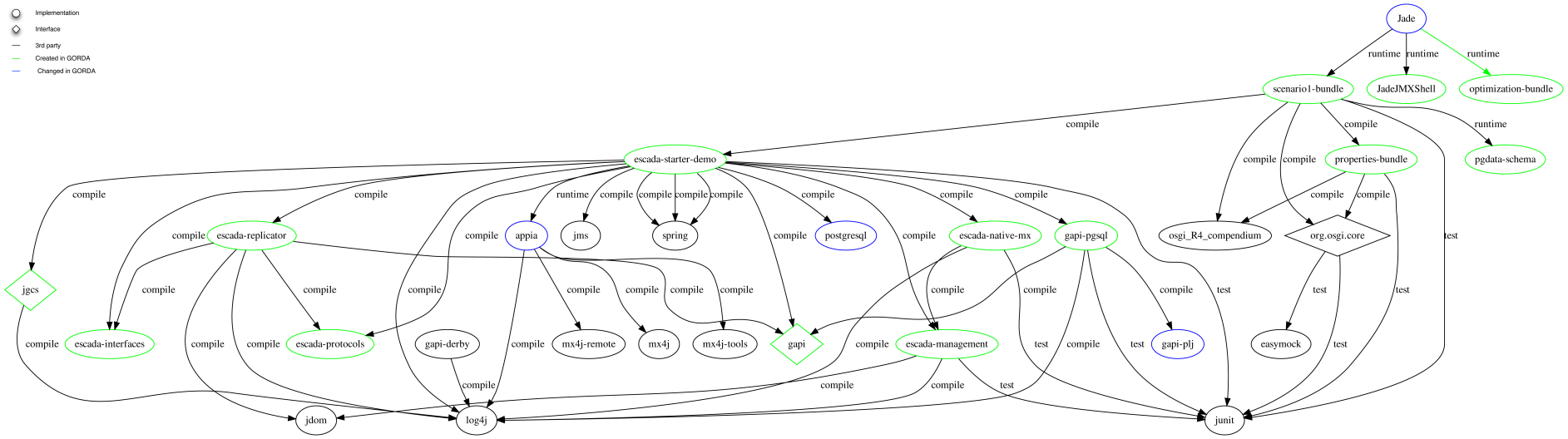

Figure 7.2: Scenario 1 software dependencies graph

7.3 Scenario 2:Realistic Workload

This scenario demonstrates the performance of a GORDA configuration identical to Scenario1 but running the workload of the industry standard TPC-C benchmark with proper scaling. The outline of the software dependencies graph is presented in Figure 7.3.

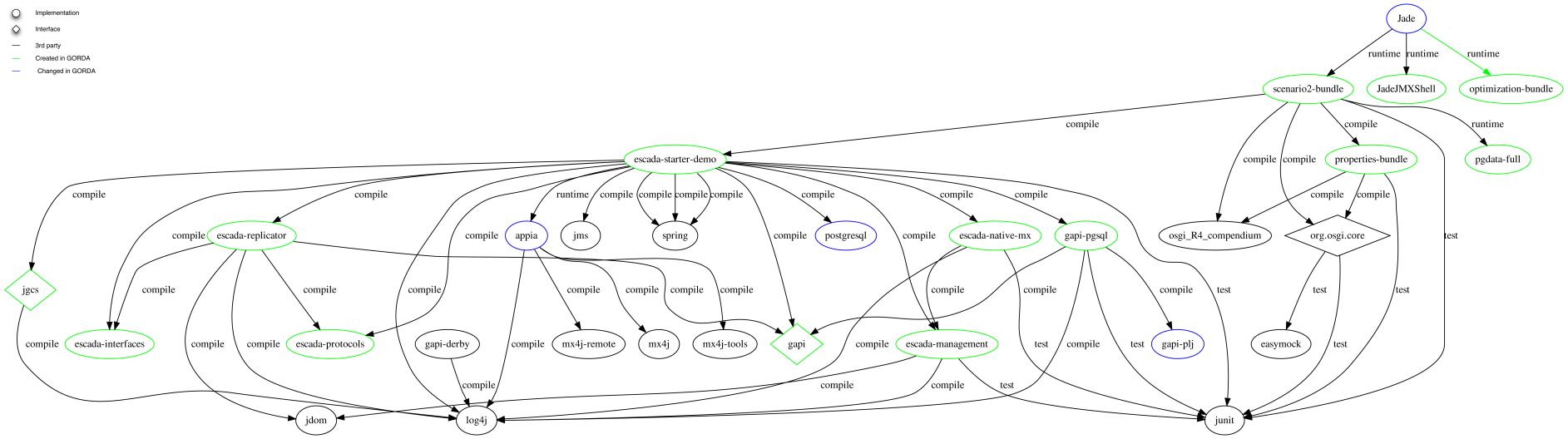

Figure 7.3: Scenario 2 software dependencies graph

Bibliography

[1] HOWTO postgresql/g. http://gorda.di.uminho.pt/community/pgsqlg/, 2006.

[2] ESCADA replicator. http://sourceforge.net/projects/escada/, 2007.

[3] Jade: A framework for autonomic management. http://sardes.inrialpes.fr/jade.html, 2008.

[4] 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.

[5] The Apache Software Foundation. Apache maven project. http://maven.apache.org/, 2002.

[6] H. Miranda, A. Pinto, and L. Rodrigues. Appia: A flexible protocol kernel supporting multiple coordinated channels. In *21st International Conference on Distributed Computing Systems (ICDCS 2001)*, Phoenix, Arizona, USA, April 2001.

[7] Transaction Processing Performance Council (TPC). Tpc benchmark^{TM} c standard specification revision 5.0, February 2001.