Project no. 004758

GORDA

Open Replication of Databases

Specific Targeted Research Project

Software and Services

Interface and Modules Performance Assessment Report

GORDA Deliverable D5.3

Due date of deliverable: 2008/03/31
Actual submission date: 2008/04/28

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
José Pereira, U. Minho
Luís Soares, U. Minho
Luís Rodrigues, U. Lisboa
Nuno Carvalho, U. Lisboa
Robert Hodges, Continuent
Rui Oliveira, U. Minho

Abstract

This report presents methodologies and procedures used to conduct performance assessment on the GORDA reflection API and its corresponding mappings. We have relied on a model-driven development approach, in which simulation played an important role in the initial stages of the development cycle. This has been previously detailed elsewhere. Eventually, every abstract component got replaced by its respective real implementation, subject to stress testing and benchmarking. The procedures for this undertaking are detailed in this document.

Contents

Chapter 1

Introduction

This document presents methodologies and procedures used to conduct performance assessment on the GORDA reflection API and its corresponding mappings. GORDA proposes several components and a reflection API for database replication, which required early and frequent tests during the development cycle. A derivative of these tests, eventually led to the foundations of the validation and performance assessment benchmarking processes.

We have relied on a model-driven development approach, in which simulation played a big role in the initial stages of the development cycle. This has been thoroughly detailed in D5.1 Report. Eventually, every abstract component got replaced by its respective real implementation, subject to stress testing and benchmarking. The procedures for this undertaking are detailed in the rest of this document.

1.1 Objectives

This report aimas at:

- presenting the methodologies and procedures used to assess the GORDA components;
- providing insight of the major distinguishing details between the several mappings regarding the group communication and database APIs;
- provide representative performance results of the existing prototypes.

1.2 Relationship With Other Deliverables

The descriptions and results herein directly relate to the prototypes presented by deliverables D3.3 - Replication Modules Reference Implementation, D3.5 - Group Communication Protocols Report, D4.3 to D4.6 - In-Core Proof-of-concept, Middleware Proof-of-concept and Hybrid Proof-of-concept, respectively.

This reports complemenets deliverable D5.1 - Performance and Reliability Assessment Report.

1.3 Document Structure

This document is structured as follows: Chapter 2 presents relevant technical details that may play a major role in performance evaluation, either of the APIs as well as their implementations. In particular, Section 2.1 brings forth issues related to the Group Communication APIs (jGCS) and its bindings. Moreover, Section 2.2 does the same for GORDA database reflection interfaces (GAPI), by analyzing reference implementations of the API. Chapter 3 presents the performance assessment results. Section 3.1 shows results obtained when assessing message throughput and performance overhead when using jGCS and two of its bindings (for Appia and JGroups). On the other hand, Section 3.2 presents the performance results for transaction processing when using a GORDA replication stack, for all three GORDA reference implementations. Finally, Chapter 4 concludes the report.

Chapter 2

APIs and Components

The GORDA architecture is comprised of three sets of components: i) the Application Programming Interfaces (e.g., for database reflection, GAPI, and for group communication, jGCS); ii) the API mappings (either GAPI or jGCS implementations); and iii) the database monitors (which required instrumentation for in-core implementations). Not all of these components were benchmarked individually, but all of them were ultimately assessed by benchmarking a fully compliant GORDA software stack.

Apart from these functional components, others, geared towards benchmarking and performance assessment, were developed, either from scratch or from formal specifications. The latter, may be classified in different categories depending on their focus. Classification is performed according to the following categories:

Workload generators. Their task is to provide realistic data in the form of inputs to the system. The workload is much more valuable as it is more close to reality. This was a major concern when deciding which benchmarks were to be used.

b-probes. Benchmarking probes were developed to extract raw structured data to be fed to numeric computation tools responsible to address statistical processing; *b-probes* differ from *f-probes* which are responsible to extract information during run-time and feed decision making modules that operate actuators on the system, handling gracefully and autonomically particular extemporaneous behavior. *b-probes* do not induce any relevant overhead on the overall system behavior, otherwise they would be jeopardizing their purpose.

Statistical software. As already mentioned above, *b-probes* output raw, but yet structured, data related to the system behavior. This data needs to be parsed and fed to statistical engines that transform the provisioned data into

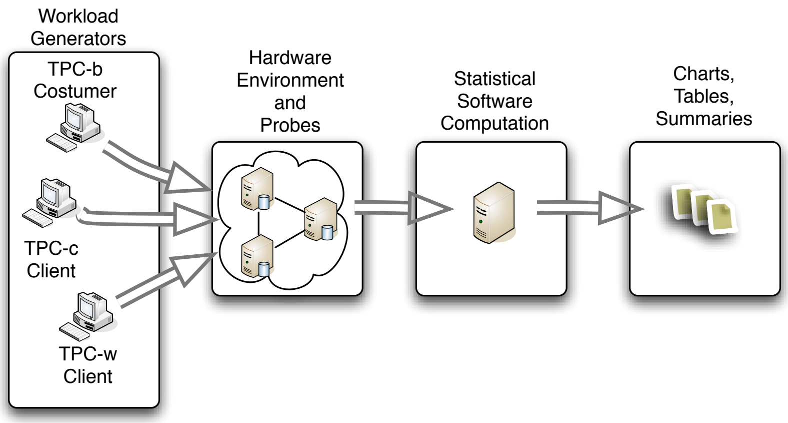

Workload Generators
TPC-b Costumer
TPC-c Client
TPC-w Client
Hardware Environment and Probes
Statistical Software Computation
Charts, Tables, Summaries

Figure 2.1: Component interaction.

human readable formats (charts, tables, summaries). Parsing and transformation is performed by using Python scripting and statistical engines rely on the R Project for Statistical Computation [3]. Plotting is performed by using gnuplot [2].

Hardware environment. These are hardware components required to conduct the performance evaluation.

2.1 Group Communication

The set of group communication interface (jGCS) was implemented in several group communication toolkits and primitives: Appia [13], JGroups [5] and Spread [4]. To validate the generality of the service, jGCS was also implemented using IP Multicast and NeEM [15]. All these bindings are open source and available on SourceForge.net [11]. These toolkits and their implementations are described in the following paragraphs.

Appia binding

Appia is a layered communication support framework that was implemented in the University of Lisbon. It is implemented in Java and aims at high flexibility to build communication channels that fit exactly in the user needs. The QoS offered by a channel can be statically configured by an XML file or dynamically assembled by the application at run time. The application can create several channels with different QoSs and send messages to different channels, depending on the QoS required by each message. In contrast with traditional layered protocols, components of Appia channels can be shared and thus offer multiple related Qualities of Service (QoS). This makes it easy, for instance, that several channels can be bound to the same group membership.

Although Appia is protocol independent, in the sense that it can be used to compose any protocol as long as it respects the predefined interface, it includes an extensive layer library targeted at view synchronous group communication. Namely, it has protocols that implement virtual synchrony, causal order, and several implementations of total order algorithms.

The implementation of jGCS is built directly on the Appia protocol composition interfaces as an additional layer. jGCS configuration objects thus define the micro-protocols that will be used in the communication channels. Each Service identifies an Appia channel and messages are sent through the channel that fits the requested service. As Appia supports early delivery in totally ordered multicast, this is exposed in the jGCS binding using a jGCS service listener interface. The Appia binding implements all extensions of the jGCS session control interface, depending on the channel configuration.

JGroups binding

JGroups is a group communication toolkit modelled on Ensemble [9] and implemented in Java. It provides a stack architecture that allows users to put together custom stacks for different view synchronous multicast guarantees as well as supporting peer groups. It provides an extensive library of ordering and reliability protocols, as well as support for encryption and multiple transport options. It is currently used by several large middleware platforms such as JBoss and JOnAS.

The JGroups implementation of jGCS also uses the configuration interface to define the micro-protocols that will be used in the communication channel. JGroups can provide only one service to the applications, since configurations only support one JGroups channel per group communication instance. The JGroups binding implements all extensions of the jGCS session control interface.

Spread binding

Spread/FlushSpread [4] is a toolkit implemented by researchers of the Johns Hopkins University. It is based on an overlay network that provides a messaging service resilient to faults across local and wide-area networks. It provides services ranging from reliable message passing to fully ordered messages with delivery guarantees. The Spread system is based on a daemon-client model where generally long-running daemons establish the basic message dissemination network and provide basic membership and ordering services, while user applications linked with a small client library can reside anywhere on the network and will connect to the closest daemon to gain access to the group communication services. Although there are interfaces for Spread in multiple languages, these do not support the FlushSpread extension, which provides additional guarantees with a different interface.

The Spread and FlushSpread bindings of jGCS use the configuration interface to define the location of the daemon and the group name. The implementation to use (FlushSpread or just Spread) is also defined at configuration time. In Spread, the quality of service is explicitly requested for each message, being thus encapsulated in Service configuration objects.

Other bindings

To prove the generality of jGCS, we also provide two implementations, based on the well known IP Multicast and on the Network-friendly Epidemic Protocol (NeEM) [15]. The NeEM protocol is an epidemic multicast protocol (also called probabilistic or gossip-based) in wide-area networks that uses multiple TCP/IP connections in a non-blocking fashion. The resulting overlay network is automatically managed by the protocol. The implementations of jGCS that use IP Multicast and NeEM allow peers to join and leave the multicast group, and send and receive messages to/from other peers. One application that uses only these functionalities can easily be ported to other implementations.

2.2 Database

In this section, we provide a brief description of each of the GAPI bindings, including information about the number of lines of code required to implement them, on each architecture.

Apache Derby Binding

Apache Derby 10.2 [1] is a fully featured database management system with a small footprint that uses locking to provide serializability. It can either be embedded in applications or run as a standalone server. It was developed by the Apache Software Foundation and distributed under an open source license; It is also distributed as IBM Cloudscape and in the Sun JDK 1.6 as JavaDB.

The GAPI prototype implementation takes advantage of Derby being natively implemented in Java to load meta-level components within the same JVM and thus closely coupled with the base-level components. Furthermore, Derby uses a different thread to service each client connection, thus making it possible that notifications to the meta-level are done by the same thread and thus reduce to a method invocation, which has negligible overhead. This is therefore the preferred implementation scenario.

Implementation Effort. The total size of the Apache Derby engine is 514941 lines of code. In order to implement the GAPI interface, 29 files were changed by inserting 1250 lines and deleting 25 lines; in total, 9464 lines of code were added in new files.

PostgreSQL Binding

PostgreSQL 8.1 [8] is a fully featured database management system distributed under an open source license. Written in C, it has been ported to multiple operating systems, and is included in most Linux distributions as well as in recent versions of Solaris. Commercial support and numerous third party add-ons are available from multiple vendors. Since version 7.0, it provides a multi-version concurrency control mechanism supporting snapshot isolation.

A challenge in implementing the proposed architecture in PostgreSQL is the mismatch between its concurrency model and the multi-threaded meta-level run-time. PostgreSQL 8.1, as all previous versions, uses multiple single-threaded operating system processes for concurrency. This is masked by using the existing PL/J binding to Java, which uses a single standalone Java virtual machine and inter-process communication. This imposes an inter-process remote procedure call overhead on all communication between base and meta-level.

Therefore, the prototype implementation of the GORDA interface in PostgreSQL 8.1 uses a hybrid approach. Instead of directly patching the reflector interface on the server, key functionality is added to existing client interfaces and as loadable modules. The proposed meta-level interface is then built on these modules. The two layer approach avoids introducing a large number of additional dependencies in the PostgreSQL code, most notably on the Java virtual machine. As an example, transaction events are obtained by implementing triggers on transaction begin and end. A loadable module is then provided to route such events to meta-objects in the external PL/J server.

Implementation Effort. The size of PostgreSQL is 667586 lines of code; the PL/J package adds 7574 lines of C code and 16331 of Java code. In order to implement the GAPI interface on PostgreSQL 21 files changed by inserting 569 lines and deleting 152 lines, 1346 lines of C code were added in new files, and 11512 lines of Java code added in new files.

2.2.1 MySQL Binding

MySQL [14] is a fully featured database management system. MySQL AB provides a freely downloadable version of its open source database. It is written in C/C++ and it has ports to multiple operating systems. The most popular storage engine, and the one that we have used, is the one created by InnoBase and named InnoDB [10]. It provides serializable isolation level and ACID properties. The core is multi-threaded, meaning that each connection is assigned its own thread, unlike PostgreSQL which uses a single-threaded process per connection. This has major impact on the overall performance as it avoids inter-process communication and eases the implementation of extensions points that are to be executed within the context of the original thread.

In the current pre-production version - 5.1 - the MySQL core has been revamped and as consequence there are now interfaces that provide hooks for pluggable modules. Several plugin types exist, most noticeable for storage engines, but still, none for replication. Nevertheless, the architecture favored the implementation of a plugin type for replication which holds a subset of the GORDA interface.

Although the plugin was implemented both in C++ and Java, execution context between MySQL and ESCADA was preserved by resorting to a Java bridge coded using Java Native Interface [16], keeping overhead at the minimum.

Implementation Effort. The size of the patch to MySQL codebase is 1123 lines: 2 files added and 4 changed. The plugin that interfaces MySQL and Java was implemented in 6 files in a total of 879 C++ lines.

Sequoia Binding

Sequoia 3.0 [7] is a middleware package for database clustering built as a server wrapper. It is primarily targeted at obtaining replication or partitioning by configuring the controller with multiple backends, as well as improving availability by using several interconnected controllers.

Nevertheless, when configured with a single controller and a single backend, Sequoia provides a state-of-the-art JDBC interceptor. It works by creating a virtual database at the middleware level, which re-implements part of the abstract transaction processing pipeline and delegates the rest to the backend database.

The current prototype exposes all context objects and the parsing and execution objects, as well as calling from meta-level to base-level with a separate connection. It does not allow calling from base-level to meta-level, as execution runs in a separate process. It can however be implemented by directly intercepting such statements at the parsing stage. It does not either avoid that base-level operations interfere with meta-level operations, and this cannot be implemented as described in the previous sections as one does not modify the backend DBMS. It is however possible to the clustering scheduler already present in Sequoia to avoid concurrently scheduling base-level and meta-level operations to the backend, thus precluding conflicts.

Implementation Effort. The size of the generic portion of Sequoia is 137238 lines, which includes the controller and the JDBC driver; additional 29373 lines implement plugable replication and partitioning strategies, that are not used by GAPI. In order to implement the GAPI interface on Sequoia, 7 files were changed by inserting 180 lines and deleting 23 lines, and 8625 lines of code were added in new files.

2.2.2 Notes on the GAPI Implementation Effort

The effort required to implement a subset of the GAPI 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 numbers presented in the previous sections show that it is possible to implement the GAPI interface in various different architectures, with consistently low intrusion in the original source code. This translates in low effort both when implementing it but also when maintaining the code when the DBMS server evolves.

Load Test	Description
Evaluator	Runs mix of INSERT, UPDATE, DELETE, and SELECT statements with ability to change percentages, alter number of clients, and adjust number of rows

Table 2.1: Bristlecone main load test.

Note also that a significant part of the additional code is shared, namely in the definition of the interfaces (6144 lines). There is also a firm belief most of the rest of the code could also be shared, as it performs the same container and notification support functionality. This has not happened as each implementation was developed independently and concurrently.

Finally, it is interesting to note that the amount of code involved in developing a state-of-the-art server-wrapper is in the same order of magnitude as a fullyfeatured database (i.e. hundreds of Klines of code). In comparison, implementing the GAPI involves 100 times less effort as measured in lines of code.

2.3 Benchmarking Tool-set

Continuent Bristlecone Test Tools

Bristlecone [6] test tools suite was designed to allow rapid performance benchmarking across a wide variety of scale-out architectures. It is specifically targeted at performance of primary-backup, state-machine, and certification based architectures supported by GORDA. The main benefits in addition to addressing performance issues of interest to GORDA are rapid set-up and easy-to-read output.

Continuent has released Bristlecone Test Tools under GPLv2 license and the source code is therefore publicly available [6]. Contributions from other GORDA participants are integrated in the code and are to be maintained on an on-going basis. Continuent is also using the tools as a vehicle to promote the overall GORDA architecture. (Plans for additional productization are discussed further in D6.5.)

It provides a mean to easily conduct mixed load tests in which mixed transactions are sent to the database to test overall scaling. Table 2.1 summarizes the current main load test that is available.

Figure 2.2 shows typical output showing response time and request rates in a test run. This graph also illustrates the ability to scale load during the test.

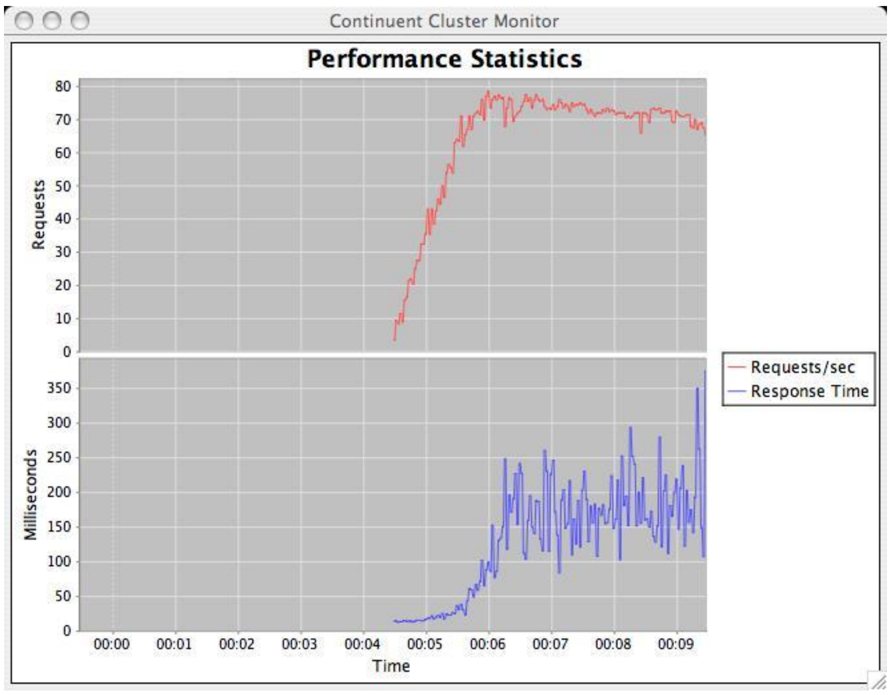


Figure 2.2: Typical graphical output generated from a bristlecone run.

Statistical Analysis and Data Plotting

R bindings to python [18] and python itself were used to conduct analysis over raw structured data files. R provided means to easily conduct numeric/statistical computation over the samples collected. This automated the process and eliminated error prone *ad hoc* approaches to computation that one might be tempted to create.

As for plotting data, gnuplot [2] was the tool selected. It provides a simple, easy and clear script language to create advance plots in little time.

Chapter 3

Performance Evaluation Results

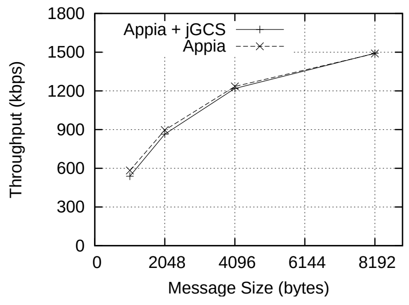
3.1 Group Communication

We have done a number of experiments to assess the overhead imposed by the use of jGCS to wrap different group communication toolkits. Namely, we wanted to assess the impact of the extra level of indirection between the application and the toolkit introduced by jGCS. For this purpose we have defined two different sets of tests. In the first set we carried standalone throughput measurements for two different toolkits, both with and without jGCS. In the second set of tests we have integrated jGCS in a production environment, namely in the Sequoia database clustering middleware.

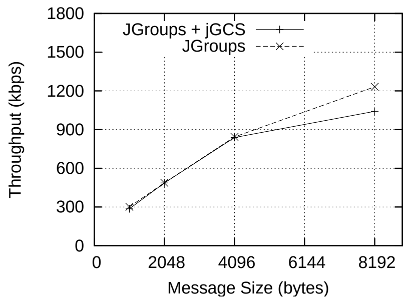
Scenario I: jGCS for Appia and JGroups

To measure the impact of the jGCS on the maximum throughput of existing group communication toolkits we have selected Appia and JGroups. To run the experiments, we have implemented three different versions of a test application that transmits a number of messages of a configurable payload size to the group. One version uses the Appia native interface, other uses the JGroups native interface, and the last version uses jGCS. This allowed us to run four different configurations: *i)* the test application with Appia; *ii)* the test application with JGroups, *iii)* the test application with jGCS, configured to use Appia and *iv)* the test application with jGCS, configured to use JGroups.

Measurements were obtained with the following environment. The JGroups and the Appia protocol stack were created using similar configurations. All tests used a virtual synchrony protocol stack and a token based total order protocol. Furthermore, they were made with a group of three members, each member sending 10^4 totally ordered messages to the group. Each member of the group runs in a Pentium IV/2.8GHz server with 1GB of memory. The three machines



(a) Appia Throughput.



(b) JGroups Throughput.

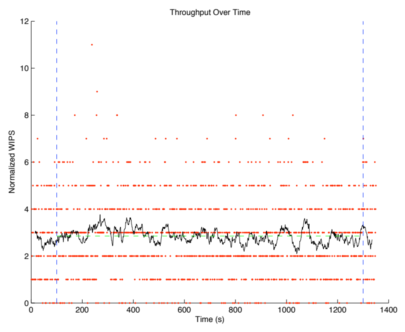
Figure 3.1: Throughput of the toolkits with and without the jGCS.

were connected through a 100Mbps ethernet switch. Each test was carried with different message sizes. Figure 3.1 shows the throughput of the two group communication toolkits, using directly the interface provided by the toolkit and using jGCS. As we can see in Figure 3.1(a), the Appia implementation of jGCS does not cause a significant overhead and this overhead is increasingly less noticeable as the message size grows. In the case of JGroups, in Figure 3.1(b), the overhead caused by the jGCS is also very small but it grows as the message size increases. This is explained as follows: For improved performance, JGroups delivers messages in a buffer that can be reused later by the protocol, forcing the application to locally copy data during delivery. The native JGroups test application does not perform this copying, and thus has better performance. On the other hand, the current jGCS binding does this copying in order to provide the same service as other bindings and thus incurs in additional overhead. In the future, this decision should probably be left to the configurator, thus making it possible to achieve the same performance as with the native interface.

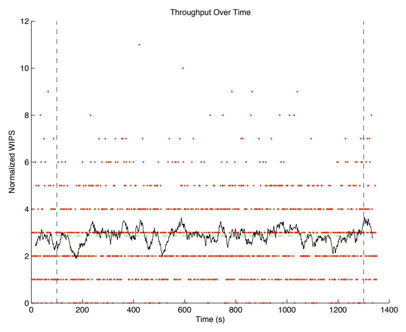
Scenario II: jGCS in Sequoia

The second set of tests measures the overhead of having jGCS in a real application. To do these tests we used Sequoia that exports a JDBC interface to applications and routes client requests to a set of databases. Sequoia is composed by a JDBC driver, that is used by applications that want to access the databases and a controller that receives the client requests and forward them to a set of databases. For availability and fault tolerance, the Sequoia controller can be replicated. Each controller manages a set of databases. In a system with more than one controller, the application can use any controller to make the requests. The controllers exchange their requests using total order multicast, to execute the same set of requests in the same order in all databases.

The implementation of primitives that make use of group communication is distributed as a separate package, Hedera (formerly ObjectWeb Tribe). In detail, it provides access to an application specific subset of group communication



(a) with jGCS



(b) without jGCS

Figure 3.2: Throughput of Sequoia in WIPS.

and additional functionality for explicitly acknowledged messages, multiplexing and dispatching. Hedera has been previously implemented against JGroups and Appia. We ported Hedera to jGCS which allowed us to use Sequoia with any jGCS implementation that supports the required service guarantees.

Performance figures were obtained in a system configured as follows. The clients are a Java implementation of the TPC-W [12] that use the Jakarta Tomcat to make requests to a database. The requests are made to a sequoia controller that replicates the requests. Sequoia is configured to use three controllers, each one controls one MySQL database. The emulated browsers of the benchmark used and Tomcat run in one machine. The other three machines have one instance of the Sequoia controller and one instance of the MySQL database each. All four machines are connected by a 100Mbps ethernet switch and have the same memory and processing power of the machines used in the previous tests. In these tests, the benchmark was configured to have always 20 clients (emulated browsers) making requests to the database, in the Ordering Mix (50% of write operations).

Figure 3.2, shows the throughput of the Sequoia controller in normalized Web Interactions Per Second (WIPS), one of the metrics defined by the TPC-W implementation used. The Figure shows that the throughput of the system is not affected by the usage of jGCS in the whole system, since the number of WIPS over time is equivalent. Table 3.1 details even more, as it shows that latency results, when using the Appia toolkit, either through the native interface or through jGCS, are in practice the same. In fact, the difference is not statistically relevant, even with a very low confidence level, as confidence intervals overlap significantly. This shows that the use of jGCS is negligible in the overall performance of a complex system.

Implementation	**Mean**	**Std. Dev.**	**Samples**
Native	39.96	41.10	3846
With jGCS	40.26	52.97	3832

Table 3.1: Latency of client requests of TPC-W (ms).

Component	**Node 1 (lhufa)**	**Node 2 (lhona)**	**Node 3 (inha)**	**Node 4 (ucha)**
Processor	Dual AMD Opteron(tm) 2.4 GHz		Dual AMD Opteron(tm) 1.5 GHz	
Memory	4 GB			3 GB
Storage	One 55 Gb dedicated volume for each node.			
Network	1 Gbps (Ethernet)			
Operating System	Ubuntu 7.10			

Table 3.2: Hardware and SO Specifications.

3.2 Transaction Processing

The GORDA interface has been implemented on four different systems, namely, Apache Derby, PostgreSQL, MySQL and Sequoia. These bindings illustrate the effort required to implement the GAPI using different apporaches. Details about components implementations were already provided in Section 2.2.

The systems under test were deployed on a cluster of four HP Proliant machines. All shared a storage area network (SAN) by means of a fibre switch which connected the nodes to an EMC storage tank with 1 Terabyte capacity in RAID-5 configuration. A dedicated volume, sized at 55 GB, was assigned for each of the nodes. Complete hardware, network infrastructure and Operating System specifications is depicted in Table 3.2. Additionally, Figure 3.3 provides a graphical layout of the involved hardware parts. Node 3 was used to start the emulated clients that connected to the databases using a regular JDBC connection. Node 1 and 2 hosted the database instances. Finally, Node 4 was used to compute raw data output from benchmark runs.

Performance evaluation was conducted by resorting to the TPC-b [17] benchmark. TPC-b is a stress test oriented benchmark with an update intensive workload. It is characterized by significant disk I/O and moderate system and application execution time. Focus is put on the performance of the database management system with particular interest in the transaction processing. CPU stressing occurs due to the absence of user think time between operations, and I/O gets stressed, due to the mix of large amounts of small read and write operations within each transaction.

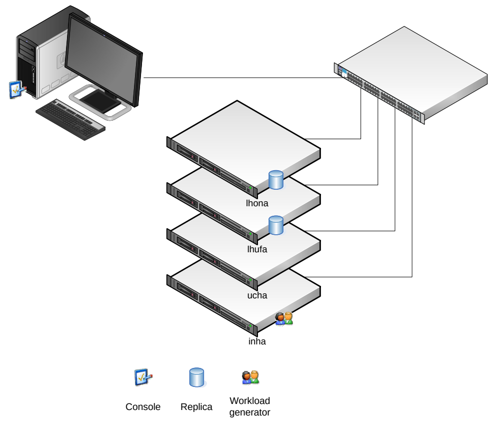


Figure 3.3: TPC-b benchmark runs hardware layout and description.

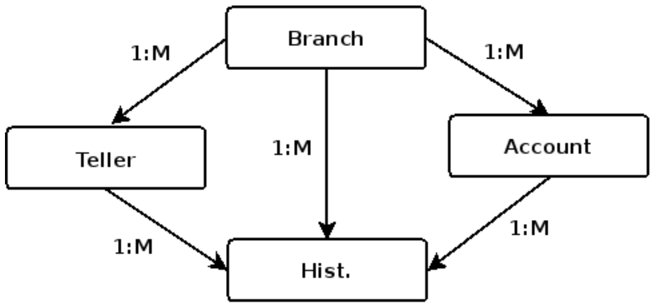


Figure 3.4: TPC-b Database Schema.

TPC-b simulates a bank information system. The bank is comprised of one or more branches each of which having several tellers and holding accounts, one for each costumer. As a whole, the database tracks the cash position of each entity (branch, teller and account). Alongside, there is also a record or history of recent monetary transactions run by the bank. Whenever a deposit or a withdrawal is issued by a customer, a transaction is performed by a teller at some given branch.

The database contains four tables: *branches*; *tellers*; *accounts*; and *history*. Their logical relation is depicted in Figure 3.4. Only one type of transaction profile exists in the system and an instance of this transaction is issued by every emulated customer everytime it iterates. Five SQL statments compose the transaction and are presented in Figure 3.2. *Aid*, *Tid*, *Bid* and *Delta* are generated according to the TPC-b specification on each transaction instantiation, i.e. on each costumer application iteration.

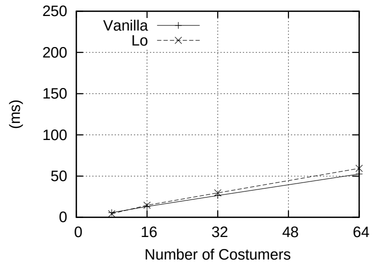
```
UPDATE accounts
        SET Abalance = Abalance + :delta
        WHERE Aid = :Aid;

SELECT Abalance
        INTO :Abalance FROM accounts
        WHERE Aid = :Aid;

UPDATE tellers
        SET Tbalance = Tbalance + :delta
        WHERE Tid = :Tid;

UPDATE branches
        SET Bbalance = Bbalance + :delta
        WHERE Bid = :Bid;

INSERT INTO history(Tid, Bid, Aid, delta, time)
        VALUES (:Tid, :Bid, :Aid, :delta, CURRENT);
```

Figure 3.5: TPC-b transaction profile.

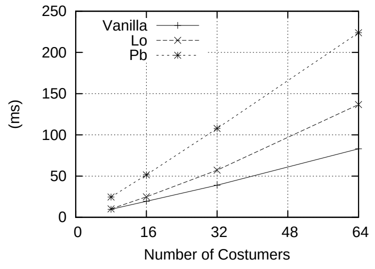
TPC-b costumer emulation was performed by integrating its implementation into Bristlecone. Bristlecone was also used to conduct experiments to assess read scalability of GORDA middleware reference implementation, Sequoia. These benchmarks were conducted by using Bristlecone workloads specifications.

The TPC-b benchmark choice was driven by three reasons: *i)* it provides an update intensive workload, meaning that the GORDA reflection API pipeline is mostly covered; *ii)* given its nature, it stresses out the reference implementations and API mappings; and *iii)* it is a benchmark that is supported on all the prototypes developed during the GORDA project.

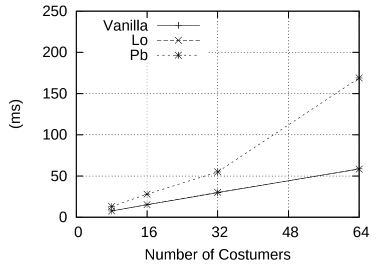
Ultimately, performance assessment relies on three metrics: transaction execution latency; transaction overall throughput; and transaction abort rate. Transaction execution latency provides a mean to measure the end-to-end overhead as observed by the end user. Overall throughput measures the average amount of work done successfully within a given period of time (we are considering one minute period). Although throughput already presents some hints on how the system behaves, it does not distinguish contention, transactions that get blocked waiting for others, from aborts, transactions that abort due to deadlocking or conflict resolution. This is our third metric, the abort rate. The results were obtained by conducting the runs several times and using a percentile of 0.95 when calculating the average values. This eliminated outliers values from the samples read resulting in excelent overall accuracy.



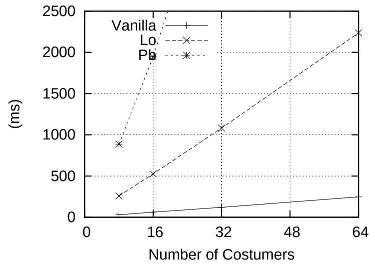
(a) Apache Derby



(b) PostgreSQL



(c) MySQL



(d) Sequoia

Figure 3.6: Average execution latency (ms).

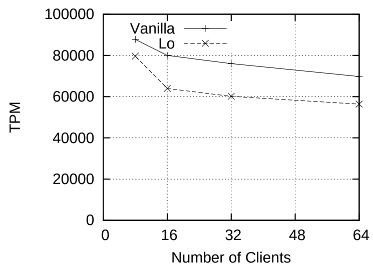
Finally, we have also conducted micro-benchmarking for assessing read scalability on several Sequoia proxies. The metric used in this case is the average number of queries per second.

3.2.1 Scenarios

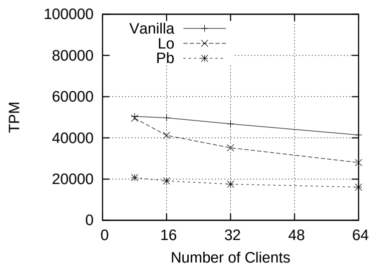
Scenario I: No Modifications (vanilla). TPC-b runs were performed against an unmodified version of the original component (either in-core - derby and postgresql, or middleware - sequoia). Results obtained here, represent the baseline for the evaluation. No intrusive GORDA patchs or GORDA overhead of any kind are imposed.

Scenario II: Loopback (lo). Overhead of the GORDA API reflection interface is assessed by performing runs against all three reference implementations. For each of the three prototypes, a single database instance is configured for GORDA notifications but coordination and communication stacks are set to loopback nature. The resulting notifications are handled by a fake listener that emulates the loopback behavior. No network communication and transaction dissemination overheads are imposed in this setting.

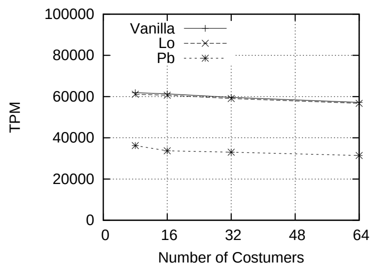
Scenario III: Primary-Backup (pb). Overhead of a replicated database in a cluster environment is assessed by conducting runs of TPC-b in a synchronous



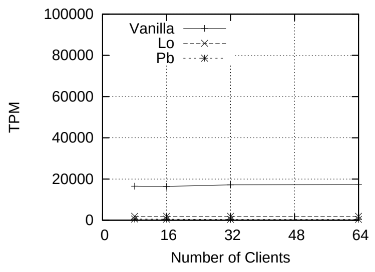
(a) Apache Derby



(b) PostgreSQL



(c) MySQL



(d) Sequoia

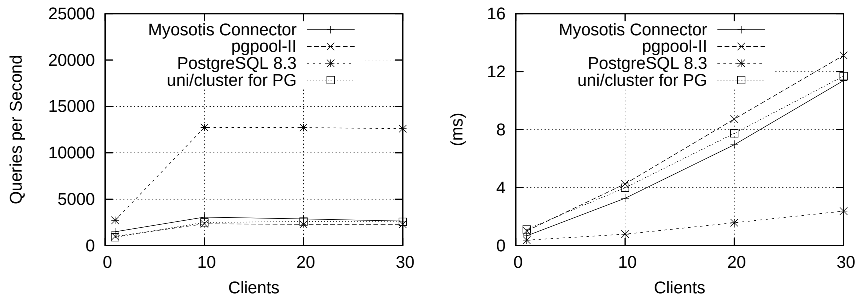
Figure 3.7: Average throughput (TPM).

primary-backup database replication scenario. The cluster contains two replicas, a primary and a backup. Reflection, communication and coordination overheads are assessed, given the synchronous nature of the replication protocol. The synchronous property, means that the primary only replies to the customer once the backup replica acknowledges the reception of the transaction. Node 1 and 2, from Table 3.2 were used to host primary and backup instances, respectively.

Scenario IV: Micro-Benchmarks Micro-benchmarks provide detailed tests of specific operations with the ability to vary test parameters like result set sizes and numbers of clients systematically. Table 3.3 describes types of micro benchmarks developed during the GORDA project timeframe. Bristlecone permits many different types of tests of scale-out architectures. For example, it can help evaluate proxy throughput and latency. Such latency is an important consideration for scale-out solutions that use proxies. Sequoia mapping is itself a proxy, as it is based on the original Sequoia software. Bristlecone provides a basic benchmark for latency entitled ReadSimpleScenario. This scenario selects all rows from a table of configurable length using a simple SQL query of the form shown below:

```
SELECT * from benchmark_scenario_0
```

Each client in the test repeats this query as fast as it can. The table used for testing is relatively small and there is no other activity on the database,



(a) Query Throughput. (b) Proxy Latency Overhead.

Figure 3.8: Query micro-benchmarks.

which guarantees that the table contents will be in the server buffer cache, thus rapidly accessible. Such a benchmark is helpful in establishing boundary conditions for proxy performance, since it makes overhead induced by proxies very easy to distinguish. Benchmark tests were run using the following hardware configuration: Dell SC 1425 dual-core dual-CPU Xeon, 1 GB memory, SATA disks.

3.2.2 Discussion

Apache Derby. Figure 3.6(a) shows that increasing transaction latency as more customers are added to the system. This is due to higher contention rate in queuing and locking. There is also an increased latency penalty as we go from a vanilla deploy to a full synchronous primary-backup replication setting. The additional latency in the loopback (lo) runs are due to extra calls to GAPI notification methods. Such calls are performed by a different thread from the one in transaction execution, forcing a context switch thus worsening latency penalty. Although not a bug, this is sure lacking an optimal implementation, which is scheduled to happen in one of the next releases. The impact on throughput is presented in Figure 3.7(a). It shows that increased load (number of customers) leads to higher contention rate in locks resulting in a slight decrease in the overall throughput. Note that the benchmarks were conducted with *read committed* isolation level, thence no aborts have been observed. **Note:** We were unable to run Primary-Backup scenario using Apache Derby. We faced some exceptions during the execution and were unable to solve them in time to deliver this report.

PostgreSQL. Figure 3.6(b) presents latency results for PostgreSQL. We can find the same pattern as the one observed in Apache Derby. However, unlike Apache Derby, PostgreSQL uses operating system processes instead of threads

Micro-Benchmark Category	Description
Read Latency	Measures latency on queries with ability to generate result sets running to many millions of rows. Ideally suited to testing performance overhead of middleware solutions like Sequoia.
Read Scaling	Measures performance of queries that perform resource intensive queries. Current benchmarks stress CPU and buffer cache (shared memory).
Write Latency	Benchmarks to test performance of simple update, updates that include queries (e.g., SQL UPDATE requests), as well as mixed transactions.
Deadlocks	GORDA testing and is incorporated in the benchmark suite. Measures number of deadlocks/aborts as transaction size scales. Designed to test middleware and certification approaches that offer one-copy serializability.

Table 3.3: Micro-benchmarks types developed during GORDA timeframe.

to handle incoming connections and foster concurrent transaction processing. This makes difficult interaction between our Java-based replication engine and the GAPI PostgreSQL binding. Although masked by using the existing PL/J binding to Java, as explained previously, the context switch and the inter-process communication (and the respecive necessary serialization) contribute to the increased latency. Furthermore, collecting a transaction write set is done by installing table level triggers that fire changes into an in-memory hash table. This poses an additional negative performance impact. The synchronous primary-backup replication results, very much like Apache Derby, present the same overhead induced by serialization and data propagation. Nevertheless, although serialization and propagation are pretty much the same in Apache Derby and PostgreSQL, because are both handled by the ESCADA replicator, applying remote updates in the backup replica poses much more *background noise*.[1] Ultimately, this leads to more load on the backup replica related to all the requirements needed for the apply process, thence delaying delivery acknowledgment to primary, resulting indirectly in latency penalty on other transactions.

MySQL. MySQL matches Apache Derby, in terms of latency results shown in the *Vanilla* and *Lo* runs. This is depicted in Figure 3.6(c). MySQL uses a multi-threaded core, which leverages concurrency and optimizes resource consumption. As such this has direct impact on the overall performance and throughput. Hooks placed inside MySQL core pose negligible overhead when compared with

[1]Remote updates are applied using JDBC client connections, but it seems that PostgreSQL performs worse and needs more resources than Apache Derby to do the same amount of work.

the unmodified version of the DBMS, as it is shown by the loopback run. This means that even the most complex operation of extracting the write-set, which is done by inspecting MySQL binary-log events, has little impact on the average execution latency. The primary-backup replication run, shows that there is a significant latency penalty as more customers are added to the system. The additional latency is imposed by queuing, serialization and message propagation leading to a significant performance loss. The overall throughput of the system is depicted in Figure 3.7(c). One can easily note that as the primary-backup run exhibits twice the latency of the unmodified single database and loopback run, the throughput is cut in half. Furthermore, there is a trend in increased latency and constant throughput. This happens due to queuing effect which results in a steady debit of committed transactions but increased latency due to time spent waiting in the resource queues.

Sequoia. Sequoia approach presents an overwhelming latency increase even in the vanilla runs, as it performs table level locking. In TPC-b there is only one type of transaction (see Figure 3.2), that accesses all tables. Consequently, transaction execution is serialized, increasing queuing and therefore contention rises. Figure 3.6(d) shows the results. Note that the scale is tenfold of the previous two approaches. In the loopback results, there is additional overhead related to the GAPI notifications, featuring the collection of the write set. This is done resorting to table triggers that fire changes into temporary tables, existing during connection lifecycle. This has a cascading effect and worsens as more costumers are added (number of connections increases). As for the primary-backup replication scenario, the latency overhead builds on the loopback issues and on the need for extracting write sets, which is done using client-side JDBC connections and SELECT statements. Additionally, data serialization and propagation to the backup replica also impact latency. Queuing and contention impact negatively on throughput, especially because this is a closed system in which clients are sequential, thence if execution latency increases, less transactions per second will be submitted. As a consequence, less transactions per second will be committed. This is shown in Figure 3.7(d).

Micro-benchmarks. Figure 3.8 diagram shows typical graphical output generated from the run. This experiment compares read throughput of several proxies including Myosotis, which is a proxy used to provide native DBMS client wire-protocol support for Sequoia.

Figure 3.8(a) and 3.8(b) show comparisons of read latency using runs of 1, 10, 20, and 30 clients operating directly against the PostgreSQL database as well as through 3 proxy implementations:

- Myosotis connector (Continuent's transparent proxy for native database clients);
- Pgpool-II (A load-balancing proxy for PostgreSQL written in C);
- Sequoia (Continuent's uni/cluster implementation of Sequoia)

Figure 3.8(a), shows query throughput for all proxies as loads scale from 1 to 30 clients. The database is the base—native throughput is bound by database host CPU and tops out at slightly over 12,700 queries per second. This is the maximum possible throughput. Proxy output by contrast tops out around 3,000 queries per second. In both cases, throughput is bound by CPU on the database host machine, which also has the proxy.

Figure 3.8(b), by contrast evaluates the exact latency induced by a proxy on a single request. Query response from the native PostgreSQL server sets the baseline response time. The other series show response time for queries sent through proxies. The query latency can be read off the graph by comparing the baseline response for each query. This graph demonstrates that query latency increases linearly for larger numbers of clients, as available CPUs are completely occupied.

Scale-out environments may need to handle enormous numbers of queries. The foregoing numbers are helpful in capacity planning for such scale-out. For example, an environment that processes 6000 queries per second might need to budget 3 hosts of the type shown here when proxying through Sequoia. Interestingly, this test also indicates that proxies and middleware are a significant performance drag for applications that have very short queries on data entirely resident in memory. Middleware offers a performance scaling benefit only as the cost of queries begins to outweigh the latency effects.

Chapter 4

Conclusion

Results show that the jGCS interfaces can be implemented using most of the state of the art group communication toolkits. It is also shown that the overhead caused by the jGCS service is negligible and do not affect real applications, improving modularity and configurability. This service was implemented in Java and is hosted at SourceForge.net (http://jgcs.sf.net).

Performance of GORDA interfaces was assessed in four different mappings: Apache Derby, PostgreSQL, MySQL and Sequoia controller. Results have shown that the middleware approach has some disadvantages as it redoes most of the DBMS work outside in the middleware and presents a very coharse concurrency control grain. Nevertheless, it performs well for read-oriented workloads. On the other hand, other mappings, either as a plugin, user level procedures or deep in-core implementations show good performance even in stressful update-intensive workloads, as is the case for TPC-b. Implementations are available at the GORDA web-site (http://gorda.di.uminho.pt).

Bibliography

[1] Derby. http://db.apache.org/derby/.

[2] GnuPlot. http://www.gnuplot.info/.

[3] The R Project for Statistical Computing. http://www.r-project.org/.

[4] Y. Amir, C. Danilov, and J. Stanton. A low latency, loss tolerant architecture and protocol for wide area group communication. June 2000.

[5] B. Ban. Design and implementation of a reliable group communication toolkit for java, 1998.

[6] Continuent. Bristlecone Project. http://bristlecone.continuent.org/.

[7] Continuent. Sequoia version 2.9. http://sequoia.continuent.org, 2006.

[8] PostgreSQL Global Development Group. PostgreSQL. http://www.postgresql.org, 2003.

[9] M. Hayden. *The Ensemble System*. PhD thesis, January 1998.

[10] InnoBase. InnoDB. http://www.innodb.com.

[11] Lasige Research Lab. jGCS: Group Communication Service for Java. http://jgcs.sourceforge.net/.

[12] Mikko H. Lipasti. Tpc-w in java. http://www.ece.wisc.edu/ pharm/tpcw.shtml.

[13] H. Miranda, A. Pinto, and L. Rodrigues. Appia, a flexible protocol kernel supporting multiple coordinated channels. April 2001.

[14] SUN Microsystems Inc MySQL AB. MySQL. http://www.mysql.com, 1995.

[15] J. Pereira, L. Rodrigues, M. J. Monteiro, R. Oliveira, and A.-M. Kermarrec. Neem: Network-friendly epidemic multicast. In *Proceedings of the 22th IEEE Symposium on Reliable Distributed Systems (SRDS'03)*, pages 15–24, Florence,Italy, October 2003.

[16] Inc Sun Microsystems. Java Native Interfaces. http://java.sun.com/j2se/1.5.0/docs/guide/jni/spec/jniTOC.html, 2003.

[17] Transaction Performance Council. TPC Benchmark B. http://www.tpc.org/tpcb.

[18] Walter Moreira and Gregory R. Warnes. RPy (R from Python). http://rpy.sourceforge.net/.